Complex Musical Behaviours via Time-Variant Audio Feedback Networks and Distributed Adaptation: a Study of Autopoietic Infrastructures for Real-Time Performance Systems

Dario Sanfilippo

Doctor of Philosophy
Creative Music Practice
Reid School of Music
The University of Edinburgh
2019

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

Dario Sanfilippo

Acknowledgements

I would like to thank:

my supervisors Martin Parker, Michael Edwards, Michael Newton, for the priceless support and guidance that they have provided from the beginning until the very end of my studies, both as professionals and as friends;

the people who were part of my life during my years in Edinburgh;

my friend and mentor Agostino Di Scipio, whose brilliance has never ceased to amaze me;

my family, for their constant support and love.

To my dear brother, who has always made my world a better place.

Abstract

The research project presented here is a study of the application of complex adaptive systems (CASes) for live music performance and composition by fully autonomous or semi-autonomous machines. The fundamental artistic concept and the motivating idea behind this project are that complex systems are an optimal model for creative music practice as they operate at the *edge of chaos*: that is, a condition where there is an interplay between order and disorder, or patterns and surprise. Arguably, this is an essential element found in music regardless of its genre or style.

The central research questions addressed by this project are: how to realise music systems with an abstract yet structurally coherent and contextually complex output that display organicity and resemble aliveness? How to design music organisms so that artificial expressiveness and formal developments emerge and generate musical behaviours that contribute to the ongoing exploration at the edges of new music practices? How to create audio networks that are responsible for their structure and organisation where a substantial autonomy expands the paradigm of human-machine interaction?

The methodology used in this project for the implementation of the music systems applies theories from complexity and adaptation, biology, and philosophy within nonlinear time-variant self-modulating feedback networks. The structural coupling between system and performer is realised by following a cybernetic approach in human-machine interaction and human-machine interfacing.

The music systems developed for the creative works in this research are a combination of interdependent algorithms for the processing of information and the synthesis of sound and music. A technique formulated to improve the complexity of music systems is referred to as *distributed adaptation*, related to the notion of *evolvability* in biology and genetic algorithms. Distributed adaptation consists of making the adaptation infrastructure itself adaptive and time-variant by employing emergent sensing mechanisms for the generation of information signals, and emergent mapping functions between information signals and state variables. This framework realises the idea for which information and information processing recursively determine each other in a radical constructivist fashion with the important consequence that the machine ultimately constructs its reality as a self-sensing, self-performing, and context-dependent entity.

This research includes seven music performances, each implementing CASes with or without human intervention. Also included is a library of original software algorithms for low-level and high-level information processing written in Faust. Chronologically ordered, the performances depict the progress of the study, starting with systems having basic adaptive characteristics and eventually revealing the more advanced ones where the distributed adaptation design is applied. Through self-reflection and post-hoc analysis, case studies illustrate that the combination of CASes and cybernetic performance and interfacing, and particularly distributed adaptation systems with or without human agents, produce a sophisticated musical output whose evolutions are complex, coherent and expressive.

These results suggest that the emergent behaviours of such systems can be deployed as a means for the exploration of new music in practice. Furthermore, the autonomy and contextual nature of these systems suggest that promising results can be achieved when applying them to other fields involving audio, especially where interactivity with the surrounding environment is crucial, or when using them as musical instruments for users with special needs.

Contents

Abstract viii

1 Introduction 1

1.1 A brief history of complexity and music feedback systems . . . 2

1.1.1 Examples of feedback-based music . . . 3

1.2 A remarkable practitioner in the field of adaptive ecosystems: Agostino Di Scipio 5

1.3 The aesthetics of music complex adaptive systems . . . 6

1.3.1 From failure to *extreme functioning* . . . 6

1.3.2 The objectivity of the machine . . . 8

1.3.3 Losing control to gain complexity . . . 11

1.4 Innovative aspects of this research . . . 12

1.5 Portfolio . . . 14

1.5.1 Audio documentation . . . 15

1.6 Remarks . . . 16

2 Complex adaptive systems 19

2.1 Feedback mechanisms . . . 20

2.1.1 Properties of feedback mechanisms . . . 21

2.2 Chaos . . . 23

2.2.1 Fixed-point attractors, periodic oscillations, strange attractors, chaos . . . 24

2.3 Emergence . . . 28

2.3.1 The role of the observer . . . 29

2.4 Autopoiesis . . . 30

2.4.1 Structural coupling . . . 31

2.5 Complexity and adaptation . . . 32

2.5.1 Case study: *Phase Transitions* (2018-2019) . . . 34

2.6 Remarks . . . 37

3 Distributed adaptation 39

3.1 System-context structural coupling and information . . . 40

3.1.1 Detecting and affecting infrastructures . . . 41
3.1.2 Adaptation and time-variance . . . 41
3.1.3 Case study: *Inexorable Shifting 2* (2019) . . . 42
3.2 Emergent infrastructures . . . 46
3.2.1 Case study: *Order from Noise (Homage to H. von Foerster)* (2016-2019) . 47
3.2.2 Case study: *Constructing Realities* (2019) . . . 49
3.3 Information processing techniques . . . 51
3.3.1 Low-level information processing . . . 51
3.3.2 High-level information processing . . . 74
3.4 Remarks . . . 80

4 Audio processing techniques and DSP implementation 83
4.1 Sound transformations . . . 84
4.1.1 Granulation . . . 84
4.1.2 Sampling . . . 90
4.1.3 Nonlinear transfer functions . . . 91
4.1.4 Modulations . . . 92
4.1.5 Feedback delay networks . . . 93
4.1.6 Filtering . . . 94
4.2 Stability processing . . . 95
4.2.1 Bounded saturators . . . 95
4.2.2 Lookahead limiting . . . 96
4.2.3 Adaptive compression . . . 98
4.3 Faust: a functional programming language for real-time signal processing . . . 99
4.3.1 *Phase Transitions* code . . . 100
4.3.2 *Inexorable Shifting 2* code . . . 106
4.3.3 *Constructing Realities* code . . . 107
4.4 Pure Data: a patchable environment for audio analysis, synthesis, and processing 114
4.5 Remarks . . . 115

5 Performance modalities and interfaces 117
5.1 Cybernetic improvisation . . . 117
5.1.1 Case study: *Human Network: Machine Nostalgia* (2016-2018) . . . 120
5.2 Cybernetic mapping . . . 122
5.3 Reduced intervention . . . 123
5.3.1 Case study: *Single-Fader Versatility* (2016) . . . 124
5.4 Electroacoustic devices and the environment as interfaces . . . 127
5.4.1 Case study: *Audible Icarus* (2016-2018) . . . 128
5.5 Remarks . . . 130

6 Conclusion **133**

6.1 Future work 136

A *Edge of Chaos* library code **139**

A.1 alleoc.lib 139

A.2 auxiliary.lib 140

A.3 delays2.lib 142

A.4 edgeofchaos.lib 145

A.5 filters2.lib 146

A.6 information.lib 172

A.7 maths2.lib 196

A.8 oscillators2.lib 243

A.9 outformation.lib 251

A.10 stability.lib 264

List of Figures

1.1 Human-machine improvisation configuration. The "human" element represents a human agent performing actions on a non-autonomous machine. The "machine" element is a sound-generating system that responds to the actions of the human. The output of the machine, eventually, reaches the human agent back after a delay. The human agent is not affected by other signals other than the output of the machine as, for this specific case, it is assumed to bypass the environment. 9

2.1 General feedback configuration. In this diagram, "system" represents a generic function that transforms an input signal and outputs the result. The result is then fed back into the system after a delay and combined with the input. 20

2.2 Nonlinearity and iteration as minimum requirements for chaotic behaviours. . . . 23

2.3 Lorenz system state space for 10^5 iterations depicting a fixed point. $\rho = 10$. . . . 26

2.4 Lorenz system state space for 10^5 iterations depicting a periodic oscillation. $\rho = 18$. 26

2.5 Lorenz system state space for 10^5 iterations depicting a strange attractor. $\rho = 28$. 27

2.6 Lorenz system state space for 10^5 iterations depicting a chaotic behaviour. $\rho = 120$. 27

2.7 Lorenz system state space for 10^5 iterations depicting a long-period oscillation. $\rho = 125$. 28

2.8 Emergent process as retroactive relationships between the parts (low level) and the whole (high level). 29

2.9 Autopoietic configuration. The mutually determining components-organisation couple represents a system that retroactively affects and is affected by its environment ("env"). 31

2.10 Adaptation configuration. The system can change its organisation and internal configuration of its agents through the dynamics of its context mediated by a goal. The direct feedback loop between agents and context represents an exchange of energy. 33

2.11 *Phase Transitions*: outer layer block diagram. The "input" block contains the initial conditions through which the system is initialised. "Nltf" contains four sets of four different bounded saturators. "Dcblocker" prevents the build-up of steady signals as they would not be optimal for the development of the behaviours of this system in particular. The resulting signals are then duplicated and processed through scattering matrixes in the blocks "matrix". The "fb" module contains the adaptive mechanism that is applied to each signal to determine its magnitude. The resulting signals are then processed through delay lines whose delay is as well adaptive and determined by the signals themselves, which are ultimately fed back into the "input" block. The signals from "dcblocker" are mixed and used as main outputs. 36

3.1 System-context structural coupling and information as an emergent process. The goal of the system is implicitly contained in the information processing infrastructure. The direct feedback loop between agents and context represents an exchange of energy. 41

3.2 *Inexorable Shifting 2*: outer layer block diagram. "Ins" contains the initial condition that triggers the network. The "delays" block contains the non-transposing delay lines with the adaptive mechanisms to determine the delay lengths. "Limiters" are lookahead limiters and "agents" contains the SSBM units with adaptive shifts. The resulting signals are mixed and used as main outputs while also being sent back and redistributed through a scattering matrix. 45

3.3 *Order from Noise* Pure Data main patch. 48

3.4 *Constructing Realities*, outer layer block diagram. "Agents" contains the six adaptive networks carrying out the audio processing of ZC granulation, state-variable filtering (high-pass/low-pass/band-pass/all-pass/band-stop), sampling, reverberation, feedback delay network with state-variable filtering. "Topologies" contains four topology types that are selected based on complexity analysis, using linear interpolation among the topologies. The topologies are then used in "matrix" to combine the agents. The outputs from the agents are processed through adaptive delays, limited, and used as main outputs after being mixed. The outputs from "limiters" are also sent back to the agents through adaptive feedback coefficients and mixed with an external signal. 50

3.5 Block diagram of the spectral tendency algorithm. The green blocks are external signals: "in" is the analysed signal, "window" sets the analysis rate. "Spec_bal" outputs the difference between the RMS of the high-passed signal and the low-passed signal coming out of a crossover. The output is divided by the RMS of the input signal and multiplied by the analysis rate. The result is then divided by the samplerate and integrated. The signal is finally raised to the power of 4, fed back to the cut-off of the crossover in "spec_bal", and mapped over $Nyquist$. 52

3.6 Spectral tendency response with a 15 Hz analysis window and a 1 kHz sinusoid as test signal. 57

3.7 Spectral tendency response with a 15 Hz analysis window and white noise as test signal. 58

3.8 Block diagram of the reduced-computation spectral tendency algorithm. The green blocks are external signals. "In" is the analysed signal that is first processed through a raw-coefficients crossover. Balance computes the RMS difference between upper and lower parts from the crossover which is then divided by the RMS of the input for normalisation. The output is then integrated, squared, and sent back to the crossover cut-off. 59

3.9 Reduced-computation spectral tendency response with a 1 kHz sinusoid as test signal. 59

3.10 Reduced-computation spectral tendency response with noise as test signal. 59

3.11 Block diagram of the noisiness algorithm. The green blocks are external signals. "In" is the input signal to be analysed and "window" is the analysis rate. "Spec_ten" and the multiplication by "ny" calculate the spectral tendency of the input in Hz. The result drives the analysis rate of the "zcr4" block, which is a zero-crossing rate calculation with four cascaded one-pole low-pass filters. The ZCR is mapped through a logarithmic scale and then differentiated over a period that is determined by the spectral tendency. The magnitude of the ZCR variations is averaged and divided by a constant for normalisation. 60

3.12 Noisiness response with a 1 Hz analysis window and a 1 kHz sinusoid as test signal. 62

3.13 Noisiness response with a 1 Hz analysis window and band-passed noise (cut-off 1 kHz, Q 1) as test signal. 63

3.14 Noisiness response with a 1 Hz analysis window and band-passed noise (cut-off 10 kHz, Q 1) as test signal. 63

3.15 Noisiness response with a 1 Hz analysis window and white noise as test signal. . 64

3.16 Block diagram of the roughness algorithm. The green blocks are external signals. "In" is the analysed input and "window" sets the analysis rate. "Instant_amp" outputs the instantaneous amplitude of the input signal. The result is differentiated over a 0.001-second period and normalised through its RMS. The magnitude is averaged over a 75 Hz rate, differentiated over a 1/150 seconds period, averaged again and finally normalised. 64

3.17 Magnitude response of the analytic signal. 66

3.18 Phase difference between real and imaginary parts. 67

3.19 Roughness response with a 1 Hz analysis window and a 1% duty cycle rectangular chirp as test signal. 68

3.20 Block diagram of the lowest partial algorithm. This algorithm is an extension of the spectral tendency algorithm. The main difference is a low-pass filter at the top of the chain with its cut-off being driven by the feedback-path signal. 70

3.21 Lowest and highest partial response with a 15 Hz analysis window and eight sinusoids with randomly chosen frequencies as test signal. 71

3.22 Block diagram of the spectral spread algorithm. The green blocks are external signals. "In" is the analysed input, and "window" is the analysis rate. The measurement of spectral tendency of the input, in Hz, drives the cut-off of a biquadratic band-pass with a Q of 10. The RMS of the output of the filter is divided by the RMS of the input signal. Finally, the output is spectral spread is calculated as the complement of the ratio. 72

3.23 Spectral spread response with an analysis window of 5 Hz and noise band-passed with a progressively increasing Q as test signal. 74

3.24 Block diagram of the dynamicity algorithm. The green blocks are external signals. There are three main processing paths to calculate the dynamicity based on RMS, spectral tendency, and noisiness. Each path maps the input on a logarithmic scale, differentiates over a 1/15-second period, and averages the magnitude of the variations. The three paths are then nonlinearly combined through a hyperbolic tangent function. 75

3.25 Block diagram of the complexity algorithm. The green blocks are external signals. There are three main processing paths based on RMS, spectral tendency, and noisiness. Each path maps the inputs logarithmically and calculates the heterogeneity. The result is normalised based on the state space size of the input in each path, averaged, and differentiated over a 16-second period. The magnitudes of the variations are summed, averaged, and nonlinearly combined through a hyperbolic tangent function. 76

4.1 Reconstruction of a sinewave at 1 kHz through 879 grains per second and random grain positions. Comparison of spectra between original and reconstructed sinewaves. 89

4.2 1000 grains per second looping of a portion of a drum sample. Visualisation of grains start. 90

4.3 Block diagram of the lookahead limiter. The green blocks are external signals. “In” is the input signal, “lim” is the limiting threshold. The input is processed through a peak holder and smoothed out with a low-pass filter followed by a peak envelope. The resulting signal is used to divide the limiting threshold to calculate a scaling factor. Amplification factors are discarded while attenuation factors are applied to the input after being delayed by 0.002 seconds. 96

4.4 Block diagram of the dynamical compressor with RMS analysis. The green blocks are external signals. “In” is the input signal, “window” is the analysis rate, and “curve” is the exponent of the scaling curve. The input signal is multiplied by the complement of the RMS of the input signal itself, after being raised to an exponent to render nonlinear behaviours. 98

5.1 *Human Network: Machine Nostalgia* score. “Loudness”, “brightness”, “noisiness”, and “density” are the information criteria. “All”, “env”, “right”, and “left” are the sources from which to extract information. “><”, “<>”, “low”, and “high” are the functions to be applied to the criteria. “Section” is the row for the sections of the piece. “Duration” is the row for the durations of each section. 122

5.2 *Single-Fader Versatility*: Pure Data main patch. 126

5.3 *Audible Icarus*: Pure Data main patch. 130

Chapter 1

Introduction

What we call knowledge does not and cannot have the purpose of producing representations of an independent reality, but instead has an adaptive function.

Ernst von Glasersfeld

The first chapter of this thesis provides a brief introduction to feedback-based music, starting from the early practitioners up until the most advanced methods and state-of-the-art works realised in the field. Some of the most relevant techniques developed over almost six decades of investigations in the area of recursive systems for electronic music are exposed to show the variety and richness that a single specialised domain can have, providing examples of how scientific and philosophical principles could be translated into music.

The historical context is key to understand the evolution of the field. Feedback-based music arose during the same years in which cybernetics, together with other disciplines, were experiencing a profound transformation, as we will see in the following section. This introductory chapter also provides a quick overview of how such disciplines changed, highlighting the connections between seemingly distant areas such as philosophy, biology and engineering, and presenting key terms, some of which will be investigated thoroughly in chapter 2 Complex adaptive systems. The development of feedback-based music appears to have followed somewhat closely the evolution of systems thinking.

Furthermore, this chapter explores questions of musical aesthetics related to the use of complex autonomous systems in live performance, and it describes the most salient innovative aspects of the research.

1.1 A brief history of complexity and music feedback systems

The Macy Conferences, which lasted for nearly twenty years starting from the early 1940s, was a fundamental series of events attended by several scholars from different disciplines sharing the perspective of cybernetics and systems thinking. Some of the most prominent figures who attended the meetings were Ross Ashby, Margaret Mead, Gregory Bateson, Heinz von Foerster, and Norbert Wiener. The latter is considered the father of cybernetics as he first defined the term in his *Cybernetics or control and communication in the animal and the machine* [Wiener, 1948]. Modern cybernetics began as a discipline interconnecting different fields such as network theory, mechanical engineering, evolutionary biology, and psychology, with a focus on the study of systems with interdependent agents having nontrivial behaviours. Eventually, cyberneticians had the intuitions that the process of analysing a system could itself be nontrivial, and they started to consider the issues related to the role played by the observer. As stated by Margaret Mead and Gregory Bateson in an interview from 1976 [Brand, 1976], classic cybernetics was considering systems as elements independent from their surround environment or observer. Cybernetics then evolved into second-order cybernetics, which Heinz von Foerster defined as "the cybernetics of observing systems" [Von Foerster, 2003a].

It is possible to notice an emphasis on the system as a cognitive and self-referential entity capable of sensing its context [Etxeberria et al., 1994, Barandiaran and Moreno, 2006]. Coherently to this theory, during the same period, the notion of radical constructivism was being formulated by Ernst von Glasersfeld. To approach the concept of self, he used the metaphor of an invariant condition resulting from the counterbalancing mechanism of mutually affecting changes. He identified such a metaphor in the cybernetic domain as a self-regulating closed-loop, essentially paraphrasing the notion of circular causality: what we see in a feedback loop is the past being affected by the present which is about to be compensated by the immediate future. Hence, it is not possible to depict the state of such a mechanism through a single element, for, by nature, it consists of one or more relationships, and relationships are between things rather than in things [von Glasersfeld, 1979]. What we experience as *self*, it takes place in the circular relationship between the entity and its surroundings.

The early studies on complexity are from at least the 1970s with Edgar Morin and V. Rao Vemuri [Morin, 1977, Vemuri, 2014], but several others were researching the same field from different directions. For example, Prigogine was investigating dissipative structures, non-equilibrium thermodynamics, and the function of time in biological systems [Prigogine and Nicolis, 1985]; the theories on autopoiesis by Maturana and Varela [Maturana and Varela, 1980]; Kauffman and his work on random Boolean networks showing the emergent evolution of their self-organisation [Kauffman, 1984]; the research on artificial life by Langton [Langton, 1986]. In the 1970s, John H. Holland implemented a computational model of adaptation in

evolutionary systems inspired by the work of Rosenblatt [Rosenblatt, 1958]. Holland published his early research on genetic algorithms in 1975 [Holland John, 1975], and he was a distinguished researcher in the field of adaptivity [Wilensky and Rand, 2015]. Holland also became part of the Santa Fe Institute[1] in 1985, a place where some of the most prominent CASes thinkers like Melanie Mitchell and James Crutchfield currently work.

CASes are now used in several fields, and they have gained substantial importance during the years. Some of their applications are to predict and understand complex real-world phenomena through computational models for economic trends or the development of technological progress [Farmer, 2002, Farmer and Lafond, 2016], the evolution of intelligence [Krakauer, 2011], or global behaviours in societies [Lagi et al., 2011]. CASes are also applied in the implementation of artificially intelligent systems like self-repairing software and intelligent anti-virus systems [Forrest et al., 1997], or robotics and linguistics [Steels, 1997, Steels, 2003].

1.1.1 Examples of feedback-based music

Feedback loops, both within the system and between system, environment and observer, are essential elements in second-order cybernetics that, due to the success of the discipline, became popular in different areas including music. Some of the earliest examples of feedback-based music are from the 1960s. Roland Kayn drew inspiration from cybernetics and implemented self-regulating music systems based on feedback, both as models for instrumental pieces as well as analogue networks for sound generation [Patteson, 2012]. During the same period, technologies and techniques for sound reinforcement were being investigated, which favoured the studies in signal processing and acoustics for analogue audio equipment. Specifically, the phenomenon of Larsen, named after the Danish scientist Sren Absalon Larsen who discovered it a few decades earlier, raised particular interest and was a significant concern in the field of analogue audio and sound reinforcement. While sound engineers were researching techniques to prevent the Larsen effect [Boner and Boner, 1966], some music practitioners had the intuition that such a phenomenon may be a powerful creative means for music composition and performance.

Robert Ashley's *The Wolfman*, from 1964, is a work for tape, voice and feedback. In the same year, John Cage composed *Electronic Music for Piano*, which is a piece for piano and Larsen network. Steve Reich's *Pendulum Music* from 1968 also explores the Larsen effect and its modulation through the repositioning of microphones with respect to the loudspeakers. Specifically, he uses microphones as pendulums over the loudspeakers, hence investigating two fundamental aspects of Larsen itself: phase, given by the angle of incidence of sound on the microphone, and gain, given by the distance between microphone and loudspeaker. In 1969, Alvin Lucier composed *I Am Sitting in a Room*, which is a particularly interesting use of the Larsen phenomenon. Lucier's piece consisted of two tape recorders, a microphone and a loudspeaker. The microphone recorded his voice as he was reading a short text that lasted about

[1] https://www.santafe.edu/about. Accessed on the 29th of August 2019.

75 seconds. The recorded voice would then be played back in the room through the loudspeaker and recorded on the other tape recorder through the microphone. Thus, the two tape recorders were needed to perform the task of playing back and recording simultaneously. Today, the same process could be achieved by using a long-enough delay line. The exciting aspects of Lucier's piece are that he stretched out in time the recursive process of the Larsen effect, that is, the circular action of reproducing and capturing sounds through microphones and loudspeakers. Lucier gave the possibility to inspect this recursive process on a different time scale, but he also demonstrated that slowing down such a systemic process would result in different emergent behaviours, rather than merely slowing down the final output itself. Furthermore, Lucier very consciously recognises the environment as part of the process – the room is indeed a filter that shapes the Larsen – underlining the strong ecosystemic nature of the process itself.

In 1966 and 1967, Gordon Mumma composed, respectively, *Diastasis, as in Beer* and *Hornpipe*. Mumma realised groundbreaking works as they are arguably some of the very first music pieces implementing elementary forms of CASes. In particular, Mumma implemented analogue feedback networks coupled with the environment through instruments that he used as filters or transfer functions in general. Control signals that he mainly generated using envelope followers piloted some of the parameters in the processes that transformed the sounds from the instruments, resulting in self-regulating adaptive networks [Mumma, 1967]. Nicolas Collins realised his *Pea Soup* in 1974. Similar to the work of Mumma, Collins also implemented envelope followers to pilot the parameters of a self-regulating network. In his case, in particular, the network only consisted of one positive feedback loop and one negative feedback loop and their cooperation: the positive feedback loop was the Larsen effect given by the iterated amplification of a signal through microphones and loudspeakers; the negative feedback loop resulted from controlling a phase shift in the input signal through its envelope contour. Loud signals, which indicated the onset of Larsen, phase-shifted the signal of a higher amount. Since feedback is a function of phase, the result was that the Larsen effect would suppress itself whenever enough energy built up, giving the possibility to different tones to emerge in different areas of the spectrum. Despite the simplicity of the network, the expressiveness and articulations of the sonic output were outstanding. For a more detailed analysis of some relevant feedback-based pieces see [Sanfilippo and Valle, 2013].

Today, along with Agostino Di Scipio, who will be discussed in thenext section, Alice Eldridge has provided some of the most important contributions in the creative practice with adaptive systems. She is a composer and performer working with improvisation with a background in psychology, adaptive and evolutionary systems, and informatics. Her work focuses on ecosystemic approaches of music creation and performance with emergent evolutionary models and adaptive systems [Eldridge, 2007, Eldridge et al., 2008]. Within the field of feedback systems, in particular, in collaboration with Chris Kiefer, she has developed the *Feedback Cello*, which is a self-oscillating instrument that bridges physical tools and computational methods.

The Feedback Cello is an autonomous instrument capable of adaptation that can be implemented in human-machine interactions performances, or sound installation for the display of complex autonomous behaviours [Eldridge and Kiefer, 2017].

1.2 A remarkable practitioner in the field of adaptive ecosystems: Agostino Di Scipio

Agostino Di Scipio has contributed significantly to the field of computer music starting from the early '90s, and he is regarded as one of the most influential figures in the area of ecosystemic live electronic music.

His publication from 1994, *Formal processes of timbre composition challenging the dualistic paradigm of computer music* [Di Scipio, 1994], is a landmark conceptual work and distinctive formulation that proposes composition techniques capable of merging what he referred to as the dualistic paradigm of computer music. The dualism can be summarised into the two main approaches that see practitioners either *composing sounds* or *composing with sounds*. This paradigm fundamentally describes the standard workflow of computer music composition where a performer generates a set of sonic materials that are subsequently arranged into a music piece. Di Scipio proposes a *theory of sonic emergence*, that is a process-based composition paradigm where structures defined at a lower level for the generation of timbres would allow for the emergence of musical forms at a higher level. Di Scipio had then begun his exploration of emergent music systems following autopoietic metaphors and ecosystemic designs.

Another milestone in his works is the publication from 2003 *'Sound is the interface': from interactive to ecosystemic signal processing* [Di Scipio, 2003]. In this work, Di Scipio describes his aesthetics of music composition that has reached a clear identification and connotation: Di Scipio's focus is on the exploration of the dynamical behaviours that emerge through the coupling of an autonomous system such as a DSP network of interdependent components with the environment that is hosting the performance. The concepts from second-order cybernetics are crucial as well as the structural coupling between system and environment, which is typical of autopoietic and living systems. Furthermore, Di Scipio formulates a radical paradigm shift in live performance, from *interactive composing* to *composing the interactions*, questioning the basic design of interactive music (performer-controllers-DSP-sound), which is substantially linear without explicit closed-loops – as it is instead implied by the notion of interaction – and proposes a design strategy where interaction is a structural characteristic of music systems taking place in the domain of sound.

Notable examples of Di Scipio's music are the *Audible Ecosystems* series, some of which are published on CD by the record label Editions RZ, and the *Modes of Interference* series. The first series includes works such as *Impulse Response Study* (2004), *Feedback Study* (2003-2004),

and *Background Noise Study* (2005). *Impulse Response Study* explores formal developments as a series of responses from an autonomous system in an environment that is subject to different kinds of impulses. *Feedback Study* investigates the phenomenon of Larsen, that is the positive electroacoustic feedback between microphones and loudspeakers, that is used as a means for the generation of the initial sonic material that then recirculates and shapes itself. The work has also been performed in a different setup using the vocal tract of a performer as a resonant chamber: an acoustic filter. Finally, *Background Noise Study* studies the accumulation process of background noise into relatively large buffers and its self-organisation through a network of thresholds and transformations. The works in the first series focus mainly on the autonomous behaviours of sonic audio ecosystems with reduced human intervention. The second series explores the combination of autonomous systems with human agents as direct perturbations for the overall outcome.

Di Scipio and I started collaborating in 2013 after we had decided to combine our independent practices in the field of autonomous music to explore the possibility of a meta-system given by the combination of our systems and performance approaches. Between 2013 and 2014, we realised a series of studio performances and recordings to which we gave the name of *Machine Milieu*. Our project explores the hybridisation of autonomous machines, with or without human intervention, and the formulation of a set of performance modalities in which the human and the machine are structurally interacting agents in a higher-level system made of systems [Sanfilippo and Di Scipio, 2017, Di Scipio and Sanfilippo, 2019].

1.3 The aesthetics of music complex adaptive systems

1.3.1 From failure to *extreme functioning*

There are several examples of composers and sound artists who deliberately altered the technologies and techniques used in order to achieve new sonic results. For instance, some compositions are based on scores in which the degree of details and required performing skills are intentionally conceived to challenge the capabilities of the performers. This strategy allows us to achieve indeterminacy and musically interesting awkwardness as a result of mistakes and inaccuracies (see the works *Cheet Sheet* by Michael Edwards and *Unity Capsule* by Brian Ferneyhough). Cage's prepared piano has shown how changing the mechanisms of the instrument and the conventional performing techniques can lead to a new domain of sonic materials, sometimes profoundly different than the original ones. Pierre Schaeffer, while experimenting with turntables, altered the internal mechanism of such a device so that he could eventually change the rotation of the turntable, from clockwise to counterclockwise, thus achieving the reverse effect playing the recordings backwards. Other examples come from the hardware hacking and circuit bending practitioners that initially appeared in the 1960s and 1970s, where people like Gordon

Mumma, Nicolas Collins and Reed Ghazala (to name a few) realised practically brand-new sound generators by manipulating the electric circuits of everyday devices or existing synthesisers. In more recent times, we can see the same approach in the digital domain where devices and software are pushed towards conditions where failure takes place – what has been referred to as *the aesthetics of failure*. Artists have explored this practice in various ways: by drawing on the rear side of CDs and eventually producing irregular patterns and clicks when playing them on a CD player (Oval) [Cascone, 2000]; by letting a digital sampler crash through a particular configuration of the parameters in order to create unexpected sounds (Fennesz) [Bridda, 2004]; or by pushing the computational capabilities of computers so that CPU overloads result in unwanted yet interesting sonic behaviours (Galarreta) [Private conversation]. What is common to all these practices, while operating in different domains and with different approaches, is the possibility to achieve new outcomes out of tools not explicitly designed for the resulting materials and behaviours. This can be a crucial aspect in the process of sound creation and in general something that strongly characterises the originality of a work.

The role and influence that technology has in the creative sonic practice have progressively become a central question for composers, sound artists, philosophers and critics. The concept according to which a technological determinism is plausible, and where technology serves as a neutral tool for ideas and eventually final results, has been discussed and, in some cases, refuted [Hamman, 2002, Di Scipio, 1997, Di Scipio, 1998, Di Scipio, 2000, Feenberg, 1992, Rognoni, 1966]. Ideas themselves change and evolve during the creation process concerning the technology and techniques adopted. Meanwhile, the technologies and techniques used can, in turn, be affected according to the evolving ideas and results. The design process of the systems involved in sound creation and the techniques formulated and implemented in such a practice is indeed a fundamental stage whose outcome will significantly determine the final results.

Composers can use pre-existing technologies and techniques to create works reflecting their ideas and purposes, yet they will operate in a creative domain where traces of an aesthetics intrinsically sculpted in the pre-existing devices, software and techniques are present. To some extent, this can homogenise the work of the artist, for a common root will be shared between all the works using a particular technology or technique. Today, through the advent of powerful computers relatively accessible to most people, and very sophisticated software for audio programming also available for free, composers can design their tools and systems from scratch. Even if such software makes it possible to operate at a lower level, technology and techniques still seem to have a noticeable influence on the final result. Even though environments like SuperCollider or Pure Data are powerful and flexible, some of their building blocks are more high-level and can still have very characteristic features. On the other hand, specific sound synthesis and processing techniques, even if implemented with very low-level programming languages, can still have very peculiar and recognisable features.

Technology and techniques can be altered and shaped into something original. After this

process, they operate in a meta-state whose behaviour may be different from the one they were designed for. As a result, the exploration of different methods leading to such a meta-state corresponds to the exploration of potentially new creative practices and aesthetics. There are different ways to achieve this. In the case of Cage's prepared piano or hardware hacking, the kind of action required is that of manipulating the structure of the tools themselves. On the other hand, when discussing this approach in the digital domain, we have seen that the tendency is that of pushing digital systems towards failure and malfunctioning. A conceptually and technically different approach, instead, is that of using the feedback mechanism [Sanfilippo, 2013]. Through feedback, it is possible to turn elementary tools into what Heinz von Foerster defined as a non-trivial machine. That is a system that is self-related, autonomous, unpredictable, and analytically non-determinable [Von Foerster, 2003a]. Feedback is also ubiquitous in the discussions regarding complex and emergent systems [Gershenson and Heylighen, 2005, Kitto, 2006, Morin, 2007], which we will see in the next chapter, and emergence is indeed a key aspect in this approach for the alteration of technology.

Emergent systems are different from the sum of their parts because of synergetic relationships between their components [Corning, 2002], and this particular characteristic is what allows such systems to operate in that meta-state that leads an alteration. With feedback, it is possible to radically alter the behaviour of analogue and digital devices as well as synthesis and processing techniques, so that hidden characteristics can be unfolded. Different materials and forms, two aspects that with such an approach become inseparable [Di Scipio, 1994], can then be discovered. Microphones, loudspeakers, mixing boards and any other device within the feedback loops, as well as audio manipulation techniques, are no longer sound capturers, reproducers or transforming units, but they become sonic generators with their identity and fundamental systemic role in the emergent whole. Feedback systems, unlike the examples previously discussed, are not cases of manumission or malfunctioning. The state characterising feedback systems, instead, could be referred to as extreme functioning: a condition where the operational level coincides with the limits of the machines themselves, and where equilibrium and instability coexist. Feedback systems push technologies and techniques towards their extremities, up to the threshold where the original identity of such technologies and techniques is replaced with an emergent one.

1.3.2 The objectivity of the machine

The concept of *objectivity of the machine* is useful to understand the importance of autonomous systems – specifically CASes – in the practice of music composition and performance. The idea has been informed by several years of experience as an electronic music composer and performer, after realising the difficulties related to predictability in improvised music and redundant procedures in the use of electronic instruments by looking backwards at my practice as a musician. I underline that my observations apply specifically to my practice and that other practitioners

may have never achieved similar findings.

Radical improvisation is by definition the approach that offers the highest degree of freedom in the live act of music performing, although it is by no means an approach that guarantees an acceptable degree of variety and complexity. Formally, radical improvisation can be described as a feedback loop coupling the human performer and the sound-generating device, which we can generally refer to as *machine* [Pressing, 1984]. The human and the machine constitute a hybrid higher-level system where the input is the human's auditory system, and the output is the set of actions performed by the human through the sonic results generated by the machine, as depicted in the diagram below.[2] We are assuming the basic case in which the machine is not autonomous.

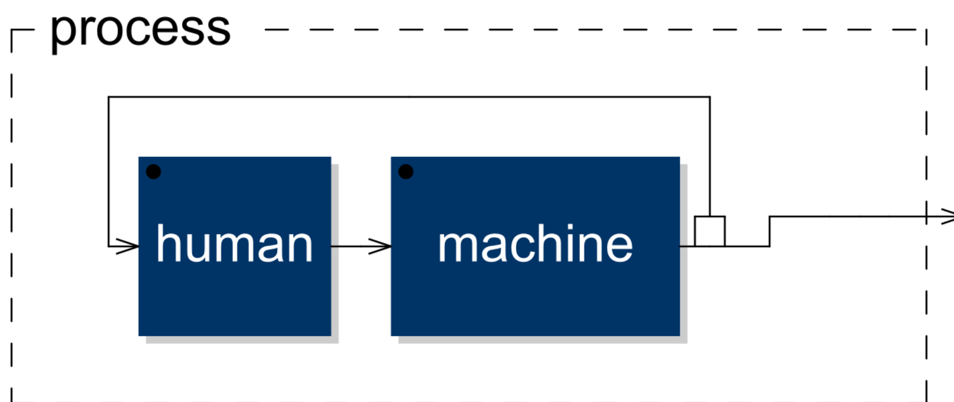

Figure 1.1: Human-machine improvisation configuration. The "human" element represents a human agent performing actions on a non-autonomous machine. The "machine" element is a sound-generating system that responds to the actions of the human. The output of the machine, eventually, reaches the human agent back after a delay. The human agent is not affected by other signals other than the output of the machine as, for this specific case, it is assumed to bypass the environment.

Similarly, the practice of electronic music composition in the studio also resembles a feedback configuration, although the process takes place in a non-real-time domain. Namely, the composer working in the studio and following a standard workflow, for example, using software for the generation and processing of sounds that are eventually arranged into pieces through editors and digital audio workstations, cycles through two main phases: *modification* and *result-checking*. The iterative process is repeated until the piece results complete, i.e., when it does not require any further modifications.

Feedback loops and chaos theory will be discussed in details in the next chapter; for now, it will suffice to know that such configurations can result in behaviours that tend towards steady-state outputs, which are called *attractors*, or outputs that oscillate through a limited

[2] All block diagrams are generated in Faust 2.18. https://faust.grame.fr/. Accessed on the 29th of August 2019.

number of states, which are called *periodic oscillations* [Gleick, 2011]. I hypothesise that the principles of feedback systems and chaos theory may also apply to feedback configurations involving individuals, such as the practices of improvised electronic music and electronic music composition in the studio, as explained above.

Pierre Boulez claimed that improvised music is often confined within a process where the states of *excitement* and *relaxation* alternate [Bowers, 2002]. Paraphrasing Boulez's statement, improvised music can in some cases be reduced to a process where musical tension progressively builds up and finally breaks down after it has reached a peak so that the process can start again. This kind of behaviour resembles a typical response of positive feedback configurations [Heylighen and Joslyn, 2001], and intrinsically limits the variety of an improvised live performance as the global development becomes redundant. Of course, experienced improvisers can avoid these behaviours and can make sure that redundancy is diminished as much as possible, although the tendency towards redundancy intrinsic to some feedback configurations should be taken into account and it may still play a role in the unfolding of an improvised music piece. Empirically, what I could also observe in my practice as an improviser, besides a tendency towards alternating tension and relaxation, is a tendency towards retracing a limited set of configurations in the software parameters to recall familiar or otherwise known sonic environments, which, ultimately, also resulted in a large-structure redundancy.

Composers and performers have developed techniques to avoid recurring patterns in improvised music. Structured improvisation (hence non-radical) is an example, although in this case a reduced tendency towards a global redundant behaviour is achieved at the expenses of the freedom of the performer, especially considering that the structures may be unrelated to the specific sonic context. Other techniques are the definitions of sound-related constraints or rules. In this case, the improvising performer has the potential freedom of exploring any number of sonic environments, although the limitations will arise from the specific sonic environment, which can be different each time the piece is executed. See, for example, *Duet I* (1960) by Christian Wolff. For background on other works following a similar approach see [Dahlstedt et al., 2015].

The implementation of CASes in the area of live performance and improvised music is yet another technique that can be used to enhance the variety in the development of music performance and limit the tendencies towards redundancy of some improvised music. CASes are structurally coupled with their context, which becomes an essential element for their very existence, and exhibit a nonlinear and nontrivial behaviour that is always a function of the surrounding sonic environment. This allows musical CASes to explore their parameters objectively, making it possible to have a variety of combinations that is virtually only limited by the data type representing the state variable of the system. The adaptive behaviour of such systems can then explore a theoretically much larger number of sonic environments, which in turn result in a much larger number of developing curves so that the redundancy of both timbres and forms

can be contrasted.

1.3.3 Losing control to gain complexity

The matter concerning the degree of control delegated to the machine, particularly in the case of autonomous machines, is a complicated topic that raises philosophical questions of ontological nature as well as more practical and empirical ones about the work of electronic music composers and performers.

It may appear as if the electronic musician has no aesthetic or musical ability whatsoever, and that the machine is the sole entity responsible for the musical results. After several years of working in the field of electronic music, it also seems to be rather common to consider the work of a machine as intrinsically less valuable than the work of the human. The music generated by machines is sometimes subject to an *a posteriori* judgement: despite the result may initially seem objectively convincing for the listener, knowing that a machine generated the work may penalise it and, in some cases, negatively affect the whole experience.

The fact that machine-generated music may be considered less valuable by non-practitioners is not a relevant issue. Though, the same misconception is often found in professional environments of electronic music. There is still today the misunderstanding that music technologies and devices are means that solely serve the ideas of composers without affecting the aesthetic content of the musical outcome. Even human-machine interaction performances, as stated by Di Scipio [Di Scipio, 2003], follow a linear workflow, which establishes an implicit hierarchy where the machine is subordinated to the performer.

Mutuality and reciprocal alteration are implicit and fundamental features in the notion of interaction. These conditions are achievable when both the human and the machine can perform independent actions while, at the same time, being able to react to external stimuli to adjust their actions accordingly. In other words, both entities should be capable of generating autonomous behaviours and nonlinear dynamics.

It is indispensable to distinguish between autonomous and automated behaviours. Autonomous is in some cases considered a synonym for automated, although the implications of an autonomous design are substantially different from standard automated systems. In several computer music workflows, the formal development and scheduling of sound events are driven by stochastic processes, i.e., pseudo-randomness generators, or other algorithmic rules that determine the unfolding of the piece through a predefined path. The formal domain, or *high level*, is independent of the domain where music develops, that is, the domain of sound itself or the *low level*.

The autonomous behaviour of CASes, on the other hand, is a consequence of the interrelatedness between sound self-generation and formal self-organisation, that is, the tight bond between low level and high level in a network of nonlinear interactions that result in complex evolutions and emergent behaviours. CASes are then particularly well-suited for applications

of human-machine interaction performance and autonomous music systems, and the notion of *aesthetics of the machine* is central within this framework. The implementation of nonlinear feedback delay networks and recursive loops is a widespread methodology not only in areas of acoustic physical modelling [Karplus and Strong, 1983, Cook, 1992, Rocchesso and Smith, 1997] but also in biocybernetics and complexity for the design of artificial intelligent behaviours and artificial life [Heylighen and Joslyn, 2001, Maturana and Varela, 1980]. "All processes containing feedback loops can be the origin of the emergence of being and existence" [Morin, 1977]. The emergent nature of CASes is a key element for the creation of a radically new aesthetics that can challenge and enhance the creative output of composers and performers. While musical CASes are designed to produce a sonic output that is structurally complex concerning the time scales and thresholds of human perception (a discussion on the complexity and the observer is in section 2.5 Complexity and adaptation), the particular articulations and evolutions are unique features of each system or, of each performance, more precisely, that determine the personality and musicality of the artificial entity.

The action of deliberately losing control, the action of *not doing*, hence the conscious choice as a composer/performer to discover the hidden aesthetics of an autonomous machine such as a CAS, is a process that carries a critical potential. The creative practice of musical CASes is a shift from the synthesis of sound to the synthesis of formal evolutions and artificial expressiveness. The role of the composer becomes primarily concerned with composing music systems that compose music rather than assembling sonic materials into a music piece. The aesthetics of the machine becomes vital for the exploration of new music through the cooperation of the human and the artificial.

1.4 Innovative aspects of this research

This work is about the investigation of autonomous machines exhibiting complex emergent behaviours for the exploration of new music in the area of live electronics. Arguably, the most significant contribution of this research is the development and application of the technique of *distributed adaptation* discussed in chapter 3 Distributed adaptation. As mentioned earlier, this technique is related to *evolvability* in biology and genetic algorithms. Evolvability is the ability of genomes to produce adaptive variants. Similarly, distributed adaptation is the application of adaptive mechanisms at all levels and to all infrastructures – including the adaptive infrastructures themselves – of a recursive network, the result of which is a system that continuously redefines its internal organisations as an ongoing process where new adaptive modalities emerge again and again. The result of such an infrastructural rearrangement has shown a substantial improvement in the long-term variety of the autonomous performance of these networks, followed by an increased overall musical complexity. Due to the lack of literature related to this specific technique, it is plausible to assume that, at least in the music domain, it has not been

explored thoroughly yet.

The practice of electronic music through CASes also carries an influential impact on the general practice of composition and performance, for example, through the paradigm shift from composing music to composing systems that compose music. The creativity of the composer then lies on a different domain, that is the construction of networks capable of dynamical sonic outputs with evolutions at the level of timbre and form. Working within this framework also has consequences on the modes of interactions that can be established between humans and machines. Palle Dahlstedt et al., in a publication from 2005 [Dahlstedt et al., 2015], propose a methodology for structured improvisation that they call *The Bucket System*. The technique is based on the random selection of "signal state interpretations" that are assigned to the performers so that they can act according to specific interaction rules such as *lead, support, opposition*. These indications appear to relate to global behaviours through the interpretation that each performer gives to the rules. There appears to be no explicit reference to the mechanisms of positive and negative feedback or the low-level or high-level sonic characteristic of a sound event.

The formulation of the idea of *cybernetic improvisation* discussed in section 5.1 Cybernetic improvisation, on the other hand, proposes a lower-level and infrastructural approach for an emergent and self-determining network of interactions among human agents. The outcome of such a low-level structuring maintains extensive performative freedom and open interpretations for the analysis of the contextual sonic environment while supporting the emergent expressiveness and formalism that are inherent in the self-structuring network of interactions.

Information processing is an essential aspect of CASes as they must obtain an understanding of their surrounding environment; this research proposes a set of original techniques for the processing of information discussed in section 3.3 Information processing techniques. Algorithms for low-level information processing such as brightness or noisiness are based on frequency domain analysis. The low-level algorithms presented here, instead, are based on time-domain calculations, some of which following an adaptive design, and provide a somewhat accurate measurement at a low computational cost. Furthermore, two high-level algorithms for the measurement of complexity and dynamicity of long sound events are introduced, which apply the theories of complexity and dynamical systems combined with average absolute deviation and recurrence quantification analysis.

Furthermore, some original approaches for audio processing are proposed. Some of these include windowless granular processing techniques implementing zero-crossing detection for the handling of discontinuities, as well as non-homogeneous-windowing granulation; modulations based on the variation of cut-off frequencies; and nonlinear distortion in which sound output and transfer functions dynamically and recursively determine each other. These techniques are all particularly suitable for the generation of noisy texture, in particular, textures with high inner activity and varying shape.

Finally, the development of a library in the Faust programming language dedicated to the implementation of CASes for music is another relevant aspect that can contribute to the spreading of this creative practice both in professional and non-professional environments.

1.5 Portfolio

Attached to this thesis, is also a music portfolio with audio documentation of six of the seven live performance projects presented here in the case studies. There is one audio file per project, except for the case study in subsection 5.1.1 Case study: *Human Network: Machine Nostalgia* (2016-2018), recorded either in a live public performance or in a studio live performance.

It is essential to underline that the audio files serve merely as audio documentation to provide an overview of the capabilities of the performance projects; experiencing them in a live situation – the *hic et nunc* condition these projects have been designed for – is a crucial element to appreciate these works completely. Even in the case of performances with non-real-time systems that use pre-recorded sound materials, experiencing the performance live is substantially different from listening to its recording. From a radical constructivist perspective, arguably, sounds cannot be objects as they are always subject to the perception of the individual. Practically, pre-recorded sounds in a live performance, too, will be shaped by the resonant characteristics of the space, while the observer's experience is also dependent on its position in the environment. In his publication from 1971 *Towards an ethic of improvisation*, Cornelius Cardew questioned the meaning and utility of improvised music recordings, arguing that they are mostly empty as they cannot convey any sense of time and place [Cardew, 1971].

This emptiness is particularly apparent for music that follows an ecosystemic approach, which recalls notions from second-order cybernetics and the uncertainty principle. In ecosystemic music where system and environment are structurally coupled, the very presence of an observer is sufficient to alter the output of the system, as that will affect the resonant characteristics of the space. It is virtually impossible to observe the system without affecting it. The ecosystemic approach is then an intrinsically participatory music practice that necessarily involves the observer in its creation. Within this framework, then, the notion of *sweetspot*, that is, an ideal listening position for the observer to experience a music work, is surpassed: the presence of the observer in a particular position will shape the final result, making the overall experience unique and individual.

It is also fundamental to emphasise that the performance projects presented here, regardless of the presence or absence of a human agent, have a strongly improvisational nature. The music-making processes based on improvisation or studio composition are briefly reviewed in subsection 1.3.2 The objectivity of the machine. A key difference between the two processes is the real-time domain of the first and the non-real-time domain of the second. Prigogine's work investigates the role of time in complex dynamical systems. He argues that the irreversibility

of the process is a critical characteristic of such systems and that it may lead to new types of dynamic states that he calls *dissipative structures* [Prigogine, 1978].

Improvisation, too, is an irreversible process as it follows a strict continuity and sequentiality. The irreversibility of the process is one of the reasons why improvisation is substantially different from studio composition. The DSP systems implemented in these performance project are inherently dependent on the temporal dimension as they are entirely causal. These DSP systems, indeed, are implemented with no stochastic elements and are thus, completely deterministic, at least when isolating them from the real environment, hence considering only the digital domain. When operating as closed systems, initial conditions determine the entire evolution of the process and what happens in the present is always a result of the history of the system. Arguably, to consider these performances as processes representing a *developing organism* is an optimal mindset to experience and appreciate the work thoroughly.

It is advisable to listen to the audio documents in conjunctions with the corresponding case studies, after reading the sections where the works are described. This way, it will be possible to have a clear idea about the overall setup and conditions for the work to acquire a better understanding after listening to the results.

1.5.1 Audio documentation

The audio documentation for *Phase Transitions* was recorded in the studio in 2019 with a closed system configuration and using a Dirac impulse as the initial condition. For a live performance, the system can be coupled with the environment using microphones, or it can be triggered using environment-dependent initial conditions. The audio documentation is an adjustment of the work to fit the perceptual thresholds and time scales of music. The Rotting Sounds[3] research project originally commissioned the work to represent the idea of *digital deterioration*. The original work is an adaptive process lasting about three months that implements the idea of sonic deterioration through saturators. As of August 2019, the work is operating at the Auditorium of Rotting Sounds at the Music and Performing Arts University in Vienna.[4]

The audio documentation for *Inexorable Shifting 2* was recorded in the studio in 2019 with a closed system configuration and using a Dirac impulse as the initial condition. For a live performance, the system can be coupled with the environment using microphones, or it can be triggered using environment-dependent initial conditions.

The audio documentation for *Order from Noise (Homage to H. von Foerster)* was recorded in the studio in 2019 using an environment-dependent impulse as the initial condition. This work has been presented live on several occasions; for example, live performances at the Sibelius Academy in Helsinki in April 2018, at Zentrale in Vienna in October 2018, and at Echoraum in Vienna in June 2019. This recording, in particular, shows a single initialisation of the system.

[3] https://rottingsounds.org/ Accessed on the 29th of August 2019.
[4] https://rottingsounds.org/category/threads/auditorium/ Accessed on the 29th of August 2019.

The audio documentation for *Constructing Realities* was recorded in the studio in 2019. As of August 2019, this work has not been premiered in a public concert yet. Especially for this audio documentation, the network was initialised with a Dirac impulse in a self-oscillating closed system configuration, hence without any connection with the environment, to show the long-term formal variations in the absence of external perturbations.

The audio documentation for *Single-Fader Versatility* was recorded live at the Edinburgh College of Art Sculpture Court in March 2016. This project has been presented live on several occasions, for example, in Vienna for the VELAK series in November 2018. A recent live performance of this project took place in Vienna in June 2019. Despite the low quality of the audio recording, video documentation of the last live performance can be seen at the following link: https://www.youtube.com/watch?v=hIrqmDif5uQ. Accessed on the 29th of August 2019.

The audio documentation for *Audible Icarus* was recorded live in Edinburgh at the Biscuit Factory in October 2017.

All audio files are strictly non-edited, and they can be found at the link https://era.ed.ac.uk/handle/1842/37190.

1.6 Remarks

We have discussed the development of systems thinking from the early cyberneticians to the modern theories and techniques in complex systems. In the 1940s, the interaction and exchange between groups of scholars from different disciplines set the beginning of the formalised study of cybernetics and systems science that greatly contributed to today's growth of the field. The application of these studies to a large number of domains ranging from psychology and sociology to engineering and biology underlined their highly transdisciplinary nature and connection to several real-life phenomena.

The early investigations in cybernetics mainly concerned understanding the behaviours of systems regardless of their interactions with the environment or observers. In the 1960s, scholars investigated the possible connections between nontrivial systems and their surroundings, which included the environment and the observers. The direction of the study of cybernetics shifted towards considering nontrivial systems as tightly interrelated with their environment and observers, or even considering such systems as larger systems that encapsulate the environment and the observers as components that contribute to the global behaviours. This new conception was a key aspect in second-order cybernetics, which Heinz von Foerster defined as "the cybernetics of observing systems". Systems as system-environment couples were then regarded as self-referential entities following metaphors of self-awareness. In philosophy, von Glasersfeld was investigating the idea of radical constructivism and the notion of *self*, which he later described through concepts of second-order cybernetics, particularly circular causality. According to his idea, the experience of self takes place as retroactive relationships between

the entity and its surroundings. We can see that these disciplines contributed to each other's growth.

The development of feedback-based music was also significantly influenced by this way of thinking, and it followed somewhat closely the evolution of these fields. Feedback-based music was initially based on rather simple mechanisms while it expanded over the 1960s and 1970s into more articulated techniques where the environment and self-regulation had a central role. Today, we have seen examples of highly advanced music systems implementing large networks of adaptations through which complex musical structures emerge.

The aesthetics of musical CASes is a critical aspect to investigate as they open to new paradigms in music composition and performance. Three of the most relevant points in the aesthetics of musical CASes proposed in this chapter are the differences between failure and operational criticality; the machine as a self-referential entity that thoroughly explores its state variables; and the fundamental relationships that arise from the interaction between a human and a highly autonomous machine in live performance.

The first point describes the operational state of these systems as substantially different from the approaches found in music based on failure and glitch aesthetics. A "far-from-equilibrium" dynamics characterises CASes, which represents a very high sensitivity to environmental perturbations as well as a strong nonlinearity and unpredictability. These conditions, combined with iterative configurations and adaptation, produce emergent behaviours and radical novelty through which conventional tools for electronic music can be turned into new instruments.

The second point analyses these systems from the perspective of self-performing machines. Due to adaptation, these machines can reorganise their internal configurations and the variables that define their behaviours, replacing the role that human performers have in conventional human-machine interaction performance. Particularly in the case of improvised performance, it is observed that patterns and recurrent trends may appear, for improvisation can be formalised as a recursive mechanism, hence a process that may follow the cybernetic principles of positive and negative feedback. These principles are discussed in section 2.1 Feedback mechanisms. The self-regulation and adaptation in CASes allow for counterbalancing mechanisms that prevent repetitions, hence increasing the variety and potential of the music system by exploring a broader set of combinations of parameters and configurations. See chapter 2 Complex adaptive systems and chapter 3 Distributed adaptation for discussions on adaptive behaviours.

The third point proposes a new perspective on human-machine interaction performance with CASes. This view highlights and favours the development of the aesthetics of the machine, that is a conscious decision of the performer to interfering as little as possible to achieve a compelling musical output by observing the evolution of the system. This approach raises questions concerning the need for a new conception of musicianship and virtuosity in the performance of live music with systems with a high degree of autonomy. Formalised performance modalities based on this idea are discussed in chapter 5 Performance modalities and interfaces.

To conclude, the chapter summarises the most relevant aspects of the research and presents a portfolio of live performances underlining the importance of an adequate listening mode and mindset to appreciate these works completely.

Chapter 2

Complex adaptive systems

All processes containing feedback loops can be the origin of the emergence of being and existence.

Edgar Morin

This chapter introduces the fundamental principles and characteristics of complex adaptive systems (CASes). Starting with the description of positive and negative feedback, these two essential mechanisms, applied at different levels and domains in the design of systems, are shown to be the key building blocks in the implementation of complex behaviours.

Chaos theory is distilled and presented through a set of examples based on the well-known Lorenz system to describe the possible responses of chaotic systems under different configurations. Furthermore, other examples from creative practitioners are discussed to provide examples of musical applications of chaos theory.

The concept of emergence is analysed within the framework of observer-dependent phenomena, and the idea of autopoiesis is explained relating the notion of structural coupling. Finally, adaptation and complexity are presented as strictly related concepts, and one of the works from the portfolio is illustrated to provide a practical application of all the theories discussed in the chapter.

2.1 Feedback mechanisms

Feedback loops are mechanisms that relate a system with itself or several components by making them interdependent. Feedback loops are ubiquitous in all kinds of networks exhibiting complex behaviours, and they are used to model systems in a variety of fields. Climate is an example of an incredibly intricate network of feedback connections among an enormous number of components. Societies, too, can be seen as large networks of interacting components (individuals who are, in turn, vast networks of feedback relations) shaped by feedback loops. Stock markets are yet another example together with ecosystems, animals predator-prey relationships, galaxies, and several other systems at very different scales.

The basic definition of feedback loop takes into account a system that performs some transformation to an input value or signal, and that outputs the result by also feeding it back into the input after a certain delay [Heylighen and Joslyn, 2001]. Despite the large family of behaviours that can be modelled through feedback network, there are only two types of feedback relations, although by combining them in different amounts it is possible to achieve a great variety of responses in virtually any area.

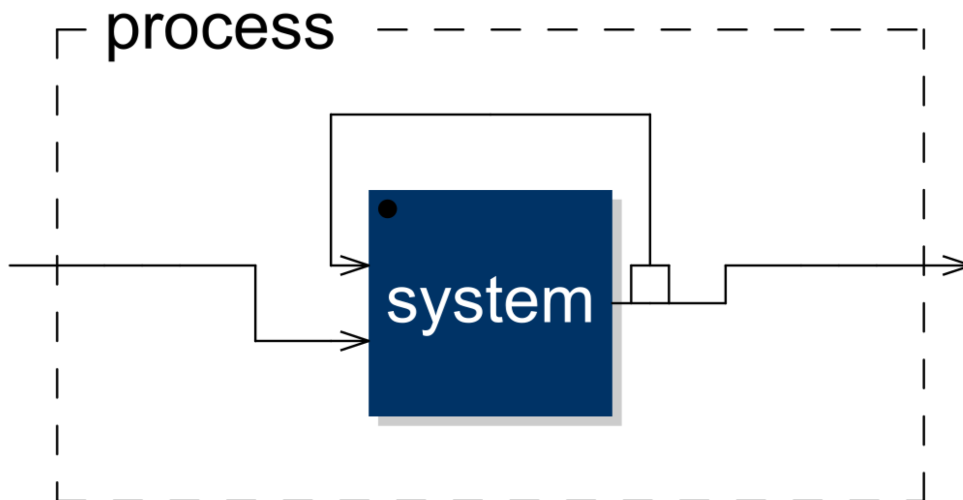

Figure 2.1: General feedback configuration. In this diagram, "system" represents a generic function that transforms an input signal and outputs the result. The result is then fed back into the system after a delay and combined with the input.

The two main feedback types are *positive* and *negative* feedback loops. On a system subject to stimuli, a positive feedback loop can create a response with exponential growths or decays from the system's natural equilibrium point. Often, positive feedback is indeed seen as a chain reaction where perturbations that tend towards a certain direction will be amplified recursively

by the system and will result in even further shifts towards the same direction. In positive feedback loops, then, the input-output relation is direct, and an increase (or decrease) in the input will cause an increase (or decrease) in the output [Wiener, 1948, Ashby, 1957, Heylighen et al., 2001, Heylighen and Joslyn, 2001, Gershenson, 2007]. Negative feedback loops have complementary behaviour to positive feedback, and they tend to oscillate around a stability point. Indeed, negative feedback systems are said to be in *dynamical equilibrium* and they perform a counterbalancing action on the external stimuli to keep balance. In this case, the input-output relation is inverse, and an increase (or decrease) in the input will cause a decrease (or increase) in the output.

Heylighen and Joslyn propose an alternative definition of the positive and negative feedback loops concepts by examining systems with a *causal relation* between two variables [Heylighen and Joslyn, 2001]. Positive feedback takes place when an increase (or decrease) in the first variable causes an increase (or decrease) in the second variable, which, in turn, has a similar effect in the first variable, recursively. An example of positive feedback is a crowd subject to a panicking event. If the two variables are the number of people running and the overall panic level in the crowd, we can see that the higher the level of panic and the more people will run, and the more people will run, the higher the panic in the crowd will be. Conversely, a negative feedback loop is a mechanism regulating the body temperature through sweat. If the temperature and the amount of sweat are the variables, we see that if the temperature rises, the amount of sweat will rise too, causing a decrease in body temperature. On the other hand, lower body temperature will decrease the amount of sweat, which will then result in a further increase in body temperature. This way, the body temperature oscillates around an ideal range. Generally speaking, negative feedback represents *convergence* towards equilibrium, while positive feedback represents *divergence* from equilibrium.

2.1.1 Properties of feedback mechanisms

Despite the structural simplicity of feedback loops, such recursive mechanisms have significant consequences in the behaviours of systems or interdependent components in general. While in open loops the distinction between origin and consequence is rigorous and linear, closed loops depict a condition where distinguishing between causes and effects becomes impossible. Heinz von Foerster and other scholars taking part in the Macy Conferences called this particular condition *circular causality*. Von Foerster describes it as the characteristics of a closed loop that, if opened, the causes for an effect in the present can lie either in the past or the future, depending on the point where the loop is opened [Von Foerster, 1952]. For von Foerster, circular causality fills the gap between effective and final cause, or motive and purpose. Furthermore, von Foerster goes on by saying that a closed loop allows decreasing the degree of uncertainty as it is no longer necessary to provide the initial conditions for the system: the final conditions themselves can already provide them.

Heylighen and Gershenson explain how circular causality is intrinsically related to the nonlinearity of a system [Heylighen et al., 2001, Gershenson, 2007]. The mathematical definition of linearity uses the superposition principle, which is in turn defined by the two simpler properties of additivity and homogeneity. The additivity property consists of a function that, given two inputs, the sum of the individually processed inputs is the same as the sum of the inputs subsequently processed by that function. The homogeneity property consists of a function that, given an input and a scalar, the processed product between the two is the same as the product between the scalar and the processed input. If these properties are not satisfied, a system or function is said to be nonlinear. In systemic terms, the nonlinear interactions between components determine a nonlinear behaviour in the system. It means that apparently negligible causes can result in significant long-term effects, whereas, on the other hand, large causes may not result in significant effects. In other words, there is no proportionality between the input and output of a system.

Ross Ashby identified another fundamental property of feedback systems that he referred to as *coupling* [Ashby, 1957]. Essentially, a feedback connection that relates a system with itself (self-coupling) or two or more components among themselves (cross-coupling) creates a structural coupling that redefines the system as a whole. A system then becomes self-referential or self-affecting, while cross-coupled components determine their states by mutually affecting each other. This type of relationship defines the very concept of interaction at the structural level as a continuous and ongoing exchange of energy and possibly information, too, between the coupled parts. Synergetic phenomena and structural interactivity highly characterise feedback systems and their behaviours as higher-level wholes. The identity of a system is the result of a distributed cooperation between all the interconnected parts.

Feedback loops are also essential for self-organising behaviours to take place. Self-organisation has received different definitions from different fields such as thermodynamics [Prigogine and Nicolis, 1985], computer science [Mamei et al., 2006, Kohonen, 1990, Heylighen and Gershenson, 2003], cybernetics [Von Foerster, 2003c, Ashby, 1991, Heylighen et al., 2001], biology [Camazine et al., 2003], and mathematics [Lendaris, 1964]. As the many definitions suggest, there is no generally accepted definition of such a property, although some approaches in defining the term allow for a more general application. Notably, a rather standard definition of self-organisation takes into account statistical entropy. This approach is particularly useful as it applies to any system for which a state space can be defined, hence information systems, too [Gershenson, 2007]. Within this framework, self-organisation is defined as the spontaneous decrease of statistical entropy in a system, which corresponds to an increase in the order of a system or to a decrease in the degree of uncertainty.

Notwithstanding the many definitions given to the term, self-organisation has requirements that apply to all fields in which it is discussed. Self-organisation implies a distributed control, and it is in contrast with the idea of a limited part of a system piloting organisational

behaviours. Self-organisation indeed takes place as the local and parallel interaction between all the components in a system, allowing for patterns to emerge at a global scale. The pattern formation is a consequence of the process of self-organisation where we have a decrease in the uncertainty, as the system will ultimately shift through a limited set of states which is a subset of the original state space. The self-organising process and the opposite one through which unpredictability increases, hence letting a system shift over different state subspaces of different sizes, are possible through nonlinear interactions and a combination of cooperating negative and positive feedback loops. These are fundamental elements for the production of complexity.

2.2 Chaos

The term "chaos" is ubiquitous but frequently misused. Often, especially in everyday situations, the term is used to indicate a form of disorder with a definite metaphorical and negative acception. Chaos is also very often considered a synonym of unpredictability and indeterminism, although the symmetry between the terms does not always imply. Stochastic systems that use randomness show unpredictability, albeit the generation of randomness and the overall behaviour of the system usually follow a linear relationship, i.e. they lie within an open loop. It means that, despite randomness determines part or the entire output of the system, the stochastic elements are not affected by the overall outcome. Thus, the stochastic elements operate on an independent domain. On the other hand, chaos is a mathematically well-defined theory that relates past, present, and future through a tight bond, and its most profound essence is the description of systems that are sensitive to their initial conditions [Mitchell, 2006].

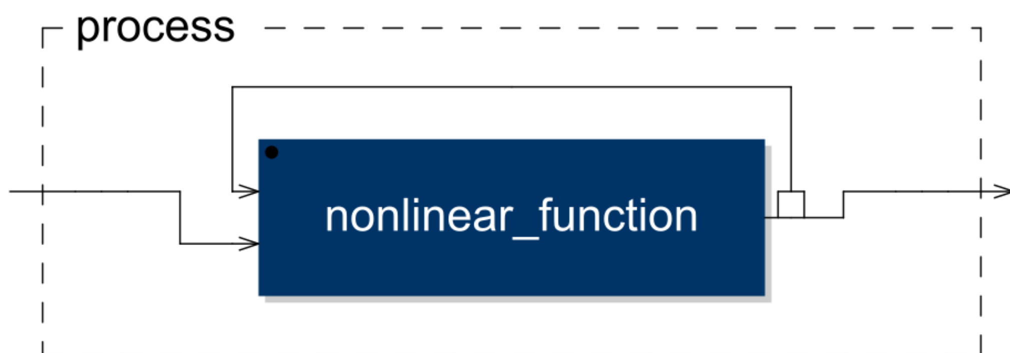

Figure 2.2: Nonlinearity and iteration as minimum requirements for chaotic behaviours.

An essential mathematical requirement for chaos to appear is nonlinearity and iteration [Kellert, 2009]. Hence, a system that performs some nonlinear processing and that is in a feedback configuration, has the requirements for chaotic behaviours to emerge. Some common

chaotic systems are the logistic map [May, 1976]; the Bunimovich stadium [Bunimovich, 1979]; the Lorenz system [Lorenz, 1964]; the Chua's circuit [Matsumoto, 1984]; the double pendulum [Levien and Tan, 1993]; the Duffing equation [Thompson et al., 2002]. The last chaotic system example, in particular, has been implemented by Tom Mudd as agents of larger networks for the realisation of physical models with chaotic behaviours achieving excellent results. [Mudd, 2017] The instruments generated in Mudd's work show a somewhat unique organicity in their dynamical behaviours that significantly contributes in making the sound of these physical models even more realistic [Gaver, 1993]. Chaotic systems can offer possibilities for music composition both at timbre level or formal level. Di Scipio has explored these techniques extensively for the generation of timbres. [Di Scipio, 1999]. Less formalised practices in the analogue world are, for example, from Toshimaru Nakamura, who is an exceptional improviser. Nakamura has been performing with what is often called *no-input mixing board* for more than 20 years, which simply consists of an analogue mixer in a feedback configuration. Analogue circuits may have nonlinear behaviours, which can then produce chaos in iterative setups such us the feedback ones. Nakamura has built his aesthetics around the unpredictability and sensitivity to small variations of these systems.[1]

My composition for MIDI piano *For different values of x and r*,[2] realised for the Slippery Chicken[3] symposium at Goldsmiths University in London in 2016, is an example of exploration of the logistic map for formal musical developments. The logistic map was used to generate short-to-long melodic patterns, rhythms, dynamics, as well as higher-level structures for the realisation of the global development of the piece. The main idea for the piece was to use the high sensitivity that the equation has to initial conditions and parameters. The generation of the elements of the piece (melodies, rhythms, dynamics) were realised using logistic maps with very slightly different initial conditions and parameters. The piece starts as a self-similar structure, but the elements would progressively decorrelate as the individual logistic maps deviate from the initial paths, showing an articulated process where the degree of disorder increases over time.

2.2.1 Fixed-point attractors, periodic oscillations, strange attractors, chaos

The behaviour of chaotic systems depends on internal parameters that regulate the nonlinearity, as well as on the initial conditions that trigger the systems. It is possible to identify at least four fundamental behaviours in chaotic systems: fixed-point attractor, periodic oscillation, strange attractor, chaos [Gleick, 2011]. A fixed-point attractor is a condition in which the system, unless it is forced away from it, remains on the same state indefinitely. It is a static behaviour without

[1] https://www.youtube.com/watch?v=dqfGbtqDVDk. Accessed on the 29th of August 2019.
[2] https://tmblr.co/Zhtq9x2B4B40D. Accessed on the 29th of August 2019.
[3] http://michael-edwards.org/sc/. Accessed on the 29th of August 2019.

changes. Periodic oscillations are conditions where the system's output oscillates periodically, hence in a predictable way, through two or more states. The system is then orbiting a limited state space, and it is returning to the same trajectory again and again. A strange attractor is a special case in which the system also shifts through a subset of states, although the main difference is that the trajectories will change at each cycle. It is a situation where we can see local unpredictability and global stability, or what can be called dynamical equilibrium. The last behaviour, which we can simply call chaos, is the case where the system travels around a wide variety of states without the formation of defined patterns, thus with fewer recurring trajectories or adjacents similar trajectories. This represents a condition of unpredictability with no apparent order, although, as we will see in the next session, the role of the observer is crucial to determine such properties.

To examine these behaviours more closely, we can consider the Lorenz system, originally developed by Edward Lorenz as a mathematical model for atmospheric convection [Lorenz, 1963, Song and Liang, 2013]. The differential equations of the Lorenz system are:

$$\begin{cases} \frac{\partial x}{\partial t} = \sigma(x - y) \\ \frac{\partial y}{\partial t} = x(\rho - z) - y \\ \frac{\partial z}{\partial t} = xy - \beta z \end{cases} . \tag{2.1}$$

The discrete-time equations then become:

$$\begin{cases} x[n+1] = x[n] + \sigma(x[n] - y[n])\partial t \\ y[n+1] = y[n] + (\rho x[n] - x[n]z[n] - y[n])\partial t \\ z[n+1] = z[n] + (x[n]y[n] - \beta z[n])\partial t \end{cases} . \tag{2.2}$$

For the following examples, σ is set to 10, β is set to $8/3$, and ∂t is set to 0.005. The initial conditions for $x[0]$, $y[0]$, and $z[0]$ are, respectively, 1.2, 1.3, and 1.6. They are fixed for all the examples. The parameter ρ will be varying to achieve different behaviours. The figures below show the state space of the system with different ρ values over 10^5 iterations.[4]

With $\rho = 10$, we can see that the system settles on a fixed point after an initial oscillation.

[4] All plots are generated in Gnuplot 5.2. http://www.gnuplot.info. Accessed on the 29th of August 2019.

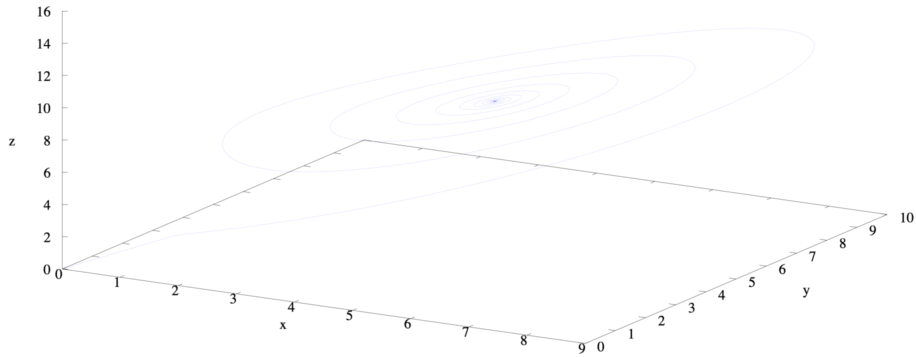

Figure 2.3: Lorenz system state space for 10^5 iterations depicting a fixed point. $\rho = 10$.

With $\rho = 18$, the system will settle on a periodic oscillation after an initial quasi-periodic phase.

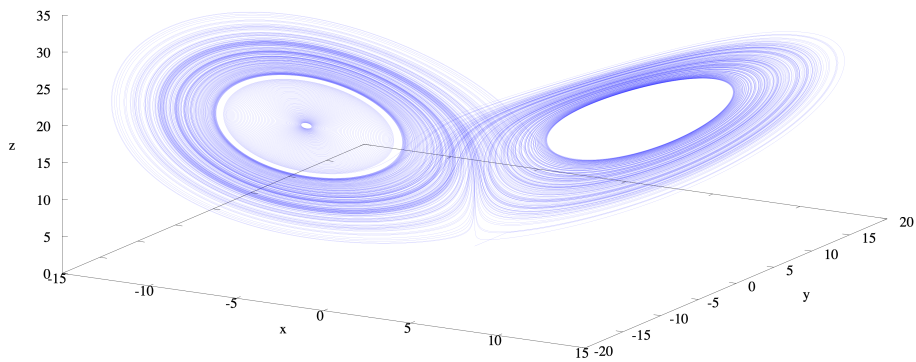

Figure 2.4: Lorenz system state space for 10^5 iterations depicting a periodic oscillation. $\rho = 18$.

With $\rho = 28$, we have a strange attractor.

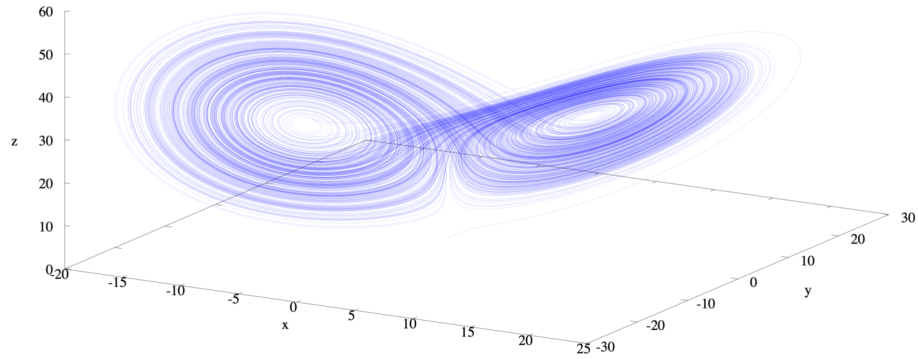

Figure 2.5: Lorenz system state space for 10^5 iterations depicting a strange attractor. $\rho = 28$.

With $\rho = 120$, the system is in a critical state as it is approaching its stability threshold. In this case, we can see a rather chaotic behaviour with irregular trajectories.

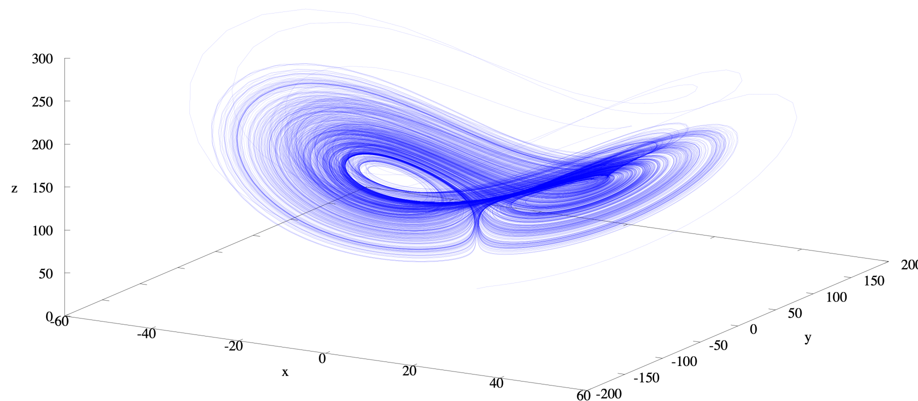

Figure 2.6: Lorenz system state space for 10^5 iterations depicting a chaotic behaviour. $\rho = 120$.

Interestingly, if we move the ρ parameter further up to 125, we see that a rather long periodic oscillation with a nontrivial path emerges. Here, the system is extremely close to losing stability, and a more chaotic behaviour may have been expected. Instead, the system returned to a more ordered state space, highlighting the nonlinearity and asymmetry of these systems.

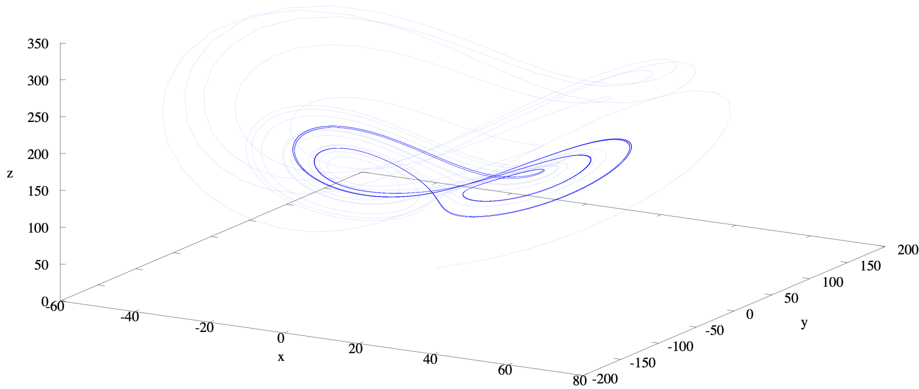

Figure 2.7: Lorenz system state space for 10^5 iterations depicting a long-period oscillation. $\rho = 125$.

2.3 Emergence

Some of the earliest mentions of the concept of emergence date back to the times of Aristotle when, in his *Metaphysics* book, he proposes a comparison between the behaviour of individual parts and the parts themselves working as a whole, underlining how different properties appear when observing the two different cases. Similar to self-organisation, emergence is a concept that can be investigated in a variety of fields and has thus received several definitions that may not always agree with each other [Gershenson, 2007]. One of the most common definitions proposes an intuitive description of the phenomenon, and it describes a system of interactive parts, working together, where the global behaviour resulting from their cooperation displays properties that are not present in the individual parts. The system is said to be "more than the sum of its parts" [Mitchell, 2006]. Morin also believes that the study of systems both as parts and as wholes is central, but he also thinks that it is not enough and that it is vital to focus on the relations themselves between the parts and the whole. Morin agrees that the whole is more than the sum of the parts, but he continues arguing that the whole is also *less* than the parts, since some of the properties of the individual parts are overcome or restrained by the organisation resulting from the system as a whole. Morin goes on by showing that *the whole is more than the whole*, as the whole affects the parts retroactively, and the parts will, in turn, affect the whole. Therefore, the whole is not a mere global entity: it is something with a dynamic organisation [Morin, 1992]. According to Morin, this is the framework where concepts such as life, existence, and being should be understood, as these concepts are not inherent qualities but cases of emergent phenomena.

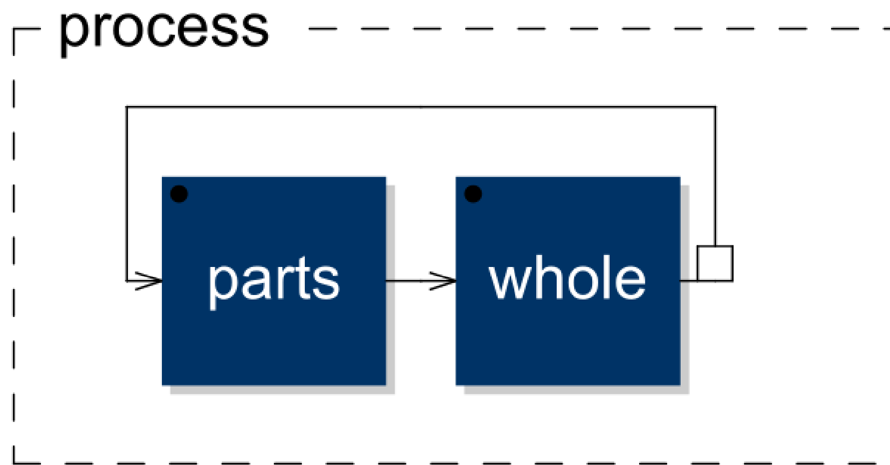

Figure 2.8: Emergent process as retroactive relationships between the parts (low level) and the whole (high level).

2.3.1 The role of the observer

From a more formal perspective, there are two approaches for the definition of emergence that are well-characterised and largely accepted [Bonabeau and Dessalles, 1997]. One approach takes into account models; specifically, a phenomenon can be classifiable as emergent when the model that describes the natural behaviour of a system is no longer sufficient in predicting the future behaviours of a system or describing the current ones, therefore requiring a new model for an exhaustive prediction or description [Cariani, 1991, Rosen, 1978]. The other approach is based on organisational levels. According to this idea, phenomena are emergent when they appear at a high level out of processes and elements defined at a lower level, and when the characteristics exhibited at the higher level are not deducible by combining or processing the characteristics at the lower level [Lewes, 1874]. Bonabeau and Dessalles argue that both approaches implicitly involve an observer. In the case of organisational levels, the emergence of a structure at a higher level can only be determined through some observational tools. On the other hand, the link between a model and an emergent event represents the relationship between an observer and an observed system [Bonabeau and Dessalles, 1997].

Other scholars before Bonabeau and Dessalles had already discussed the issues related to an observer in identifying emergent phenomena, especially in the area of self-organisation. Ashby noted several decades ago the importance of the role of the observer in systems with self-organising characteristics. He stated that: “a substantial part of the theory of organisation will be concerned with properties that are not intrinsic to the thing but are relational between observer and thing” [Ashby, 1968]. Stafford Beer also underlined how, depending on the specific

context within which a phenomenon is observed, the same can appear as either self-organising or self-disorganising [Beer, 1966]. It then becomes clear that emergent phenomena can be defined as such because of a contextualising action performed by an observer, even though emergent systems need some architectural requirements to be called so.

2.4 Autopoiesis

The idea of the whole and the parts recursively affecting each other was also a fundamental intuition in the work on autopoiesis by Maturana and Varela. The Chilean biologists realised that emergent processes follow both a top-down and bottom-up causation, that is, the organisational structure emerging in the global behaviours of a system is affecting the local structures of the parts of the system, while these, in turn, will change back the system as a whole [Benkirane et al., 2002]. Autopoiesis means self-creation and self-production; Maturana first formulised the idea to find what uniquely characterises living organisms. In 1970, he defined the term as "the organisation of living systems as discrete autonomous entities that exist as closed entities of molecular production," and conversely he says that "living systems are molecular autopoietic systems" [Maturana, 2002].

Maturana also claims that autopoiesis only occurs at the molecular level, although the principles of this phenomenon applied as a metaphor to other domains such as that of a computational machine can contribute to the understanding of music systems with complex adaptations and behaviours that actively work towards shaping their organisation. Earlier in their publication from 1980 [Maturana and Varela, 1980], Maturana and Varela define autopoietic machines and discuss their characteristics. Their starting point is a machine that can keep its variables constant or within a limited range of values. To describe this behaviour in the framework of the organisation, they consider machines as entities where processes are completely defined within their boundaries. These machines, in turn, are delimited by their organisation. They call such machines *homeostatic*, where all the feedback loops take place internally, and they extend this concept by saying that if a machine has what may appear to be an external feedback loop connecting itself through the environment, what we have is essentially a larger machine that includes the environment in its organisation. They go on by saying that "an autopoietic machine continuously generates and specifies its own organisation through its operation as a system of production of its own components, and does this in an endless turnover of components under conditions of continuous perturbations and compensation of perturbations." The essence of autopoiesis is a self-referentiality of the machine and recursivity between components that define their network of relations that, in turn, defines the components. The machine incessantly creates and defines its organisation through processes as a system that generates its components, while the components are continuously producing perturbations and adapting to perturbations [Maturana and Varela, 1980].

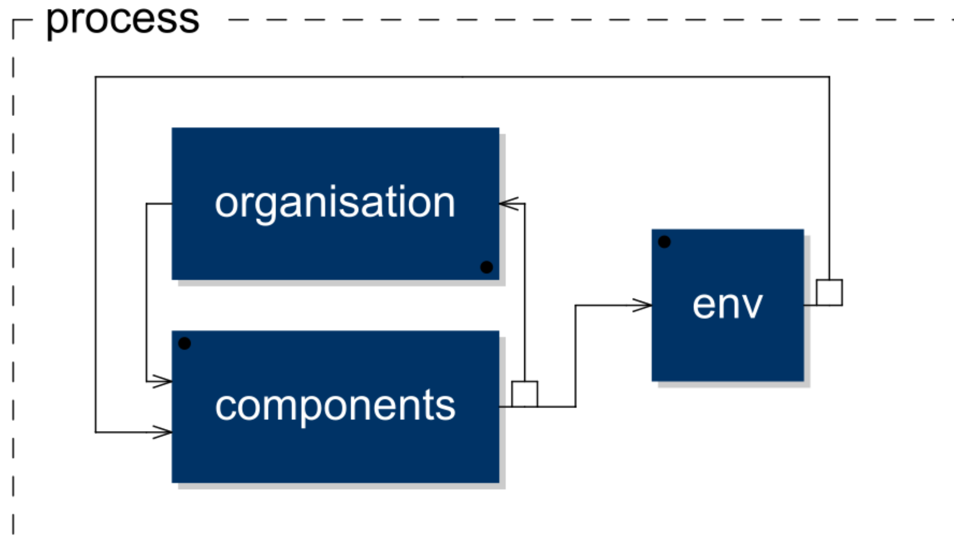

Figure 2.9: Autopoietic configuration. The mutually determining components-organisation couple represents a system that retroactively affects and is affected by its environment ("env").

2.4.1 Structural coupling

Structural coupling is related to the general concept of coupling discussed by Ashby [Ashby, 1957], although structural coupling emphasises the ideas of *structure* and *organisation* in a system and investigates their relationships. Maturana and Varela suggest the notion of *structure-determined system* to indicate systems where their structure at a given time entirely determines anything that happens within the system at that particular time [Maturana and Varela, 1980, Maturana, 2002]. This is undoubtedly the case for the music systems presented here as they are entirely deterministic computational systems, although the actions and perturbations of an external medium must be taken into account when one is present. Structure and organisation are the two main features constituting structure-determined systems. The organisation of a system constituted by composite components is the set of relationships among the components that determine the identity of the system as a whole. The structure of the system, on the other hand, is what characterises the components of the system themselves and their internal relationships and compositions. The organisation of a system can remain unchanged while the characteristics of the individual components – the structure of the system – can change. The organisation of a system is then a subset of the possible structural configurations that a system can undergo while maintaining its identity. Maturana and Varela distinguish between two types of structural changes: those that retain the organisation and identity of the system, and those that disintegrate the organisation of the system by producing a new system class and entity [Maturana and Varela, 1980, Maturana, 2002]. The structural coupling can take place between a composite component and its medium, which acts as a composite component, or between two

or more composite components. Maturana and Varela define structural coupling as a reciprocal triggering of structural variations while the organisation of the components is unaltered. In chapter 3 Distributed adaptation, we will see that structural coupling and the two types of structural changes relate to the techniques of distributed adaptation implemented in some of the works in the portfolio.

2.5 Complexity and adaptation

Complexity is strictly related to the concept of emergence, and for it, too, the observer plays a significant role for the study of the phenomenon [Bonabeau and Dessalles, 1997]. Complex systems can generally include systems showing complex outcomes with or without adaptive properties, but those discussed in this thesis are all CASes. Particularly for adaptation, rather than being considered as a phenomenon on its own, it will be considered as a means that may lead to complex behaviours in trivial systems, or to more complex ones in systems that already exhibit some degree of complexity. In chapter 3 Distributed adaptation, there will be a further discussion on adaptation techniques for complex behaviours where distributed adaptation and emergent adaptive infrastructures will be introduced.

CASes have now been studied for decades, but their intrinsically non-reductionist and interdisciplinary nature makes it difficult to find a single and exhaustive definition [Booker et al., 2005]. Some fundamental characteristics common to all CASes have been identified [Baranger, 2000], although the description of CASes is often approached from various fields, which can result in different terminologies and in focusing on different aspects of such systems. Here, I will describe CASes from the perspective of sound and music while reflecting the more general definitions found in the literature.

Adaptation is generally considered the capability of interdependent parts to respond, locally or globally, to changes in their context and environment or changes in the components themselves, resulting in temporary or long-term state variations and reconfigurations [Mitchell, 2009]. Adaptation implies that these variations and reconfigurations tend towards the achievement of explicit or implicit goals, or that better-fitting states are found through evolutions of the system [Maes, 1993]. Some examples of goal-directedness concerning the music domain will be seen in chapter 3 Distributed adaptation, particularly in subsection 3.2.2 Case study: *Constructing Realities* (2019). For complexity, instead, two fundamental characteristics are emergent behaviours and the *edge of chaos*, that is, an interplay between order and disorder [Waldrop, 1993, Cilliers, 2002].

From a technical and objective point of view, the design characteristics of CASes have been investigated thoroughly by researchers such as Waldrop, Holland, Mitchell, Crutchfield and others. For example, complex systems are often described as networks of nonlinearly interacting components with positive and negative feedback loops [Mitchell, 2006]. The definitions of

adaptation provided above suggest that an adaptive system has a structural coupling with its context [Maturana, 2002]. I will generally refer to *context* as the group of signals that are directly affecting a system, including environmental responses and perturbations in the case of open systems. It is also important to clarify which aspects of the system are being affected by the process of adaptation. According to Holland, the elements of CASes are adaptive agents. It means that, as the agents adapt, the elements themselves change. Particularly, these changes are reconfigurations of the state variable of the agents; the structural coupling is then between the context and the internal variables and components of the agents. This configuration shows that the system, as a whole, can locally affect the agents and vice versa, a combination of bottom-up and top-down causation that is a ubiquitous feature in complex systems [Holland, 2014].

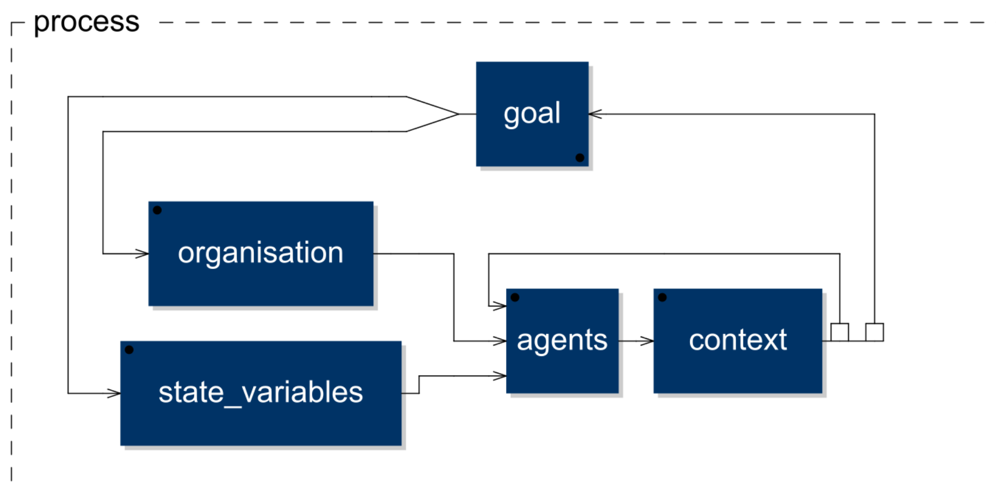

Figure 2.10: Adaptation configuration. The system can change its organisation and internal configuration of its agents through the dynamics of its context mediated by a goal. The direct feedback loop between agents and context represents an exchange of energy.

In music, the observer (listener) has an active role, and the perception of sound has subjective aspects [Bregman, 1994]. Determining whether a system is complex and adaptive based on its resulting behaviour can be counterintuitive and not always possible. As discussed earlier, Bonabeau and Dessalles have approached the problem of defining emergence in a nonmusical context using the notion of *detection*. In particular, in their definition, they show that two different conceptions of emergence, that of high-level structures and that relative to a model, are strictly dependent on an observer, and that the observer can be an abstract entity such as a computational device or a mathematical formula, as long as it is capable of detection [Bonabeau and Dessalles, 1997]. Intuitively, we can see how this applies to the musical domain.

If we assume that the perception of the observer is the only factor determining whether a system is adaptive and complex, we can consider the situation where both the listener and the system are in the same environment. Another critical assumption is that the observed

system is a network of interacting components without automation (i.e., centralised control) or stochasticity (i.e., it is deterministic at least in the digital domain). These assumptions are necessary to make sure that only the network interactions are causing behaviours that exhibit nonlinear dynamics. The system, of course, is also open and sensitive to environmental conditions. The time scale parameter for the detection would be an essential factor. Depending on the time scale, behaviours may either lack or exceed variation, failing to meet the edge of chaos requirement that implies an interplay between order and disorder. Similarly, the amount of state variation needed to detect a phase transition would play a significant role. Time scales and thresholds are thus essential in the detection of musical complexity. Furthermore, systems can show an explicit or implicit response to the environment, but that may not be enough to say if adaptation is taking place.

Are systems with a technically adaptive infrastructure but a limited dynamical behaviour (no significant long-term changes) adaptive? As we have seen earlier, whether a behaviour is sufficient – i.e., it exhibits enough long-term variety – or not for a system to be considered adaptive is observer-related and thus inherently subjective. I suggest that both the objective and the subjective analysis of CASes are useful, especially for music. All systems with a technically adaptive infrastructure, hence context-dependent recursive time-variant systems, can be considered adaptive. Whether the system exhibits a convincing adaptive behaviour or not is a problem of *quality of adaptation*, which belongs to the observer. The fundamental role of the composer is to design an adaptive behaviour that results in a compelling musical complexity.

2.5.1 Case study: *Phase Transitions* (2018-2019)

Phase Transitions is a long-form performance for an autonomous adaptive system operating in real-time which explores the idea of digital deterioration and the behavioural changes in dynamical systems with varying parameters. The work is based on feedback delay networks with nonlinear transfer functions, specifically, saturating units (also known as soft-clipping functions) whose purpose is to ensure stability in self-oscillating conditions but also to make the deterioration process possible.

Saturators transform signals so that amplitude values within a certain range are passed through almost untouched, while values outside that range are compressed never to exceed the limits of the saturators. The more the signals are far from the allowed range, the more the signals are distorted (i.e., signals deteriorate), which results in more frequency components being added to the network.

The feedback coefficients determine how much of a signal is fed back into the network, while the lengths of the delay lines determine after how long a signal will start to recirculate. The first parameter is responsible for the deterioration process as it affects the magnitude of the signals going through the saturators, which will also change the spectral output because of

the added frequencies, while the second parameter is responsible for reinforcing or dampening specific frequencies in the spectrum. These are the two varying parameters in the system.

The delay lengths are chosen as powers of prime numbers. This will favour a homogeneous distribution of the energy over the whole spectrum. The initial feedback coefficients correspond to the self-oscillating threshold of the network. It means that, after being initiated, the network could theoretically operate endlessly without any external energy being provided. In this case, the network is initially triggered by an ideally short impulse (a Dirac) that sets the system into an operating state, producing sparse tones.

The system is coupled with the environment through a microphone (input) and a number of loudspeakers (outputs). The signal from the microphone is processed to extract information and used to pilot the delay lengths. The analysis window of this process is one hour. It means that there is no immediate cause-effect relationship between what happens in the environment and the output of the system: theoretically, a perturbation in the environment will reach its maximum effect after one hour, but the system's output is continuously affected by the past environmental conditions.

The feedback coefficients are set to grow by a magnitude of 2 in about three months. Roughly, that is the limit after which the saturators will be full and will have no further effect except producing broadband noise. That is indeed what sets the life span of the system.

A nonlinear system with several interacting feedback loops and varying parameters is expected to generate phase transitions. These are a radical change in the state of the system, and they show particularly rich and nontrivial dynamical behaviours. The last and most important aspect of the work is an adaptive mechanism for the growth of the feedback coefficients used to explore microscopically such behaviours. This mechanism is used to detect phase transitions in the system through variations in the spectral tendency of the output. When a phase transition is detected, the feedback growth is radically decreased to almost freeze the current state variables until the transition is over. Interestingly, the detection process itself can either trigger or suppress a phase transition, resulting in a system that, by observing itself, will also affect its behaviour, which is what generates the formal development of the piece.

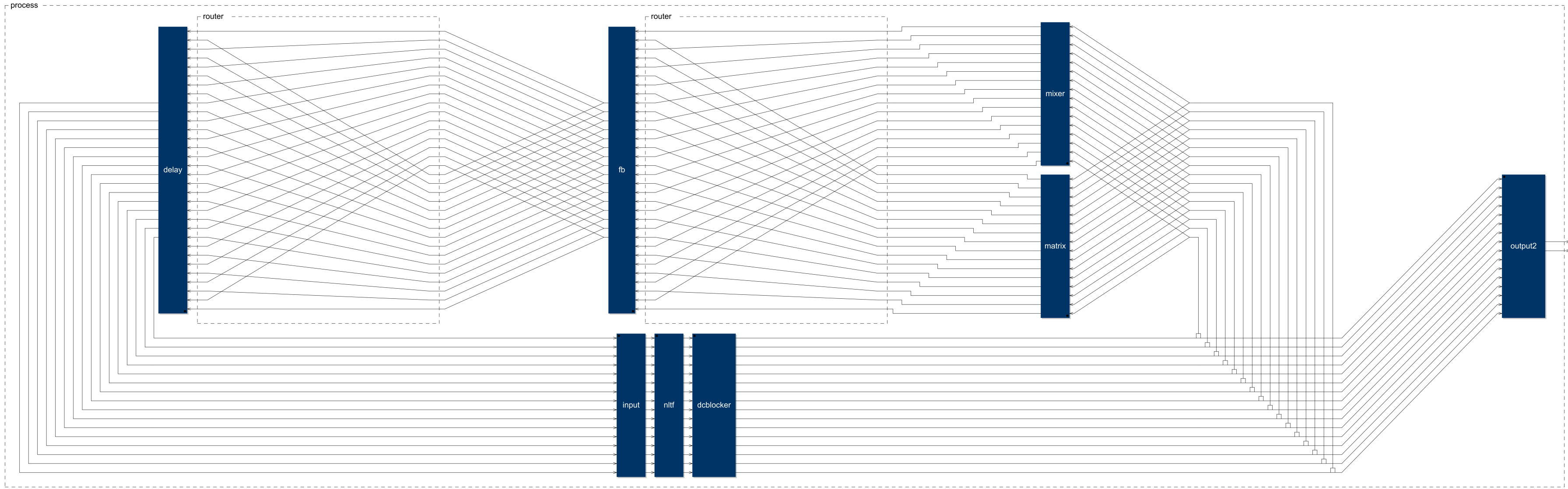

Figure 2.11: *Phase Transitions*: outer layer block diagram. The “input” block contains the initial conditions through which the system is initialised. “Nltf” contains four sets of four different bounded saturators. “Dcblocker” prevents the build-up of steady signals as they would not be optimal for the development of the behaviours of this system in particular. The resulting signals are then duplicated and processed through scattering matrixes in the blocks “matrix”. The “fb” module contains the adaptive mechanism that is applied to each signal to determine its magnitude. The resulting signals are then processed through delay lines whose delay is as well adaptive and determined by the signals themselves, which are ultimately fed back into the “input” block. The signals from “dcblocker” are mixed and used as main outputs.

2.6 Remarks

In this chapter, we have seen an analytical framework in the study of CASes to describe the main features and the fundamental principles of their design. We have discussed the mechanisms of positive and negative feedback both following the theories of cybernetics and by providing real-life examples of how these processes operate. The two mechanisms were then defined according to the more intuitive notions of *divergence* and *convergence* to explain their behaviours.

The essence of chaos theory, the sensitivity to initial conditions that may result from the combination of nonlinearities and iterative processes, was presented through a set of numerical examples based on the Lorenz system showing the characteristic behaviours of chaotic systems. These examples served as the ground for the explanation of other crucial features of CASes such as asymmetry and nonlinearities that extend to different scales and domains.

We have seen that an important intuition from some scholars who studied emergent phenomena is a mutual influence between the parts and the whole in a system. It means that the domain where interactions among components take place and the domain where the result of such interactions manifest are recursively affecting each other. The paradigm that considers the whole more than the sum of the parts was extended into a wider conceptual core where the whole is a dynamical entity rather than a mere result. The whole contributes to the organisation of the parts while the parts, collectively, contribute to the organisation of the whole. The idea of emergence as a bidirectional exchange between parts and the whole is compatible with the idea of autopoiesis. An autopoietic organism is indeed represented by a tight bond between its components, its organisation, and its environment. Such a system can continuously rearrange its organisation while the organisation affects the individual components back in a process mediated by environmental perturbations.

The role of the observer in emergent phenomena is also a central topic, and it is argued that both emergence and complexity are observer-related phenomena, allowing for parallelism with the music domain as it is, too, a process intrinsically dependent on an observer. Furthermore, the concepts of complexity and adaptation are exposed as extensions of each other's domains rather than as independent properties of a system. The idea that the observer is fundamental in the emergence of behaviours in a system, and the importance of a goal, allow interrelating complexity and adaptation so closely.

Practical applications of these theories also reinforced the conception of the observer as a key element. The work *Phase Transitions* (2018-2019) realised for the Rotting Sounds project is a long-form performance/installation without (intentional) human intervention that develops over three months. Such an extended period for the development of the performance showed that time scales and thresholds in an observer are fundamental in detecting complexity. Listeners who experienced the work on a single day could notice some dynamical behaviours within a somewhat stationary structure, hence lacking the necessary long-term variety for the emergence

of the *edge of chaos* state. On the other hand, listeners who visited the installation several times on different days could appreciate the long-term evolutions and global variety of the piece, besides the dynamical behaviours taking place at micro time scales. Similarly, the detection of complex behaviours may be performed by an algorithm, although different temporal criteria and thresholds to determine phase transitions would result in different complexity measures. The design of an algorithm for the measurement of complexity is presented in subsection 3.3.2 High-level information processing.

Chapter 3

Distributed adaptation

The world, as we perceive it, is our own invention.

Heinz von Foerster

This chapter is arguably the core of the research as it discusses the techniques through which autopoiesis and evolvability are implemented in audio feedback networks. The chapter first provides an analysis of the relationships between system and context in complex adaptive systems and the role that information plays in the dynamical realisation of these relationships.

We discuss the characteristics of agent-based modelling for complex adaptive systems, defining the fundamental elements of adaptive agents, and subsequently provide insights on the connections between adaptation and time-variance in audio systems from a musical perspective.

Lastly, the chapter introduces the idea of emergent infrastructures in adaptive agents, that is infrastructures that, at the same time, determine and are determined by the specific context of each agent. Original information processing techniques, indispensable for adaptive systems, are presented, as well as case studies to give practical examples of the possibilities offered by these techniques.

3.1 System-context structural coupling and information

The structural coupling between context and state variable establishes an interface between these two elements through a continuous and bilateral connection [Maturana, 2002]. There is a relationship – a *communication* – between system and context, and both complexity and adaptation strictly relate to information and information processing [Gell-Mann, 1995]. Shannon is considered the father of information theory with his *A mathematical theory of communication*, a landmark publication from 1948. There, he gives a mathematical definition of information using probability theory and the concepts of entropy and redundancy [Shannon, 1948]. Today, other areas involving the study of information criticise his work because he did not take into account context and meaning, and they argue that what Shannon considered to be parallel, that is, information and entropy, are opposites [Logan, 2014]. It has also been shown that his theory is insufficient to describe information in autonomous agents or biotic systems, for which information is described as "instructional" and containing constraints or boundary conditions that pilot the flow of free energy to do work [Kauffman et al., 2008]. For the study of information, too, it is possible to recognise two primary tendencies where on one side the notion is approached from an objective and engineering perspective, while, on the other, questions regarding the observer, context, and interpretation are central. Here, the focus is on a model based on the second way of approaching information, designing the adaptive systems discussed in this thesis as self-aware, context-aware, and self-structuring entities.

In computational systems for generating music, the context is a stream of samples. Specific algorithms are implemented to process that stream in order to generate new streams that will affect the state variable of the system. In turn, the output of the system will alter the context creating a recursive loop. This setup is consistent with Bateson's idea of *elementary unit of information*, which he describes as "a difference that makes a difference" [Bateson, 1979]. Paraphrasing Bateson's statement, it is possible to see how it involves the kind of relationships found in complex systems where the causal and interdependent components of such networks recursively affect each other and spread local changes in the single units throughout the whole system. We may apply this reasoning to a higher-level structure such as the observer-context one or, more generally, the system-context couple (since the observer is itself a system), obtaining a configuration in which the perceived changes in the context are also affecting the way such changes are sensed. This underlines how context plays a fundamental role, showing that information and the way it is interpreted are emergent processes, and that information arises from a contextualising system rather than existing on its own [Logan, 2014].

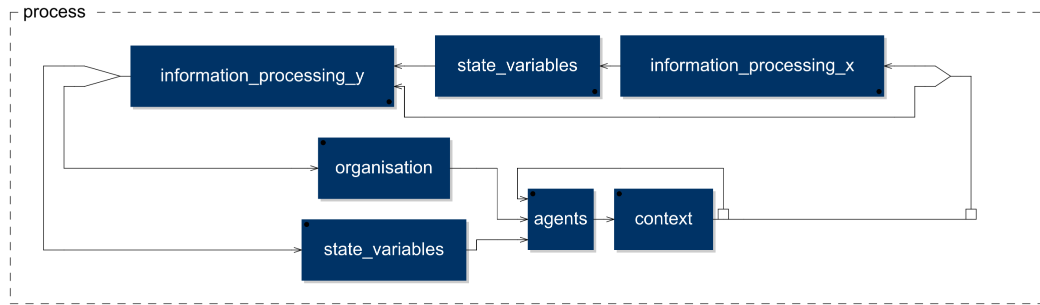

Figure 3.1: System-context structural coupling and information as an emergent process. The goal of the system is implicitly contained in the information processing infrastructure. The direct feedback loop between agents and context represents an exchange of energy.

3.1.1 Detecting and affecting infrastructures

The agents in CASes can be seen as cooperation between two networks, one containing *detecting* parts, the other containing *affecting* parts [Holland, 2014]. The detecting parts are senses that allow the agents to translate perturbations in their local environment into signals that pilot their affecting parts, which will, in turn, change the environment. In musical CASes, the agents will deliberately make apparent sonic characteristics in their surroundings to obtain an interpretation. The resulting signals are nonlinearly mapped into ranges to drive the variables of audio processing units. Finally, these units process the context signal itself (the signal from the overall system or other local agents) and output the results – evolving sounds – to close the loops. The synergy between information signals and mapping functions, through the effectors, is what establishes positive and negative feedback relationships in the context-agent couples. We now have the necessary infrastructure for an adaptive agent, which will make it possible to realise networks of co-evolving agents with complex adaptations.

The detectors include algorithms to observe the environment through low-level and high-level sonic qualities. The main distinction between the two types is that the first one processes the signals locally, focusing on single characteristics over a comparatively short time frame. The second one performs an event-scale observation by considering one or more low-level characteristics and their variations over time to describe global behaviour. Examples of low-level characteristics are the spectral tendency, amplitude power, roughness, and noisiness; dynamicity and complexity measurements of sound events are high-level characteristics. I will discuss these algorithms and a few others later in this chapter.

3.1.2 Adaptation and time-variance

A useful distinction is between *fast dynamic* and *slow dynamic* systems which, in this view, correspond to time-invariant and time-variant systems. A fast-dynamic system can produce significant short-term changes in the state while the internal variables remain fixed. In slow-

dynamic systems, the internal variables change over time according to the characteristics of the context. As a result, the system can produce short-term and long-term changing behaviours through state variations and reconfigurations in the network of interactions [Holland, 2014].

Information signals, either low-level or high-level, are processed using infrasound analysis periods to render the slow-dynamic behaviours that characterise adaptive systems. Namely, information signals vary at a frequency of maximum 15 Hz. The choice for this specific range is based on both perceptual and technical criteria. Firstly, variations or sonic events occurring at a rate above approximately 15 Hz tend to be perceived as timbral effects, i.e., the variations or events tend to be merged into a single stream [Vassilakis and Kendall, 2010]. Secondly, high-rate modulations would introduce and spread energy throughout the spectrum. There would be two main consequences. The changes in the network of interactions would be reduced to simple state variations, and adaptation would not result in long-term evolutions and formal developments; the introduction of energy in the system via fast changes in the variables would produce a mostly broadband output, which would also reduce the long-term variety. Therefore, information signals will usually be at a much lower rate to favour complexity. For convenience, information signals are computed as values in the range [0; 1].

These signals can then be processed together to build connections and enhance their interrelatedness through many-valued logic operators (NOT, AND, NAND, OR, NOR, XOR, NXR, and others) [Mizumoto and Tanaka, 1981, Schumann and Smarandache, 2007]. Time-variance can additionally be extended to information signals themselves by combining processes such as integration and differentiation with thresholds and relational operators. For example, if a signal is above (or below) a value, a quantity can be accumulated to generate increasing (or, respectively, decreasing) signals, or the variations of a signal can be accumulated up to a certain level before inverting the direction of the process.

3.1.3 Case study: *Inexorable Shifting 2* (2019)

Inexorable Shifting 2 is a performance piece for an autonomous complex adaptive network whose focus for the generation of musical complexity mainly lies on the nonlinear interplay between the emergence of sound and the emergence of silence. The network has a structure similar to the model of artificial reverberation based on feedback delay networks with Hadamard matrix [Davis, 2013], and it has been implemented in Faust. This project uses a 16th-order network.

The network is a self-oscillating system with feedback coefficients exceeding the stability threshold by 0.1%. For the audio documentation of this piece in the portfolio, the system was triggered using a Dirac impulse, although it may also be coupled with the environment using microphones to achieve different evolutions each time that the piece is performed. The stability of the network is ensured by using lookahead limiters within the feedback loops.

The agents in each feedback loop of the network combine fractional delay lines (FDL) and single-sideband modulation (SSBM) processing. SSBM shifts the components in a signal

upwards or downwards depending on the sign of the shift. SSBM will be discussed further in chapter 4 Audio processing techniques and DSP implementation. The adaptive infrastructure in this system combines RMS and spectral tendency information processing through many-valued logic to determine the delay in each feedback loop, and it uses spectral tendency information to determine the direction of the shift in the SSBM units. Specifically, the lengths of the delay lines in each loop are selected using a sample-and-hold (SAH) module; the condition for the SAH module to select the new length and output the result is for the RMS in the loop to be higher than 0.9. The combination of RMS and spectral tendency through nonlinear many-valued logic operators [Schumann and Smarandache, 2007] determines the actual lengths of the delay lines, calculated in milliseconds. There are four logic operators homogenously distributed through the 16 delay lines, AND, NAND, OR, NOR, which are described by the following relationships:

$$\begin{aligned} y_{AND}[n] &= x_1[n] \cdot x_2[n] \\ y_{NAND}[n] &= 1 - x_1[n] \cdot x_2[n] \\ y_{OR}[n] &= x_1[n] + x_2[n] - x_1[n] \cdot x_2[n] \\ y_{NOR}[n] &= 1 - x_1[n] - x_2[n] + x_1[n] \cdot x_2[n] \end{aligned} \quad . \tag{3.1}$$

Rather than using standard FDL, which are based on some interpolation algorithm to transition among successive samples in a buffer, this system uses what are sometimes referred to as non-transposing delay lines (NTDL). These delay lines work by linearly interpolating between two independent delay lines that share the same input signal. The mechanism is to use one delay line as output and to have the second delay line operating in the background for whenever the delay is changed. Thus, when the delay changes, the non-operating delay line is set to the target delay, and the transition happens by linearly interpolating between the outputs of the delay lines in the desired time. The delay line that was initially non-operating is now the active one, while the former operating delay line becomes idle, and so on.

The main difference between the standard FDL and the non-transposing one is that the latter has no Doppler effect or pitch shift. Although, for the particular case of FDL in self-oscillating feedback loops, the fundamental difference is that the input-output phase relationship in NTDL is not preserved, as self-oscillation itself is a function of phase. In a self-oscillating feedback loop containing standard FDL, changing the delay will result in a variation in the pitch and the state of self-oscillation will be preserved. When using NTDL, the state of self-oscillation is interrupted during the period in which the delay line transitions from the current delay to the new one, and the self-oscillation will be reestablished once the transition is complete. The interpolation time in this system is relatively long: 60 seconds. The use of NTDL is the fundamental mechanism that allows for lasting silent parts to emerge, which is one of the distinctive aspects of the

aesthetics of the work.

The frequency shift magnitude for each feedback loop is fixed. For i being an integer from 1 to 16, the shift amount is

$$shift_i = \sqrt{i}/1000 \quad . \tag{3.2}$$

Information obtained through spectral tendency processing determines the sign of the shift, hence its direction. Namely, if the spectral tendency is higher than 0.1, then the sign is negative, and the frequency components will be pushed down. If the spectral tendency is equal or below 0.1, then the shift is positive, and the components will be pushed up in the spectrum. Let us recall that the spectral tendency output is a real value between 0 and 1 whose extremes correspond to DC (0 Hz) and $Nyquist$.

SSBM within a feedback loop, hence as an iterative circuit, is essentially a glissando: a certain shift amount moves the frequency components towards one direction, and the result of such a process is then subject to the same process again and again, recursively. Perceivable glissandi hold a rather strong musical directionality and predictability, as, intuitively, it is possible to foresee the next stages of the process. The choice of minimal shift amounts, such as the ones in this work, are indeed necessary to avoid such predictability by stretching in time the process of frequency gliding. The slow frequency shifting affects the long-term behaviour of the system by redistributing the components based on a counterbalancing mechanism, but it affects the short-term behaviour, too, by constructively or adversely influencing the self-oscillating state of each feedback loop. This will also contribute to complexifying the interplay between the emergence of sound and silence or non-audible sounds.

Lastly, the responsiveness parameter for the global behaviour of the system is set to 1/180 Hz, which is used for the analysis window of the RMS and spectral tendency information processing.

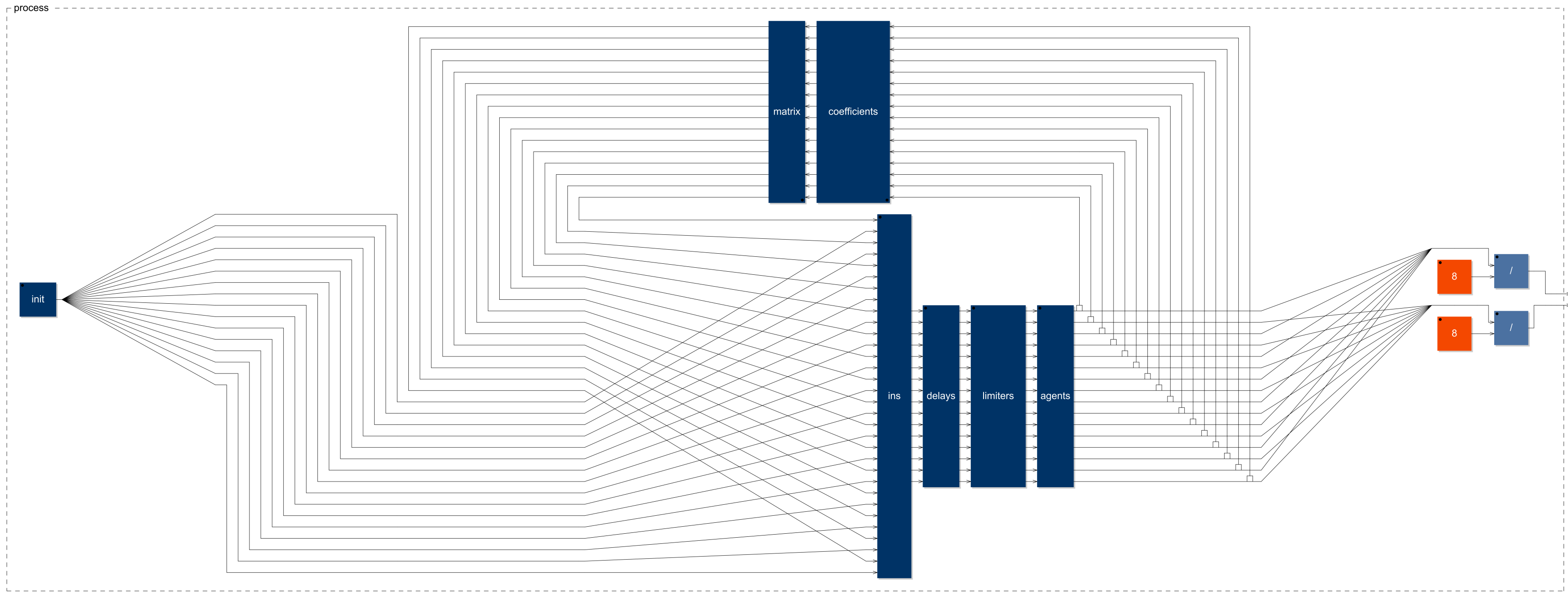

Figure 3.2: *Inexorable Shifting 2*: outer layer block diagram. “Ins” contains the initial condition that triggers the network. The “delays” block contains the non-transposing delay lines with the adaptive mechanisms to determine the delay lengths. “Limiters” are lookahead limiters and “agents” contains the SSBM units with adaptive shifts. The resulting signals are mixed and used as main outputs while also being sent back and redistributed through a scattering matrix.

3.2 Emergent infrastructures

Some of my early investigations of the idea of emergent infrastructures are from 2016. I discuss them in an article from 2018 where I present a work implementing emergent mapping functions (nonlinear and monotonic) to connect information signals and audio processing variables [Sanfilippo, 2018]. Essentially, in the early investigations, the information processing algorithms are context-independent and they have static parameters. The assignment of information signals to audio processing variables is fixed. The minima and maxima of the mapping functions, though, is dynamical and context-dependent, resulting in narrower or broader ranges of action over the audio processing variables, but also in the functions shifting from increasing to decreasing and vice-versa.

Emergent mapping functions can affect the agent-context relationship and can produce aesthetically convincing results, although this is not enough for the infrastructures themselves to be fully emergent. The cooperation between information processing and mapping functions, which we can now more appropriately call *mapping processing*, determines the agent-context relationship. By making both information processing and mapping processing emergent, it is possible to alter the agent-context relationship profoundly, which is a crucial aspect of the idea of distributed adaptation that can significantly enhance long-term variety and complexity.

Characteristics of the context will determine changes in information processing. For example, the type of information signals flowing into logic operators can change, or the operators themselves can vary over time. Besides, the assignment of signals to audio variables can be time-variant, as can be the analysis periods used to compute the information signals, in order to have different responsiveness. This design shows that the paradigm of *information signals affecting information signals* (or *meta-information processing*) can be applied recursively to theoretically any number of levels to realise a model of radical constructivism. That is a configuration where the perceived context affects the perception process itself.

Adaptation can be distributed to even more aspects such as the topology of the network, implementing what is referred to as *adaptive wiring* in complex networks [Boccaletti et al., 2006], and the type of audio processing algorithms in the agents. The connections between agents could be activated or deactivated to transition, for instance, from a fully-connected network to a circular one; the audio processing in an agent could shift from a low-pass filter to a high-pass or granulator. The system would then be able to recompose itself autonomously and continuously in all its characteristics, realising the idea of autopoiesis to its full extent.

As mentioned earlier, distributed adaptation is related to the notion of evolvability in biology and genetic algorithms, which is the genome's ability to produce adaptive variants [Wagner and Altenberg, 1996]. The connection with evolvability is indeed the ability of systems implemented through distributed adaptation to generate mutations of themselves, hence systems with new infrastructures for the interpretation and alteration of context and self. In fact, the essence of

timbral and formal evolutions through such a design is the trace of a process where the system incessantly questions and accepts its identity; an endless process of genesis and dismemberment from which music and complexity emerge.

3.2.1 Case study: *Order from Noise (Homage to H. von Foerster)* (2016-2019)

Order from Noise is a performance project following the *reduced intervention* paradigm discussed in section 5.3 Reduced intervention implementing an environment-dependent impulse response in a closed system. This project was originally the prototype exploring the formulation of *distributed adaptation*, and its most important feature is an emergent mapping processing infrastructure. Namely, the ranges through which the variables in the agents adapt vary based on the analysis of local signals in each agent. The effect of such an emergent mapping infrastructure is that the variables in the agents orbit through state spaces of different size, but also that the relationship between context and variables can switch from positive feedback to negative feedback and vice versa.

The DSP network consists of five adaptive agents implementing the following audio processing techniques: recursive comb filtering; state-variable (high-pass/low-pass) filtering; reverberation; sampling; pulse-width modulation. A quasi-full (anti-identity) feedback network topology with adaptive feedback coefficients interconnects the agents, and lookahead limiters, discussed in subsection 4.2.2 Lookahead limiting, guarantee the stability of the network. The delay in the feedback loops is eight seconds.

The information processing infrastructure is entirely based on measurements of polarity tendency, presented in subsection 3.3.1 Low-level information processing, subsequently processed through nonlinear transfer functions. Furthermore, the resulting signals are integrated and processed again through nonlinear transfer functions to implement inner time-variance, as explained in subsection 3.1.2 Adaptation and time-variance.

Von Foerster's *order from noise* principle inspired this work. The performance takes place with the human agent triggering the DSP network with a 0.001-second impulse. The signal from a microphone in the environment is sampled for 0.001 seconds and sent to the agents. The network is thus in an open configuration for only a very short period and is subsequently a closed and self-oscillating system. The short impulse of background noise taken from the environment is the initial energy that recirculates in the network, and that determines the entire development of the system.

For this work, the performer is mostly an observer whose primary role is to determine the end of a system's evolution that displays coherence and completeness from the musical perspective. After the end of a machine's performance, the DSP network is reset to its initial state, ready to be initialised again with a new background noise impulse.

Statistically, the background noise in the environment will never be the same, considering a time scale of 0.001 seconds, corresponding to 96 samples at a samplerate of 96 kHz. Of course, for human perception, there would be no change among successive impulses, though the differences in the sequence of samples that trigger the system are determinant for the system: *"a difference that makes a difference"* [Bateson, 1979]. An identical initial condition would result in an identical unfolding of the system's output.

The initial state variable is identical before the impulse triggers the system, and it is possible to remark similarities at the beginning of each evolution. The essence of the formal musical development of this work is to display the high sensitivity to initial conditions by reinitialising the system with minimally different impulses, resulting in a succession of dynamical behaviours that share the same origin and that progressively diverge through radically different state spaces in the long-term.

The individual evolutions of the system may last between three to ten minutes, and the process of reinitialising the system may be performed between three to ten times depending on the length of each development. Approximately, the performance of the piece lasts between 15 and 30 minutes.

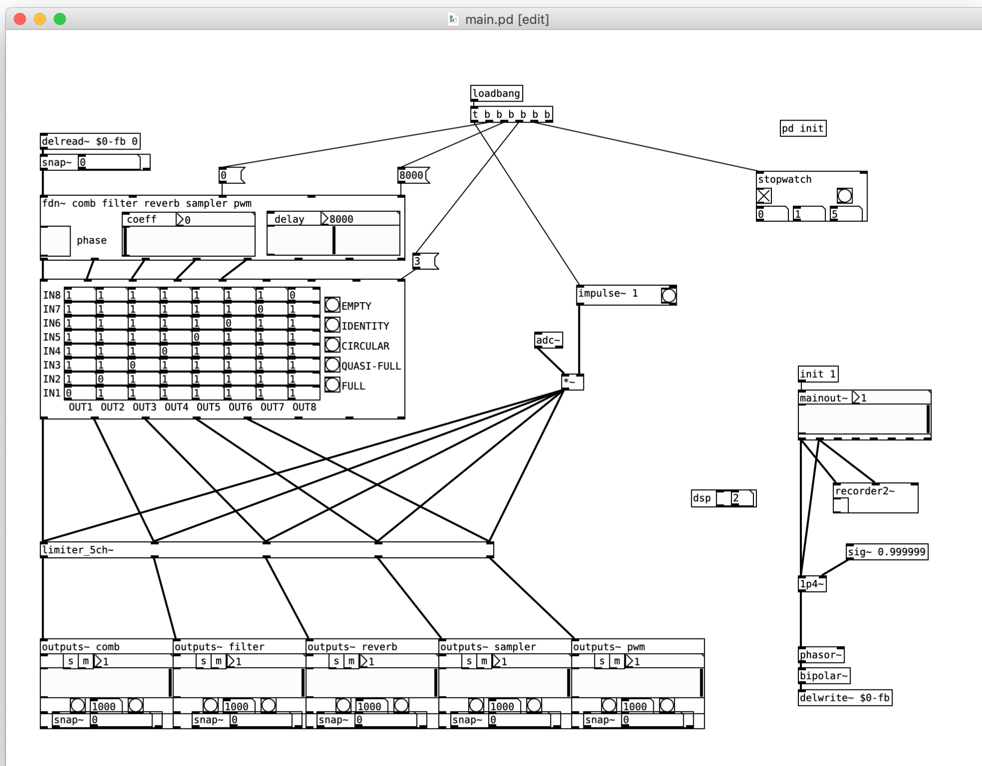

Figure 3.3: *Order from Noise* Pure Data main patch.

3.2.2 Case study: *Constructing Realities* (2019)

Constructing Realities is the project that realises the paradigm of *emergent infrastructures* reviewed in section 3.2 Emergent infrastructures in its entirety, following the idea of radical constructivism [von Glasersfeld, 1979, Von Foerster, 2003b] applied to the design of CASes. The work is a performance for a DSP network in an ecosystemic configuration with fundamentally no human intervention, except that of ending the performance after a sense of completeness is perceived.

The network implements six adaptive agents with the following audio processing techniques: zero-crossing (ZC) granulation; state-variable filtering (high-pass/low-pass/band-pass/all-pass/band-stop); sampling; reverberation; feedback delay network with state-variable filtering.

The network is connected through the environment with a microphone, and the information processing infrastructure relies mainly on two sources for the generation of self-organising signals: *global*, when agents use the signal from the environment; *local*, when each agent uses the signal at its input.

The information processing infrastructure of each agent consists of a set of three low-level information measurements: RMS, spectral tendency, and noisiness, described in subsection 3.3.1 Low-level information processing. The responsiveness of the calculation of the information signals is dependent on the dynamicity (see subsection 3.3.2 High-level information processing) calculated from the global environment. Mapping processing is applied to the three information signals to pilot three key variables in each agent. The information signals that are assigned to each variable vary dynamically based on the polarity tendency calculated from the local environment. The mapping functions, too, vary dynamically based on the polarity tendency of the local environment, modifying the cybernetic relationship between agents and context. Even though information signals and mapping functions change according to the same information processing from the same source, they are differentiated through nonlinear transfer functions and integrators.

The topology of the network can vary between identity, anti-identity, circular, and fully-connected. The variations in the network topology are the result of the measurement of complexity (see subsection 3.3.2 High-level information processing) combined with nonlinear transfer functions and integration.

Finally, the feedback network is kept stable using lookahead limiters, discussed in subsection 4.2.2 Lookahead limiting, while the feedback coefficient and the delay of each feedback loop are determined by polarity tendency measures of the local environment.

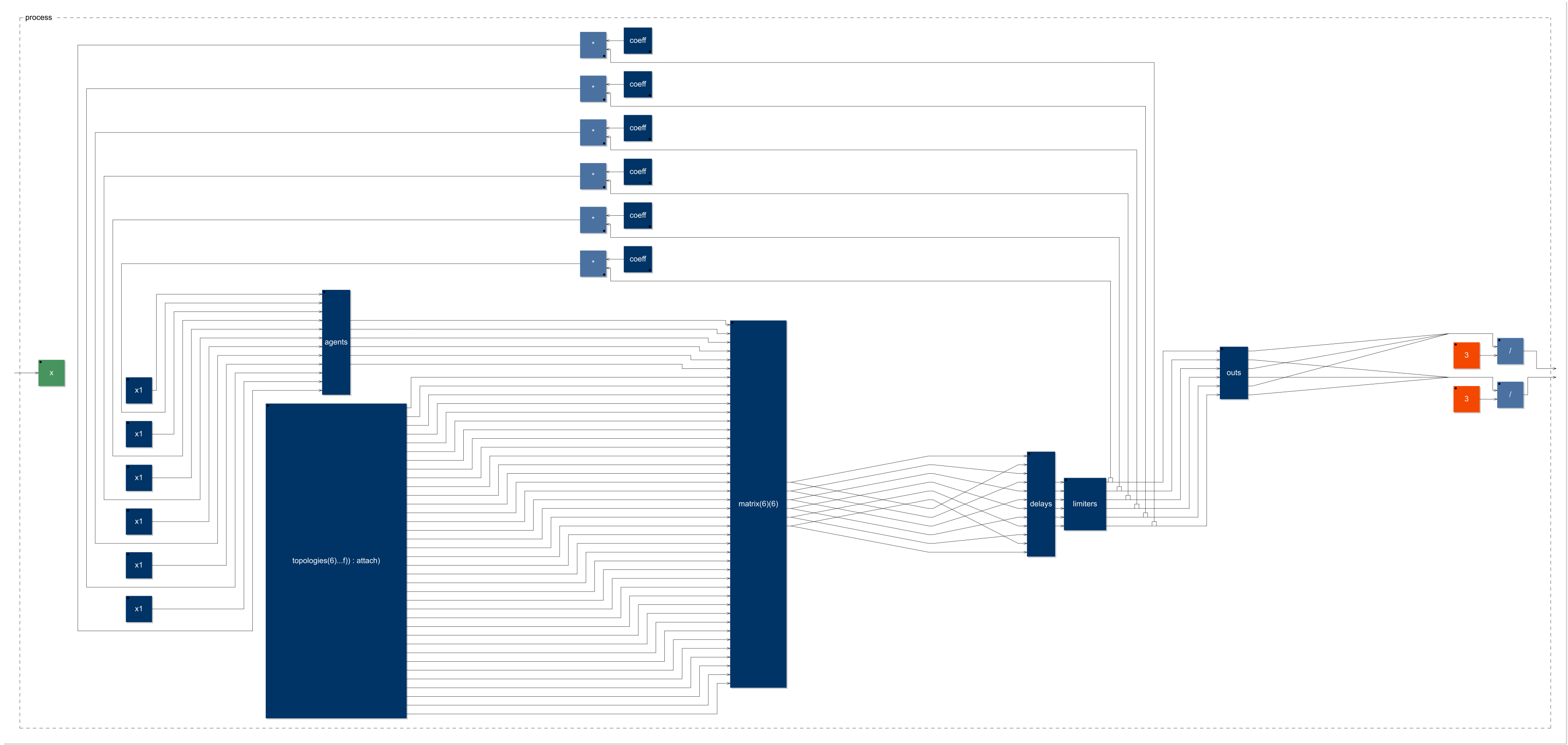

Figure 3.4: *Constructing Realities*, outer layer block diagram. “Agents” contains the six adaptive networks carrying out the audio processing of ZC granulation, state-variable filtering (high-pass/low-pass/band-pass/all-pass/band-stop), sampling, reverberation, feedback delay network with state-variable filtering. “Topologies” contains four topology types that are selected based on complexity analysis, using linear interpolation among the topologies. The topologies are then used in “matrix” to combine the agents. The outputs from the agents are processed through adaptive delays, limited, and used as main outputs after being mixed. The outputs from “limiters” are also sent back to the agents through adaptive feedback coefficients and mixed with an external signal.

3.3 Information processing techniques

The advent of complex adaptive systems (CASes) and their close connection to information have favoured the development of algorithmic techniques for information processing and the analysis of low-level and high-level behaviours. This work, in the music domain, concerns two types of information processing. One tries to model specialised senses for sonic characteristics directly correlated to human perception. We can call it *modelling information processing*. The other describes sonic features not strictly associated with human perception. We can refer to this as *abstract information processing*, although such features can still be meaningful for the machine and can be applied to establish structurally-coupling connections. Both approaches can be used to create positive and negative feedback relationships as well as distributed adaptation. Conceptually, and this may, of course, affect the musical outcomes, the main difference in the creative practice is whether the focus is on implementing a humanly-modelled autonomous system, or on highlighting aspects related to the aesthetics of the machine and abstract – although coherent – emergence and adaptation.

3.3.1 Low-level information processing

Several feature-extraction algorithms for audio implement frequency-domain techniques [Puckette et al., 1998, Brent, 2010, Peeters et al., 2011]; the fast Fourier transform (FFT) is today an efficient and optimised algorithm, and concerns linked to CPU usage may or may not be a problem depending on the application. However, FFT still requires a considerable amount of resources, and it comes with a trade-off between time and frequency resolution. The relationship between time and frequency is also known as Gabor Limit, which is Gabor's application of the Uncertainty Principle to signals. Briefly, it states that, in a signal, it is not possible to observe behaviours in both time and frequency accurately and that the resolutions of the two domains are inversely proportional [Gabor, 1946]. Consequently, if a particular computation requires a high resolution in frequency, there will be a discontinuity between the measurements. The higher the resolution, the more significant the discontinuity, which can result in audible artefacts such as unwanted rhythmic patterns.

The information processing tools that I will show here are a set of heuristic and relatively simple algorithms in the time domain. These systems are not flawless or meant as a replacement for existing high-precision or scientific measuring instruments, but they should be considered as an alternative for live music purposes when the CPU load can become an issue. An advantage of these algorithms is that they provide a continuous stream, that is, the measurement is carried out on a sample-by-sample basis. Furthermore, the simplicity of these algorithms is also made possible by the implementation of adaptive mechanisms, which shows that they can be beneficial in audio technology in general, beyond musical applications.

First, I will introduce low-level perceptual algorithms such as spectral tendency, noisiness,

and roughness. Then I will move on to the abstract ones like spectral spread, spectral peak, spectral minimum, spectral maximum, polarity tendency, and steepness.

Spectral tendency

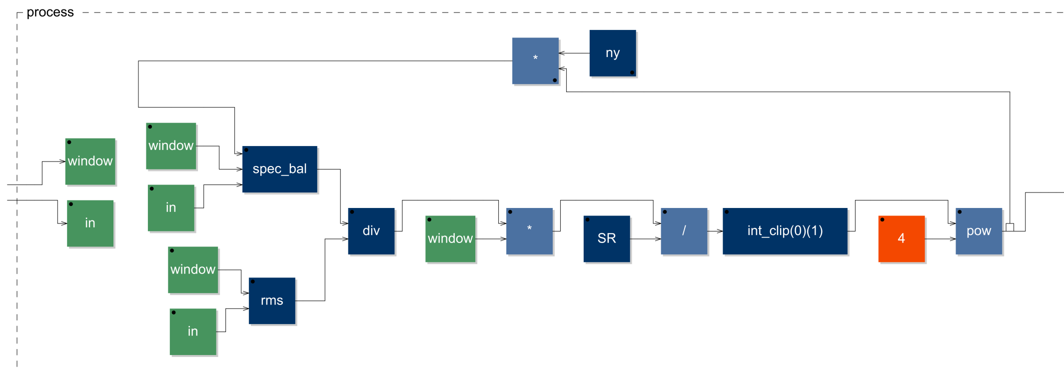

Figure 3.5: Block diagram of the spectral tendency algorithm. The green blocks are external signals: "in" is the analysed signal, "window" sets the analysis rate. "Spec_bal" outputs the difference between the RMS of the high-passed signal and the low-passed signal coming out of a crossover. The output is divided by the RMS of the input signal and multiplied by the analysis rate. The result is then divided by the samplerate and integrated. The signal is finally raised to the power of 4, fed back to the cut-off of the crossover in "spec_bal", and mapped over $Nyquist$.

The spectral centroid is probably the most common measure of spectral tendency connected to the perception of brightness, and it is also considered the best fit for that perceptual feature [Grey and Gordon, 1978]; The spectral tendency algorithm presented here implements an adaptive mechanism via a negative feedback loop. It is related to a centroid calculation as it finds a balancing point in the spectrum where the power of the components above that point equals the power of the components below it.

The input signal is filtered through a first-order crossover with one-pole-one-zero filters. A crossover is a unit that divides a signal into two parts, one with low-frequency components and one with high-frequency ones. It is realised using a low-pass filter and a high-pass filter in parallel with linked cut-off frequency. For a given input $x[n]$ and a cut-off $f_{cross}[n]$ (assumed to be between 0 and $Nyquist$, which is $samplerate/2$), with

$$\begin{aligned}\alpha_{cross}[n] &= (1 - \sin(\omega_{cross}[n]))/\cos(\omega_{cross}[n]) \\ \omega_{cross}[n] &= 2\pi f_{cross}[n]/samplerate\end{aligned}\quad, \tag{3.3}$$

the coefficients for low-pass and high-pass are calculated, respectively, from

$$
\begin{aligned}
a_{0l}[n] &= (1 - \alpha_{cross}[n])/2 \\
a_{1l}[n] &= (1 - \alpha_{cross}[n])/2 \\
b_{1l}[n] &= \alpha_{cross}[n]
\end{aligned}
\tag{3.4}
$$

and

$$
\begin{aligned}
a_{0h}[n] &= (1 + \alpha_{cross}[n])/2 \\
a_{1h}[n] &= -(1 + \alpha_{cross}[n])/2 \\
b_{1h}[n] &= \alpha_{cross}[n]
\end{aligned}
\quad .
\tag{3.5}
$$

The filters sections are

$$
\begin{aligned}
y_l[n] &= a_{0l}[n]x[n] + a_{1l}[n]x[n-1] + b_{1l}[n]y_l[n-1] \\
y_h[n] &= a_{0h}[n]x[n] + a_{1h}[n]x[n-1] + b_{1h}[n]y_h[n-1]
\end{aligned}
\quad .
\tag{3.6}
$$

$y_l[n]$ is the output of the low-pass, while $y_h[n]$ is the output of the high-pass. The design for this filter is by Cliff Sparks.[1] In this case in particular where the filters have a common input and cut-off frequency, the high-pass filter can be calculated efficiently only with an extra operation besides those needed for the low-pass filter following a state-variable design [Zavalishin, 2012]. Equation (3.6) can then be rewritten as follows, and the coefficients for the high-pass filter are no longer necessary:

$$
\begin{aligned}
y_l[n] &= a_{0l}[n]x[n] + a_{1l}[n]x[n-1] + b_{1l}[n]y_l[n-1] \\
y_h[n] &= x[n] - y_l[n]
\end{aligned}
\quad .
\tag{3.7}
$$

The spectral tendency is calculated as the cut-off frequency in a crossover where the overall power of the resulting outputs is equal. The algorithm computes the root mean square (RMS) of the crossover outputs and the RMS difference between high and low parts. A technique to

[1] http://www.arpchord.com/pdf/coeffs_first_order_filters_0p1.pdf. Accessed on the 29th of August 2019.

calculate the RMS in real-time is to square the input signal, average it with a one-pole low-pass filter, and take the square root [Zölzer, 2008]. The cut-off of the low-pass filter sets the analysis window. It is worth mentioning that the RMS is correlated to the loudness of a sound; it can be coupled with the spectral tendency algorithm and the ISO 226:2003 equal-loudness contour for a more accurate measurement of loudness that takes frequency into account.

The one-pole low-pass filter, for a general input signal $x[n]$, is based on Chamberlin's design [Chamberlin, 1985], which models the charge and discharge periods of a capacitor. Hence, the RMS calculation will reach its final value after a period that is the inverse of the cut-off frequency of the filter. The coefficients are calculated as follows:

$$\begin{aligned} a_{0lp}[n] &= 1 - b_{1lp}[n] \\ b_{1lp}[n] &= e^{-\omega_{lp}[n]} \\ \omega_{lp}[n] &= 2\pi f_{lp}[n]/samplerate \end{aligned} \quad , \tag{3.8}$$

and the filter section is

$$y_{lp}[n] = a_{0lp}[n]x[n] + b_{1lp}[n]y_{lp}[n-1] \quad . \tag{3.9}$$

Recalling the one-pole low-pass filter design, the RMS for a given input signal $x[n]$ is

$$y_{rms}[n]^2 = a_{0lp}[n]x[n]^2 + b_{1lp}[n]y_{rms}[n-1]^2 \quad . \tag{3.10}$$

If the RMS of the low spectrum is

$$L_{rms}[n]^2 = a_{0lp}[n]y_l[n]^2 + b_{1lp}[n]L_{rms}[n-1]^2 \quad , \tag{3.11}$$

and the RMS of the high spectrum is

$$H_{rms}[n]^2 = a_{0lp}[n]y_h[n]^2 + b_{1lp}[n]H_{rms}[n-1]^2 \quad , \tag{3.12}$$

we need to find a crossover cut-off for which

$$\Delta_{rms}[n] = H_{rms}[n] - L_{rms}[n] \approx 0 \quad . \tag{3.13}$$

If the difference is positive, we know that the high spectrum has higher magnitude, whereas if it is negative, the low spectrum is predominant. The sign of the difference, thus, represents the direction towards which the cut-off should move to reach an equilibrium point, while the absolute value of the difference expresses the magnitude of the imbalance, therefore how quickly to move towards that direction to overcome the inequality. If we integrate the resulting difference, we obtain a signal that will either increase or decrease at a particular rate depending on the sign and magnitude of the input of the integrator.

The integrator for a given input $x[n]$ and integration rate $r[n]$ is

$$y_{int}[n] = x[n]r[n]/samplerate + y_{int}[n-1] \quad . \tag{3.14}$$

The integrated $\Delta_{rms}[n]$ can be mapped over $Nyquist$ and can be fed back into the crossover to pilot its cut-off. Though, before doing so, we can raise the output of the integrator to the power of 4 to improve the stability of the system with low-frequency inputs. The resulting signal, $y_{bal}[n]$, is

$$\begin{aligned} y_{bal}[n] &= Nyquist \cdot y_{linear}[n]^4 \\ y_{linear}[n] &= \Delta_{rms}[n]r[n]/samplerate + y_{linear}[n-1] \end{aligned} \quad . \tag{3.15}$$

(3.3), for the crossover, can be rewritten as:

$$\begin{aligned}\alpha_{cross}[n] &= (1 - \sin(\omega_{cross}[n]))/\cos(\omega_{cross}[n]) \\ \omega_{cross}[n] &= 2\pi y_{bal}[n-1]/samplerate\end{aligned} \quad . \tag{3.16}$$

The closed loop will make the self-regulating system (minimally) oscillate, i.e., be in dynamical equilibrium, around the equal-RMS split-point of the spectrum.

A few more adjustments are necessary to improve the performance as well as the stability of the system and prevent it from entering attractors [Gleick, 2011]. The first correction is to clip the output of the integrator and limit it to the range [0; 1] so that the cut-off does not exceed 0 Hz or *Nyquist*. The integrator becomes:

$$y_{int}[n] = \begin{cases} 1, & \text{if } x[n]r[n]/samplerate + y_{int}[n-1] > 1) \\ x[n]r[n]/samplerate + y_{int}[n-1], & \text{if } 0 \leq x[n]r[n]/samplerate + y_{int}[n-1] \leq 1 \\ 0, & \text{if } x[n]r[n]/samplerate + y_{int}[n-1] < 0 \end{cases} \cdot \tag{3.17}$$

The output of the integrator, which is in the range [0; 1], is suitable as the overall output of the system and can be mapped on a logarithmic scale to model the perception of frequencies as follows:

$$y_{log}[n] = \log_b(x[n](b-1)+1) \quad . \tag{3.18}$$

b acts as a 'tension' parameter for the logarithmic curve. For the implementations showed here, $b = 10$. Otherwise, it is possible to just use the output of the integrator before it is raised to a power and fed back into the system for a logarithmic-like behaviour.

Currently, the response of the system is influenced by the overall amplitude of the input signal as it will affect the magnitude of the difference between the crossover outputs. The responsiveness can be amplitude-independent by using the amplitude of the input signal as a normalisation factor, that is, by dividing the RMS difference between the crossover outputs by the RMS of the input signal. Equation (3.15) becomes

$$
\begin{aligned}
y_{bal}[n] &= Nyquist \cdot y_{linear}[n]^4 / y_{rms}[n] \\
y_{linear}[n] &= \Delta_{rms}[n] r[n] / samplerate + y_{linear}[n-1]
\end{aligned}
\quad . \tag{3.19}
$$

The analysis window of the RMS units and the integration rate also affect the system response. Here, they are linked for consistency and used as overall responsiveness parameters. In the illustrations below we can see the behaviours of the algorithms using a 1 kHz sinusoid and white noise as test signals. All tests are performed at 96 kHz samplerate. In the diagrams, the black dot in each module indicates the top position of the inputs, which is reversed in the feedback paths. The small empty square indicates a one-sample delay, which is implicit in feedback loops.

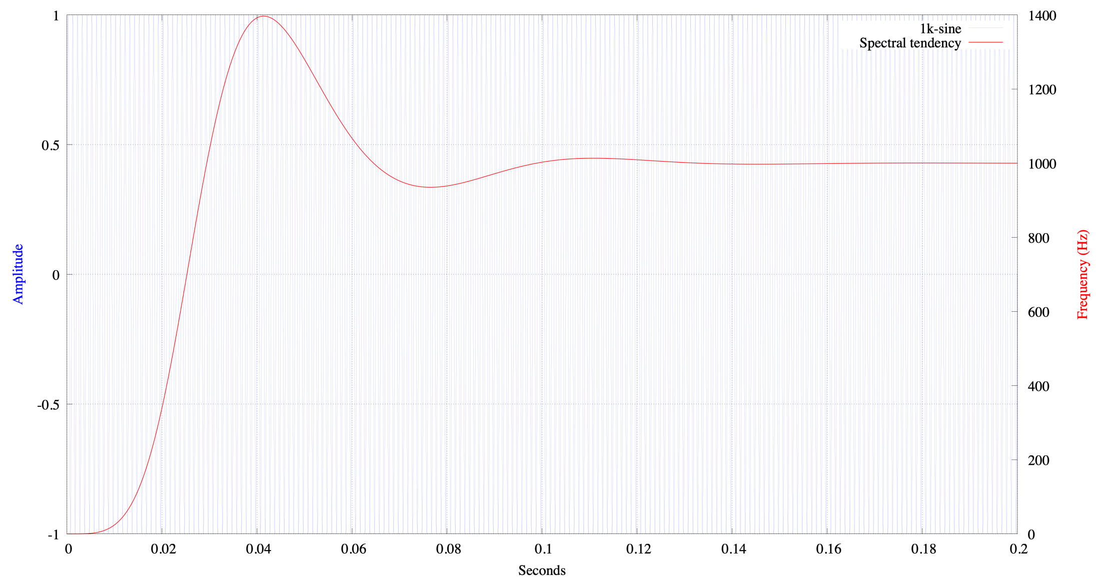

Figure 3.6: Spectral tendency response with a 15 Hz analysis window and a 1 kHz sinusoid as test signal.

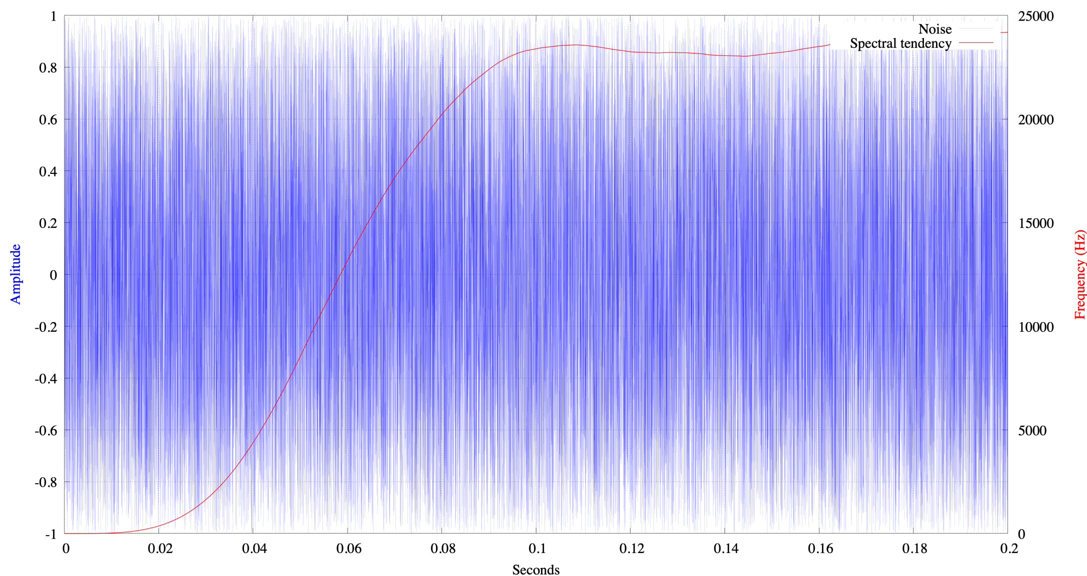

Figure 3.7: Spectral tendency response with a 15 Hz analysis window and white noise as test signal.

If accuracy is not a concern, an optimised version of the crossover can implement one-pole filters, as their coefficient calculation is less CPU-expensive. Besides, the responsiveness could be fixed to limit the calculations even further. Another optimisation is to approximate the coefficients of the one-pole-one-zero filters through linear relations. This way, assuming a normalised cut-off frequency $fn_{cross}[n]$ between 0 and 1, the low-pass filter will have the following coefficients:

$$\begin{aligned} a_{0l}[n] &= fn_{cross}[n] \\ a_{1l}[n] &= fn_{cross}[n] \\ b_{1l}[n] &= 1 - 2fn_{cross}[n] \end{aligned} \quad . \tag{3.20}$$

If we consider the particular context in which these algorithms are applied, that is, the musical one, we can see how a simple notion of *high* or *low* for a specific observed characteristic can be enough to establish meaningful relationships between context and agents, and that accuracy in the analysis may not be essential to achieve convincing musical behaviours.

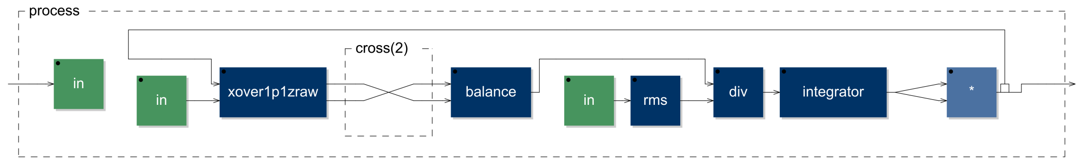

Figure 3.8: Block diagram of the reduced-computation spectral tendency algorithm. The green blocks are external signals. "In" is the analysed signal that is first processed through a raw-coefficients crossover. Balance computes the RMS difference between upper and lower parts from the crossover which is then divided by the RMS of the input for normalisation. The output is then integrated, squared, and sent back to the crossover cut-off.

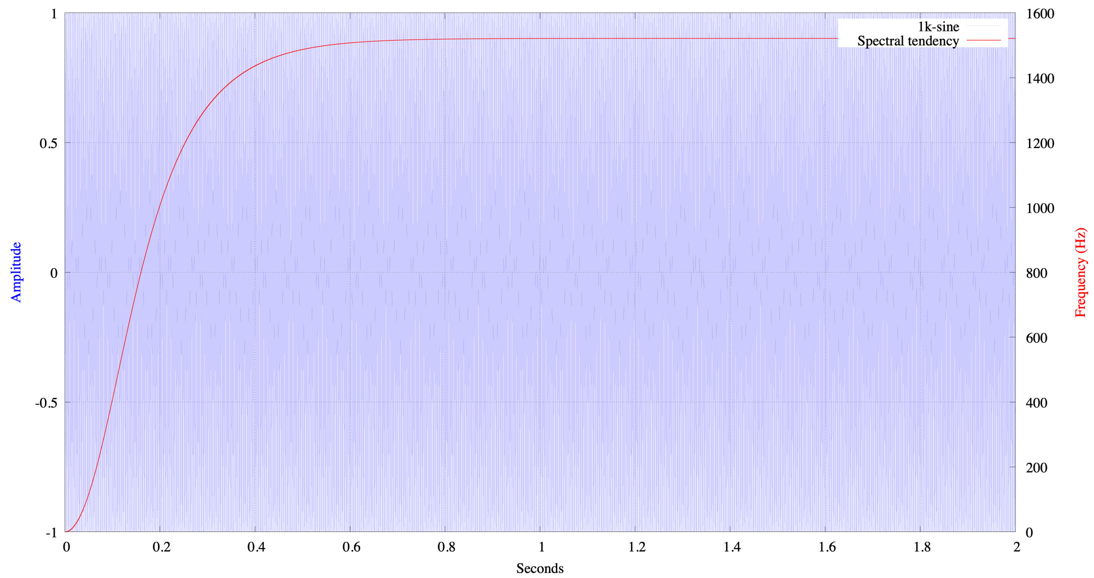

Figure 3.9: Reduced-computation spectral tendency response with a 1 kHz sinusoid as test signal.

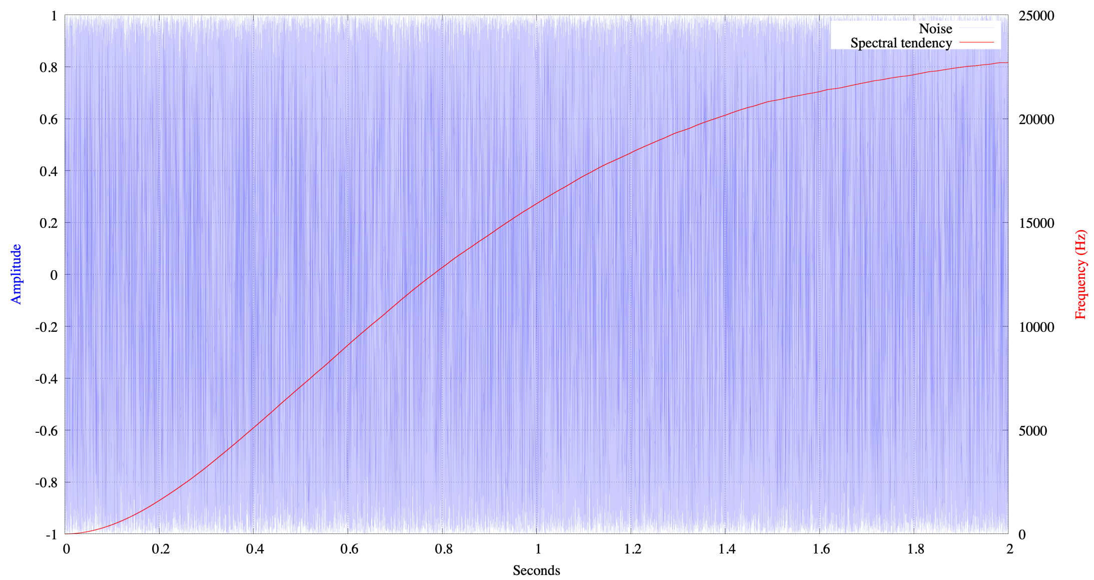

Figure 3.10: Reduced-computation spectral tendency response with noise as test signal.

Noisiness

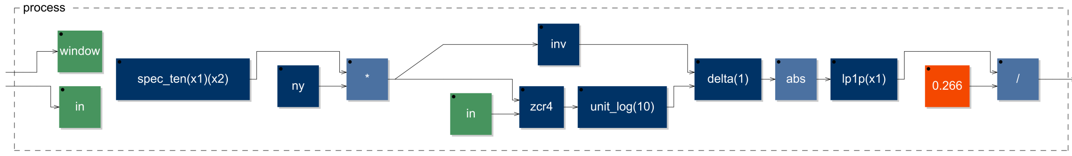

Figure 3.11: Block diagram of the noisiness algorithm. The green blocks are external signals. "In" is the input signal to be analysed and "window" is the analysis rate. "Spec_ten" and the multiplication by "ny" calculate the spectral tendency of the input in Hz. The result drives the analysis rate of the "zcr4" block, which is a zero-crossing rate calculation with four cascaded one-pole low-pass filters. The ZCR is mapped through a logarithmic scale and then differentiated over a period that is determined by the spectral tendency. The magnitude of the ZCR variations is averaged and divided by a constant for normalisation.

The measure of noisiness in a signal correlates with its zero-crossing rate (ZCR) [Giannakopoulos and (Auth.), 2014, Peeters et al., 2011]. The ZCR measures the occurrence of ZCs per unit time, which corresponds to the number of times the amplitude of a signal changes in sign. The ZCR has been implemented both for the analysis of percussive and voice sounds [Gouyon et al., 2000, Tindale et al., 2004, Bachu et al., 2008]. It is a computationally efficient technique, and it can also be used as a frequency detector for sinusoidal signals. The advantage is also that, provided that *DC* components are removed, it gives an amplitude-independent measurement. Although, as it becomes clear from the references above, it works efficiently in isolated situations when two particular types of sounds are compared, for example, voiced and unvoiced sounds, or when sounds within a limited set are considered, as in percussive sounds. In a more general case, the ZCR of a sinusoid at a given frequency does not differ enough from, for instance, band-limited noise centred around the same frequency. Another noise-detection technique operates in the frequency domain, and it computes the noisiness as the ratio between the geometric mean and the arithmetic mean of the energy spectrum [Peeters et al., 2011].

A characteristic of noisy signals is the non-periodicity. Rather than the ZCR itself, what provides useful information for the measurement of noisiness in a more general case is the *rate of change* of the ZCR. In a real-time environment, the ZCR can be calculated through a ZC detector followed by a one-pole low-pass filter to accumulate the ZC occurrences. Such a detector checks if the multiplication between the current sample and the previous one is negative. For this algorithm, I am averaging using four cascaded one-pole-low-pass filters to properly smooth out the ZCR and minimise unrealistically high indexes with high-frequency periodic signals.

$$y_{zc}[n] = \begin{cases} 1, & \text{if } x[n]x[n-1] < 0 \\ 0, & \text{if } x[n]x[n-1] \geq 0 \end{cases} . \tag{3.21}$$

The period of the low-pass filters gives the time frame of the ZCR analysis. The ZCR is then:

$$\begin{aligned} y_{zcr}[n] &= a_0[n] y_{lp3}[n] + b_1[n] y_{zcr}[n-1] \\ y_{lp3}[n] &= a_0[n] y_{lp2}[n] + b_1[n] y_{lp3}[n-1] \\ y_{lp2}[n] &= a_0[n] y_{lp1}[n] + b_1[n] y_{lp2}[n-1] \\ y_{lp1}[n] &= a_0[n] y_{zc}[n] + b_1[n] y_{lp1}[n-1] \end{aligned} \quad . \tag{3.22}$$

The algorithm implemented here works by differentiating the ZCR over a time frame matching that of the ZCR analysis so that variations are effectively detected. Essentially, we are filtering the output of the ZCR with a feedforward comb filter whose coefficient is -1, and whose period is the same as the ZCR frame. Specifically, we have

$$\begin{aligned} y_{diff}[n] &= y_{zcr}[n] - y_{zcr}[n-D] \\ D &= t \cdot samplerate \end{aligned} \tag{3.23}$$

where t is the differentiation period in seconds.

Since the direction of the variation is not relevant, we take the absolute value of the output of the differentiator and finally use another low-pass filter to accumulate these variations over the desired time to obtain an index. Considering that some analysis and differentiation periods may not be suitable for all kinds of signals, an improvement to the algorithm is to calculate the spectral tendency of the input signal (in this case, using the linear scale) to tune in both the ZCR and the differentiator on a rate matching that of the predominant partials.

With $y_{bal}[n]$ being the spectral tendency mapped over $Nyquist$, the coefficients for the low-pass filters in the ZCR become

$$\begin{aligned} a_0[n] &= 1 - b_1[n] \\ b_1[n] &= e^{-\omega_{zcr}[n]} \\ \omega_{zcr}[n] &= 2\pi y_{bal}[n]/samplerate \end{aligned} \quad , \tag{3.24}$$

and (3.23) can be rewritten as

$$
\begin{aligned}
y_{diff}[n] &= y_{zcr}[n] - y_{zcr}[n-D] \\
D &= samplerate/y_{bal}[n]
\end{aligned}
\quad . \tag{3.25}
$$

Considering the logarithmic perception of frequencies, using (3.18), the ZCR output is mapped on a logarithmic scale so that small variations in low-frequency signals contribute as much as larger variations in high-frequency signals. This way, the noisiness index for equal-Q noise signals in different spectral areas is more consistent. Lastly, the output is normalised to 1 based on the index peak when processing white noise. With this technique, with a samplerate of 96 kHz, the noisiness index difference between a sinusoid at 1 kHz and white noise band-passed with a biquadratic filter centred at 1 kHz, having a bandwidth of 1 kHz, is of a factor of about 400 (~ 0.001 against ~ 0.4), whereas the index difference exclusively based on the ZCR is of a factor of about 4 (~ 0.02 against ~ 0.08). The second result is less realistic considering that the comparison is among nearly opposite cases.

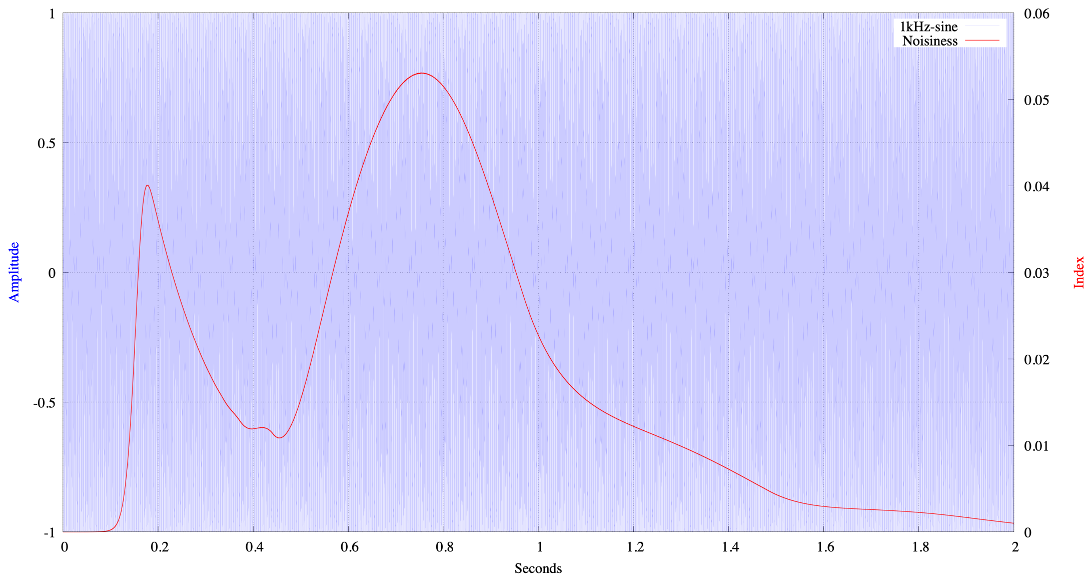

Figure 3.12: Noisiness response with a 1 Hz analysis window and a 1 kHz sinusoid as test signal.

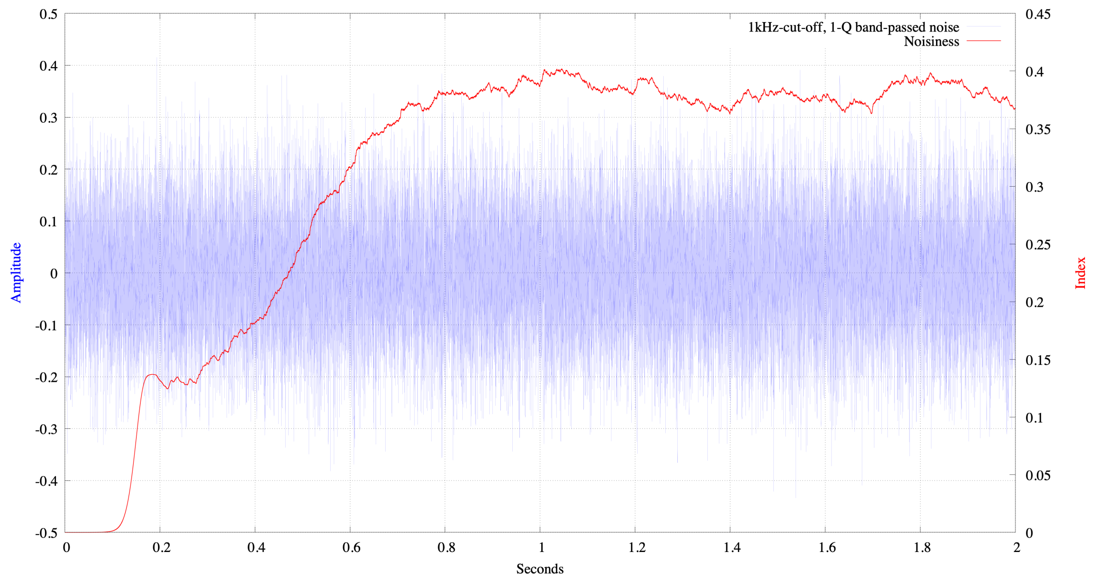

Figure 3.13: Noisiness response with a 1 Hz analysis window and band-passed noise (cut-off 1 kHz, Q 1) as test signal.

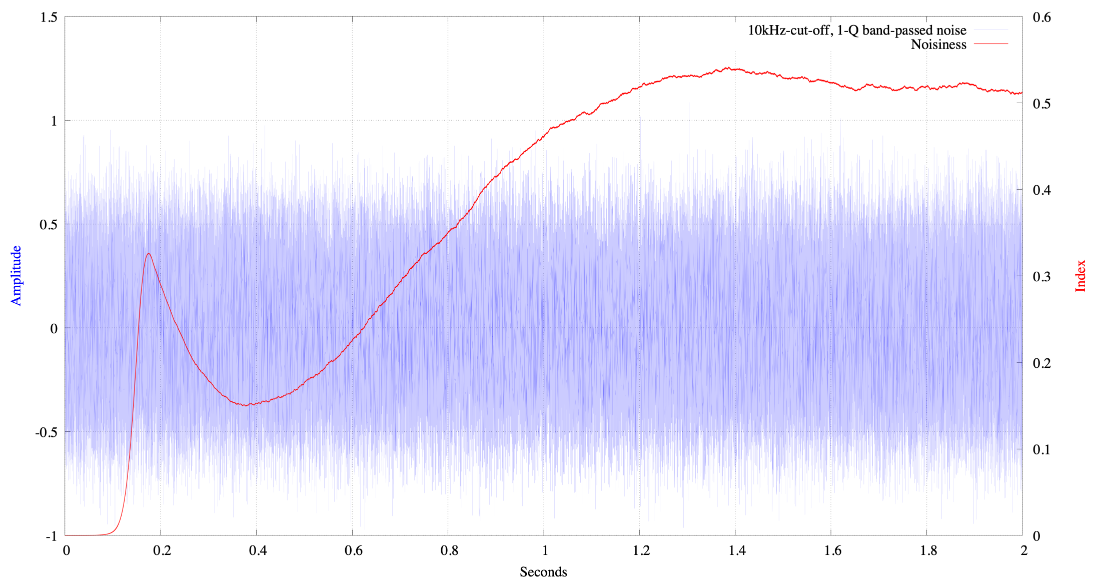

Figure 3.14: Noisiness response with a 1 Hz analysis window and band-passed noise (cut-off 10 kHz, Q 1) as test signal.

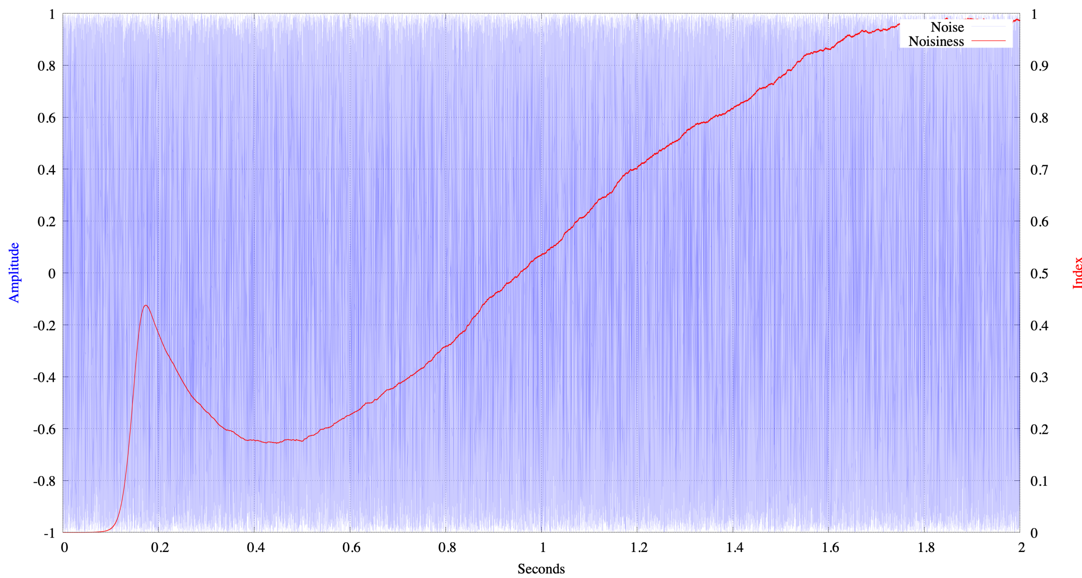

Figure 3.15: Noisiness response with a 1 Hz analysis window and white noise as test signal.

Roughness

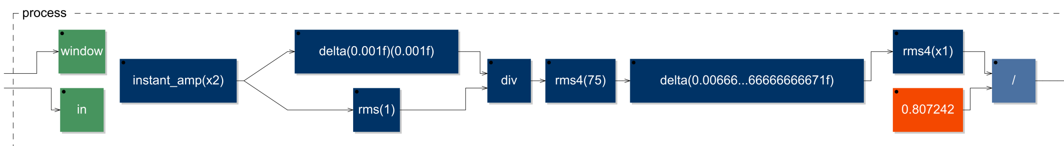

Figure 3.16: Block diagram of the roughness algorithm. The green blocks are external signals. "In" is the analysed input and "window" sets the analysis rate. "Instant_amp" outputs the instantaneous amplitude of the input signal. The result is differentiated over a 0.001-second period and normalised through its RMS. The magnitude is averaged over a 75 Hz rate, differentiated over a 1/150 seconds period, averaged again and finally normalised.

The concept of auditory roughness was first discussed by Helmholtz in the late 19th century to describe a feeling of harshness related to sounds [Helmholtz, 2013]. Vassikalis, in his publication from 2010, refers to auditory roughness within Western music as one of the perceptual correlates for the multidimensional concept of dissonance, and he also presents time-domain and frequency-domain techniques based on formalisations informed by previous experiments [Vassilakis and Kendall, 2010]. Roughness correlates to audio signals with rapid amplitude fluctuations – in the simplest case, two sinusoids with close enough frequencies – and Vassilakis identifies three overlapping fluctuation rate ranges describing three different behaviours. With up to 15 fluctuations per second, we perceive the amplitude variations as individual changes or beats; increasing the fluctuation rate, we enter a range where the perception of roughness reaches its peak; from 75 to 150 fluctuations per second, roughness starts decreasing until it entirely disappears.

The algorithm that I am discussing here is not a precise measurement of roughness – which is also dependent on whether the fluctuation rate is within the critical bands [Terhardt, 1974] –

although it can provide an index that is somewhat proportional to that quantity. This algorithm is mainly implemented for the detection of transients in audio signals, i.e., how non-smooth the amplitude profile of a sound is, but it does so within the roughness range described in the literature. Molla and Torrésani provide a detailed mathematical model based on wavelets in a publication from 2004 [Molla and Torrésani, 2004]. Moore identifies a limit of about 100 transients per second in the human hearing, after which a continuous and non-fluctuating amplitude is perceived [Moore, 1995]. The transient rate considered for analysis here includes fluctuation rates from close to 0 Hz to about 150 Hz, taking into account that the 15 − 75 Hz range is the roughness peak.

The first stage of the algorithm is to get the envelope profile of the signal by calculating the instantaneous amplitude. The instantaneous amplitude can be obtained by turning a real signal into a complex one using a Hilbert filter to perform a frequency-independent 90-degree phase shift. The original and shifted signals constitute an analytic signal with orthogonal real and imaginary parts, and the instantaneous amplitude can be calculated as:

$$y_{IA}[n] = \sqrt{x_{re}[n]^2 + x_{im}[n]^2} \quad . \tag{3.26}$$

The Hilbert filter is based on Olli Niemitalo's fantastic design, which implements two sections of four cascaded second-order filters, one for the real part, the other for the imaginary part.[2] Each second-order filter section has the following form:

$$y[n] = c(x[n] + y[n-2]) - x[n-2] \quad . \tag{3.27}$$

The real signal path needs be delayed by an extra one-sample delay, and these are the c-coefficients used in each of the four filters in, respectively, the real and imaginary paths:

$$\begin{aligned} c_{1_{re}} &= 0.47944111608296202665 \\ c_{2_{re}} &= 0.87624358989504858020 \\ c_{3_{re}} &= 0.97660296916871658368 \\ c_{4_{re}} &= 0.99749940412203375040 \end{aligned} \quad , \tag{3.28}$$

[2] https://dsp.stackexchange.com/questions/37411/iir-hilbert-transformer/59157#59157. Accessed on the 29th of August 2019.

and

$$
\begin{aligned}
c_{1_{im}} &= 0.16177741706363166219 \\
c_{2_{im}} &= 0.73306690130335572242 \\
c_{3_{im}} &= 0.94536301966806279840 \\
c_{4_{im}} &= 0.99060051416704042460
\end{aligned}
\quad . \tag{3.29}
$$

The plots below show the magnitude response of the real and imaginary parts as well as their phase difference.[3] It is possible to see how good of an approximation of the Hilbert transform this filter is.

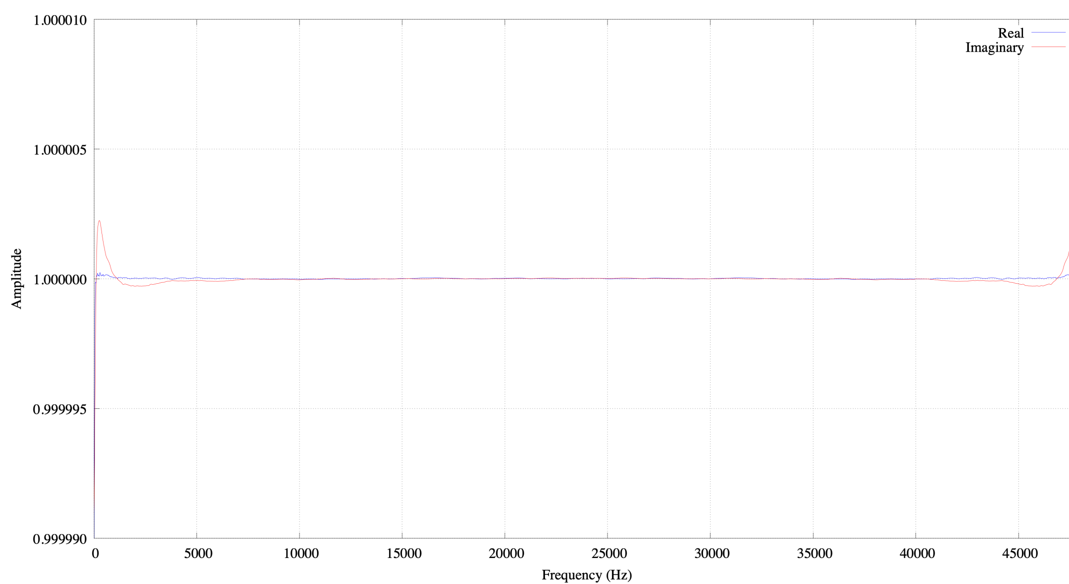

Figure 3.17: Magnitude response of the analytic signal.

[3]The FFT analysis is performed in Python 3. https://www.python.org/. Accessed on the 29th of August 2019.

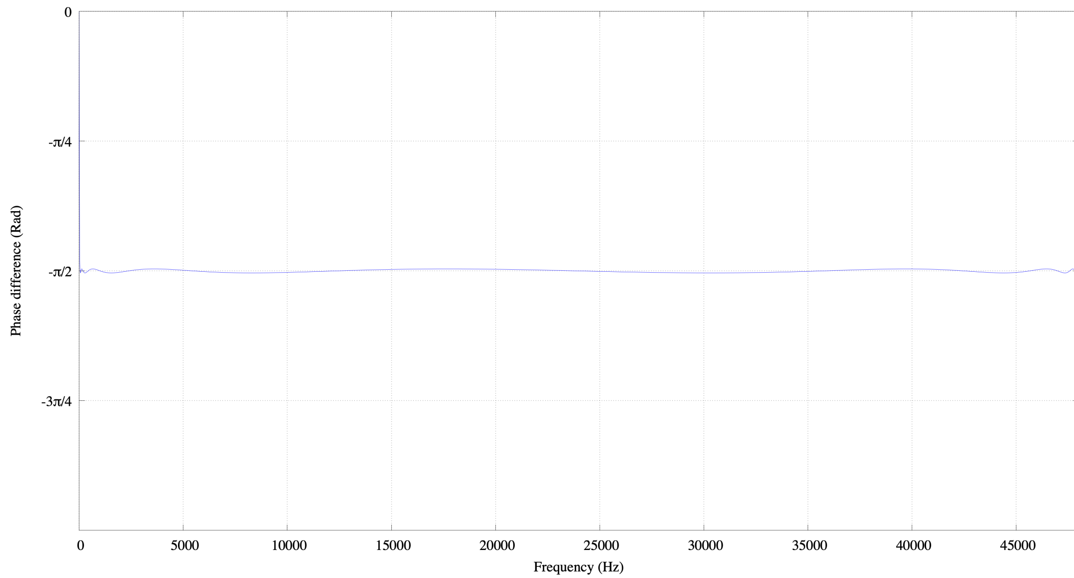

Figure 3.18: Phase difference between real and imaginary parts.

The instantaneous amplitude is differentiated over a period of 0.001 seconds so that fluctuations taking place within this frame or a smaller one can be detected. The magnitudes of the fluctuations are accumulated through an RMS unit with four cascaded one-pole low-pass filters with a 75 Hz cut-off. The RMS output is then differentiated over a period of 1/150 seconds and the magnitude of the output is then accumulated with a one-pole RMS unit whose cut-off determines the responsiveness of the system. The effect of a 75 Hz RMS is to allow fluctuations up to that frequency and to smooth out fluctuations at a higher rate. The second differentiator, with a 1/150-second period, has a positive-sine-shaped frequency response with zeroes at 0 Hz and 150 Hz, besides the zeroes at multiples of the fundamental frequency. The magnitude peak is at 75 Hz and its multiples. The exact magnitude response can be calculated using the formula:

$$G(f) = 2|sin(\pi f d)| \quad , \tag{3.30}$$

where f is the frequency in Hz and d is the delay in seconds.[4]

With these settings, fluctuations at a rate higher than 75 Hz will be smoothed out into a more continuous stream which is progressively attenuated by the comb filter. When reaching about 150 fluctuations per second and above, the magnitudes are averaged into a rather steady stream and almost completely removed. Individual fluctuations below a rate of 15 per second can be attenuated by using a responsiveness of 1 Hz or lower so that they are distributed over a

[4] https://ccrma.stanford.edu/~jos/waveguide/Feedforward_Comb_Filter_Amplitude.html. Accessed on the 29th of August 2019.

large-enough time frame. The last improvements to the algorithm are to divide the output of the first comb filter by the RMS of the analysed to have an amplitude independent measurement, and finally to normalise the output to 1 using a reference signal. In this case, the reference signal is a 45 Hz rectangular wave with a 1% duty cycle. In Figure 3.19 Roughness response with a 1 Hz analysis window and a 1% duty cycle rectangular chirp as test signal., it is possible to see that, for this particular test, the system is slightly overshooting. This behaviour can be corrected by using, for instance, one of the saturators discussed in subsection 4.2.1 Bounded saturators, even though the measurement does not require corrections in most cases.

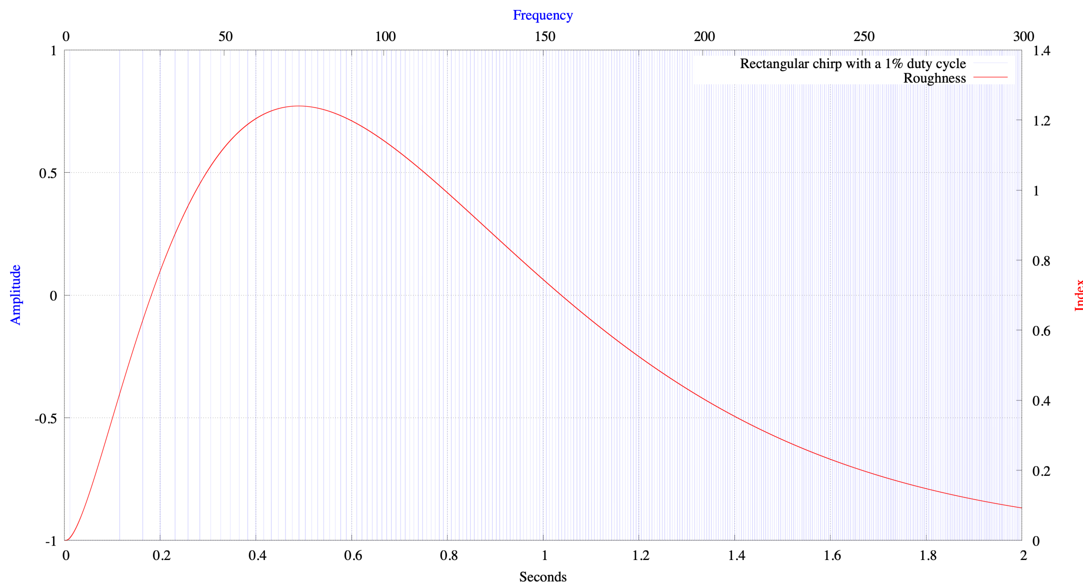

Figure 3.19: Roughness response with a 1 Hz analysis window and a 1% duty cycle rectangular chirp as test signal.

Polarity tendency and steepness

I will now introduce some algorithms for abstract information processing, used to extract information from some of the characteristics of sounds not strictly connected to perceptual features. Two elementary algorithms can be used to calculate the polarity tendency and the steepness of a signal. The first one consists of low-passing a signal, which means to average the samples over a period given by the cut-off frequency of the filter. In my applications, I implement four cascaded one-pole low-pass filters with cut-off frequencies as low as hundredths or thousandths of Hz. The output can then be transformed into unipolar to have 0 and 1 as minimum and maximum. The processed signal merely tells us if the average is positive or negative, and how much it diverges from the origin. Statistically, if we consider a somewhat uniformly distributed signal, for example, background noise in a room, the average tends to 0. As a result, the filtered signal can be very faint, especially when using large averaging periods. One possibility is to expand the low-frequency components by using dynamical normalisation so that the levels of the processed signal and those from the input signal are roughly the same. Dynamical

normalisation can be achieved by dividing the envelope of the input signal by the envelope of its low-passed version to calculate a scaling factor, which is then applied to the filtered signal itself. Depending on the needs, the envelope can be calculated as a peak or RMS contour of the amplitude. We have seen how to compute the RMS in (3.10). A peak envelope can be computed as follows:

$$y_{peak}[n] = \begin{cases} |x[n]|, & \text{if } |x[n]| \geq \tau y_{peak}[n-1] \\ \tau y_{peak}[n-1], & \text{if } |x[n]| < \tau y_{peak}[n-1] \end{cases} \tag{3.31}$$

where τ is a coefficient that determines a decay of 60dBs in a desired time in seconds $t_{60} > 0$. It can be calculated as:

$$\tau = 0.001^{1/(samplerate \cdot t_{60})} \quad . \tag{3.32}$$

The analysis window for the envelope calculation of the input signal should be larger than the rate of the filtered signals to minimise distortions of the contour, which can be caused by intermodulation between the two signals. In a digital feedback environment, this algorithm can be used as a DC-offset detector to pilot the sign of feedback coefficients, as they determine resonances at 0 Hz and $Nyquist$.

The steepness of a signal is simply computed by using a one-sample delay differentiator. The differentiator returns the first derivative, which describes the slope of a signal sample-by-sample. Positive values indicate a rising waveform, whereas negative ones show the opposite. The absolute value of the derivative indicates the steepness of the curve, which is the only characteristic taken into account as the increasing or decreasing behaviour is not relevant in this case. The output of the differentiator is divided by 2 so that the magnitude does not exceed unity, and its absolute value is accumulated using a one-pole low-pass filter. Like in some previous cases, I want to calculate the index as a relative measure, so the accumulated slopes are normalised by dividing them by the RMS of the input signal computed over a one-second frame. The scaled differentiator is the most straightforward form of FIR high-pass filter, while low-passing the absolute value of a signal is a form of envelope following. High-frequency components will contribute more to the growth of the envelope than low-frequency components: this algorithm can be used as a rudimentary brightness processor.

Lowest partial, highest partial, and spectral bandwidth

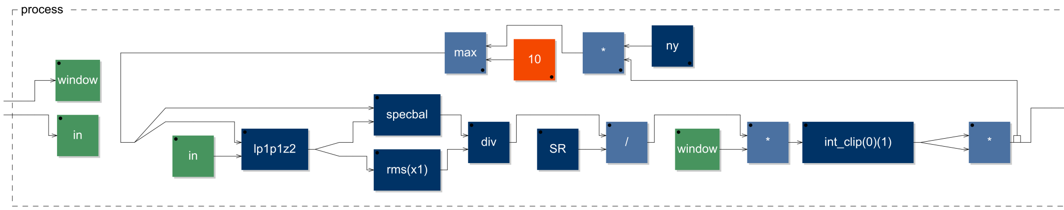

Figure 3.20: Block diagram of the lowest partial algorithm. This algorithm is an extension of the spectral tendency algorithm. The main difference is a low-pass filter at the top of the chain with its cut-off being driven by the feedback-path signal.

A simple extension to the spectral tendency algorithm that I discussed above is to insert a low-pass filter in the feedback loop, at the top of the circuit. As we have seen, the imbalance from the output of the crossover pushes the cut-off frequency towards the predominant side, creating a negative feedback loop. By using the same imbalance to pilot the cut-off of the low-pass filter, the result is a positive feedback loop: the low-pass filter will weaken the upper part of the spectrum, and the imbalance will be pushing towards the lower part even further. This recursive process of frequency attenuation, from high to low components, will terminate when no other elements are left on the lower side of the spectrum, and the negative feedback loop will then be oscillating around the equal-RMS point, which is, roughly, the frequency of the lowest partial. The same principle can be used by replacing the low-pass with a high-pass to implement a system which removes all frequency components up to the last one in the upper part of the spectrum. The combination of the two, clearly, can be used as a measurement of the bandwidth of a signal.

The order and type of the filters implemented in these algorithms have a significant impact on the overall behaviour. For the spectral tendency processor, using one-pole-one-zero filters for the crossover seems to be a good compromise: considering that the RMS difference of their outputs is what matters, the fact that the filters have wide transition bands is not a problem as they will overlap and counterbalance each other out. With the algorithm discussed here, the quality of the low-pass or high-pass filters needs to be high enough to remove the components. Otherwise, the non-attenuated parts may affect the accuracy of the result. The selectivity of the filter, i.e., how narrow its transition band is (which dependents on the order and filter type) can be used as a parameter to pilot the sensitivity and threshold of the system to determine if a component is a signal or noise. Some techniques for the implementation of fractional-order filters could be used, and they can be found in [Tsirimokou et al., 2017]. Alternatively, some preliminary prototypes where I linearly interpolate between the outputs of cascaded one-pole-one-zero filters have also shown convincing results. On the other hand, some high-order filters may in some cases remove the last partial before the system reaches equilibrium, hence resulting in an erroneous calculation. Assuming that the sensitivity of the system is fixed,

a good compromise between accuracy, computation, and stability is to use one-pole-one-zero filters for the crossover, and two cascaded one-pole-one-zero filters for the high-pass or low-pass filters.

One case to take into account is that, after the iterated attenuation of the partials has taken place, if the input signal suddenly moves into an area of the spectrum that is currently being attenuated, then it will be immediately filtered out, failing to be detected. To prevent this, the algorithm for the lowest partial can be improved by adding a positive constant to the RMS output of the high spectrum of the crossover. The constant needs to be small enough to be negligible in the detection process of the lowest partial, but big enough to force the system out of its attractor. By adding the constant, if the input suddenly disappears or moves to an attenuated region, the system can bring the cut-off of the low-pass up to $Nyquist$ so that all partials can subsequently pass and be processed. Similarly, the highest partial algorithm will need the constant to be added in the RMS of the low spectrum of the crossover. The constant 0.00001 works as expected.

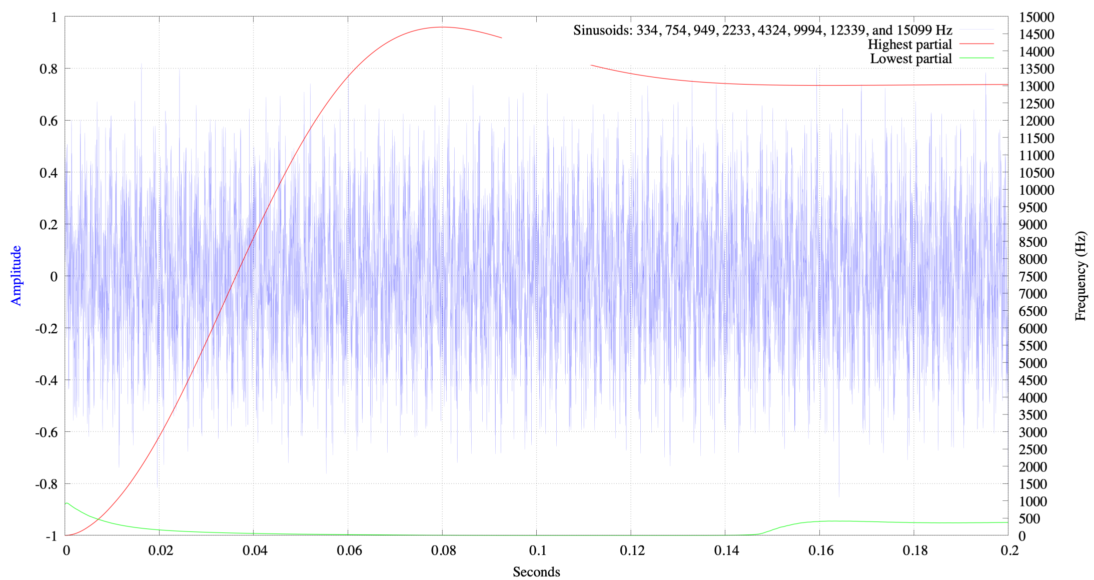

Figure 3.21: Lowest and highest partial response with a 15 Hz analysis window and eight sinusoids with randomly chosen frequencies as test signal.

Spectral spread

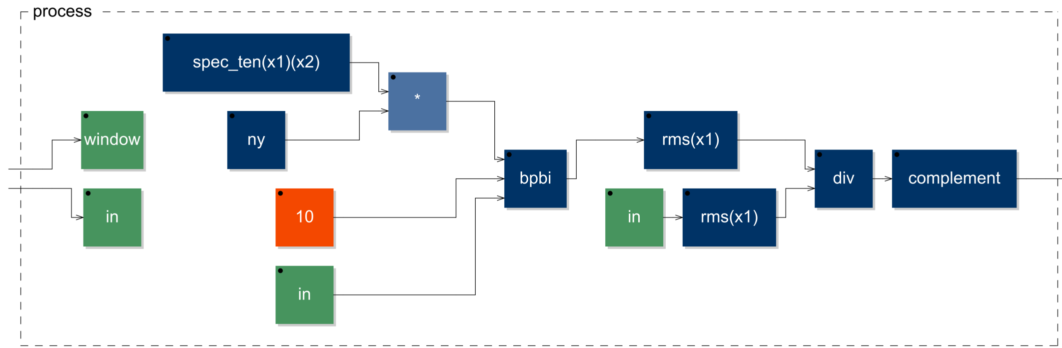

Figure 3.22: Block diagram of the spectral spread algorithm. The green blocks are external signals. "In" is the analysed input, and "window" is the analysis rate. The measurement of spectral tendency of the input, in Hz, drives the cut-off of a biquadratic band-pass with a Q of 10. The RMS of the output of the filter is divided by the RMS of the input signal. Finally, the output is spectral spread is calculated as the complement of the ratio.

Measurement of spectral spread can be realised by combining the spectral tendency algorithm with a band-pass filter. The spectral tendency is used to pilot the centre frequency of the band-pass filter so that it is adaptively shifted towards the spectral area of maximum magnitude. The Q of the filter has a fixed value of 10. The band-pass filter is a biquadratic section:

$$y_{bp}[n] = a_{0bp}[n]x[n]+a_{1bp}[n]x[n-1]+a_{2bp}[n]x[n-2]-b_{1bp}[n]y_{bp}[n-1]-b_{2bp}[n]y_{bp}[n-2] \quad . \tag{3.33}$$

The coefficients for a desired cut-off frequency and Q are calculated using Robert Bristow-Johnson's formulas.[5] With $\alpha[n]$ and $\nu[n]$ defined as

$$\begin{aligned} \nu[n] &= 1 + \alpha_{bp}[n] \\ \alpha_{bp}[n] &= \sin(\omega_{bp}[n]/(2Q[n])) \quad , \\ \omega_{bp}[n] &= 2\pi f_{bp}[n]/samplerate \end{aligned} \tag{3.34}$$

the coefficients are

[5] https://www.w3.org/2011/audio/audio-eq-cookbook.html. Accessed on the 29th of August 2019.

$$
\begin{aligned}
a_{0bp}[n] &= \alpha_{bp}[n]/\nu[n] \\
a_{1bp}[n] &= 0 \\
a_{2bp}[n] &= -\alpha_{bp}[n]/\nu[n] \\
b_{1bp}[n] &= -2\cos(\omega_{bp}[n])/\nu[n] \\
b_{2bp}[n] &= (1-\alpha_{bp}[n])/\nu[n]
\end{aligned} \quad . \tag{3.35}
$$

If $y_{bal}[n]$ is the spectral tendency of the input, and $Q = 10$, we can rewrite (3.34) as

$$
\begin{aligned}
\nu[n] &= 1+\alpha_{bp}[n] \\
\alpha_{bp}[n] &= \sin(\\
omega_{bp}[n]/20) \\
\omega_{bp}[n] &= 2\pi y_{bal}[n]/samplerate
\end{aligned} \quad . \tag{3.36}
$$

If $RMS_{bp}[n]$ and $RMS_x[n]$ are, respectively, the RMS of the band-passed input and the RMS of the input, then the spectral spread $y_{ss}[n]$ is simply

$$
y_{ss}[n] = 1 - RMS_{bp}[n]/RMS_x[n] \quad . \tag{3.37}
$$

The responsiveness of the spectral tendency and window of the RMS units can be adjusted to change the overall responsiveness of the system.

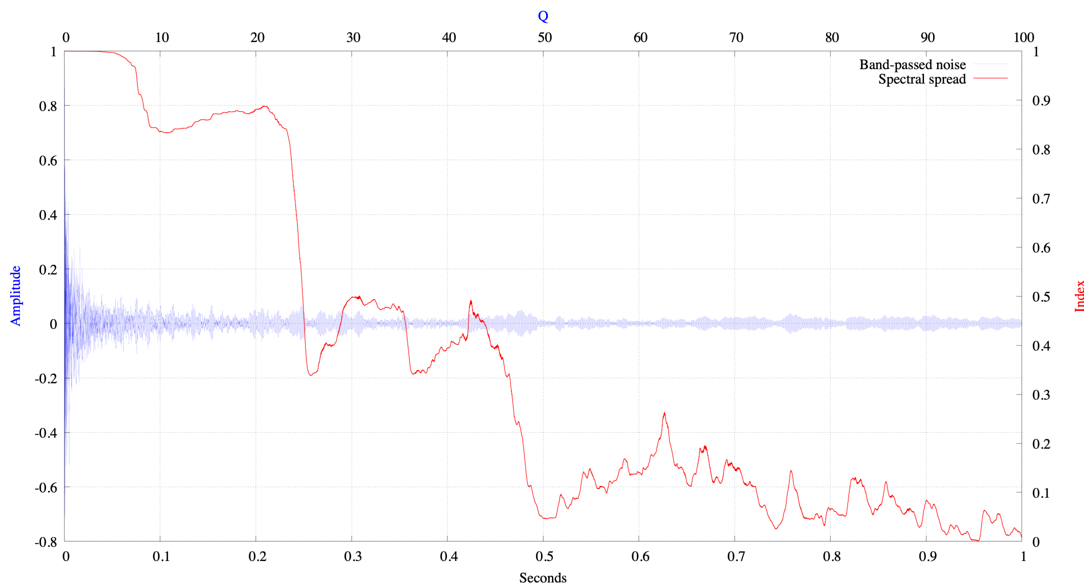

Figure 3.23: Spectral spread response with an analysis window of 5 Hz and noise band-passed with a progressively increasing Q as test signal.

3.3.2 High-level information processing

In music information retrieval, we have several techniques for high-level symbolic and rhythmic descriptions. These can be useful for conventional kinds of music, although sound-event-based characteristics can be more suitable in other cases, as in the works discussed in this article and many other electronic music works. Within the framework of human interaction with digital music collections, a review of techniques for measurements of musical complexity, both in the symbolic and audio domain, can be found in [Streich et al., 2006]. The two high-level algorithms in this section provide information on the dynamicity and complexity of sound events. The notion of dynamicity is related to the number of variations in the characteristics of a sound event at the formal time scale, that is, variations below 15 Hz. The complexity algorithm provides a measurement of the edge of chaos and the nontrivial interactions within that region to determine the complexity index of a sound event. Both algorithms use some of the low-level analysis presented earlier combined with average absolute deviation, concepts from recurrence quantification analysis, and the theory of dynamical and complex systems.

Dynamicity

Figure 3.24: Block diagram of the dynamicity algorithm. The green blocks are external signals. There are three main processing paths to calculate the dynamicity based on RMS, spectral tendency, and noisiness. Each path maps the input on a logarithmic scale, differentiates over a 1/15-second period, and averages the magnitude of the variations. The three paths are then nonlinearly combined through a hyperbolic tangent function.

The dynamicity algorithm focuses on the combined action of variations in low-level characteristics. We know that approximately 15 fluctuations per second is the limit above which individual sound events tend to be merged into a single stream [Vassilakis and Kendall, 2010]. Considering that no formal variations can be perceived at such a rate, 15 Hz is also the area that separates a highly dynamical behaviour – concerning a specific sonic characteristic – from a highly static one. On the other hand, the brain capacity of relating sound events separated by long periods decreases considerably as the period reaches four-to-six seconds or more [Demany and Semal, 2008]. According to Snyder et al., the maximum short-term memory (STM) capacity for sound events to be grouped as a whole in the present is limited to about eight seconds. This is likely to be the largest possible period for sound events to be perceived as related unless the listener performs some action of rehearsing or remembering [Snyder and Snyder, 2000].

The temporal frame for this algorithm comprises fluctuations from about 0.125 Hz, which is a period of eight seconds, to about 15 Hz. The algorithm processes three low-level descriptors and nonlinearly combines them to calculate an index. The three descriptors are amplitude tendency (RMS), spectral tendency, and noisiness. Due to the logarithmic perception of amplitudes, the RMS is mapped onto a logarithmic scale. The outputs of the spectral tendency and noisiness algorithms are already mapped logarithmically. By differentiating over a period of 1/15 seconds, we obtain a positive-sine-shaped frequency response with zeroes at 0 Hz and 15 Hz. This frequency response is suitable for the frequencies considered for the analysis, as we expect fluctuations to peak when occurring in the mid-area of that range, and to decrease as they approach the lower and upper limits. The responsiveness of the low-level information signals themselves is 15 Hz; thus, fluctuations occurring at a higher frequency are averaged and attenuated at an earlier stage.

The low-level information signals are differentiated individually. Even in this case, the direction of the variation is not relevant, and only the magnitude of the output of the differentiators is essential. It is reasonable to assume that the contributions of the individual magnitudes are non-additive when combined, i.e., they sum up in a nonlinear way. If one of the three

low-level characteristics shows a high variation rate while the other ones show a low variation rate, it is counterintuitive to weigh the overall dynamicity of a sound event at only 1/3 of the maximum. Individual characteristics can significantly contribute to the overall dynamicity of a sound event. To model the nonlinear behaviour, the magnitudes of the fluctuations are summed and then processed through a hyperbolic tangent function. Lastly, a one-pole low-pass filter with different periods can average the output for arbitrary responses in the overall index.

Complexity

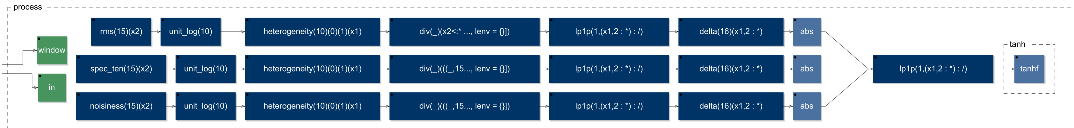

Figure 3.25: Block diagram of the complexity algorithm. The green blocks are external signals. There are three main processing paths based on RMS, spectral tendency, and noisiness. Each path maps the inputs logarithmically and calculates the heterogeneity. The result is normalised based on the state space size of the input in each path, averaged, and differentiated over a 16-second period. The magnitudes of the variations are summed, averaged, and nonlinearly combined through a hyperbolic tangent function.

As we have seen earlier, the most crucial characteristic of a complex system is the edge of chaos and the nonlinear interplay between order and disorder. We can imagine the state space of a system distributed over a horizontal line. At one extreme of the line, we have entirely predictable patterns and behaviours, and at the other, entirely irregular structures and chaotic behaviours. Using terms from information theory, it is also possible to link redundancy and entropy to these respective extremes. Complexity is maximal somewhere in between the extremes, where a new state space emerges from the nonlinear interaction of forces pulling towards the edges. This scenario technically represents dynamical networks containing mutating mechanisms of positive and negative feedback loops: the struggle between these two interfaces, one tending towards equilibrium, the other tending towards instability, allows complexity to emerge.

The measurement of complexity also focuses on the processing of three low-level information signals. Similar to the dynamicity algorithm, the complexity analysis processes RMS, spectral tendency, and noisiness, on a logarithmic scale, at a rate of 15 Hz. Additional low-level information signals can be processed for more-accurate results, but for simplicity and efficiency I will focus on only three key characteristics.

The concept of recurrence for dynamical systems was initially introduced by Poincaré in 1890. (Poincaré 1890) Today, recurrence quantification analysis (RQA) and recurrence plots are common tools that enable the investigation of patterns and high-dimensional dynamics in complex systems by looking at simple two-dimensional plots [Mocenni et al., 2011]. Rimoldi and Manzolli have also applied RQA-based studies in music, as in the analysis of Agostino Di Scipio's works [Rimoldi and Manzolli, 2016]. The complexity algorithm that I present here also implements a form of RQA. The state space of the low-level information signals, which are

$\{\in \mathbb{R} \in [0;1]\}$, is divided into ten equally-spaced sub-spaces.

If m is the number of equally-spaced sub-spaces through which the analysed signal can orbit, the lower and upper boundaries of each sub-space s_i can be calculated as follows:

$$\begin{aligned} l_{s_i} &= L + (H - L)(i - 1)/m \\ h_{s_i} &= L + (H - L)i/m \end{aligned} \quad . \tag{3.38}$$

L and H are the minimum and maximum values that the input signal can reach, respectively 0 and 1 in this case, and i is an integer number $\in [1; m]$. With $m = 10$, (3.38) can then be simplified as:

$$\begin{aligned} l_{s_i} &= (i - 1)/10 \\ h_{s_i} &= i/10 \end{aligned} \quad . \tag{3.39}$$

The state of each low-level signal is analysed through a ten-channel unit to detect the occurrences in each sub-space. The output of each occurrence detector is connected to a leaky integrator. If a channel is activated, that is if the signal value is within that sub-space, then the integrator acts as an accumulator and tracks the number of occurrences of the signal over time. When the channel is off, hence when the signal value is within a different sub-space, then the leaky integrator discharges after a certain period. For an information signal z[n], the occurrence detection can be written as:

$$y_{s_i}[n] = \begin{cases} 1, & \text{if } l_{s_i} < z[n] \leq h_{s_i} \\ 0, & \text{otherwise} \end{cases} \quad . \tag{3.40}$$

The output of the leaky integrator for each channel is then:

$$y_{leaky_i}[n] = y_{s_i}[n] + \tau y_{leaky_i}[n-1] \quad , \tag{3.41}$$

recalling that τ can be calculated using (3.32). Subsequently, the output of the integrators is processed through a peak-holder:

$$y_{ph_i}[n] = \begin{cases} y_{ph_i}[n-1], & \text{if } y_{leaky_i}[n] < y_{ph_i}[n-1] \quad \wedge \quad C_i[n] < D \\ y_{leaky_i}[n], & \text{otherwise} \end{cases} \tag{3.42}$$

with

$$C_i[n] = \begin{cases} 0, & \text{if } y_{leaky_i}[n] \geq y_{ph_i}[n-1] \quad \vee \quad C_i[n] \geq D \\ C_i[n-1] + 1[n-1], & \text{otherwise} \end{cases} \quad . \tag{3.43}$$

$D = t \cdot samplerate$ is the holding period in samples, and $C_i[n]$ is a counter that is reset when a new peak is detected, or when the holding time is out. The peak-holder output is then smoothed out through one-pole low-pass filters. This configuration provides the algorithm with a temporary memory on the states of the low-level signals to trace their recurrences over the ten states.

The notions of heterogeneity and homogeneity can help to describe the order or disorder degree of a signal. A heterogeneous state space suggests that an ordered behaviour or small pattern for the signal circulates through a limited number of states. Conversely, a homogeneous state space characterises chaotic behaviours because of the diversity of the states, which are distributed more evenly throughout the space. The processing for a heterogeneity index, for each low-level signal, is carried out through normalised average absolute deviation (AAD) of the outputs of the integrators.

With a set of signals $\{y_{ph_1}[n], y_{ph_2}[n], ..., y_{ph_i}[n]\}$ indicating the recurrence of the input in any of the states, and $\overline{y}_{ph}[n]$ being the average of the set, the equation for AAD is

$$\frac{1}{m}\sum_{i=1}^{m} |y_{ph_i}[n] - \overline{y}_{ph}[n]| \quad . \tag{3.44}$$

With $\{y_{ph} \in \mathbb{R} : y_{ph} \geq 0\}$, the formula can be modified slightly to obtain a normalised output in the range $[0; 1]$; (3.44) then becomes:

$$\frac{1}{2(m-1)\overline{y}_{ph}[n]} \sum_{i=1}^{m} |y_{ph_i}[n] - \overline{y}_{ph}[n]| \quad . \tag{3.45}$$

If the signal is in only one channel, the index is one; if it uniformly distributes in all the channels, it is zero. The central feature to detect is the variation between order and disorder, or heterogeneity and homogeneity. The AAD output is thus averaged using one-pole low-pass filters and differentiated to identify shifts. Following the same non-additive principle from the dynamicity algorithm, the magnitudes of the shifts between heterogeneity and homogeneity of the three low-level signals are summed together and then processed through a hyperbolic tangent function.

Music perception can inform the choice of some parameters in the algorithm as well as an improved design for better measurements. On the other hand, CASes can model some of the mechanisms through which we perceive and evaluate music. Order and disorder must play a significant role for they determine recurring structures (patterns, predictability) and evolving forms (surprise, irregularity). Based on the studies on auditory scene analysis [Bregman, 1994], it is also sensible to think that music is processed starting from low-level characteristics such as loudness, brightness, and noisiness. One hypothesis, which has inspired the complexity algorithm presented here, is that music perception follows a *features-frames* representation. The y-axis depicts a set of low-level features; the x-axis depicts the frames that contain the contour of the features. According to studies on timbre, there is an interaction between STM and long-term memory when processing sounds [Siedenburg and McAdams, 2017]. It is possible that the comparison of successive frames gives the evaluation of the degree of redundancy in long sound events. Namely, at least two frames need to contain similar enough contours to determine a pattern, hence a redundant global state. This corresponds to differentiating to detect variations. We know that musical STM has a time capacity of about eight seconds, although its length can vary based on several factors [Snyder and Snyder, 2000]. Musical STM is then dependent on the context, and the frames are likely to have a variable length depending on the contours that they contain. Besides, the concept of *state space dimension* is fundamental within this framework. A system can exhibit complex behaviours despite circulating a limited state space. Likewise, the behaviours of the low-level signals can exhibit edge-of-chaos behaviours even if operating through a limited set of values. Variations in the dimensions of the state space are also a key aspect in the nonlinearity of the interplay between order and disorder.

In this algorithm, the analysis window for the detection of variations in the heterogeneity index is 16 seconds or two successive eight-second frames. Both the averaging period of the low-pass filters and the period of the differentiators is 16 seconds. The points that I discussed above

suggest that the size of the channels for the AAD index to be set according to the maximum and minimum values reached by the low-level signals. Hence, the number of channels is the same, though their ranges can be smaller or larger to take into account different dimensions in the state space. Variations in this dimension should also contribute to the complexity index. The peaks determining the channel size also rely on a temporary memory by being held for two frames unless a new upper or lower peak is detected. In the same way, the discharge of the integrators following the channel on/off state starts after 16 seconds, and the recurrence state is held for two frames before it resets to a neutral position. The frame length, instead, varies according to the dynamicity index as it is a meaningful signal in terms of redundancy and entropy. Both extreme situations of very high entropy or very high redundancy suggest that shorter analysis windows may be enough to detect a coherent pattern, for they both represent a behaviour with no formal developments and no tendency towards evolutions. Ultimately, an interaction matrix could establish direct low-level interactions between the source signals, while the heterogeneity index and the state space dimension could nonlinearly affect each other.

3.4 Remarks

This chapter presented the fundamental techniques developed in this research for the realisation of systems that determine the structure and the network of relationships of their components trough evolvability and autopoiesis, which we have discussed in detail in section 2.4 Autopoiesis.

We have seen the importance of the role of information in adaptation concerning the strict relationship that is established between systems and their context when following notions of self-referentiality, self-observation, and self-regulation. Furthermore, the idea of information is analysed following the historical developments coming both from engineering and philosophy, underlining the differences between approaches that consider information from a quantitative and static view, without considering the particular circumstances taking place in its environment, from perspectives that see information as a dynamical and morphing entity that is strongly related to an observer. Information is then regarded as a fundamentally emergent process that is affected by the very act of detection itself, suggesting an analogy with some of the insights proposed in the theories of radical constructivism.

We have discussed the primary principles in the design of CASes through networks of adaptive agents, providing clear identification of the infrastructures that are responsible for the processes of interpretation and modification of context and the relationships that develop between the two processes. The implementation of agents with the ability to dynamically determine their infrastructures is a key development that, together with the ability of a system as a whole to rearrange the relationships among agents, allows for the continuous generation of adaptive variants that evolve differently. These are organisms that actively take part in their design and express a paradigm shift, from "composing the interactions" [Di Scipio, 2003] to *self-composing*

interactions, with significant impact on the notion of autonomy that is connected to these systems.

The very origin of the idea of distributed adaptation is strongly related to the notion of autonomy. The early investigations with feedback systems, which eventually resulted in this research, date back to 2006. After discovering that some elementary DSP modules in a feedback loop, without human intervention, produced some musically convincing – even if limited – variations, there was the decision to explore feedback systems solely. The initial experimentations and applications in live performance implemented several essential feedback networks that would be interconnected and modulated by a performer on-the-fly. Fundamentally, these sub-systems were used as a set of complex oscillators that could respond to external stimuli, both in the form of energy, as audio signals entering the sub-systems, or information, given by the performer who would modulate the DSP parameters based on musical decisions. Discovering the work of Di Scipio, a few years later, was a critical step forward as it introduced the realm of adaptation and self-regulation in computer music, which became the new research focus. The live performance practice with adaptive and self-regulating systems mainly concerned the variation of the relationships between the agents and variations on the mapping functions that mediated the feedback relationships within the agents. Distributed adaptation turned the variations that were first controlled by the performer into autonomous and emergent features of the systems, extending the principle to more aspects such as how information is processed, which information signal is assigned to the state variables, and the multi-modality of the agents.

Given the importance that information has in the design of adaptive systems, the chapter concludes with a section on original techniques for the processing of low-level and high-level information. These techniques are a set of heuristic algorithms in the time domain, providing CPU-efficient alternatives to the standard techniques found in the literature. These algorithms show the importance of adaptation in information processing, and that hard-coded designs such as the weighted average in the frequency domain for the measurement of spectral tendency, or the ZCR for the measurement of noisiness, can be improved or replaced by self-regulating mechanisms to enhance efficiency and accuracy, as it is shown in section 3.3.1 Spectral tendency and section 3.3.1 Noisiness. Furthermore, it is shown that adaptive mechanisms can be valuable to model complex perceptual aspects such as the detection of redundancy in different sonic contexts, as it is discussed in section 3.3.2 Complexity.

Finally, the chapter describes the three most advanced case studies of the portfolio, giving musical examples of systems that compose and perform themselves.

Chapter 4

Audio processing techniques and DSP implementation

The principle assumption made in Artificial Life is that the 'logical form' of an organism can be separated from its material basis of construction, and that 'aliveness' will be found to be a property of the former, not of the latter.

Christopher Langton

This chapter is dedicated to the technical aspects of digital audio signal processing and the programming languages through which algorithms are implemented. The first part of the chapter describes some of the conventional techniques for audio transformation and also introduces alternative designs to the standard approaches, for instance, in granular processing and nonlinear transfer functions.

The second part of the chapter discusses several techniques to guarantee stability in recursive systems, hence systems with feedback mechanisms that may result in exponential growths exceeding the boundaries of audio-domain representations. These techniques include soft-clipping through digital saturators, and adaptive processing for the self-regulation of the amplitude of signals. The musical implications of the use of different techniques for stability processing are exposed.

Finally, the chapter introduces the two primary developing environments, Pure Data and Faust, showing the strengths and weaknesses of the programming languages as well as some of their main functions.

4.1 Sound transformations

4.1.1 Granulation

In 1946 and 1947, Dennis Gabor published two crucial articles that would lay the theories and foundations of granular processing [Gabor, 1946, Gabor, 1947]. Gabor developed a conceptual framework for the analysis of audio signals linking quantum theory, Heisenberg's uncertainty principle, and Mach's analysis of sensation to formulate a key relationship between time and frequency domain in the sonic realm [Di Scipio, 2016, Heisenberg, 1985, Mach, 1914]. The formula from his publication from 1946,

$$\Delta t \Delta f \approx 1 \quad , \tag{4.1}$$

represents the interpretation of Heisenberg's uncertainty principle for acoustic signals. Δt and Δf are the uncertainties for the temporal and frequency locations of an oscillation, linked by multiplication and equality to a constant to express their inverse relationship, that is, that reducing the uncertainty in one quantity increases it in the other and vice versa.

In music and DSP, granular processing is a technique for the generation and transformation of sounds through fragmentation and reorganisation. Signals are decomposed into sonic grains, each of them having independent features such as pitch, amplitude, and duration (normally below 0.05 seconds), and are then rearranged following different criteria to obtain a new stream with original global characteristics [Roads, 1988, Truax, 1988].

Non-real-time techniques for granular processing are typically based on wavetables whose portions can be read at different speeds using some interpolation scheme, and at different positions depending on the array index start-point. Real-time applications of granular processing, on the other hand, are usually based on fractional delay lines (FDL), that also allow for the reproduction of sound at different speeds using linear or nonlinear interpolation to transition among successive samples. Wavetables and FDL have some characteristics in common: they are both based on buffers and interpolation to achieve pitch-shifting, although the working mechanism is somewhat different.

The buffer of wavetables is usually fixed, whereas the buffer of FDL is circular, that is, it is continuously updated with new data, making it suitable for live applications. Wavetables are played back by going through the indexes of the buffer to read their content. It is then the index increment (or decrement when reading audio in reverse) that determines the pitch of a sound. If $i[n]$ is the buffer index position, then the pitch transposition factor $t[n]$ is given by

$$t[n] = i[n] - i[n-1] \quad . \tag{4.2}$$

FDL, instead, can be read at different rates by modulating the amount of delay at which they reproduce the content of their buffer. Specifically, if $d[n]$ is the delay in samples, then the pitch transposition $t[n]$ is given by

$$t[n] = 1 - (d[n] - d[n-1]) \quad . \tag{4.3}$$

with $\{d[n] \in [0; L]\}$, where L is the length of the FDL in samples. Accordingly, the delay amount is what determines the reading position in the buffer of the FDL. An FDL is essentially a model for a closed-loop rotating tape with a writing head and a reading head. We can imagine the tape rotating at a constant speed, the writing head fixed at some point in the tape, and the reading head being able to jump at any position in the tape in virtually no time. The delay determines the distance between the two heads.

Gabor describes the design for a mechanical machine capable of frequency compression and expansion in his publication from 1946. Years later, in the 1960s, Springer developed an analogue device, the Tempophon, based on the principles described by Gabor, which implemented a set of rotating reading tape heads for the modulation of, independently, tempo or pitch. A revolutionary aspect of the granular theories is indeed the fact that tempo and pitch are no longer linked and that they can be altered individually. The Tempophon worked with six reading heads arranged in a circle so that they could rotate at different speeds and read portions of the tape, hence grains, at different pitches. The tape could change in speed too, and this allowed to change the tempo independently provided that the speed of the reading heads relative to the movement of the tape remained constant. Specific configurations of digital granular processing can then implement live pitch shifting and time stretching.

The mechanical implementation produced a smooth transition between successive reading heads. In the digital case, one FDL is potentially enough to achieve a continuous stream as it can move in virtually no time throughout the tape. The digital implementation, though, requires a design that avoids discontinuities, as they are likely to occur when transitioning between grains. The windowing of grains is also a key element in granular processing. Windowing means to ring-modulate two signals, so the window type and its spectral content play a significant role in the outcome. Windowing also results in a non-continuous stream as there are regions where the

amplitude decreases drastically. The problem is resolved by using two windowed FDL which are out of phase by 180 degrees [Puckette, 2007]. The overlapping grains then add up to a continuous or quasi-continuous stream depending on the windowing function that is used. One design that shows particularly convincing results is the implementation of non-homogeneous overlap-add-to-1 linked to a non-homogenous pitch modulation within each grain, although this technique has not been explored in a piece yet.

The regularity or non-regularity of how the pitch, the duration, and the buffer position is selected for each grain defines synchronous and asynchronous granular processing. Randomness and stochasticity are often used to render asynchronous granular processing. In my investigations, the focus is on having parameters such as pitch, duration and grain position determined by the sonic context of the granulator itself, so that a self-modulation leads to a nonlinear iterated configuration for potential chaotic behaviours and asynchronous granular processing characteristics.

Alternative techniques for the handling of discontinuities in granular processing are possible; zero-crossing (ZC) detection can be deployed to achieve a smooth transition between grains. In Pure Data, I had already done some experiments with this technique during the past few years using audio samples and static (non-circular) buffers, though Pure Data had some limitations for which a working implementation was not efficient enough for live applications. Faust, on the other hand, offered compelling tools and the possibility to do sample-wise operations with very efficient computational costs. For the early implementations, I used Faust's read-write tables as circular buffers as the read and write indexes can be piloted through signals, making the synchronisation among several buffers easy.

The main idea behind a ZC granulator is that grains start and end at a ZC. If we have a fixed audio source on a wavetable, then we can scan the table and store all the ZC positions in an array of as many elements as the ZC occurrences so that they can be recalled at a later time. However, signals can be irregular, and so can be ZC occurrences, as a consequence. If we want both the start and end of grains to be at a ZC, it means that the duration of each grain is variable and dependent on the signal itself. The fundamental condition for a sequence of grains of duration D without discontinuities is that each successive grain should be triggered after the time D has passed, at the first ZC occurrence. It means that the output of the granulator must be continuously inspected to detect a ZC, and such information must be sent back into the section that generates each grain. It is the minimum requirement for a continuous stream without discontinuities, although harmonics, noise, and aliasing may be introduced. It is important to underline that the technique presented here is not intended as a replacement for the well-established and standard techniques for granular processing. Instead, it should be considered as an investigation of the new sonic possibilities offered by a design that does not contain the intrinsic artefacts of windowing [Cavaliere and Piccialli, 1997].

One crucial aspect is to have consistency among the sign of the derivatives at the end and

beginning of consecutive grains. The first derivative gives the slope of a signal. Hence:

$$y_{up}[n] = \begin{cases} 1, & \text{if } x[n] - x[n-1] > 0 \\ 0, & \text{otherwise} \end{cases}. \tag{4.4}$$

$$y_{down}[n] = \begin{cases} 1, & \text{if } x[n] - x[n-1] < 0 \\ 0, & \text{otherwise} \end{cases}. \tag{4.5}$$

The ZC positions of the input that we want to process can then be stored into two different arrays: one for the ZC occurring in ascending signals, the other for the ZC in descending signals. Similarly, the output of the granulator can be analysed for direction and ZC so that the position of the next grain is selected from the corresponding set of ZC indexes. If both the end and beginning of grains are at a ZC position and signals have the same slope, then they might be at a very close value which may result in the repetition of two samples and the generation of noise. By skipping one sample at the beginning of each grain, there is a better continuity and smoothness in the resulting signal. Practically, this is implemented by subtracting 1 from the grain positions to move the reading pointer ahead of one step. Furthermore, particularly when transitioning between grains that have different slopes, the one-sample correction may not be enough. For a more general position correction, the position start of the next grain can be determined by the ratio between the derivatives at the end and the beginning of consecutive grains. In this case, a third delay line is necessary to store the derivative of the input signal to be recalled at a later time.

If the input signal is not fixed and we are using a circular buffer (CB) to update it continuously, then we can use two CB of the same size to store the ZC indexes of ascending and descending signals. We can sample-and-hold (SAH) the indexes at which a ZC is detected so that any recalled position in the ZC buffers corresponds to a ZC position in the input buffer. A SAH unit has two inputs: $c[n]$, a Boolean value, controls the sampling process; $x[n]$ is the signal to be sampled:

$$y_{SAH}[n] = \begin{cases} x[n], & \text{if } c[n] = 1 \\ y_{SAH}[n-1], & \text{if } c[n] = 0 \end{cases}. \tag{4.6}$$

If the size of the CB is L, then the writing index, $i[n]$, cycles through integers from 0 to $L-1$. $i[n]$ is the signal that we want to store in the ZC CB, whereas the conditions to trigger the SAH in ascending and descending signals are, respectively:

$$\begin{aligned} & y_{zc}[n] \wedge y_{up}[n] \\ & y_{zc}[n] \wedge y_{down}[n] \end{aligned} \quad . \tag{4.7}$$

A ZC is detected using (3.21). In Faust, tables do not implement fractional indexes and are not ideal for pitch transposition, but we can use FDL for live granular processing with pitch transposition. In the case of tables, recalling a ZC index is rather straightforward, and it is enough to read the input buffer at that position. With delay lines, since we move around the buffer by setting a delay relative to the position of the writing index, a few more steps are necessary.

In delay lines of length L samples, too, the writing index of the CB, $i[n]$, cycles through integers from 0 to $L-1$ [Rocchesso, 2003]. $i[n]$ is what we sample-and-hold when the ZC is detected; it represents the time after which, relative to the beginning of the process, a ZC has occurred. It is essentially a time offset, and we can recall a ZC that has occurred at previous time P by setting the delay to $i[n]-P$. Of course, if P is greater than the current index i[n], then the negative value should be wrapped around the range $[0; L]$. A general wrapping function for ranges with min and max boundaries has the following form:

$$\begin{aligned} & y_{wrap}[n] = y_{decimal}[n](max - min) + min \\ & y_{decimal}[n] = (x[n] - min)/(max - min) - \lfloor (x[n] - min)/(max - min) \rfloor \end{aligned} \quad . \tag{4.8}$$

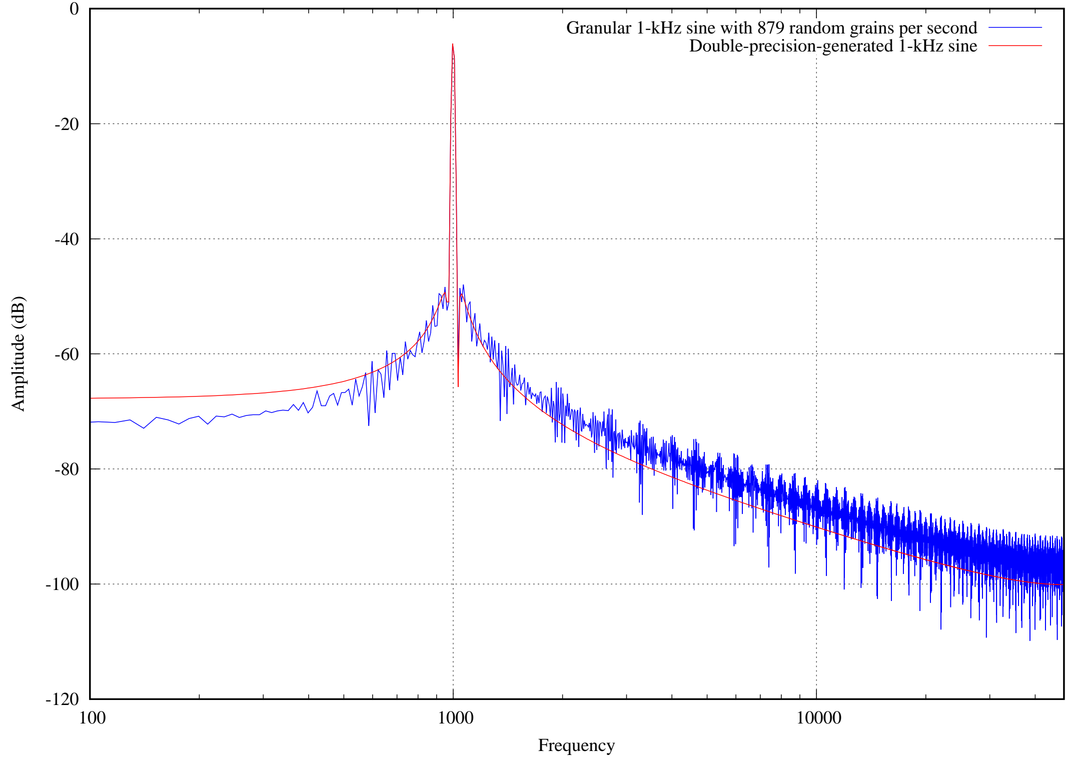

Figure 4.1: Reconstruction of a sinewave at 1 kHz through 879 grains per second and random grain positions. Comparison of spectra between original and reconstructed sinewaves.

By simply reorganising the input signal rearranging the grains at different ZC positions in the buffer, we have a granulator without transposition. At this point, the pitch transposition of each grain can be implemented as a delay shift starting from the selected ZC position. If the desired grain rate is $r[n]$, assuming it positive, which determines the grain duration $1/r[n]$, then the delay shift $y_{shift}[n]$ for a given positive pitch factor $pitch[n]$ can be calculated as follows:

$$y_{shift}[n] = (1 - pitch[n])/r[n]y_{line}[n]samplerate \quad . \tag{4.9}$$

$y_{line}[n]$ is a signal that grows from 0 to 1 in $1/r[n]$ seconds. A line can be implemented as follows:

$$y_{line}[n] = r[n]/samplerate + y_{line}[n-1] \quad . \tag{4.10}$$

Similarly, a time transposition can be achieved by consistently offsetting the grain positions relative to the motion of the reading head. In this case, we have that for a desired time

transposition factor $time[n]$, the position of each grain is given by a phasor (as implemented in (4.8) and (4.10)) whose rate is $(1 - time[n])samplerate/L$, and whose output is mapped over the length of the delay line, L.

Particularly, if $time[n] = 0$, we have that the granulator is looping – without discontinuities – a specific region of the buffer whose size is, approximately, 1/r[n]. Hence, depending on its parameters settings, the granulator can transition between click-free looping, wavetable oscillation with complex waveforms, and granular synchronous or asynchronous processing. Furthermore, different regions of the buffer can be explored by offsetting the position and by wrapping around in the range $[0; L]$.

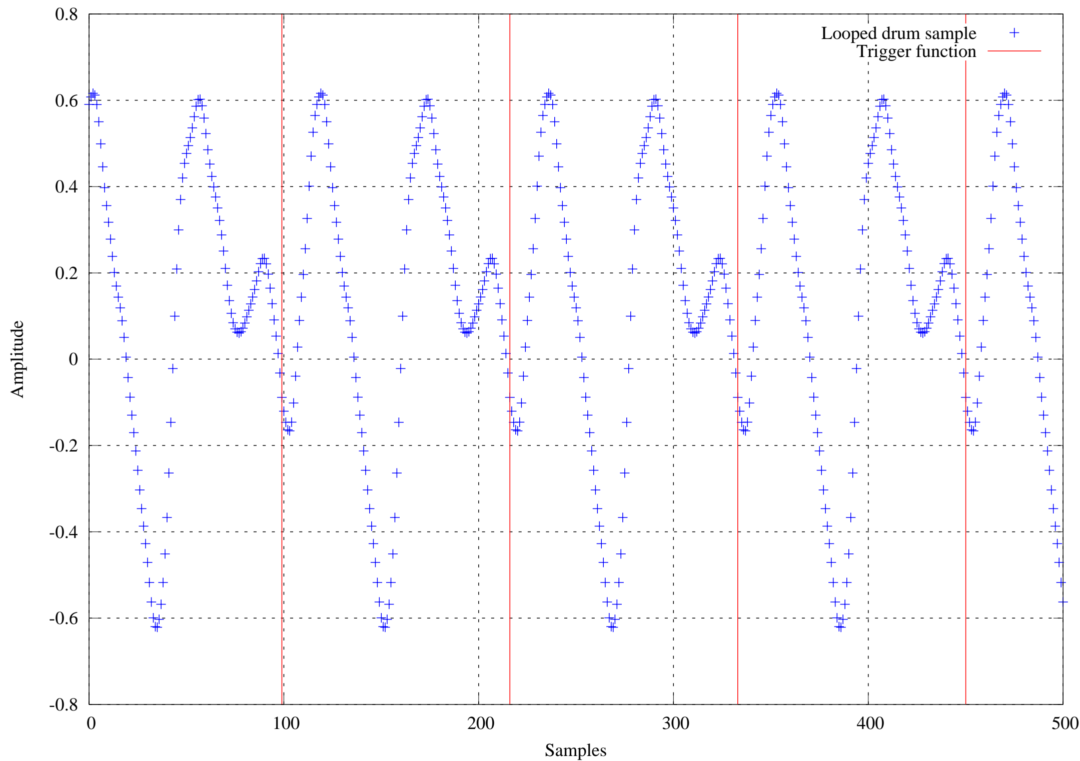

Figure 4.2: 1000 grains per second looping of a portion of a drum sample. Visualisation of grains start.

For a constant pitch shift during playback, $r[n]$ and $pitch[n]$ can be sampled-and-held at the start of each grain. The noisy textures generated by this algorithm are particularly satisfying, as the windowless design has a distinct sharpness and fullness even at lower grain rates with no overlapping grains.

4.1.2 Sampling

Sampling with FDL is very similar to granular processing with FDL. The primary parameters implemented in the sampling modules used in the performance projects of this thesis are pitch, frame size, and buffer position. Pitch and frame size can be calculated using (4.9). The area of the buffer to be read, on the other hand, can be obtained using an offset, that is, adding a

constant to the shift amount that modulates the FDL.

These modules, in some case studies, do not implement windowing and frames overlapping, whereas in other cases they do. Sampling using consecutive frames without windowing may result in discontinuities and audible clicks. As shown in the section above, ZC detection is a solution to avoid discontinuities, although, for these applications, in particular, it was not necessary as mainly very long frames were used and discontinuities were not noticeable.

4.1.3 Nonlinear transfer functions

Nonlinear transfer functions (NTF) are mathematical functions that do not follow the superposition principle discussed in subsection 2.1.1 Properties of feedback mechanisms. These functions are often used for digital distortion, which is also called *waveshaping*. The saturators discussed in subsection 4.2.1 Bounded saturators are examples of NTF.

Alternatively, transfer functions can be implemented using table-lookup techniques for CPU optimisation, that is, by storing the values of a function in an array [Le Brun, 1979]. The size of the array, N, determines the resolution of the stored function. The table positions, from 0 to $N-1$, represent the function response through minimum and maximum values. The input signal, assumed to be in the range $[-1; 1]$, is then linearly mapped onto the table range $[0; N-1]$. The values between consecutive positions, which are integer values, are obtained via interpolation.

The NTF technique implemented for the case studies presented here follows a recursive design: the output of the transfer function determines the shape of the transfer function itself, which in turn determines the output of the system. FDLs allow for signals to be written cyclically on buffers and are thus indicated for this kind of implementation. The (potentially interpolated) output of a FDL $y_{fdl}[n]$ of size S seconds that has an input $x[n]$ and delay in seconds $d_s[n]$ is given by:

$$y_{fdl}[n] = x[n - d_s[n] \cdot samplerate] \quad , \tag{4.11}$$

with $0 \leq d_s[n] \leq S$. In the recursive case, $x[n]$ is replaced by the output of the delay line itself and $1/samplerate \leq d_s[n] \leq S$ as there is at least a one-sample delay in digital feedback loops. If the delay line has a length S in seconds, then we can linearly map an external signal $x_{nltf}[n]$ (in the range $[-1; 1]$) over the entire buffer S of a FDL to modulate its delay and output the transfer function. We can call the resulting signal $d_{nltf}[n]$. Although, it is necessary to offset the delay to make the reading pointer static concerning the motion of the writing pointer. With the offset, the reading pointer will refer consistently to the same region of the buffer when

the input does not change. Notably, the offset must be a phasor that cycles from 0 to S at a rate of $1/S$. A phasor ($y_{phasor}[n]$) can be implemented as a line function followed by a decimal function, which we have seen in (4.8) and (4.10). Lastly, the overall delay should be wrapped around using (4.8) to keep the signal within the working range $[0; S]$. With $y_{nltf}[n]$ being the output of the NLTF system, we have:

$$\begin{aligned} y_{nltf}[n] &= x_{init}[n] + y_{nltf}[n - d_{nltf}[n] \cdot samplerate] \\ d_{nltf}[n] &= S((1 + x_{nltf}[n])/2 + y_{phasor}[n]) \end{aligned}, \tag{4.12}$$

where $x_{init}[n]$ is an initial input necessary to trigger the mechanism. As we can see from the behaviour of the Chebyshev polynomials, the richness or harmonic content of the output of a NTF is somewhat proportional to the ZCs in the transfer function [Roads, 1979]. It is reasonable to filter the signal that determines the transfer function with a low-pass at a cut-off of H/S, where H, approximately, is the number of ZCs in the transfer function. The low-passed signal, though, is likely to have a low amplitude after being filtered; hence dynamical normalisation (see subsection 3.3.1 Low-level information processing) may be deployed to have a full-amplitude transfer function. Moreover, the signal that writes the transfer function may also require limiting and DC-offset filtering for stability and optimal functioning.

Since the transfer function is written cyclically with irregular signals, it is possible to experience discontinuities in the output. A viable design for time-variant transfer functions without discontinuities is to implement relatively complex math equations bounded to the range $[-1; 1]$ where some parameters can be varied smoothly. However, even though math equations can be complex, individually, they may not be able to provide as much variety in the shape of the transfer functions as the audio signals. Depending on the application, either design may be more suitable.

4.1.4 Modulations

The two principal modulation techniques used in the case studies discussed earlier are pole modulation, and single-sideband modulation otherwise referred to as frequency shifting.

The pole modulation technique essentially consists of modulating, within the stability thresholds, the feedback coefficient of a one-pole system. The modulation of the feedback coefficient results in a smooth transition between low-pass and high-pass filter. The sign of the coefficient determines the filter type, and the cut-off shifts from DC, when reaching 1, to $Nyquist$, when reaching -1, passing through 0 which has no action on the input signal.

In some implementations, the system performs self-modulation through its output after being processed with NTF, delays, and low-pass filters to change the modulation rate.

Frequency modulation implements the complex multiplication between an analytic signal and a quadrature oscillator. The generation of the analytic signal is discussed in subsection 3.3.1 Low-level information processing when describing the roughness algorithm. The quadrature oscillator is based on the excellent design by Martin Vicanek,[1], which is arguably the most advanced and complete recursive quadrature oscillator design available in the literature. The difference equation for the oscillator is

$$\begin{aligned} w[n] &= u[n] - k_1[n]v[n] \\ v[n+1] &= v[n] + k_2[n]w[n] \\ u[n+1] &= w[n] - k_1[n]v[n+1] \end{aligned} \quad , \tag{4.13}$$

with

$$\begin{aligned} v[0] &= 0 \\ u[0] &= 1 \\ k_1[n] &= \tan(\omega_{quad}/2) \\ k_2[n] &= 2k_1[n]/(1+k_1[n]^2) \\ \omega_{quad} &= 2\pi f_{quad}/samplerate \end{aligned} \quad . \tag{4.14}$$

This quadrature oscillator is stable under high-rate modulations as well as accurate in the low-frequency range.

4.1.5 Feedback delay networks

Feedback delay networks (FDN) have been explored widely for the realisation of artificial reverberation, see, for instance, [Stautner and Puckette, 1982, Jot and Chaigne, 1991], which is one of the main applications in some of the music projects discussed here, although the principles for artificial reverberation through FDN have also informed other practices in this research besides reverberation effects.

Artificial reverberation through FDN explores the ability of feedback configurations to model sonic interactions (see, for example, [Cook, 1992]). The real-world phenomenon of reverberation is a consequence of sound originating from a point in space and spreading throughout the environment. The acoustic waves bounce against surfaces and obstacles such as walls and rejoin

[1] https://vicanek.de/articles/QuadOsc.pdf. Accessed on the 29th of August 2019.

in the room where they interact and intermodulate. There is a vast number of possible paths sounds can go through, hence a vast number of delays after which sonic interactions among reflecting waves take place. Reflecting waves are subject to phase transformations, which make the combination of waves from different paths rather nontrivial. Furthermore, reflection and air friction influence the frequency content of the acoustic waves as well as their amplitude, making the phenomenon of reverberation somewhat complicated as different frequencies decay at different rates [Rocchesso and Smith, 1997].

FDNs can model these behaviours, to some extent, while requiring adequate CPU resources for real-time applications. The number of delay lines determines the order of FDNs; the higher the order, the more accurate the response. The primary design principle for artificial reverberation with FDNs is to implement a system whose impulse response is as close as possible to uniformly distributed noise, which can then be modelled into specific environments through the processing of the internal feedback paths. Conventional approaches are to design FDNs with prime or coprime delay lengths to minimise ringing modes, and to implement scattering matrixes that maximise the energy diffusion over the entire spectrum [Rocchesso, 1997, De Sena et al., 2015]. Filters can then be used within the loops to model the frequency response of specific environments.

Some of these design principles appeared to be useful for the design of complex feedback networks. In particular, the minimisation of ringing modes and the maximisation of energy distribution along the spectrum may decrease the likelihood of feedback systems to enter attractors, while increasing the possibility of emergence for diverse frequency ranges, resulting in a potentially greater variety.

This paradigm was used for different implementation layers, from the realisation of basic reverberator units or pseudo-reverberators containing nonconventional filtering processing to self-oscillating FDN reverbs with very small room sizes resembling continuously-excited metallic percussion instruments. Alternatively, for higher-level implementations where the processing units within FDNs are complex adaptive agents, as we have seen in the case studies above, which are networks of networks or systems of systems.

4.1.6 Filtering

Filtering techniques are mostly based on the filter designs discussed throughout subsection 3.3.1 Low-level information processing, although the zero-delay feedback topology proposed in [Zavalishin, 2012] is also used for one-pole and two-pole state-variable filters (SVF). Zero-delay SVFs allow the consecutive generation of low-pass, high-pass, and all-pass in the one-pole design, and low-pass, high-pass, band-pass (normalised and non-normalised), low-shelving, high-shelving, band-stop, notch, peak, and all-pass. All these filters can be found in the *Edge of Chaos* library presented in Appendix A *Edge of Chaos* library code.

4.2 Stability processing

Stability processing is an essential part of feedback systems, especially self-oscillating systems, as it provides the necessary conditions for the systems to operate. A typical stable configuration for a self-oscillating feedback system is to control a positive feedback loop through a negative feedback loop. While the positive feedback loop contributes to reinforcing the energy present in the system, the negative feedback loop guarantees that the energy flowing within the loop is restrained to operational boundaries to avoid exponential growths that may result in damaging the equipment, in the analogue case, or in software failure in the digital case due to denormal numbers.[2]

4.2.1 Bounded saturators

Hard clipping would provide stability, but transformations in the amplitude of a signal can affect the spectral content significantly. Hard clipping is the most invasive way to maintain stability; it is implemented through a pair of *if-then-else* statements to replace the signal with a constant if the signal exceeds some boundaries, or to let the signal go through when it is within boundaries. Though the action of adjusting the amplitude in such a sharp way introduces harmonics that can be way above $Nyquist$, and that can result in aliasing.[3] In the digital domain, a sudden sample-wise amplitude change with a perfectly steady transition between amplitudes – for example, a portion of an ideal square wave – corresponds to introducing an infinite number of harmonics, which is the reason why aliasing happens.

Soft clipping provides an alternative to hard clipping by transitioning from the allowed range to the limited range in a smooth way. A smooth transition results in fewer harmonics being introduced and less aliasing artefacts. Some nonlinear transfer functions implement soft clipping techniques, which are also referred to as *bounded saturators*, as we have seen in subsection 2.5.1 Case study: *Phase Transitions* (2018-2019). The most common saturator is probably the hyperbolic tangent function, that is mathematically defined as:

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \quad . \tag{4.15}$$

Zavalishin in his *The art of VA filters design* [Zavalishin, 2012] provides a set of saturators that can be deployed in feedback systems for stability purpose. The saturators differ in the way they smooth out the transition between non-limited and limited range, which results in different spectral responses. Other than the hyperbolic tangent function, Zavalishin suggests

[2] https://en.wikipedia.org/wiki/Denormal_number. Accessed on the 29th of August 2019.
[3] https://en.wikipedia.org/wiki/Aliasing. Accessed on the 29th of August 2019.

the following saturators: $sin(arctan())$, $parabolic()$, and $hyperbolic()$. These are defined as:

$$sin(arctan(x)) = \frac{x}{\sqrt{1+x^2}} \quad , \tag{4.16}$$

$$parabolic(x) = \begin{cases} x(1-|x|/4), & \text{if } |x| \leq 2 \\ sign(x), & \text{if } |x| > 2 \end{cases} \quad , \tag{4.17}$$

$$hyperbolic(x) = \frac{x}{1+|x|} \quad . \tag{4.18}$$

Another saturator that is used often, especially for the simulation of guitar distortions, is the *cubic nonlinear distortion* [Sullivan, 1990]. This saturator is given by the relations:

$$CND(x) = \begin{cases} -2/3, & \text{if } x \leq -1 \\ x - x^3/3, & \text{if } -1 < x < 1 \\ 2/3, & \text{if } x \geq 1 \end{cases} \quad . \tag{4.19}$$

4.2.2 Lookahead limiting

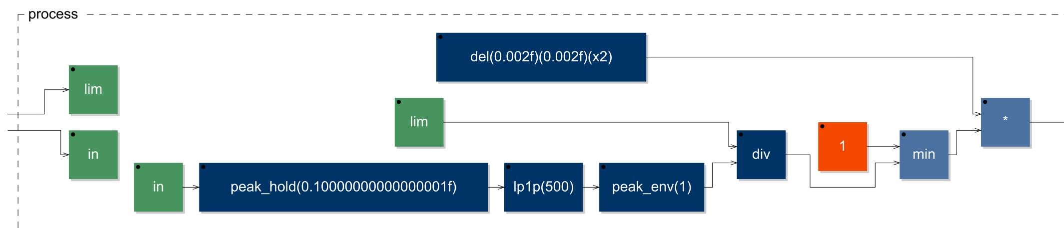

Figure 4.3: Block diagram of the lookahead limiter. The green blocks are external signals. “In” is the input signal, “lim” is the limiting threshold. The input is processed through a peak holder and smoothed out with a low-pass filter followed by a peak envelope. The resulting signal is used to divide the limiting threshold to calculate a scaling factor. Amplification factors are discarded while attenuation factors are applied to the input after being delayed by 0.002 seconds.

Even though saturators can perform a smooth attenuation of signals, they can still generate a high amount of distortion, especially in recursive configurations such as the feedback systems discussed here.

Years ago, I performed some tests on lookahead limiters for the stability processing in feedback networks. The principle of lookahead limiting is to attenuate based on an analysis of the signal instead of using transfer functions. Besides, a key mechanism is to delay the input signal by a small amount, as signals can be attenuated more efficiently if the analysis is performed in advance, and the attenuation takes place early enough. Lookahead limiters can then perform an *ad-hoc* scaling resulting in very little or no audible distortion.

The initial design in my tests followed two side-chained amplitude curves: a fast one for attack transients, and a slow one for sustained sounds, which had to be slow enough to avoid intermodulation distortions. The fast curve was based on peak envelope estimation, while the slow one was based on RMS to have a smoother profile. The curves were in a master-slave mode so that the effect of the fast curve would decrease proportionally to the growth of the slow curve: it was necessary to avoid that the processed signal was scaled down twice by both curves. The algorithm produced acceptable results, but it needed some empirical calibrations, and it also involved a rather large number of calculations that would considerably load the CPU.

A simplified design used only one amplitude curve calculated with a long-decay peak envelope (10 seconds). Peak envelopes would allow detecting fast attack transient and, at the same time, the long decay would create a smooth curve for the sustained sounds. A one-pole low-pass with a cut-off matching the delay time of the input in the limiter filtered out the peak envelope curve. A one-pole low-pass filter simulates the charging curve of a capacitor [Giannoulis et al., 2012]; thus, the peaks detected by the peak envelope would reach their maximum smoothly after the same period of the delay, so that the attenuation curve and the attenuated signal resulted synchronised, reducing the distortion even further. The system was a valid compromise, although such a long decay is not desirable: occasionally, silences occurred after high-amplitude impulses for the attenuating curve decreased very slowly, and the input was still subject to attenuation even if below the limiting threshold.

Iohannes Zmölnig, in a blog post,[4] suggests a design based on Peter Falkner's *Entwicklung eines digitalen Stereo-Limiters mit Hilfe des Signalprozessors DSP56001*. This design is based on a peak holder with an exponential decay curve. The peak holder is described in (3.42); the exponential decay can be achieved by connecting a peak envelope to the output of the peak holder. The chain that analysis the input signal is then a peak holder, a peak envelope, and a one-pole low-pass whose cut-off is the inverse of the period in the delayed path.

A limiter attenuates signals whose magnitude exceeds a threshold and leaves unaltered signals within the threshold. If $x_{env}[n]$ is the amplitude profile, the scaling factor to be applied

[4] http://iem.at/~zmoelnig/publications/limiter/. Accessed on the 29th of August 2019.

to the delayed path is calculated as:

$$y_{scaling}[n] = \begin{cases} threshold[n]/x_{env}[n], & \text{if } threshold[n]/x_{env}[n] < 1 \\ 1, & \text{otherwise} \end{cases} \quad . \tag{4.20}$$

The hold and decay time is set to 0.1 seconds. Theoretically, this should result in a consistent amplitude profile for signals as slow as 5 Hz, for that is the peak-to-peak time at that frequency, and the attenuating signal would decrease fast enough not to result in long silences or gaps.

4.2.3 Adaptive compression

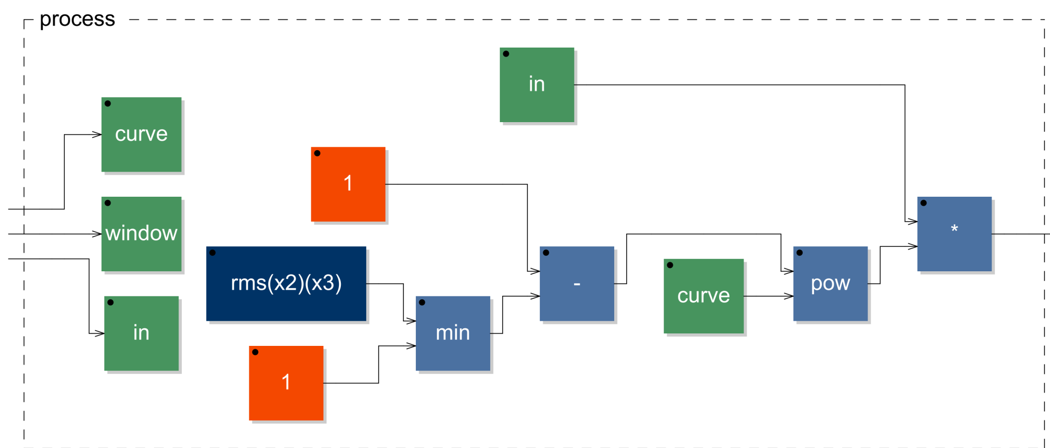

Figure 4.4: Block diagram of the dynamical compressor with RMS analysis. The green blocks are external signals. "In" is the input signal, "window" is the analysis rate, and "curve" is the exponent of the scaling curve. The input signal is multiplied by the complement of the RMS of the input signal itself, after being raised to an exponent to render nonlinear behaviours.

What is here referred to as *adaptive compression* is similar to the design of lookahead limiters as it, too, involves an analysis of the amplitude profile of the input signal and scaling based on the acquired information. Unlike lookahead limiters, though, adaptive compression does not operate based on a specific threshold. Instead, it establishes a continuing relationship with the amplitude of the input that inversely relates the input and output of the system. Adaptive compression is essentially a negative feedback loop between input and output. The analysis of the input profile can be based on RMS or peak envelope measurements or other types of amplitude tendencies, depending on the specific application. Specifically, RMS provides control on the attack and release of the system, which are linked, by setting different analysis windows. Peak envelope has an infinitely fast attack and an adjustable release. The attack time, though, can be modifying by using a one-pole low-pass, as shown earlier, allowing to have independent

control on attack and release of the system. The scaling factor to be applied to the input signal is calculated as the complement of the input amplitude profile:

$$y_{scaling}[n] = 1 - x_{env}[n] \quad . \tag{4.21}$$

The scaling factor can then be raised to a positive power for a nonlinear response, hence adjusting the sensitivity of the system concerning the magnitude of the input.

4.3 Faust: a functional programming language for real-time signal processing

Faust (Functional AUdio STream) is an open-source functional programming language designed for high-performance, sample-wise implementation of audio digital signal processing algorithms [Orlarey et al., 2009]. Faust, unlike other high-level programming languages for DSP such as Pure Data, MaxMSP, or Supercollider, is a compiled language, which allows the compiler to perform several optimisations before the Faust code is translated into C++ code. Furthermore, Faust is an intrinsically stream-based programming language, that is, there is no separation between audio and control domain as everything is a signal, making it particularly suitable for the time-variant networks discussed in this research.

Faust combines the paradigm of functional programming with an algebraic approach to block diagram construction [Orlarey et al., 2004]. Faust has five essential operators for the composition of DSP networks: *sequential*, *parallel*, *split*, *merge*, and *recursive*. The sequential and parallel composition operators are used to implement series or parallel connections between functions. The split and merge operators are used to implement one-to-many and many-to-one connections between functions outputs and inputs. Lastly, the recursive operator is used to implement feedback loops.

Faust allows to define local signals by means of the *with* environment. Faust also offers the *letrec* environment that can be used to define DSP relationships mathematically, particularly for systems of equations, by means of difference equations. For example, the Lorenz system in (2.2), using the *letrec* environment in Faust, can be implemented as follows:

```
import("stdfaust.lib");
lorenz(x0, y0, z0, a, b, r, dt) = x, y, z
letrec {
'x = x+a*(y-x)*dt+(x0-x0');
'y = y+(r*x-y-x*z)*dt+(y0-y0');
```

```
'z = z+(x*y-b*z)*dt+(z0-z0');
};
process = lorenz; .
```

Faust comes with all the math operators available in the C++ math library, and it includes an enormous library of Faust-implemented functions offering a vast possibility to work with different types of filters, band-limited oscillators, physical modelling, and more. It also includes a set of primitives such as sample-wise delay operations as well as logic operators.

Lastly, Faust provides a graphical user interface through which the user can affect the DSP network in real-time, and the Faust compiler can generate stand-alone applications for many platforms.

The software library *Edge of Chaos*, specifically designed for the implementation of musical complex adaptive systems, described in Appendix A *Edge of Chaos* library code, is written in Faust 2.18.[5] The case studies *Phase Transitions*, *Inexorable Shifting 2*, and *Constructing Realities* are also implemented in Faust 2.18 and the code is shown in the next sessions. These last two projects heavily rely on the Edge of Chaos library, whereas the first one has no dependency whatsoever.

4.3.1 *Phase Transitions* code

```
import("stdfaust.lib");

// SYSTEM PARAMETERS

// positive or negative FB coeff.
fb_phase = 1;
// global responsiveness of the system in Hz
//response = 1 ,
//          (lp1pint(1.1, .1) : min(_, 1-1/3600)) : -;

response = 1/36000;

// prime numbers starting from the nth position
prime_offset = 1;
// jump among successive primes
prime_leap = 1;
// feedback growth (unity) rate in seconds
rate = 36000;
```

[5] https://faust.grame.fr/. Accessed on the 29th of August 2019.

```
// order of the network, i.e., number of delay lines
order = 16;

// MATH

primes = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149,
151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233,
239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311);
ny = ma.SR/2;
speriod = 1/ma.SR;
// NAN-safe divider
divider(x1, x2) = ba.if(x2 : ==(0), 0, (x1 ,
                                        x2 : /));
// angular frequency
w(x) = x*2*ma.PI/ma.SR;
// 60-dB decay in a desired time
rt60(x) =   .001 ,
            (   speriod ,
                max(x, .001) : /) : pow;
prime_base_pow(n) = ba.take(n, primes) ,
                        _ : pow;
// line: in signal sets x/y ratio
line =  _ ,
        ma.SR : / : +
                    ~ _;
// rate of change
delta(in, t) =  in ,
                (in : de.delay(max_del, t*ma.SR)) : -;

// AUXILIARY CONSTANTS

// delay lengths maxima
max_del = 2^20;
// interpolation size (samples) for sdelay
int_size = 1024;
// order of Lagrange interpolation
pol_order = 6;
```

```
// stability threshold (for linear FDN)
margin = 1/sqrt(order);
// nonlinearities/saturators
nl = (tanh, sinatan, parabolic, hyperbolic);

// AUXILIARY FUNCTIONS

dirac = 1-1';
step(n) = 1 <: _ ,
               @(n) : -;
// sample counter
timer = 1 : fi.pole(1);
// minimum delay allowed for correct performance
min_del = max((pol_order-1)/2, _);
// fixed delay
del_int(in, del) = in : de.delay(max_del, del*ma.SR);
// linear interpolation
del_lin(in, del) = in : de.sdelay(max_del, 8*ma.SR, del*ma.SR);
// polynomial interpolation
del_pol(in, del) = in : de.fdelayltv(pol_order, max_del, del*ma.SR : min_del);
// 6-ch output busses for different orders
order4 = si.bus(4) <:   (si.bus(4) :> par(i, 2,   /(2))) ,
                        si.bus(4);
order8 = si.bus(8) <:   (   (   si.bus(6) ,
                                (_ : !) ,
                                (_ : !)) ,
                            (   (   par(i, 6, (_ : !)) ,
                                    (si.bus(2))
                                ) <: si.bus(6))
                        ) :> par(i, 6, /(2));
order16 = si.bus(order) :> par(i, order/2, /(2)) : order8;
// 2-ch output bus for different orders
output2 = si.bus(order) :>  /(order/2) ,
                            /(order/2);

// STABILITY PROCESSING

// hard clipping
```

```
clip(in, lower, upper) = in : max(lower) : min(upper);
// reference_power-input_power ratio (with responsiveness parameter)
norm_fact_rms(ref, target, window) =    (    ref ,
                                             window : rms) ,
                                        (    target ,
                                             window : rms) : divider;
// dynamical normalisation based on RMS
dyn_norm_rms(ref, target, window) = (ref ,
                                     target ,
                                     window) :   norm_fact_rms ,
                                                 target : *;

// STABILITY PROCESSING/NLTF (bounded saturators)

tanh(x) = (exp(2*x)-1)/(exp(2*x)+1);
cubic(x) = select3( cond,    -2/3,
                             x-(x*x*x/3),
                             2/3)
with {
    cond = (     (x : >(-1)) ,
                 (x : <(1)) : &) ,
             (x : >=(1))*2 :> _;
};
sinatan(x) = x/sqrt(1+x*x);
parabolic(x) = ba.if(abs(x) : >=(2),     ma.signum(x),
                                         x*(1-abs(x/4)));
hyperbolic(x) = x/(1+abs(x));

// FILTERS

// bounded integrator
clip_int(in, lower, upper) =     (_ ,
                                 in : + : clip(_, lower, upper)
                                 ) ~ _;
// spectrum splitter
crossover(in, cf) = lp1p1z(in, cf) ,
                    hp1p1z(in, cf);
// 1-pole lowpass based on naive integration
```

```
lp1pint(in, cf) =   (in ,
                    _ : + : *(w(cf) : clip(_, 0, 2)) : fi.pole(1)
                    ) ~ *(-1);
// 1-pole highpass based on naive integration
hp1pint(in, cf) = in-lp1pint(in, cf);
// 1-pole-1-zero lowpass
lp1p1z(in, cf) =    in*a0 ,
                    in'*a0*a1 : + : +
                                    ~ *(b1)
with {
    a0 = (1-b1)/2;
    a1 = 1;
    b1 =    (1-sin(w(cf))) ,
            cos(w(cf)) : divider;
};
// 1-pole-1-zero highpass
hp1p1z(in, cf) =    in*a0 ,
                    in'*a0*a1 : + : +
                                    ~ *(b1)
with {
    a0 = (1+b1)/2;
    a1 = -1;
    b1 =    (1-sin(w(cf))) ,
            cos(w(cf)) : divider;
};

// INFORMATION PROCESSING

// infinitely fast attack and adjustable release
peak_env(in, release) = abs(in) :   max(_, _)
                                    ~ *(release : rt60);
// root mean square measurement using lowpasses
rms(in, window) = in <: * : lp1pint(_, window) : sqrt;
// spectral energy difference (RMS) at a splitting point (Hz)
spec_bal(in, cf, window) =  in ,
                            cf : crossover <:   (   _ ,
                                                    window : rms : *(-1)) ,
                                                (   _ ,
```

```
                                                window : rms) : +;
// spectral tendency (weighted median)
spec_ten(in, window) = (    (   in ,
                                _ ,
                                window : spec_bal) ,
                            (   in ,
                                window : rms) : divider : /(ma.SR) : *(window)
                                    : clip_int(_, 0, 1) <: * : *(ny)
                        ) ~ _;

// TIME-VARIANCE AND ADAPTIVITY

// information signal
is_delay =  1 ,
            seq(i, 4, lp1pint(_, response)) ,
            response : dyn_norm_rms : tanh;
// information signal
is_fb = (   _ ,
            50 : spec_ten : /(ny)) ,
        .02 : delta <: * : *(500) : tanh ,
                                    60 : peak_env : *(10) : +(1);
// adaptive signal
as_delay(n) = is_delay : +(1) : prime_base_pow(n) : *(.0001) : -(speriod);
// adaptive signal
as_fb = 1/rate ,
        is_fb : divider : line : +(margin*1.01) : *(fb_phase);

// SECTIONS

input = par(i, order, +(dirac));
matrix = ro.hadamard(order);
mixer = si.bus(order) :> /(order) <: si.bus(order);
router = ro.interleave(order, 2);
dcblocker = par(i, order,   _ ,
                            10 : hp1p1z);
delay = par(i, order,   _ ,
                        (as_delay(i*prime_leap+prime_offset)) : del_lin);
nltf = par(i, order, ba.take(i : %(4) : +(1), nl));
```

```
fb = par(i, order, *(as_fb));
output = case{  (4) => order4;
                (8) => order8;
                (16) => order16;}(order);

// MAIN

process =   (input : nltf : dcblocker)
            ~ (si.bus(order) <: matrix ,
                            mixer : router : fb <: ro.interleave(order, 2)
                                : delay) <: output2;
```

4.3.2 *Inexorable Shifting 2* code

```
import("stdfaust.lib");
import("edgeofchaos.lib");

order = 16;
stability = 1/sqrt(order);
rate = 1/180;
curve = 48;
i_time = 60;

dll_and(in) =   (ip.rms(rate, in) : >(.9)),
                (   ip.spec_ten(rate, in),
                    ip.rms(rate, in) : m2.hp_and) : ba.sAndH/1000;

dll_nand(in) =  (ip.rms(rate, in) : >(.9)),
                (   ip.spec_ten(rate, in),
                    ip.rms(rate, in) : m2.hp_nand) : ba.sAndH/1000;

dll_or(in) =    (ip.rms(rate, in) : >(.9)),
                (   ip.spec_ten(rate, in),
                    ip.rms(rate, in) : m2.hp_or) : ba.sAndH/1000;

dll_nor(in) =   (ip.rms(rate, in) : >(.9)),
                (   ip.spec_ten(rate, in),
                    ip.rms(rate, in) : m2.hp_nor) : ba.sAndH/1000;
```

```
dll_list = (dll_and, dll_nand, dll_or, dll_nor);

dlls(e) = ba.take((e%4)+1, dll_list);

agent(sh, x) = x : op.ssbm(ip.spec_ten(rate, x) : >(.1) : *(-2) : +(1) : *(sh));

ins = par(i, order, +);
agents = par(i, order, agent(sqrt(i+1)/1000));
delays = par(i, order, (_ <:    (dlls(i)),
                                 _ : d2.del_smo(16, i_time)));
coefficients = par(i, order, *(stability*1.001));
comps = par(i, order, st.dyn_comp_rms(curve, 1/rate));
limiters = par(i, order, st.limiter(1));
matrix = ro.hadamard(order);

process(x) = (x <: si.bus(order) :   (ro.interleave(order, 2) : ins : delays
                                        : limiters : agents)
                                     ~ (coefficients : matrix)
                                        :> par(i, 2, /(order/2)));
```

4.3.3 *Constructing Realities* code

```
import("stdfaust.lib");
import("edgeofchaos.lib");

// 0.5008 good FB coeff.

// CONSTANTS

trate = 1/.001;
r1 = 1/128;
order = 6;

// ADAPTIVE MODULES

r2(x) = dynamicity(r1, x) : <(.5) : *(r1)
    : os.osc : m2.uni : m2.map_pow(48, r1, 15);

pol_ten(x) = st.dyn_norm_rms(r1/10, 1/1024, x : seq(i, 4, f2.lp1p(r1)))
```

```
    : os.osc :  m2.uni : f2.sah_inv : f2.lp1p(trate);

pol_ten_smo(x) = st.dyn_norm_rms(r1/10, 1/1024, x : seq(i, 4, f2.lp1p(r1)))
    : os.osc :  m2.uni;

// ADAPTIVE VARIANTS GENERATION

// g = global context; l = local context

ip_set(g, l) =  ip.rms(r2(g), l),
                ip.spec_ten(r2(g), l),
                ip.noisiness(r2(g), l) : par(i, 3, ba.if(>(.5), r1, -r1)
                    : os.osc : m2.uni: f2.sah_inv : f2.lp1p(trate));
ip_sel(g, l) = par(i, 3, (  (pol_ten(l) : +(i/3) : ma.decimal),
                            ip_set(g, l) : m2.interpolate_mn(3, 1)));

map_sel(n, x) = par(i, n,   (   ((pol_ten(x) : +(i/n) : ma.decimal),
                                _,
                                _ :  m2.interpolate_mn(2, 1)),
                                ((pol_ten(x) : +(i/n) : ma.decimal),
                                _,
                                _ : m2.interpolate_mn(2, 1))),
                            _ : m2.map_lin);

// AGENTS

agent1(g, l) = ip_sel(g, l) : (      -4,
                                     -1,
                                     1,
                                     4,
                                     _),
                              (      100,
                                     10000,
                                     10000,
                                     100,
                                     _),
                              (      1,
                                     -1,
```

```
                                    -1,
                                    1,
                                    _) : map_sel(3, l) : (_,_,*(l)), g+l
                                        : op.grains_dl_zc(1, 16);

agent2(g, l) = ip_sel(g, l) :   (   1/16,
                                    16,
                                    -16,
                                    -1/16,
                                    _),
                                (   0,
                                    16,
                                    16,
                                    0,
                                    _),
                                (   -.25,
                                    4,
                                    -4,
                                    .25,
                                    _) : map_sel(3, l), g+l : op.sampler(16);

agent3(g, l) = ip_sel(g, l) :   (   20,
                                    20000,
                                    20000,
                                    20,
                                    _),
                                (   .5,
                                    2,
                                    2,
                                    .5,
                                    _),
                                (   .5,
                                    2,
                                    2,
                                    .5,
                                    _) : map_sel(3, l)
                                        :   par(i, 3, _ <: si.bus(4)),
                                            audio_ins : ro.interleave(4, 4)
```

```
                                                        : filters :> _
with {
    audio_ins = g+l : f2.xover1p1z_ada(r1) :     f2.xover1p1z_ada(r1),
                                                 f2.xover1p1z_ada(r1)
                                                    : par(i, 4, st.clip(-1, 1));
    filters = par(i, 4, f2.svf2blti : par(i, 10, ma.tanh)
        :    m2.interpolate_mn(10, 1)
             ~ (ma.tanh : pol_ten_smo));
};

agent4(g, l) = ip_sel(g, l) :    (    -10,
                                      -1,
                                      -1,
                                      -10,
                                      _),
                                 (    -.9999,
                                      .9999,
                                      -.9999,
                                      .9999,
                                      _),
                                 (    20,
                                      20000,
                                      20000,
                                      20,
                                      _) : map_sel(3, l), g+l
                                          : op.rev_fdn_smo(16, 16, 1);

agent5(g, l) = ip_sel(g, l) :    (    -10,
                                      -1,
                                      -1,
                                      -10,
                                      _),
                                 (    -2,
                                      2,
                                      -2,
                                      2,
                                      _),
                                 (    20,
```

```
                                        20000,
                                        20000,
                                        20,
                                        _) : map_sel(3, l), g+l
                                           : fdn_smo_svf(16, 16, 1)
with {
fdn_smo_svf(n, max_size, it, size, fb_coeff, cf, in) =
      (summing : delays : filters : matrix : fb : limiters)
      ~ si.bus(n) :> /(n)
with {
      stability = 1/sqrt(n);
      limiters = par(i, n, st.limiter(1));
      summing = par(i, n, +(in));
      delays = par(i, n,    max_size,
                            it,
                            (size : m2.prime_base_pow(i+1)),
                            _) : par(i, n, d2.del_smo);
      filters = par(i, n, (cf, _)
        : f2.svfblti :  m2.interpolate_mn(3, 1)
                        ~ (ma.tanh : pol_ten_smo));
      matrix = ro.hadamard(n);
      fb = par(i, n, *(fb_coeff*stability));
};
};

agent6(g, l) = ip_sel(g, l) :   (   -10,
                                    -1,
                                    -1,
                                    -10,
                                    _),
                                (   -2,
                                    2,
                                    -2,
                                    2,
                                    _),
                                (   20,
                                    20000,
                                    20000,
```

```
                                        20,
                                        _) : map_sel(3, 1), g+l
                                            : fdn_smo_ssbm(16, 16, 1)
with {
fdn_smo_ssbm(n, max_size, it, size, fb_coeff, shift, in) =
     (summing : delays : shifts : matrix : fb : limiters)
     ~ si.bus(n) :> /(n)
   with {
        stability = 1/sqrt(n);
        limiters = par(i, n, st.limiter(1));
        summing = par(i, n, +(in));
        delays = par(i, n,    max_size,
                              it,
                              (size : m2.prime_base_pow(i+1)),
                              _) : par(i, n, d2.del_smo);
        shifts = par(i, n, op.ssbm(shift));
        matrix = ro.hadamard(n);
        fb = par(i, n, *(fb_coeff*stability));
   };
};

agents =     (agent1 : au.inspect(0, -10, 10)),
             (agent2 : au.inspect(1, -10, 10)),
             (agent3 : au.inspect(2, -10, 10)),
             (agent4 : au.inspect(3, -10, 10)),
             (agent5 : au.inspect(4, -10, 10)),
             (agent6 : au.inspect(5, -10, 10));

// NETWORK

cr(x) =      (par(i, order,   x,
                          _) :     agents,
                                   m2.topologies(order, order, t)
                                       : m2.matrix(order, order)
                                           <: ro.interleave(order, 2)
                                               : delays : limiters)
             ~ par(i, order, *(coeff)) : outs :> par(i, 2, st.limiter(1))
with {
```

```
    limiters = par(i, order, st.limiter(1));
    outs = par(i, order, _ <:   pol_ten_smo^8,
                                _ : *);
    delays = par(i, order, (    pol_ten*8+8,
                                _) : d2.del_smo(16, 1));
    t = complexity(16, x) : f2.lp1p(r1) : <(.75) : /(1024)
        : os.osc : m2.uni;
    coeff = pol_ten(x)*(-1/order);
};

// complexity index based on the edge of chaos of
// RMS, spectral tendency and noisiness
complexity(window, in) =   (ip.rms(15, in) : m2.unit_log(10)),
                           (ip.spec_ten(15, in) : m2.unit_log(10)),
                           (ip.noisiness(15, in) : m2.unit_log(10))
                               : par(i, 3, ip.heterogeneity(10, 0, 1, window))
                                   : (  (_ , t1 : m2.div),
                                        (_ , t2 : m2.div),
                                        (_ , t3 : m2.div)) :
                                           par(i, 3, f2.lp1p(1/(window*2)))
                                               : par(i, 3, m2.delta(
16,
                                                                         window*2)
                                                   : abs)
                                                       :> f2.lp1p(1/(window*2))
                                                           : ma.tanh
with {
      t1 = ((ip.rms(15, in) : m2.unit_log(10)) : ip.peak_hold_LH(window) : ro.cro
              : -) : f2.lp1p(1/window) : m2.map_lin(.25, 1);
      t2 = ((ip.spec_ten(15, in) : m2.unit_log(10)) : ip.peak_hold_LH(window)
              : ro.cross(2) : -) : f2.lp1p(1/window) : m2.map_lin(.25, 1);
      t3 = ((ip.noisiness(15, in) : m2.unit_log(10)) : ip.peak_hold_LH(window)
              : ro.cross(2) : -) : f2.lp1p(1/window) : m2.map_lin(.25, 1);
};

// dynamicity index based on RMS, spectral tendency and noisiness
dynamicity(window, in) =   ((ip.rms(15, in) : m2.unit_log(10)),
                           (ip.spec_ten(15, in) : m2.unit_log(10)),
```

```
                                    (ip.noisiness(15, in) : m2.unit_log(10)))
                                        : par(i, 3, m2.delta(1/15, 1/15) : abs
                                            : f2.lp1p(window)) :> ma.tanh;

process(x) =    (+(x) : cr)
                ~ ( _,
                    _ :> st.limiter(1)*0.5008 : @(ma.SR*8)) : par(i, 2, f2.hp1p1z(5));
```

4.4 Pure Data: a patchable environment for audio analysis, synthesis, and processing

Pure Data is an open-source graphical environment for object-oriented audio programming. Pure Data is particularly fast and very easy to learn, and it is thus particularly suited for fast prototyping as well as educational purposes in introductory-to-advanced classes on DSP and audio programming.

Pure Data is divided into message and audio domains, separating the scheduling and control part from the audio signal processing and generation. The software provides a large number of objects from simple scheduling operators to list processors, and from basic math operations for audio signals to fast Fourier transform processing [Puckette, 1997].

The software's strength is reliability: after many years of using Pure Data in live performance, I have never experienced a software malfunction that compromised the performance. Though Pure Data also has limitations, the most important of which is being single-precision. The 32-bit resolution for audio signals is enough for a suitable dynamic range; 24bits represent the amplitude of signals, which is about 144dBs. Although, single precision is not suitable for indexing of large tables, which results in distorted sounds, or accurate filtering when the cut-off frequency is close to the extremes, namely *DC* for low-passing and *Nyquist* for high-passing, as they require precise coefficients. Pure Data's single-precision is the main reason why Faust has now become the primary developing environment, while Pure Data is used as a testing and prototyping tool. Another fundamental reason is that Faust does not need to change the block size of its internal signal processing to implement sample-wise feedforward or feedback connections, while Pure Data does, which results in much slower performance.

The case studies discussed in subsection 3.2.1 Case study: *Order from Noise (Homage to H. von Foerster)* (2016-2019), and subsection 5.3.1 Case study: *Single-Fader Versatility* (2016), subsection 5.4.1 Case study: *Audible Icarus* (2016-2018) were implemented in Pure Data 0.49,[6]

[6]http://msp.ucsd.edu/software.html. Accessed on the 29th of August 2019.

4.5 Remarks

We have reviewed some standard and original techniques for the processing of audio in the time domain to achieve transformations and guarantee stability in feedback networks. In the implementation of audio complex adaptive systems, the processes responsible for the modification of context are also crucial as it is their combination with information processing and stability processing that will result in complex behaviours.

Especially in the case of systems where adaptation is distributed and widely deployed, being able to access the largest number of variables in algorithms for audio processing is crucial as it maximises the relationships that can be established with information signals. Hence, the analytical description of some of the standard audio processing techniques was necessary to achieve maximum control over the parameters that drive such algorithms.

The implementation and evaluation of information processing, audio processing, stability processing, and distributed adaptation are procedures that are both informed by theoretical thinking and trial-and-error approaches. Audio processing units are tested individually first as self-oscillating systems without self-regulation. Human-driven empirical testing is identifying the sonically most interesting parameters and ranges than can eventually be piloted through self-modulation and adaptation.

Information processing algorithms are deployed, and basic adaptation mechanisms are tested with positive and negative feedback configurations, as well as different mapping curves to implement nonlinearities and biases. Stability processing modules, too, are tested within these basic prototypes to guarantee stability but also to achieve timbral transformations and formal developments, particularly with long-term amplitude variations.

Lastly, distributed adaptation is tested by plotting the top-level information signals that pilot the organisation of the network and the structure of the agents. The plotting, in real-time, is useful to identify corresponding behaviours between the variations in the plots and the long-term variations in the evolutions of the systems.

The first technique that we discussed is granular processing, starting from the early theoretical developments to the early analogue prototypes and the more modern digital implementations. An alternative design for this kind of processing was proposed, which suggests the use of zero-crossing detection as an alternative to the standard windowing methods used to overcome the issue of discontinuities in fragmented signals. The results of this new method are sonically convincing as dense-like and sharp textures can be obtained with a small number of granulator voices, hence reducing the CPU usage, and the granulator can oscillate between a looping device and a noise generator, depending on the context and on how adaptation takes place.

Another alternative method is proposed for nonlinear transfer functions. Following the notions used for the granulator, an FDL implements a CB to which transfer functions are

cyclically stored and accessed through different delay amounts. This mechanism is then used within a retroactive configuration where the output of the system writes the transfer function, which, in turn, affects the output. To provide the system with control over the additional frequency components being generated, hence the richness and noisiness of spectra, the zero-crossings of the signals that determine the transfer functions can be modulated as they relate to the number of harmonics resulting from the transformation.

After reviewing other standard techniques for audio processing, the chapter focuses on stability processing, which is deployed in music systems to prevent audio networks from growing indefinitely. Stability processing, despite it appears to be a secondary task that mainly deals with a technical issue, it has a significant effect on both the short-term and long-term unfolding of the music systems presented here. In section 5.4 Electroacoustic devices and the environment as interfaces, we have seen examples of the synergetic and intrinsically nonreductionist characteristics of feedback networks and the systemic role acquired by the elements contained in the feedback loops. Similarly, the task of maintaining amplitude levels within specific ranges has effects that extend to the whole network and go beyond the amplitude characteristics of the audio streams. For example, the two main categories of stability processing algorithms presented earlier, soft-clipping by means of saturators, and adaptive mechanisms via the extraction of amplitude-related information to counteract particular responses, have radically different results when deployed in feedback systems. The first is a special kind of transfer functions, having an immediate effect on the signal, which may result in significant production of additional spectral components and overall noisiness. The second type, on the other hand, operates by smoothly scaling down the signal, hence reducing the appearance of extra spectral components to a minimum. Furthermore, the second approach allows for temporal parameters to be set to adjust the responsiveness of the scaling mechanisms, which can be used as a process to generate formal musical developments through long-term variations of the amplitude of audio streams, as we mentioned earlier.

The chapter concludes by presenting the two primary programming environments used for the implementation of the case studies: Pure Data, adopted in the past years, and Faust, for current developments. Faust, particularly, represents an ideal choice for adaptive audio systems as it follows a stream-based paradigm. Hence, any variable in a system can be modulated through audio or infrasonic signals.

Chapter 5

Performance modalities and interfaces

> *If you put yourself in a situation of unpredictability and then find that it's completely possible to accept it, then you become an observer.*
>
> David Tudor

In this chapter, we will investigate the relationships between human performers and autonomous machines within the context of live music and improvisation. Here, we propose a framework for improvisation based on cybernetic criteria to structurally couple human and artificial entities as hybrid systems of mutually-determining components. The human and the machine become entangled through incessant and mutating relationships that take place as adaptations to information derived from low-level and high-level sound characteristics.

Then, the chapter explores notions of cybernetics for the implementation of human-machine interfaces that allow the performer to reshape the network of relationships within the system's components with agility and organicity.

Lastly, this chapter investigates the role of electroacoustic devices and environments within the context of human-machine interfacing in live performance.

5.1 Cybernetic improvisation

As discussed in subsection 1.3.2 The objectivity of the machine, improvisation represents a configuration where the human and the machine are coupled through a feedback loop; the two entities realise a higher-level system where the input is the human's auditory system,

and the output is the sound generated by the machine through the actions of the human. Improvisation can be used to establish recursive loops among several entities, too. For instance, an improvisation ensemble, specifically an ensemble practising radical improvisation, can be seen as a network of humans where each agent is affected by its output as well as the output of the other agents, that is, a fully-connected network. When human performers operate with autonomous machines, the number of entities in the network is then the combination of humans and machines involved. For ensembles of human and machine performers see, for example, [Sanfilippo and Di Scipio, 2017].

The behaviour of musicians in radical improvisation setups, that is, setups where performers' actions are entirely unsupervised, are not formalised or defined according to any explicit rules, although, technically, the performers' actions are always a response to the outputs generated by the agents involved in the process. As we have seen earlier in subsection 1.3.2 The objectivity of the machine, though, despite improvisation allows for the highest degree of freedom, achieving musically compelling results can be challenging because of some inherent characteristics of feedback loops.

In structured improvisation, constraints or rules are set for the performers to drive the development of music towards specific directions, which are predefined paths that should guarantee a certain degree of musical complexity: these constraints or rules limit the freedom of the performers but can, on the other hand, avoid undesired outcomes where redundancy and predictability negatively affect the overall performance. These rules, though, are not necessarily related to the sonic context that is generated by the performers. In other words, the rules are absolute rather than relational.

Composers have explored techniques for relational rules and behaviours in improvisation, and an overview of some of these works can be found in [Dahlstedt et al., 2015], where Palle Dahlstedt et al. also describe the group improvisation system that they have developed. Their system is a relational model of improvisation with random elements based on subjective and high-level behavioural rules such as *lead, support, opposition*. These behaviours, though, are somewhat arbitrary as different performers may have a substantially different interpretation of the modalities.

Another example of formalisation of improvised performance based on relational criteria can be found in [Murray-Rust and Smaill, 2011]. Murray-Rust and Smaill propose a model where actions are mediated through analysis functions of the musical surface to create a musical context. The musical context would then allow for processing the musical outputs to construct a set of performative actions, similarly to how processes take place in Speech Act theory.

More generally, a thorough analysis of improvisation methods and models can be found in [Pressing, 1988], where the concepts of feedback are considered fundamental for the development of methods and models of the improvisation practice. The improvisation techniques described below follow the same direction.

The improvisation system proposed here follows a cybernetic approach where rules of interaction are well-defined and can then be applied to several aspects of sound at the timbral or formal level to achieve higher-level behaviours. Sounds can be analysed based on several low-level information criteria, some of which have been discussed in subsection 3.3.1 Low-level information processing, that are suitable for this kind of improvisation system. Among the standard low-level information measures, we have *loudness* and *brightness*. In the next session, where a performance project based on this system is described, these low-level features will be used together with *noisiness* and a higher-level feature of sound events called *density*, which is related to the measurement of dynamicity discussed earlier in subsection 3.3.2 High-level information processing, that provides an index of the number of recognisable variations per unit time. Other low-level criteria may include *roughness* and *spectral spread*, also presented in subsection 3.3.1 Low-level information processing.

[Heylighen and Joslyn, 2001] identify three main mechanisms of control in cybernetic and self-regulating systems; one mechanism is *feedback*. The two fundamental relationships for this improvisation system are the *diverging* and the *converging* functions. These, respectively, correspond to positive feedback and negative feedback behaviours which are described in section 2.1 Feedback mechanisms. Examples of these behaviours concerning music performance are provided in the next session.

Another control modality in cybernetics that may be used in this approach is *buffering*. It consists of absorbing the perturbations received by a system through a dampening mechanism. In the specific case of an improvising agent, a buffering function may be achieved by recording the incoming perturbations at regular intervals to set their states and transitioning smoothly among successive states. The interval would determine the dampening degree of the process, and the transitioning modality sets the linearity or nonlinearity of the process. Namely, a smooth and gradual transition would resemble a linear interpolation, while transitioning as-fast-as-possible to the next state would resemble an exponential variation.

The application of functions requires the analysis of each criterion and a quantification. The quantification can have different values according to the details required for each function. The measurement is likely to be based on subjective estimations of the agents, although computers may as well be used to provide the performers with more accurate measurements. Moreover, the analysis can be performed on different sources, which is what ultimately determines the topology of the network.

The relational scores realised with this technique can be represented as a two-dimensional array or two-dimensional grid. The x-axis, or timing axis, is divided into sections of different lengths. The timing for each piece may follow different approaches, either objective ones using a centralised timer, or individual references for each agent based on timing cues or arbitrary and subjective estimations. We will see an example in the next session. The y-axis represents the information criteria used for a piece. At the intersections between the elements of the axes, we

have the functions that set the interaction modalities between information criteria and sound sources. Furthermore, each section can also contain indications on which source to analyse, so that the network structure can dynamically change as the piece unfolds. In general, different kinds of *if-then-else* conditions can be set for each section so that the structure of the network itself becomes adaptive and varies according to the sonic context.

5.1.1 Case study: *Human Network: Machine Nostalgia* (2016-2018)

Human Network: Machine Nostalgia is a score for three or more instrumental performers based on relational mechanisms taking place in the domain of sound and distributed among the musicians. The *functions* and *descriptors* are the two main aspects of the score. Two behaviours, *diverging* and *converging*, describe the functions, which, respectively, correspond to the characteristics of positive and negative feedback mechanisms concerning a state of equilibrium in a system.

The functions are applied to the descriptors, which are a set of four sonic characteristics describing a sound event from a specified source. The descriptors are *loudness*, *brightness*, *noisiness*, and *density*. The loudness refers to the intensity of a sound. The brightness indicates the overall spectral energy distribution, i.e., what register is predominant. The noisiness describes how noisy a sound is. The density, instead, refers to the number of individual sound events per unit time.

The score is divided into sections of different durations. For each section, one of the two functions will be assigned to some of the four descriptors, and different sonic sources will be assigned to each performer.

In order to avoid ambiguous situations and to give the possibility to the performers to handle more than one variable at a time, the descriptors will be assigned two values only: *high* and *low*. Performers, thus, will be asked to perform an analysis of the incoming sound and give an estimation of one or more descriptors in real-time according to their perception and, based on the resulting values, they will act according to the function assigned to each descriptor.

Musicians are asked to perform the analysis of the descriptors based on one of the other performers, on all of the other performers as a whole (excluding her/himself), or on the surrounding environment, entirely bypassing the other performers. The source to be analysed will change from section to section, and the indication on the score will be as follows: *ALL*, referring to all the other performers; *LEFT/RIGHT*, referring to the performer to the left/right side; *ENV*, referring to the environment.

We have two functions in the score: the *diverging* function and the *converging* function. The diverging function represents the behaviour of a positive feedback mechanism, as mentioned earlier. The characteristic of this mechanism is to recursively strengthen the effects of perturbations that pushes a system away from equilibrium, in turn resulting in exponential deviations. In the music domain, this mechanism translates into producing sound events that

move even further in the same direction as the value of the estimated descriptor to which the function is applied. For example, a diverging function applied to a low brightness means to produce a sound event whose brightness is as low as the analysed one or lower. The same kind of action applies to the other descriptors and the other value.

The converging function, instead, represents a negative feedback mechanism. It means that this type of action results in a counterbalancing behaviour with oscillations around an equilibrium point. Musically speaking, the performer applying this function will produce a sound event which goes towards the opposite direction of the estimated value of a descriptor. For example, applying this function to a high loudness means to produce a sonic event with a low loudness. On the score, the following symbols will be used for, respectively, the converging and diverging functions: ><; <>.

Considering that the analysis of several descriptors at the same time could be extremely challenging for a performer, one way to simplify this task is that of using constants for some descriptors. This way, the analysis task will be limited to a maximum of two descriptors per section, and the remaining descriptors will be handled using the constants *high* and *low*, corresponding to the estimation criteria used for the analysis of the descriptors. For example, a density feature with a low constant means that the performer will have to maintain whatever her or his interpretation of a low density is for the whole section. On the score, constants will be indicated with the words *high* and *low*.

The duration of the sections are based either on a perceptual clock, given by the individual temporal estimation of each performer, or on a master clock, which will be the reference for all performers. For example, if the first section is one minute, in the first case, each performer will attempt to guess the correct time to switch to the next session. Alternatively, one or more synchronised stopwatches will be used.

The indeterminacy given by individual interpretations of the musicians is what makes this piece organic, although the indeterminacy is the result of subjective processing of the context, which is likely to be unpredictable and to change each time that the piece is performed, even when doing so with the same performers. When the estimation of a sonic feature in a section changes, there is a chain reaction that triggers a series of changes in the other performers and reorganises the piece into a new global dynamics. Furthermore, the subjective interpretation of time can have an even more profound effect, for the sections among the performers can overlap differently at each execution producing a higher-level restructuring of the network.

The score in the image below, which is only one of the many possible scores that can be generated with this mutable system of auditory relationships, was performed in the studio with a string trio of excellent musicians: Dimitris Papageorgiou (violin), Armin Sturm (double bass), Rus Wimbish (double bass). Despite recording several performances of the piece after many hours of rehearsing, due to technical issues, there is currently no recording of the piece. The performances were convincing, and it was fascinating to see somewhat different dynamical

behaviours each time that the score was performed.

Source	ENV	RIGHT	ALL	LEFT	LEFT	ALL	RIGHT
LOUDNESS	><	*low*	*high*	*low*	<>	*low*	*low*
BRIGHTNESS	*high*	><	*low*	*low*	*high*	*low*	*low*
NOISINESS	*high*	*low*	*low*	><	*low*	*high*	<>
DENSITY	*low*	*high*	><	*low*	*high*	<>	*low*
Section	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Duration	1'30"	2'45"	1'15"	2'	1'	3'	30"

Figure 5.1: *Human Network: Machine Nostalgia* score. "Loudness", "brightness", "noisiness", and "density" are the information criteria. "All", "env", "right", and "left" are the sources from which to extract information. "><", "<>", "low", and "high" are the functions to be applied to the criteria. "Section" is the row for the sections of the piece. "Duration" is the row for the durations of each section.

5.2 Cybernetic mapping

Cybernetic mapping follows the same principles as those described in the previous section for cybernetic improvisation, although rather than interrelating performing agents, the cybernetic mapping approach binds the variables of DSP agents to positive and negative feedback relationship. More precisely, the cybernetic mapping strategy was developed for human-machine interaction performance with semi-autonomous systems. A semi-autonomous system is a machine with some degree of adaptation but not able to alter the structure of the network enough to change the relationship among variables or agents.

The setup for the application of cybernetic mapping consists of a machine able to self-modulate its output and, to some extent, shape its formal developments, coupled with a human entity through a feedback loop provided by some improvisational modality. The role of the human entity, in particular, is to alter the adaptation modality and adaptation ranges in the system while the relationship between pairs of internal variables is maintained.

As discussed in section 3.3 Information processing techniques, information processing for adaptation can follow criteria based on perceptual models, or it can be based on abstract principles that are meaningful for the machine, depending on the specific goals that are set. These goals can rely on a specific characteristic of the sonic output, e.g., the amplitude, frequency, spectral distribution of energy, noisiness. For example, if the goal is to keep the overall noisiness constant in a system with two agents, a bandpass filter and a frequency modulation (FM), the modulation index in the FM and the Q in the bandpass filter should be directly proportional. Similarly, if we have a saturator and an FM unit and we still want to keep a constant noisiness, the amplitude of the saturator and the modulation index of the FM should be inversely

proportional.

Of course, this kind of relationships among variables does not guarantee that the system will always respond as expected, as these systems are highly nonlinear and subject to significant changes even with small perturbations on seemingly non-affecting parameters. Nonetheless, these criteria allow for a fundamental framework upon which networks of relationships deriving from precise criteria can be built.

Alternatively, instead of choosing relationships according to some desired characteristics of the overall output, it is possible to build the network of relationships considering a specific characteristic between pairs of variables, hence locally to let the global behaviour emerge. In this case, if we have a high-pass resonant filter with input gain, resonance, and cut-off parameters, regarding the characteristic of amplitude, input gain and cut-off counterbalance each other when directly proportional, while they contribute to an imbalance when inversely proportional: respectively, the first configuration tends towards equilibrium, while the second one tends to be far from equilibrium. The same kind of relationship takes place between resonance and cut-off, while input gain and resonance tend towards an imbalance when directly proportional, and towards balance when inversely proportional.

It is possible to have an arbitrary number of positive and negative feedback relationships in this type of networks. Otherwise, a simple procedure to have approximately an equal number of positive and negative feedback relationships is to switch the relationship every other pair of variables. If we have a system with parameters A, B, C, D and positive feedback between $A \leftrightarrow B$, then we will have negative feedback between $B \leftrightarrow C$ and again positive feedback between $C \leftrightarrow D$, and so on.

Lastly, the same set of variables could be mapped using the same principles over several chains of relationships, respectively representing different relationships based on different characteristics such as amplitude, spectral tendency, or noisiness. The human agent can then select the desired environment to operate and affect the system state variable and regions of adaptation. Furthermore, the environments can be interpolated to have a multi-dimensional space where it is possible to transition over different control chains.

5.3 Reduced intervention

The *reduced intervention* performance modality is the practical realisation of the concept of *losing control to gain complexity* explained in subsection 1.3.3 Losing control to gain complexity. This approach of human-machine interaction with autonomous systems deliberately seeks the least interference so that the machine can fully express itself. Unlike the cybernetic improvisation approach examined in section 5.1 Cybernetic improvisation where the relational network is established especially at the low level, the reduced intervention modality aims at building relationships between the human and the machine that are functions of high-level information

and analysis frames that extend over long periods.

The particular features of the output that are analysed and the consequent actions to be made significantly depend on the peculiarities of each performance. The human-machine interfacing, the trends displayed by dynamical behaviours of the system, but also the physical conditions and the characteristics of the environment where the work is performed are all determinant factors that inform the strategy for the high-level interaction chain. Some of the case studies follow these performance principles, and they show how physical, technical, and aesthetic aspects of the work affect the control (or lack thereof) strategy. The analysis and examination that performers deploy concern the complexity in the evolutions of systems; the coherence and completeness of formal developments; the tendencies towards the emergence of sound or silence; the variations in the size of the *edge of chaos* regions; and more.

Despite the high-level nature of this approach, the reduced intervention performance can still be formalised following the cybernetic principles of buffering, positive feedback, and negative feedback or, more generally, through articulated or straightforward chains of *if-then-else* conditions applied to the analysis criteria mentioned above. Similarly, constraints can also be used to drive the control process and the outcome towards specific targets or paths.

5.3.1 Case study: *Single-Fader Versatility* (2016)

Single-Fader Versatily is a human-machine interaction performance implementing the idea of *cybernetic mapping* and *reduced intervention* discussed in section 5.2 Cybernetic mapping and in the previous section. The machine is a feedback network containing eight semi-autonomous agents that include audio processing techniques such as state-variable filtering, recursive non-linear distortion, audio-driven granulation, sampling, pulse-width modulation, and frequency shifting modulation. The agents are semi-autonomous and can thus self-modulate their internal variables within the ranges set by the cybernetic mapping.

The human performer can rewire the network by switching among fully, quasi-full, circular, and diagonal (identity) topologies. The performer can also transition between *open* and *closed* configurations by modulating the amount of external signals from the environment flowing inside the network. Furthermore, the human performer can vary the output amplitude of each agent, which is a systemic action as it affects the amount of recirculating signals in the feedback loops, and consequently the system as a whole. Most importantly, the performer can operate a single fader to affect the variables of the agents and the range of adaptation and self-modulation.

For autonomous feedback systems, especially for closed systems with no interaction with the environment, a requirement for the realisation of the machine and its evolutions is self-oscillation. Arguably, autonomous music systems should be able to modulate the overall amplitude, even by attenuating it for extended periods, but a sufficient amount of energy must always recirculate within the network to make sure that sounds and evolutions can emerge continuously for an indefinite length. For this self-oscillating system, the cybernetic mapping

among variables concerns the amplitude relationships, and it aims at having an approximately equal number of positive and negative feedback relationships so that self-oscillation is maintained while having feedback coefficients slightly above the stability threshold. The system is then kept under control using adaptive compression units whose analysis modality can switch between RMS and peak measurements.

Lastly, the specific performance modality relates to the *limited intervention* approach discussed in the next section. In particular, the performer attempts to influence the system as little as possible, and major variations in the state variables take place when a dynamical behaviour has been explored sufficiently, or when a drastic drop on the overall complexity of the output occurs.

This project was last performed in Vienna, Austria, on the 29th of June 2019. Despite the low quality of the audio, the following video may still be helpful to understand the work: https://www.youtube.com/watch?v=hIrqmDif5uQ. Accessed on the 29th of August 2019.

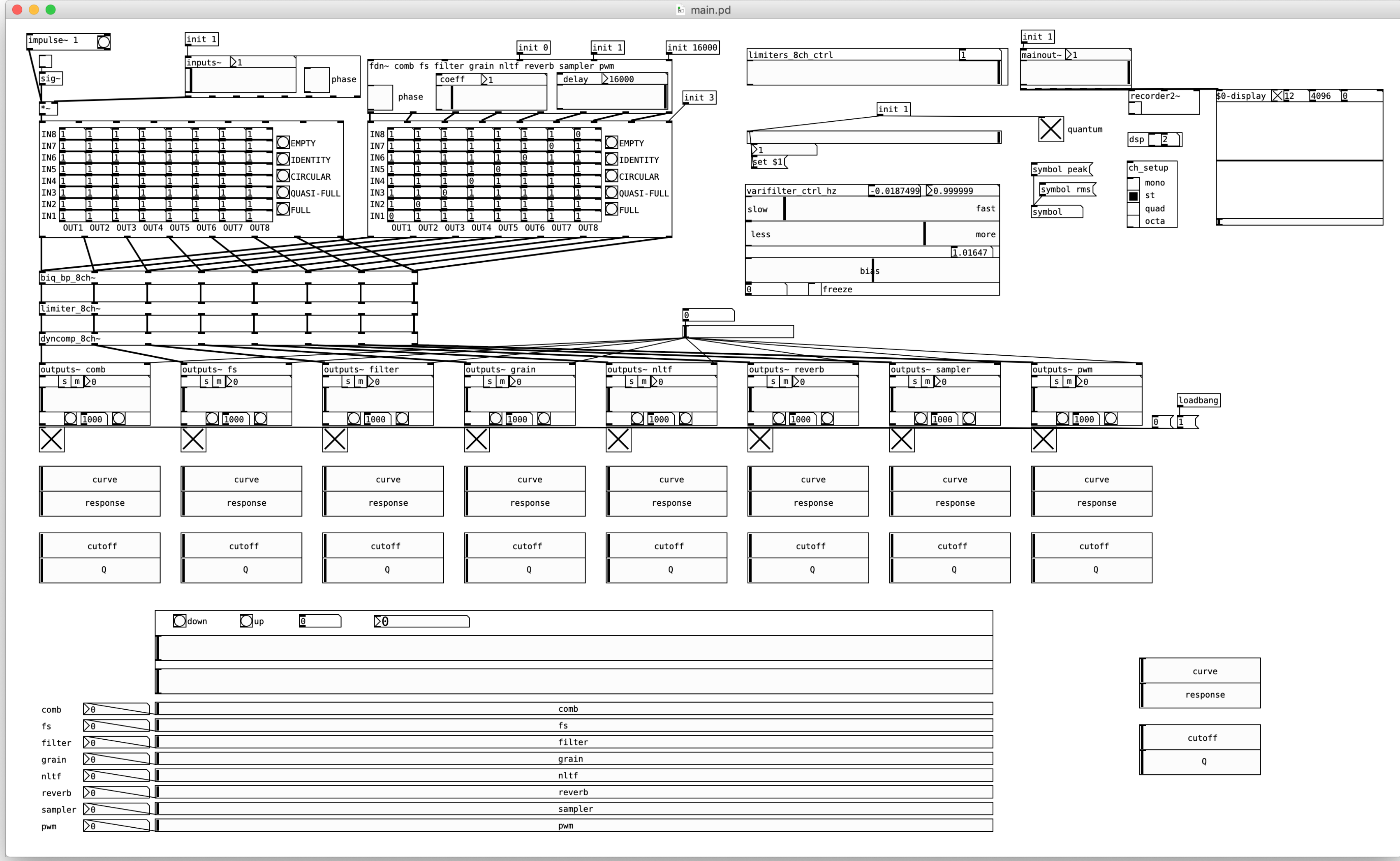

Figure 5.2: *Single-Fader Versatility*: Pure Data main patch.

5.4 Electroacoustic devices and the environment as interfaces

In recursive networks of interdependent components that interact in a nonlinear way, every single element has a potentially critical role that affects the global output of the system. As discussed in chapter 2 Complex adaptive systems, the components of these networks establish a synergetic relationship in a highly distributed self-organising control structure from which emergent behaviours originate. Their intrinsic non-reductionist nature does not allow for analytical procedures where the components are observed individually. In doing so, the relationships among components would be bypassed, and the very essence of the system would be lost. "Linearity is a reductionist's dream, and nonlinearity can sometimes be a reductionist's nightmare," said Melanie Mitchell [Mitchell, 2009]. It is then not possible to quantify the contribution of single elements to global behaviours as the system can only operate and develop as a whole.

The nonlinearity property described in subsection 2.1.1 Properties of feedback mechanisms and defined through the superposition principle is something that extends to different perspectives and observation scales. If the state variable of a complex system at the initial condition is altered and subsequently reset to the original configuration, then the output of the system is likely to be different from the initial one. Complex systems show clear asymmetry as their past shapes their present, which shapes their future, and are a demonstration of the inexorability of time and the irreversibility of the process [Prigogine, 1978].

This nonlinearity protracts towards creating an interrelatedness of the variables in a system, which organically responds to modifications. For linear and non-recursive systems in the sound and music domain, amplitude-related variables are expected to affect amplitude-related characteristics of sound, and frequency-related variables are expected to affect frequency. Examples are the threshold of a dynamic compressor or the shift amount in single-sideband modulation. When these units become part of a nonlinear feedback network, even without adaptation, variations in a single variable are likely to produce variations in all or most aspects characterising the output of the system.

Similarly to how performing human agents become part of a more extensive network if inside the feedback loops, electroacoustic devices and the environment where the performance takes place also become extensions of a meta-system within which they acquire a systemic role. Microphones, loudspeakers, and the environment shape the output of the system and, since *"sound is the interface"* [Di Scipio, 2003], they serve as the core of the connection between the human and the machine, or between the machine and itself. David Tudor and Alvin Lucier pioneered the use of microphones, loudspeakers, sound, and the environment as interfaces already in the '70s. Tudor realised his performance *Microphone* in 1973; Lucier realised *Bird and Person Dyning* in 1975. Analyses of these works can be found in [Sanfilippo, 2012b] and [Sanfilippo and Valle, 2013].

5.4.1 Case study: *Audible Icarus* (2016-2018)

Audible Icarus is a human-machine interaction performance project initially conceived in 2012 [Sanfilippo, 2012a] but developed and fully implemented in 2016. The performance utilises an autonomous ecosystem which is interfaced with a human agent through microphones, loudspeakers, and the environment. This project is indeed the realisation of the concepts addressed in the preceding section and section 5.3 Reduced intervention.

The DSP network consists of four adaptive agents based on the following processing techniques: granulation, sampling, reverberation, and pulse-width modulation. The information processing infrastructure has fixed modules for the extraction of information while the adaptation ranges and most of the variables in the agents are time-variant and dependent on measurements of polarity tendency from local signals in each agent. The agents do not implement digital feedback; hence, they do not self-oscillate unless they are coupled with themselves through the microphones, loudspeakers, and the environment.

The loudspeakers, which can vary in number, are typically placed near the corners and pointed towards the walls to maximise and enhance the resonant characteristics of the environment. There are usually one or two microphones sending signals to the agents with different routing possibilities.

A fundamental aspect of the ecosystemic approach is *calibration*. Of course, in the ecosystemic approach, the investigation of different environments to achieve different behaviours is part of the creative practice, although these systems necessitate operating at optimal conditions to express their maximum potential. Non-equilibrium can be a source of order and self-organisation from disorder [Prigogine, 1978]; complex systems, too, can benefit from *far-from-equilibrium* conditions. In self-oscillating systems, such a condition can be favoured by setting a minimally self-oscillating state, that is, a configuration of the feedback coefficients that allow for enough energy to recirculate without forcing the system towards its attractors. A state of minimal self-oscillation is optimal for the emergence of fluctuations and maximal sensitivity to perturbations. A high sensitivity, in turn, is particularly desirable in the ecosystemic approach as the environment and the observers become a source of perturbation and, consequently, the origin of new dynamics.

One calibration procedure for *Audible Icarus* is to locate as many spots in the space as many agents in the DSP network. To each spot corresponds one agent, which is calibrated individually, bypassing the output of the other agents, after placing the microphones in that position. The calibration aims to find input gains for the single agents so that minimal self-oscillating activity takes place.

The performer interacts with the system by moving in the space, holding the microphones, and exploring resonances and anti-resonances in the environment to drive the system towards different dynamics. For a realisation of the piece, in particular, all loudspeakers are placed on

one side of the space. The performer locates an area in the space far away from the loudspeakers that will be non-self-oscillation zone. By moving closer to the loudspeakers following a straight line, the performer identifies successive spots for the calibration of the agents. Ideally, the spots should be equally distant, and the last spot should be relatively close to the loudspeakers. The result is what in a linear setup would appear as a path of progressively activating agents. Though, due to the nonlinear response of the agents, the exploration of their activation and mutual interference is highly nontrivial.

In this setup, the performer starts from the non-self-oscillation area and walks a straight line towards the loudspeakers until reaching a somewhat near position. The performer then walks back to the initial position, which is where the performance ends. The piece, hence the process of walking towards the loudspeakers and back, takes between 15 and 30 minutes, homogeneously distributed along the path. Concerning the emergence of sound, the performer follows a negative feedback response while walking towards the loudspeakers, and a positive feedback one while walking back. Practically, while the performer moves forward – an action that intrinsically favours the emergence of sound as it technically increases the feedback coefficient – the performer contrasts that emergence by pointing the microphones in different directions. On the other hand, when moving backwards – towards silence – the performer supports the emergence of sound by finding and holding resonant areas with the microphones.

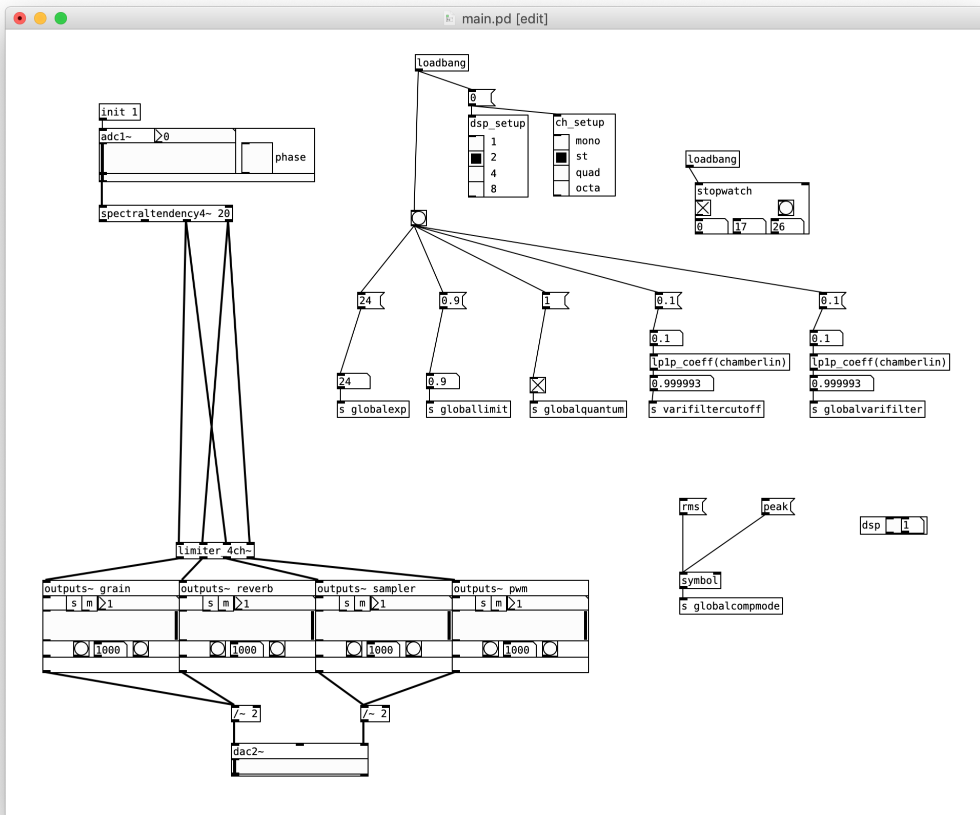

Figure 5.3: *Audible Icarus*: Pure Data main patch.

5.5 Remarks

We have discussed the relationships between human and machine and the interfaces that bridge these two entities in live performance. Within this research, performing with autonomous systems live is considered as a higher-level whole that emerges from relentless and retroactive adaptations between the human and the machine: between their actions, reactions, and organisations.

Such a framework requires an environment through which low-level and high-level interactions can be formalised to connect the human and the machine. The idea of *cybernetic improvisation* is the realisation of such an environment. The principles of cybernetics for self-regulation and control in systems of interacting components constitute the transfer functions that the human performer applies to their sonic context. The term *improvisation*, on the other hand, emphasises that the output of the performer is a continuous function of its input being processed through the cybernetic criteria.

This approach allows for the realisation of networks of interactions between humans and machines, or between several humans, based on simple principles that unfold into nontrivial

behaviours due to circular nonlinearity and adaptation. The human performer becomes an audio and music analysis algorithm that responds to the sonic context according to a set of rules. The information extracted from the output of the machine – as well as environmental perturbations, in some circumstances – can be low-level and high-level. In the first case, the improvisation modalities are applied to characteristics such as loudness, brightness, and noisiness. For high-level characteristics, we have measurements such as event density, dynamicity, and complexity. The relational criteria are represented by the positive and feedback mechanisms, as well as other control operations such as buffering and delaying.

The *cybernetic mapping* approach is based on the same principles described above. It is used to interrelate the variables in a DSP network in a systemic way to allow the performer to interact with digital audio networks in an organic and agile way. This technique identifies the fundamental sonic characteristics that are connected to DSP variables to create positive and negative feedback links. Typically using one-to-many mapping strategies with different mapping functions, the performer can operate on a single parameter to reshape the characteristics of the DSP network while maintaining the relationships among DSP agents. Alternatively, this approach can allow the performer to create relational biases by shifting the feedback relationship of DSP variables couples from positive to negative and vice versa.

The chapter concludes with a section on *reduced intervention* performance modalities, which are practical application and consequence of the concepts explored in subsection 1.3.3 Losing control to gain complexity, and with an analysis on how the performance space and the electroacoustic devices can be deployed as interfaces to explore the aesthetics of the machine.

Chapter 6

Conclusion

> *The future is uncertain, but this uncertainty is at the very heart of human creativity.*
>
> Ilya Prigogine

This research presents an approach for the exploration of new music in live performance, with or without human agents, that focuses on autonomous or semi-autonomous complex adaptive systems (CASes). The central methodology for the implementation of CASes relies on feedback delay networks design with time-variance and nonlinear processing. Particularly, this method follows the paradigms of complex systems implementation based on adaptive agents [Holland et al., 1992, Mitchell, 2006, Mitchell, 2009, Holland, 2014] combined with the principles of second-order cybernetics and self-regulation [Ashby, 1957, Ashby, 1968, Ashby, 1991, Heylighen and Joslyn, 2001, Von Foerster, 2003c, Von Foerster, 2003a].

In chapter 1 Introduction we have seen an overview of the evolution of cybernetics and complex systems with examples from early practitioners working with feedback systems as well as the most advanced techniques for live performance via adaptive systems in recent times. In chapter 1 Introduction we also discuss the aesthetics of CASes, remarking the relationships with technologies and techniques, the role (or lack thereof) of human agents and performers in human-machine interaction with CASes, and the radical novelty inherent in the essence of these systems. CASes exhibit emergent behaviour that is abstract yet coherent, which is why they are especially suited for the exploration of new music.

In chapter 2 Complex adaptive systems, we describe the main features which are common to all systems that are classified as complex and adaptive, while also providing practical examples to bridge the general case and the musical domain. We have seen that networks of competing positive and negative feedback mechanisms and the nonlinearity of the relationships between the components are necessary for designing complex behaviours [Mitchell, 2006]. Chaos is a direct consequence which can result in state spaces with different degrees of order and disorder

[Gleick, 2011], and emergence is a key feature of CASes that is at the heart of the novelty of their behaviours. Finally, it is shown that the ability to adapt and establish a symbiosis with their context or environment can significantly contribute to the overall complexity of these systems [Holland, 1995].

Arguably, chapter 3 Distributed adaptation is the core of this research for its theoretical formulations and results that these theories have provided. The chapter gives a summary of the design principle illustrated in [Holland, 2014]. Holland describes CASes as networks of adaptive agents that, in turn, are networks of interdependent *detecting* and *affecting* infrastructures. The detecting units are specialised for the processing of information from the context to generate self-controlling mechanisms that act upon the affecting units. The affecting units are dedicated to transforming their surrounding environment to reach their goal and maintain high fitness for the global entity.

This chapter includes a discussion on the relationships between information, adaptation, and the structural coupling of system and context. This discussion is essential to establishing a framework where information, context, system, and adaptation – and, consequently, complexity – are interrelated and mutually determining. Within this framework, the notion of *emergent infrastructures* is formulated, for which the detecting and affecting networks of the adaptive agents can vary through the structural coupling of system and context.

The idea of *distributed adaptation* is presented, associated to the notion of *evolvability* [Wagner and Altenberg, 1996], which consists of applying adaptation to key nodes throughout all layers. The goal is to realise systems that are time-variant and emergent in the local detecting and affecting infrastructures, but also in their global compositions through morphing network topologies and multimodal agents. Systems can then compose themselves at all levels effectuating the notion of autopoiesis.

Lastly, in chapter 3 Distributed adaptation, a set of CPU-efficient information processing algorithms for real-time applications is provided. The set includes low-level and high-level feature extraction algorithms, many of which are based on an original design. These units constitute a comprehensive collection of sensing devices that can be combined to build the necessary detecting infrastructures.

In chapter 4 Audio processing techniques and DSP implementation we cover aspects related to the affecting infrastructures of CASes. Hence, this chapter discusses some of the most relevant audio processing techniques, including some unconventional ones, that have been used throughout this research, as well as the stability processing methods necessary for the proper operability of self-oscillating systems. The end of the chapter is dedicated to technical aspects of the software implementation of DSP networks in real-time.

In chapter 5 Performance modalities and interfaces, we present a formalisation of human-machine interaction modalities as well as human-machine interfacing strategies that follow the principles of cybernetics [Heylighen and Joslyn, 2001]. This formalisation is useful for

the full integration of human agents within CASes design, providing the human performers with a specialised role and systemic goal within the whole. The extension of CASes through cybernetically linked human agents is a practice that has enhanced the capabilities of improvised performance in the case studies discussed above.

Summary of the answers to the key research questions set out in the introduction are explained below.

– *How to realise music systems with an abstract yet structurally coherent and contextually complex output that display organicity and resemble aliveness?*

The output of CASes is inherently coherent as it is the outcome of causal processes in the sequence of states. Each state is the result of deterministic and well-defined rules – even self-defined rules, concerning distributed adaptation – that are applied to the former state, which in turn carries the whole history of the system. The interdependency between the agents that constitute the network makes local perturbations spread like a chain reaction towards all other nodes to then bounce back into their origins. The sonic streams that make up the global output of the system are mutually shaping themselves where individual variations trigger evolutions for all the remaining streams. These nonlinear relationships and the circularity of the mechanism allow for a structural coupling between the system and context/environment.

The system operates as a single voice that is constantly being pulled from different directions, and that is responding to external as well as internal stimuli, relentlessly, to produce complex adaptations. The organicity of the process, that is, the ability of the music system to perform an analysis and to adapt to survive musically – to maintain musical complexity – is the realisation of a performance environment with high potential for live performance due to the aliveness of the process itself.

– *How to design music organisms so that artificial expressiveness and formal developments emerge and generate musical behaviours that contribute to the ongoing exploration at the edges of new music practices?*

The research presented in this thesis originates from the need to more deeply connect the human and the machine in live performance so that higher musical complexity is achieved, and from the intuition that the aesthetics of the machine can sometimes represent the aesthetics of the human more effectively. After several years of creative practice with complex systems, the gradual shift from performed music to self-performing music resulted in performance environments – with or without human intervention – with compelling musical outputs that have extended and better-matched the author's aesthetics.

The advantages of using CASes for the generation of sounds and formal developments lie in their emergent nature. Their emergent quality can produce radically novel behaviours that favour the discovery of novel formal developments. This radical novelty is enhanced by designing

networks of interactions based on analysis and adaptation criteria that do not strictly pre-existing styles or perceptual criteria correlated to humans. Networks of abstract relationships can be deployed so that machine aesthetics emerges and provides unconventional forms. Musical sense, on the other hand, is maintained through an artificial expressiveness resulting from the continuous circular adaptations and evolutions and the fundamental requirement of keeping the system at the edge of chaos, hence creating a nonlinear interplay between pattern-formation and form-morphing.

– *How to create audio networks that are responsible for their structure and organisation where a substantial autonomy expands the paradigm of human-machine interaction?*

The combination of adaptation and autopoiesis through evolvability and goal-orientedness has provided a framework to design music systems that can profoundly reshape their organisation and structure. We have discussed a crucial paradigm shift, from composing interactions to self-composing interactions, showing that the systems described above are capable of autonomously determining their networks of relationships. Providing music systems with such a high degree of autonomy has had the consequence of a deeper complexity, variety, and unpredictability: music systems of these kinds can now be the source of substantial formal changes that do not require human intervention.

Paradoxically, providing the machine with more independence, has the effect of creating a stronger interdependency in the human-machine couple. The human performer that operates with such autonomous systems is in a performance condition that is comparable to a duo improvisation setup: if the human accepts to perform with the machine, then the machine has the highest degree of freedom. We can see how a new kind of musicianship emerges: the virtuosity of the human agent, in this case, is strongly connected to the idea of *losing control*. The conscious action of *not doing*, in some cases, becomes the highest expression of human musicality as it may allow the machine to unfold in a way that is the aesthetically most compelling in that specific context. The machine then becomes an extension of human aesthetics, and the performance is a delicate and harmonious balance between implicit and explicit interferences.

6.1 Future work

The future work for this research will focus on further explorations of the *distributed adaptation* techniques in music performance. The design of new information processing algorithms, as well as original audio processing techniques, will be crucial too. Furthermore, considering the key role played by electroacoustic devices in feedback systems, different kinds of transducers and microphones will contribute to expanding the set of possibilities offered by these techniques.

The Edge of Chaos library will be maintained and expanded as much as possible. The

future developments for the software library will aim at turning it into a CASes generator. Within the Faust environment, all the elements in the library will be interfaced through a single command-line operation that will describe the fundamental characteristics of a complex network. Following Kauffman's approach with random Boolean networks [Kauffman, 1969, Kauffman, 1993], CASes can be generated using a single seed that feeds a set of iterated nonlinear functions to output uncorrelated pseudo-random values. These values would then be used to select the nodes in the affecting and detecting networks of each agent, the relationships between agents, the network topologies, and more. The resulting CASes would then be *black boxes* whose structures and organisations are randomly determined, although the systems would be entirely deterministic in their behaviours. Furthermore, the DSP network that is generated randomly can still be explored by the user using Faust's block diagram generation.

Outside of the artistic domain, audio CASes may have critical applications in the field of game development. The audio environment of games is a fundamental aspect of the realisation of a realistic experience; the application of intelligent virtual agents within the framework of virtual sound and emergent perception for audio environments in video games has been discussed in [Garner and Jordanous, 2016]. CASes can be coupled with their environment, which would become an extension of their natural behaviour. CASes within game development could represent specific sonic sources as sound-generating adaptive agents that shape themselves based on the surrounding conditions, consequently enhancing the user experience who would be an active part of the generation of the auditory context.

Music improvisers may also use CASes as agents to practice duos or group improvisations. By applying specific constraints, CASes may be driven towards particular music styles to meet the preferences of a broader range of musicians. Furthermore, considering the expressivity that CASes exhibit, they can be implemented as musical instruments for persons with disabilities by allowing formal developments through the manipulation of high-level parameters, for example, implementing specialised brain-computer interfaces such as those proposed in [Miranda and Brouse, 2005].

Appendix A

Edge of Chaos library code

Edge of Chaos is an open-source software library written in Faust containing the fundamental building blocks for the implementation of audio complex adaptive systems. The library consists of ten modules, organised by functionality, and is available on GitHub under the Gnu General Public License v2.0.[1]

The ten modules are *alleoc.lib, auxiliary.lib, delays2.lib, edgeofchaos.lib, filters2.lib, information.lib, maths2.lib, oscillators2.lib, outformation.lib, stability.lib*. *Alleoc.lib* gives access to all the library modules from a single point. *Auxiliary.lib* is a small module containing testing functions and inspecting functions useful for debugging. *Delays2.lib* essentially uses the same delay functions in Faust's standard library, although the parameters are expressed in seconds rather than samples. *Edgeofchaos.lib* gives access to all the library modules through a series of environments. *Filters2.lib* contains first and second-order filters implemented through biquadratic sections or zero-delay feedback topologies. *Information.lib* contains low-level and high-level information processing algorithms, some of which have been presented in section 3.3 Information processing techniques. *Maths2.lib* contains several math functions, many-valued logic operators, functions for the implementation of networks with different topologies, as well as different kinds of mapping functions to be used in adaptive infrastructures. *Oscillators2.lib* contains some of the fundamental band-limited oscillators based on band-limited impulse trains as well as some self-oscillating systems for the generation of signals. *Outformation.lib* contains a set of algorithms for the transformation of signals, some of which have been described in section 4.1 Sound transformations. *Stability.lib* contains different functions based on different techniques for the power-preserving and stability of self-oscillating feedback systems.

A.1 alleoc.lib

```
// ==========================================================================
```

[1] https://github.com/dariosanfilippo/edgeofchaos. Accessed on the 29th of August 2019.

```
// ========== alleoc.lib ==========================================================
// ================================================================================
//
// Access to all Edge of Chaos library modules from a single point.
//
// Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario at gmail dot com>
// All rights reserved.

declare name "All Edge of Chaos modules";
declare author "Dario Sanfilippo";
declare copyright "Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario
    at gmail dot com>";
declare version "1.0.0";
declare license "GPLv2.0";

import("auxiliary.lib");
import("delays2.lib");
import("filters2.lib");
import("information.lib");
import("maths2.lib");
import("oscillators2.lib");
import("outformation.lib");
import("stability.lib");
```

A.2 auxiliary.lib

```
// ================================================================================
// ========== auxiliary.lib ======================================================
// ================================================================================
//
// Auxiliary functions library for testing, analysis, inspection, and debugging.
//
// The environment prefix is "au".
//
// List of functions:
//
//     dirac,
//     inspect,
```

```
//     step.
//
// Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario at gmail dot com>
// All rights reserved.

declare name "Auxiliary Library";
declare author "Dario Sanfilippo";
declare copyright "Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario
      at gmail dot com>";
declare version "1.0.0";
declare license "GPLv2.0";

au = library("auxiliary.lib");

// au.dirac; ------------------------------------------------------------------
//
// Dirac impulse: full-amplitude, 1-sample impulse.
// y[n] = 1 if n = 0; 0 otherwise.
//
// 0 inputs.
//
// 1 outputs:
//     y[n].
//
dirac = 1 - 1';
// ----------------------------------------------------------------------------

// au.inspect(i, lower, upper, x[n]); ----------------------------------------
//
// Signal inspector: it displays the value of a signal at block-size rate.
//
// 1 inputs:
//     x[n].
//
// 1 outputs:
//     x[n].
//
// 3 compile-time arguments:
```

```
//     "i", integer identifier;
//     "lower", lower display limit;
//     "upper", upper display limit.
//
inspect(i, lower, upper) =
      _ <: _ ,
          vbargraph("sig_%i [style:numerical]", lower, upper) : attach;
// ---------------------------------------------------------------------

// au.step(M); ---------------------------------------------------------
//
// Step function: full-amplitude, M-sample step.
// y[n] = 1 if 0 <= n < M; 0 otherwise.
//
// 0 inputs.
//
// 1 outputs:
//     Step().
//
// 1 compile-time arguments:
//     "M", step size in samples.
//
step(M) = 1 <:  _ - (_ @ M);
// ---------------------------------------------------------------------
```

A.3 delays2.lib

```
// =====================================================================
// ========== delays2.lib ==============================================
// =====================================================================
//
// Delay line functions library with samplerate-independent delay parameters
// based on Faust's delay lines for integer and fractional delays.
//
// The environment prefix is "d2".
//
// List of functions:
//
```

```
//     del,
//     del_lin,
//     del_pol,
//     del_smo.
//
// Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario at gmail dot com>
// All rights reserved.

declare name "Delays Library";
declare author "Dario Sanfilippo";
declare copyright "Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario
      at gmail dot com>";
declare version "1.0.0";
declare license "GPLv2.0";

de = library("delays.lib");
d2 = library("delays2.lib");
ma = library("maths.lib");
m2 = library("maths2.lib");

// d2.del(size, D[n], x[n]); ----------------------------------------
//
// Non-interpolating delay line: the signal is delayed by a number of
// samples that is the closest integer of the specified delay in seconds
// times the samplerate.
//
// 2 inputs:
//    D[n], delay amount in seconds (approximately);
//    x[n].
//
// 1 outputs:
//    y[n], delayed x[n].
//
// Compile-time arguments:
//    "size", delay line size and maximum delay in seconds.
//
del(size, del, in) = de.delay(size * ma.SR, del * ma.SR, in);
// ------------------------------------------------------------------
```

```
// d2.del_lin(size, D[n], x[n]); ----------------------------------------
//
// Interpolating delay line: the input signal is delayed by a delay
// expressed in seconds where fractional-sample delays are achieved
// through linear interpolation.
//
// 2 inputs:
//    D[n], delay amount in seconds;
//    x[n].
//
// 1 outputs:
//    y[n], delayed x[n].
//
// Compile-time arguments:
//    "size", delay line size and maximum delay in seconds.
//
del_lin(size, del, in) = de.fdelay(size * ma.SR, del * ma.SR, in);
// ----------------------------------------------------------------------

// d2.del_pol(size, D[n], x[n]); ----------------------------------------
//
// Interpolating delay line: the input signal is delayed by a delay
// expressed in seconds where fractional-sample delays are achieved
// through 4th-order polynomial interpolation.
//
// 2 inputs:
//    D[n], delay amount in seconds;
//    x[n].
//
// 1 outputs:
//    y[n], delayed x[n].
//
// Compile-time arguments:
//    "size", delay line size and maximum delay in seconds which is the
//     closest power-of-two number of samples representing the delay in seconds.
//
del_pol(size, del, in) = de.fdelayltv(4, s, d, in)
```

```
    with {
        s = size * ma.SR : m2.round_pow2;
        d = del * ma.SR;
        //d = del * ma.SR : max((4 - 1) / 2);
    };
// ------------------------------------------------------------------------

// d2.del_smo(size, IT[n], D[n], x[n]); -----------------------------------
//
// Delay line with variable delay and no clicks or Doppler effect: the
// mechanism works by interpolating between the output of two delay lines.
//
// 3 inputs:
//    IT[n], interpolation time in seconds;
//    D[n], delay amount in seconds;
//    x[n].
//
// 1 outputs:
//    y[n], delayed input.
//
// Compile-time arguments:
//    "size", delay line size and maximum delay in seconds.
//
del_smo(size, itime, del, in) = de.sdelay(s, it, d, in)
    with {
        s = size * ma.SR;
        it = itime * ma.SR;
        d = del * ma.SR;
    };
// ------------------------------------------------------------------------
```

A.4 edgeofchaos.lib

```
// ========================================================================
// ========== edgeofchaos.lib =============================================
// ========================================================================
//
// This file provides access to all the Edge of Chaos library modules through a
```

```
// series of environments.
//
// Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario at gmail dot com>
// All rights reserved.

declare name "Access to the Edge of Chaos library environments";
declare author "Dario Sanfilippo";
declare copyright "Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario
      at gmail dot com>";
declare version "1.0.0";
declare license "GPLv2.0";

au = library("auxiliary.lib");
d2 = library("delays2.lib");
f2 = library("filters2.lib");
ip = library("information.lib");
m2 = library("maths2.lib");
st = library("stability.lib");
o2 = library("oscillators2.lib");
op = library("outformation.lib");
ed = library("alleoc.lib");
```

A.5 filters2.lib

```
// ==========================================================================
// ========= filters2.lib ===================================================
// ==========================================================================
//
// Filters library containing bilinear transform and topology preserving
// transform implementations (zero-delay feedback) of allpass, lowpass,
// highpass, bandpass, bandstop, shelving, and state-variable filters.
// Furthermore, there are implementations of crossovers, comb-integrator
// circuits, analytic filters, and integrators, among others.
//
// The environment prefix is "f2".
//
// List of functions:
//
```

```
//      analytic,
//      apbi,
//      apblti,
//      biquad,
//      bpbi,
//      bp2blti,
//      bsbi,
//      cic,
//      hpbi,
//      hpblti,
//      hp1p,
//      hp1pint,
//      hp1praw,
//      hp1p1z,
//      hp1p1zraw,
//      int_clip,
//      int_eu_b,
//      int_eu_clip,
//      int_eu_f,
//      int_trap,
//      int_trap_clip,
//      integrator,
//      leaky,
//      lpbi,
//      lpblti,
//      lp1p,
//      lp1pint,
//      lp1praw,
//      lp1p1z,
//      lp1p1zraw,
//      sah_inv,
//      slew_limiter,
//      svfblti,
//      svf2blti,
//      xover_butt,
//      xover1p1z,
//      xover1p1z_ada,
//      xover1praw,
```

```
//      xover1p1zraw,
//      xover2p2z.
//
// Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario at gmail dot com>
// All rights reserved.

declare name "Filters Library";
declare author "Dario Sanfilippo";
declare copyright "Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario
      at gmail dot com>";
declare version "1.0.0";
declare license "GPLv2.0";

ba = library("basics.lib");
fi = library("filters.lib");
f2 = library("filters2.lib");
ip = library("information.lib");
ma = library("maths.lib");
m2 = library("maths2.lib");
ro = library("routes.lib");
st = library("stability.lib");

// f2.analytic(x[n]); ------------------------------------------------------------
//
// Analytic signal using a Hilbert filter by Olli Niemitalo:
// dsp.stackexchange.com/questions/37411/iir-hilbert-transformer/59157#59157.
// Four of the following sections cascaded for each of the two outputs, plus an
// extra delay for the imaginary path:
//      y[n] = c * (x[n] + y[n - 2]) - x[n - 2].
//
// 1 inputs:
//      x[n].
//
// 2 outputs:
//      y1[n], real part;
//      y2[n], imaginary part (-90-degree shift).
//
analytic(x) =  real ,
```

```
        imaginary
    with {
        im_c = (    0.47944111608296202665 ,
                    0.87624358989504858020 ,
                    0.97660296916871658368 ,
                    0.99749940412203375040);
        re_c = (    0.16177741706363166219 ,
                    0.73306690130335572242 ,
                    0.94536301966806279840 ,
                    0.99060051416704042460);
        tf(c, y, x) = c * (x + y') - x'';
        imaginary = x' : seq(i, 4,  tf(ba.take(i + 1, im_c))
                                    ~ _);
        real = x : seq(i, 4,    tf(ba.take(i + 1, re_c))
                                ~ _);
    };
// ------------------------------------------------------------------------------------------------------------

// f2.apbi(CF[n], x[n]); -----------------------------------------------------------------------
//
// Biquad allpass (design by Robert Bristow-Johnson).
//
// 2 inputs:
//    CF[n], cut-off frequency in Hz;
//    x[n].
//
// 1 outputs:
//    y[n], -90-degree shift at CF[n].
//
apbi(cf, in) = f2.biquad(a0, a1, a2, b1, b2, in)
    with {
        cf1 = st.clip(5, m2.ny - 5, cf);
        q1 = .707;
        alpha = sin(m2.w(cf1)) / (2 * q1);
        norm = 1 + alpha;
        a0 = (1 - alpha) / norm;
        a1 = -1 * cos(m2.w(cf1)) * 2 / norm;
        a2 = (1 + alpha) / norm;
```

```
        b1 = -1 * cos(m2.w(cf1)) * 2 / norm;
        b2 = (1 - alpha) / norm;
    };
// ------------------------------------------------------------------------

// f2.apblti(CF[n], x[n]); -----------------------------------------------
//
// One-pole-one-zero allpass based on zero-delay feedback with bilinear
// transform integrator (design by Zavalishin).
//
// 2 inputs:
//     CF[n], cut-off frequency in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], -90-degree shift at CF[n].
//
apblti(cf, in) = ( cf ,
                   in) <:  f2.lpblti ,
                           f2.hpblti : -;
// ------------------------------------------------------------------------

// f2.biquad(a0[n], a1[n], a2[n], b1[n], b2[n], x[n]); ---------------------
//
// Biquad filter section.
//
// 6 inputs:
//     x[n] coeff (a0[n]);
//     x[n - 1] coeff (a1[n]);
//     x[n - 2] coeff (a2[n]);
//     y[n - 1] coeff (b1[n]);
//     y[n - 2] coeff (b2[n]);
//     x[n].
//
// 1 outputs:
//     y[n] = a0[n] * x[n] + a1[n] * x[n - 1] + a2[n] * x[n - 2]
//            - b1[n] * y[n - 1] - b2[n] * y[n - 2].
//
```

```
biquad(a0, a1, a2, b1, b2, x) = fir : +
                                       ~ iir
    with {
        fir = a0 * x + a1 * x' + a2 * x'';
        iir(fb) = -b1 * fb - b2 * fb';
    };
// ------------------------------------------------------------------------

// f2.bpbi(CF[n], Q[n], x[n]); ----------------------------------------------
//
// Biquad bandpass (design by Robert Bristow-Johnson).
//
// 3 inputs:
//   CF[n], cut-off frequency in Hz;
//   Q[n], Q-factor;
//   x[n].
//
// 1 outputs:
//   y[n], bandpassed x[n].
//
bpbi(cf, q, in) = f2.biquad(a0, a1, a2, b1, b2, in)
    with {
        cf1 = st.clip(5, m2.ny - 5, cf);
        q1 = max(q, .001);
        alpha = sin(m2.w(cf1)) / (2 * q1);
        norm = 1 + alpha;
        a0 = alpha / norm;
        a1 = 0;
        a2 = -alpha / norm;
        b1 = cos(m2.w(cf1)) * -2 / norm;
        b2 = (1 - alpha) / norm;
    };
// ------------------------------------------------------------------------

// f2.bp2blti(CF[n], Q[n], x[n]); -------------------------------------------
//
// Two-pole-two-zero normalised band-pass (shortcut from the SVF TPT 2nd-order by
// Zavalishin / Pirkle).
```

```
//
// 3 inputs:
//     CF[n], cut-off frequency in Hz;
//     Q[n], Q-factor;
//     x[n].
//
// 1 outputs:
//     y[n], bandpassed x[n].
//
bp2blti(cf, q, x) = f2.svf2blti(cf, q, 1, x) : (! ,
                                                 ! ,
                                                 ! ,
                                                 _ ,
                                                 ! ,
                                                 ! ,
                                                 ! ,
                                                 ! ,
                                                 ! ,
                                                 !);
// ----------------------------------------------------------------------------

// f2.bsbi(CF[n], Q[n], x[n]); ----------------------------------------------
//
// Biquad bandstop (design by Robert Bristow-Johnson).
//
// 3 inputs:
//     CF[n], cut-off frequency in Hz;
//     Q[n], Q-factor;
//     x[n].
//
// 1 outputs:
//     y[n], bandstopped x[n].
//
bsbi(cf, q, in) = f2.biquad(a0, a1, a2, b1, b2, in)
    with {
        cf1 = st.clip(5, m2.ny - 5, cf);
        q1 = max(q, .001);
        alpha = sin(m2.w(cf1)) / (2 * q1);
```

```
        norm = 1 + alpha;
        a0 = 1 / norm;
        a1 = -1 * cos(m2.w(cf1)) * 2 / norm;
        a2 = 1 / norm;
        b1 = -1 * cos(m2.w(cf1)) * 2 / norm;
        b2 = (1 - alpha) / norm;
    };
// -----------------------------------------------------------------------

// f2.cic(N, CF[n], x[n]); -----------------------------------------------
//
// Comb-integrator circuit lowpass filter.
// Based on Eric Lyon's: https://www.dsprelated.com/showarticle/1337.php.
//
// 2 inputs:
//    CF[n], cut-off frequency in Hz;
//    x[n].
//
// 1 outputs:
//    y[n], lowpassed x[n].
//
// 1 compile-time arguments:
//    "N", (integer) order of the filter.
//
cic(N, cf, x) = x : seq(i, N, m2.delta(1, .5 / cf) :
    fi.pole(1)) / (.5 / cf * ma.SR) ^ N;
// -----------------------------------------------------------------------

// f2.hpbi(CF[n], x[n]); -------------------------------------------------
//
// Biquad highpass (design by Robert Bristow-Johnson).
//
// 2 inputs:
//    CF[n], cut-off frequency Hz;
//    x[n].
//
// 1 outputs:
//    y[n], highpassed x[n].
```

```
//
hpbi(cf, in) = f2.biquad(a0, a1, a2, b1, b2, in)
    with {
        cf1 = st.clip(5, m2.ny - 5, cf);
        q1 = .707;
        alpha = sin(m2.w(cf1)) / (2 * q1);
        norm = 1 + alpha;
        a0 = ((1 + cos(m2.w(cf1))) / 2) / norm;
        a1 = -1 * (1 + cos(m2.w(cf1))) / norm;
        a2 = ((1 + cos(m2.w(cf1))) / 2) / norm;
        b1 = cos(m2.w(cf1)) * -2 / norm;
        b2 = (1 - alpha) / norm;
    };
// ---------------------------------------------------------------------------

// f2.hpblti(CF[n], x[n]); --------------------------------------------------
//
// One-pole-one-zero highpass based on zero-delay feedback with bilinear
// transform integrator (design by Zavalishin).
//
// 2 inputs:
//    CF[n], cut-off frequency in Hz;
//    x[n].
//
// 1 outputs:
//    y[n], highpassed x[n].
//
hpblti(cf, in) = in - f2.lpblti(cf, in);
// ---------------------------------------------------------------------------

// f2.hp1p(CF[n], x[n]); ----------------------------------------------------
//
// One-pole highpass (design by Chamberlin).
//
// 2 inputs:
//    CF[n], cut-off frequency in Hz;
//    x[n].
//
```

```
// 1 outputs:
//     y[n], highpassed x[n].
//
hp1p(cf, in) = + (in * a0)
               ~ * (b1)
      with {
           a0 = 1 + b1;
           b1 = exp((.5 - cf / ma.SR) * -2 * ma.PI) * -1;
      };
// ------------------------------------------------------------------------

// f2.hp1pint(CF[n], x[n]); ------------------------------------------------
//
// One-pole highpass based on backward Euler's integration
// (design by Zavalishin).
//
// 2 inputs:
//     CF[n], cut-off frequency in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], highpassed x[n].
//
hp1pint(cf, in) = in - f2.lp1pint(cf, in);
// ------------------------------------------------------------------------

// f2.hp1praw(CF[n], x[n]); ------------------------------------------------
//
// One-pole highpass with raw coefficients in the [0; 1] where the extremes
// correspond to DC and Nyquist, although the mapping is nonlinear.
//
// 2 inputs:
//     CF[n], cut-off frequency, scalar;
//     x[n].
//
// 1 outputs:
//     y[n], highpassed x[n].
//
```

```
hp1praw(cf, in) =  + ((1 - cf) * in)
                   ~ * (-cf);
// ----------------------------------------------------------------------------

// f2.hp1p1z(CF[n], x[n]); ----------------------------------------------------
//
// One-pole-one-zero highpass (design by Cliff Sparks).
//
// 2 inputs:
//    CF[n], cut-off frequency in Hz;
//    x[n].
//
// 1 outputs:
//    y[n], highpassed x[n].
//
hp1p1z(cf, in) =   + (in * a0 <: _ + _' * a1)
                   ~ * (b1)
      with {
           a0 = (1 + b1) / 2;
           a1 = -1;
           b1 = m2.div((1 - sin(m2.w(cf))), cos(m2.w(cf)));
      };
// ----------------------------------------------------------------------------

// f2.hp1p1zraw(CF[n], x[n]); -------------------------------------------------
//
// One-pole-one-zero highpass with raw coefficients in the [0; 1] where the
// extremes correspond to DC and Nyquist, although the mapping is nonlinear.
//
// 2 inputs:
//    CF[n], cut-off frequency, scalar;
//    x[n].
//
// 1 outputs:
//    y[n], highpassed x[n].
//
hp1p1zraw(cf, in) =    + (in * (1 - cf) <: _ - _')
                       ~ * (cf * -2 + 1);
```

```
// ---------------------------------------------------------------------------

// f2.int_clip(L[n], H[n], x[n]); -------------------------------------------
//
// Special case of bounded backward Euler's integration where CF[n] = 1 / (2PI).
//
// 3 inputs:
//    L[n], lower limit;
//    H[n], upper limit;
//    x[n].
//
// 1 outputs:
//    y[n], integrated and clipped x[n].
//
int_clip(lower, upper, in) =  (+ (in / ma.SR) : st.clip(lower, upper))
                              ~ _;
// ---------------------------------------------------------------------------

// f2.int_eu_b(CF[n], x[n]); ------------------------------------------------
//
// Backward Euler's integration.
//
// 2 inputs:
//    CF[n], in Hz, sets the input gain as the angular frequency;
//    x[n].
//
// 1 outputs:
//    y[n], integrated x[n].
//
int_eu_b(cf, x) = + (x * m2.w(cf))
                  ~ _;
// ---------------------------------------------------------------------------

// f2.int_eu_clip(L[n], H[n], CF[n], x[n]); ---------------------------------
//
// Bounded backward Euler's integration.
//
// 4 inputs:
```

```
//     L[n], lower bound;
//     H[n], upperbound;
//     CF[n], in Hz, sets the input gain as the angular frequency;
//     x[n].
//
// 1 outputs:
//     y[n], integrated x[n].
//
int_eu_clip(l, u, cf, x) = (+ (x * m2.w(cf)) : st.clip(l, u))
                           ~ _;
// ---------------------------------------------------------------------------

// f2.int_eu_f(CF[n], x[n]); ------------------------------------------------
//
// Forward Euler's integration.
//
// 2 inputs:
//     CF[n], in Hz, sets the input gain as the angular frequency;
//     x[n].
//
// 1 outputs:
//     y[n], integrated x[n].
//
int_eu_f(cf, x) =  perform
                   ~ _ :    ! ,
                            _
    with {
        perform(fb) =   x * m2.w(cf) + fb ,
                        fb;
    };
// ---------------------------------------------------------------------------

// f2.int_trap(CF[n], x[n]); ------------------------------------------------
//
// Trapezoidal integration.
//
// 2 inputs:
//     CF[n], in Hz, sets the input gain as the angular frequency;
```

```
//      x[n].
//
// 1 outputs:
//      y[n], integrated x[n].
//
int_trap(cf, x) =  + (x * m2.w(cf) / 2 <: _ + _')
                   ~ _;
// ------------------------------------------------------------------------------

// f2.int_trap_clip(L[n], H[n], CF[n], x[n]); ------------------------------------
//
// Bounded trapezoidal integration.
//
// 4 inputs:
//      L[n], lower bound;
//      H[n], upper bound;
//      CF[n], in Hz, sets the input gain as the angular frequency;
//      x[n].
//
// 1 outputs:
//      y[n], integrated x[n].
//
int_trap_clip(l, u, cf, x) =
      x * m2.w(cf) / 2 <:  _ ,
                          _' :>   (+ : st.clip(l, u))
                                  ~ _;
// ------------------------------------------------------------------------------

// f2.integrator(x[n]); ---------------------------------------------------------
//
// Special case of backward Euler's integration with CF = 1 / (2PI).
//
// 1 inputs:
//      x[n].
//
// 1 outputs:
//      y[n], integrated x[n].
//
```

```
integrator(x) =    + (x / ma.SR)
                   ~ _;
// -----------------------------------------------------------------------------

// f2.leaky(R[n], x[n]); --------------------------------------------------------
//
// Leaky integrator based on the tau constant, that is, an integrator
// with decay specified in seconds. Also special case of backward Euler's
// integration with CF = SR / (2PI).
//
// 2 inputs:
//    R[n], decay of about 9 dB in a desired time specified in seconds;
//    x[n].
//
// 1 outputs:
//    y[n], (leakly) integrated x[n].
//
leaky(t, x) = + (x)
              ~ * (ba.tau2pole(t));
// -----------------------------------------------------------------------------

// f2.lpbi(CF[n], x[n]); --------------------------------------------------------
//
// Biquad lowpass (design by Robert Bristow-Johnson).
//
// 2 inputs:
//    CF[n], cut-off frequency in Hz;
//    x[n].
//
// 1 outputs:
//    y[n], lowpassed x[n].
//
lpbi(cf, in) = f2.biquad(a0, a1, a2, b1, b2, in)
    with {
        cf1 = st.clip(5, m2.ny - 5, cf);
        q1 = .707;
        alpha = sin(m2.w(cf1)) / (2 * q1);
        norm = 1 + alpha;
```

```
            a0 = ((1 - cos(m2.w(cf1))) / 2) / norm;
            a1 = (1 - cos(m2.w(cf1))) / norm;
            a2 = ((1 - cos(m2.w(cf1))) / 2) / norm;
            b1 = cos(m2.w(cf1)) * -2 / norm;
            b2 = (1 - alpha) / norm;
    };
// ------------------------------------------------------------------------

// f2.lpblti(CF[n], x[n]); -------------------------------------------------
//
// One-pole-one-zero lowpass based on zero-delay feedback with bilinear
// transform integrator (design by Zavalishin; Faust code by Oleg Nesterov).
//
// 2 inputs:
//     CF[n], cut-off frequency in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], lowpassed x[n].
//
lpblti(cf, in) =    tick
                    ~  _ :    ! ,
                              _
    with {
            g = tan(m2.w(cf) / 2) / (1 + tan(m2.w(cf) / 2));
            tick(s) = y + v, y
                with {
                    v = (in - s) * g;
                    y = v + s;
                };
    };
// ------------------------------------------------------------------------

// f2.lp1p(CF[n], x[n]); ---------------------------------------------------
//
// One-pole lowpass (design by Chamberlin).
//
// 2 inputs:
```

```
//     CF[n], cut-off frequency in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], lowpassed x[n].
//
lp1p(cf, in) = + (in * a0)
               ~ * (b1)
    with {
        a0 = 1 - b1;
        b1 = exp(m2.w(cf) * -1);
    };
// ------------------------------------------------------------------------

// f2.lp1pint(CF[n], x[n]); ----------------------------------------------
//
// One-pole lowpass based on backward Euler's integration
// (design by Zavalishin).
//
// 2 inputs:
//     CF[n] (cut-off, Hz);
//     x[n].
//
// 1 outputs:
//     y[n], lowpassed x[n].
//
lp1pint(cf, in) =  (in ,
                   _ : + : *(m2.w(cf) : st.clip(0, 2)) : fi.pole(1))
                   ~ * (-1);
// ------------------------------------------------------------------------

// f2.lp1praw(CF[n], x[n]); ----------------------------------------------
//
// One-pole lowpass with raw coefficients in the [0; 1] where the extremes
// correspond to DC and Nyquist, although the mapping is nonlinear.
//
// 2 inputs:
//     CF[n], cut-off frequency, scalar;
```

```
//     x[n].
//
// 1 outputs:
//     y[n], lowpassed x[n].
//
lp1praw(cf, in) =  + (in * cf)
                   ~ * (1 - cf);
// ------------------------------------------------------------------------------

// f2.lp1p1z(CF[n], x[n]); -------------------------------------------------------
//
// One-pole-one-zero lowpass (design by Cliff Sparks).
//
// 2 inputs:
//     CF[n], cut-off frequency in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], lowpassed x[n].
//
lp1p1z(cf, in) =   + (in * a0 <: _ + _' * a1)
                   ~ * (b1)
      with {
            a0 = (1 - b1) / 2;
            a1 = 1;
            b1 = m2.div((1 - sin(m2.w(cf))), cos(m2.w(cf)));
      };
// ------------------------------------------------------------------------------

// f2.lp1p1zraw(CF[n], x[n]); ----------------------------------------------------
//
// One-pole-one-zero lowpass with raw coefficients in the [0; 1] where the extrem
// correspond to DC and Nyquist, although the mapping is nonlinear.
//
// 2 inputs:
//     CF[n], cut-off frequency, scalar;
//     x[n].
//
```

```
// 1 outputs:
//     y[n], lowpassed x[n].
//
lp1p1zraw(cf, in) =    + (in * cf <: _ + _')
                       ~ * (cf * -2 + 1);
// ------------------------------------------------------------------------

// f2.sah_dc(T[n], x[n]); ------------------------------------------------
//
// Sample-and-hold signals that have remained constant for t seconds.
//
// 2 inputs:
//     T[n], SAH trigger period in seconds;
//     x[n].
//
// 1 outputs:
//     y[n], sample-and-held stationary values in x[n].
//
sah_dc(t, in) = ba.sAndH(abs(m2.delta(t, in)) < ma.EPSILON, in);
// ------------------------------------------------------------------------

// f2.sah_inv(x[n]); ------------------------------------------------------
//
// Sample-and-hold signals that have changed in direction.
//
// 1 inputs:
//     x[n].
//
// 1 outputs:
//     y[n], sample-and-held x[n] upon change of direction.
//
sah_inv(in) = m2.diff(in) : ma.signum <:   abs ,
                                           _ : ba.sAndH <: _ != _' ,
                                                        in : ba.sAndH;
// ------------------------------------------------------------------------

// f2.slew_limiter(Min[n], Max[n], x[n]); ---------------------------------
//
```

```
// Slew limiter; the same as Max's deltaclip~
// (https://docs.cycling74.com/max7/refpages/deltaclip~).
//
// 3 inputs:
//    Min[n], minimum slope;
//    Max[n], maximum slope;
//    x[n].
//
// 1 outputs:
//    y[n], filtered x[n].
//
slew_limiter(in1, in2, in3) =  tick
                               ~ _
    with {
        tick(fb) = m2.if(cond1, max_plus_fb, cond3)
            with {
                max_plus_fb = max(in2, in3) + fb;
                min_plus_fb = min(in2, in3) + fb;
                cond1 = in1 > max_plus_fb;
                cond2 = in1 < min_plus_fb;
                cond3 = m2.if(cond2, max0, in1);
                max0 = max(0, min_plus_fb);
            };
    };
// -----------------------------------------------------------------------------

// f2.svfblti(CF[n], x[n]); ---------------------------------------------------
//
// One-pole-one-zero stae-variable filter based on zero-delay feedback with
// bilinear transform integrator (design by Zavalishin).
//
// 2 inputs:
//    CF[n], cut-off frequency in Hz;
//    x[n].
//
// 3 outputs:
//    y1[n], lowpassed x[n];
//    y2[n], highpassed x[n];
```

```
//      y3[n], allpassed x[n].
//
svfblti(cf, in) =  tick
                   ~ _ :   ! ,
                           _ ,
                           _ ,
                           _
       with {
             g = m2.div(tan(m2.w(cf) / 2), (1 + tan(m2.w(cf) / 2)));
             tick(s) =   y + v ,
                         y ,
                         in - (v + s) ,
                         s + 2 * v - (in - s)
                   with {
                        v = (in - s) * g;
                        y = v + s;
                   };
       };
// ------------------------------------------------------------------------

// f2.svf2blti(CF[n], Q[n], K[n], x[n]); ----------------------------------
//
// Two-pole-two-zero state-variable filter with zero-delay feedback topology
// (design from Zavalishin / Pirkle).
//
// 4 inputs:
//      CF[n], cut-off frequencyin Hz;
//      Q[n], Q-factor;
//      K[n], linear amplitude for shelving filters;
//      x[n].
//
// 10 outputs:
//      y1[n], lowpassed x[n];
//      y2[n], highpassed x[n];
//      y3[n], bandpassed x[n];
//      y4[n], bandpassed (normalised) x[n];
//      y5[n], lowshelved x[n];
//      y6[n], highshelved x[n];
```

```
//     y7[n], bandshelved x[n];
//     y8[n], bandstopped x[n];
//     y9[n], peak-filtered x[n];
//     y10[n], allpassed x[n].
//
svf2blti(cf, q, k, in) =   tick
                           ~ ( _ ,
                               _) : (  ! ,
                                       ! ,
                                       _ ,
                                       _ ,
                                       _ ,
                                       _ ,
                                       _ ,
                                       _ ,
                                       _ ,
                                       _ ,
                                       _ ,
                                       _)
    with {
        r = m2.div(1, (2 * q));
        wa = (2 * ma.SR) * tan(m2.w(cf) / 2);
        g = wa / ma.SR / 2;
        tick(s1, s2) =  u1 ,
                        u2 ,
                        lp ,
                        hp ,
                        bp ,
                        bp_norm ,
                        ls ,
                        hs ,
                        b_shelf ,
                        notch ,
                        peak ,
                        ap
            with {
                u1 = v1 + bp;
                u2 = v2 + lp;
```

```
                v1 = hp * g;
                v2 = bp * g;
                hp = m2.div((in - 2 * r * s1 - g * s1 - s2),
                            (1 + 2 * r * g + g * g));
                bp = s1 + v1;
                lp = s2 + v2;
                bp_norm = bp * 2 * r;
                b_shelf = in + k * bp_norm;
                ls = in + k * lp;
                hs = in + k * hp;
                notch = in - bp_norm;
                ap = in - 4 * r * bp;
                peak = lp - hp;
            };
    };
// ----------------------------------------------------------------------------

// f2.xover_butt(N, CF[n], x[n]); --------------------------------------------
//
// Nth-order crossover based on Faust's Butterworth filters.
//
// 2 inputs:
//     CF[n], cut-off frequency in Hz;
//     x[n].
//
// 1 outputs:
//     y1[n], lowpassed x[n];
//     y2[n], highpassed x[n].
//
// 1 compile-time arguments:
//     "N", (integer) crossover order.
//
xover_butt(N, cf, x) = low ,
                       high
    with {
        low = fi.lowpass(N, cf, x);
        high = fi.highpass(N, cf, x);
    };
```

```
// ------------------------------------------------------------------------------

// f2.xover1p1z(CF[n], x[n]); ---------------------------------------------------
//
// One-pole-one-zero crossover.
//
// 2 inputs:
//    CF[n], cut-off frequency in Hz;
//    x[n].
//
// 2 outputs:
//    y1[n], lowpassed x[n];
//    y2[n], highpassed x[n].
//
xover1p1z(cf, in) =    low ,
                       high
    with {
        low = f2.lp1p1z(cf, in);
        high = in - low;
    };
// ------------------------------------------------------------------------------

// f2.xover1p1z_ada(R[n], x[n]); ------------------------------------------------
//
// Adaptive crossover based on 1p1z filters: it equally redistributes power
// (RMS) among low and high spectra. Useful when only one microphone is
// available  and different signals are needed for internal processing.
//
// 2 inputs:
//    R[n], sets the responsiveness of the system in Hz;
//    x[n].
//
// 2 outputs:
//    y1[n], lowpassed x[n];
//    y2[n], highpassed x[n].
//
xover1p1z_ada(window, in) = (  (    _ ,
                                    in) : f2.xover1p1z <:  (   ip.rms(window) ,
```

```
                                                                            ip.rms(window) :
      ro.cross(2) : - : / (max(ma.EPSILON, (ip.rms(window, in)))) * window /
           ma.SR : f2.int_clip(0, 1) ^ 2 * m2.ny) ,
                                                                       (    _ ,
                                                                            _))
                                     ~ ( _ ,
                                         ! ,
                                         !) : (  ! ,
                                                 _ ,
                                                 _);
// ----------------------------------------------------------------------------

// f2.xover1praw(CF[n], x[n]); ---------------------------------------------------
//
// One-pole crossover with raw coefficients in the [0; 1] where the extremes
// correspond to DC and Nyquist, although the mapping is nonlinear.
//
// 2 inputs:
//    CF[n], cut-off frequency, scalar;
//    x[n].
//
// 2 outputs:
//    y1[n], lowpassed x[n];
//    y2[n], highpassed x[n].
//
xover1praw(cf, in) =  low ,
                      high
      with {
           low = f2.lp1praw(cf, in);
           high = in - low;
      };
// ----------------------------------------------------------------------------

// f2.xover1p1zraw(CF[n], x[n]); -------------------------------------------------
//
// One-pole-one-zero crossover with raw coefficients in the [0; 1] where the
// extremes correspond to DC and Nyquist, although the mapping is nonlinear.
//
```

```
// 2 inputs:
//     CF[n], cut-off frequency, scalar;
//     x[n].
//
// 2 outputs:
//     y1[n], lowpassed x[n];
//     y2[n], highpassed x[n].
//
xover1p1zraw(cf, in) = low ,
                       high
    with {
        low = f2.lp1p1zraw(cf, in);
        high = in - low;
    };
// ------------------------------------------------------------------------

// f2.xover2p2z(CF[n], x[n]); ---------------------------------------------
//
// Two-pole-two-zero crossover based on 2nd-order Butterworth filters.
//
// 2 inputs:
//     CF[n], cut-off frequency in Hz;
//     x[n].
//
// 2 outputs:
//     y1[n], lowpassed x[n];
//     y2[n], highpassed x[n].
//
xover2p2z(cf, in) =    low ,
                       high
    with {
        low = fi.lowpass(2, cf1, in);
        high = fi.highpass(2, cf1, in);
        cf1 = max(20 / ma.SR * m2.ny, min((1 - (200 / ma.SR)) * m2.ny, cf));
    };
// ------------------------------------------------------------------------
```

A.6 information.lib

```
// =========================================================================
// ========== information.lib ==============================================
// =========================================================================
//
// Information processing functions library including low-level and high-level
// algorithms both based on hard-coded and adaptive mechanisms. The
// low-level functions provide time-domain techniques for feature
// extraction that are normally based on FFT processing, such as
// spectral centroid and spectral flatness (noisiness). The high-level
// functions provide an analysis of the state space of low-level
// information signals to determine, based on notions of complexity theory and
// music perception, characteristics such as dynamicity, heterogeneity,
// and complexity of audio streams.
//
// All functions below that use one-pole lowpass filters for
// accumulation are based on a time constant that is 2pi*tau.
//
// The environment prefix is "ip".
//
// List of functions:
//
//     a_weighting,
//     complexity,
//     dynamicity,
//     env_fol,
//     highest_partial,
//     heterogeneity,
//     heterogeneity10,
//     instant_amp,
//     instant_freq,
//     instant_ph,
//     loudness,
//     lowest_partial,
//     noisiness,
//     peaks,
//     peak_env,
```

```
//      peak_env_AHR_cascade,
//      peak_env_AHR_switch,
//      peak_env_AR_cascade,
//      peak_env_AR_switch,
//      peak_hold,
//      peak_hold_H,
//      peak_hold_L,
//      peak_hold_LH,
//      recurrence,
//      rms,
//      rms4,
//      roughness,
//      spec_bal,
//      spec_balN,
//      spec_peakN,
//      spec_ten,
//      spec_tenN,
//      spec_ten_lite,
//      zc,
//      zcr,
//      zcr4.
//
// Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario at gmail dot com>
// All rights reserved.

declare name "Information Processing Library";
declare author "Dario Sanfilippo";
declare copyright "Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario
      at gmail dot com>";
declare version "1.0.0";
declare license "GPLv2.0";

au = library("auxiliary.lib");
ba = library("basics.lib");
fi = library("filters.lib");
f2 = library("filters2.lib");
ip = library("information.lib");
ma = library("maths.lib");
```

```
m2 = library("maths2.lib");
ro = library("routes.lib");
si = library("signals.lib");

// ip.a_weighting(F[n]); ------------------------------------------------
//
// Function for equal-loudness contour based on the A-weighting equation.
// The function is calibrated to output unity (0 dB) at 1000 Hz.
//
// 1 inputs:
//    F[n], frequency in Hz;
//
// 1 outputs:
//    y[n], linear amplitude at F[n].
//
a_weighting(f) = (12194 ^ 2 * f ^ 4) / ((f ^ 2 + 20.6 ^ 2) *
      sqrt((f ^ 2 + 107.7 ^ 2) * (f ^ 2 + 737.9 ^ 2)) *
           (f ^ 2 + 12194 ^ 2)) + .206;
// -----------------------------------------------------------------------

// ip.complexity(max_dt, dt[n], R[n], x[n]); ------------------------------
//
// Complexity measurement as edge-of-chaos detection, that is,  accumulation
// of the variations in the heterogeneity of a state space. The function
// outputs an index in the [0; 1] range. The input signal is assumed in
// the [0; 1] range, and the resolution is of 10 sub-regions in the state space.
//
// 3 inputs:
//    dt[n], differentiation period in seconds;
//    R[n], responsiveness of the system in Hz;
//    x[n].
//
// 1 outputs:
//    y[n], complexity index for x[n].
//
// 1 compile-time arguments:
//    max_dt, maximum differentiation period in seconds.
//
```

```
complexity(max_dt, dt, window, x) = ip.heterogeneity10(window, x) :
      abs(m2.delta(max_dt, dt)) : f2.lp1p(1 / max(ma.EPSILON, window));
// ----------------------------------------------------------------------------

// ip.dynamicity(max_dt, dt[n], R[t], x[n]); ---------------------------------
//
// Dynamicity index as accumulation of the magnitude of the delta at a
// specified dt (secs) and accumulation rate (Hz).
// It is supposed to be used with infrasonic low-level information signals such
// as RMS, spectral tendency, noisiness.
//
// 3 inputs:
//     dt[n], differentiation period in seconds;
//     R[n], responsiveness of the system in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], dynamicity index for x[n].
//
// 1 compile-time arguments:
//     max_dt, maximum differentiation period in seconds.
//
dynamicity(max_dt, dt, window, in) = abs(m2.delta(max_dt, dt, in)) :
      f2.lp1p(window);
// ----------------------------------------------------------------------------

// ip.env_fol(R[n], x[n]); ---------------------------------------------------
//
// Envelope following (Dodge and Jerse).
//
// 2 inputs:
//     R[n], responsiveness in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], amplitude envelope of x[n].
//
env_fol(window, in) = f2.lp1p(window, abs(in));
```

```
// ———————————————————————————————————————————————

// ip.highest_partial(R[n], x[n]); ————————————————
//
// It detects the highest partial in a signal. This function is an
// extension of the spectral tendency algorithm by including a positive
// feedback loop at the top of the chain: the system recursively removes
// low-frequency componenets through a highpass until no components are
// left on the upper side of the spectrum except the last partial.
// The SNR and sensitivity of the filter, approximately, could be adjusted
// by changing the order of the filters.
//
// 2 inputs:
//    R[n], responsiveness in Hz;
//    x[n].
//
// 1 outputs:
//    y[n], frequency of the highest partial in x[n].
//
highest_partial(window, in) =
    (_ <:    _ ,
            (    _ ,
                in : hp1p1z2 <: _ ,
                                _) : (m2.div(specbal, ip.rms(window)) /
    ma.SR * window : f2.int_clip(0, 1) ^ 2))
    ~ * (m2.ny)
    with {
        specbal = f2.xover1p1z :    ip.rms(window) + .000001 ,
                                    ip.rms(window)
            : ro.cross(2) : -;
        hp1p1z2(cf, x) = x : seq(i, 2, f2.hp1p1z(cf));
    };
// ———————————————————————————————————————————————

// ip.heterogeneity(N, L[n], H[n], T[n], x[n]); ——————————
//
// Provides an index of how heterogeneous the state space of a signal is, which
// correlates to the unpredictability of the signal. The algorithm
```

```
// follows principles of recurrence quantification analysis by dividing the
// state space into sub−regions of the same size. If the signal is in a
// sub−region, then an integrator is triggered to keep track of the
// recurrence of that state. When the signal moves to a different
// sub−region, then the recurrence value in the non−active sub−region is
// held for a specified time before being reset to 0. The output of the
// peak holder is smoothed out by a one−pole lowpass with the same period
// as the peak holder for attack and decay smoothing. The final
// heterogeneity index is then computed by processing the outputs of the
// sub−regions recurrences through normalised average absolute deviation (AAD).
// AAD is used instead of standard deviation to have an output in the range
// [0; 1].
//
// 4 inputs:
//    L[n], lower edge of the state space;
//    H[n], upper edge of the state space;
//    T[n], analysis period, it sets the memory of the system, in
//          seconds, determining hold and decay times;
//    x[n].
//
// 1 outputs:
//    y[n], heterogeneity index for x[n].
//
// 1 compile−time arguments:
//    N, state space resolution, that is, (integer) number of
//          equally−spaced regions in the state space.
//
heterogeneity(N, l, h, t, in) =
      par(i, N,     region(l, h, N, i) ,
                    t ,
                    in : ip.recurrence : ip.peak_hold(t) :
                        f2.lp1p(m2.div(1, t))) : m2.aad(N)
      with {
          region(lower, upper, N, i) =
              lower + ((upper − lower) / N) * i ,
              lower + ((upper − lower) / N) * (i + 1);
      };
// ————————————————————————————————————————————————
```

```
// ip.heterogeneity10(T[n], x[n]); ——————————————————————————
//
// Shortcut for a heterogeneity index for a 10-region state space in a [0; 1]
// range.
//
// 2 inputs:
//    T[n], analysis period, it sets the memory of the system, in
//          seconds, determining hold and decay times;
//    x[n].
//
// 1 outputs:
//    y[n], heterogeneity index for x[n] with a 10-region resolution in the
//    state space, assuming that its range is [0; 1].
//
heterogeneity10(t, in) = ip.heterogeneity(10, 0, 1, t, in);
// ————————————————————————————————————————

// ip.instant_amp(x[n]); ——————————————————————————————
//
// Instantaneous amplitude.
//
// 1 inputs:
//    x[n].
//
// 1 outputs:
//    y[n], instantaneous amplitude of x[n], sqrt(re^2 + im^2).
//
instant_amp(x) = f2.analytic(x) : sqrt(pow(2) + pow(2));
// ————————————————————————————————————————

// ip.instant_freq(x[n]); —————————————————————————————
//
// Instantaneous frequency.
//
// 1 inputs:
//    x[n].
//
```

```
// 1 outputs:
//     y[n], instantaneous frequency of x[n] as derivative of the instantaneous
//     phase.
//
instant_freq(x) = m2.diff(instant_ph(x)) / (2 * ma.PI) * ma.SR;
// ------------------------------------------------------------------------------

// ip.instant_ph(x[n]); ---------------------------------------------------------
//
// Instantaneous phase.
//
// 1 inputs:
//     x[n].
//
// 1 outputs:
//     y[n], instantaneous phase of x[n] as the arcotangent of the ratio
//           between its imaginary and real parts.
//
instant_ph(x) = f2.analytic(x) : ro.cross(2) : atan2;
// ------------------------------------------------------------------------------

// ip.loudness(R[n], x[n]); -----------------------------------------------------
//
// Loudness measurement through A-weighting function and spectral centroid.
//
// 1 inputs:
//     R[n], responsiveness of the system in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], loudness of x[n] as linear amplitude.
//
loudness(window, in) =
      peak_env(1 / max(ma.EPSILON, window), in) *
           (spec_ten(window, in) * m2.ny : a_weighting);
// ------------------------------------------------------------------------------

// ip.lowest_partial(R[n], x[n]); -----------------------------------------------
```

```
//
// It detects the lowest partial in a signal. This function is an
// extension of the spectral tendency algorithm by including a positive
// feedback loop at the top of the chain: the system recursively removes
// high-frequency componenets through a lowpass until no components are
// left on the lower side of the spectrum except the last partial.
// The SNR and sensitivity of the filter, approximately, could be adjusted
// by changing the order of the filters.
//
// 2 inputs:
//     R[n], responsiveness in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], frequency of the lowhest partial in x[n].
//
lowest_partial(window, in) =
      (_ <:     _ ,
                (    _ ,
                     in : lp1p1z2 <: _ ,
                                   _) :
      (specbal / max(ma.EPSILON, rms(window)) / ma.SR * window :
           f2.int_clip(0, 1) ^ 2))
      ~ (* (m2.ny) : max(10))
      with {
           specbal = f2.xover1p1z :     rms(window) ,
                                        rms(window) + .000001
                : ro.cross(2) : -;
           lp1p1z2(cf, x) = x : seq(i, 2, f2.lp1p1z(cf));
      };
// ------------------------------------------------------------------------

// ip.noisiness(R[n], x[n]); --------------------------------------------
//
// Noisiness index normalised for a 96 kHz samplerate, that is, the
// output of the function is 1 for white noise, and 0 (or very close to 0)
// for sinusoids. Unlike noisiness drectly calculated through zero-crossing
// rate (ZCR), this algorithm measures noisiness based on the derivative of the
```

```
// ZCR, as non-periodicity characterises noisy signals, whereas
// high-frequency sinusoidal signals can still have a high ZCR.
//
// 2 inputs:
//    R[n], responsiveness in Hz;
//    x[n].
//
// 1 outputs:
//    y[n], noisiness index of x[n] in the [0; 1] range.
//
noisiness(window, in) = spec_ten(window, in) *  m2.ny : max(1) <:
      m2.inv  ,
      (      _ ,
           in : zcr4 : m2.unit_log(10)) : abs(m2.delta(1)) : f2.lp1p(window) /
                .266 <: * : min(1);
// ---------------------------------------------------------------------------

// ip.peak_env(R[n], x[n]); ---------------------------------------------------
//
// Peak envelope function: infinitely fast attack and adjustable release.
// This function outputs the absolute value of the input if the
// magnitude of the signal is greater or equal than the output,
// or it performs an exponential decay of the last detected peak
// when the input is smaller than the output.
//
// 2 inputs:
//    R[n], release time in seconds;
//    x[n].
//
// 1 outputs:
//    y[n], peak envelope of x[n].
//
peak_env(RT, x) =  max(abs(x))
                   ~ * (m2.rt55(RT));
// ---------------------------------------------------------------------------

// ip.peak_env_AHR_cascade(AT[n], HT[n], RT[n], x[n]); ------------------------
//
```

```
// Peak envelope function with attack, hold, and release times in
// seconds. The effective attack, hold, and release times are
// interrelated as this design is based on cascaded filters, particularly,
// a peak holder (peak_hold) feeding into a peak_envelope (peak_env) feeding
// into a one-pole lowpass (smooth). Considering the tau*2   time constant
// where most of the final value is reached after the attack time, we will
// only have a hold segment in the resulting envelope function if the hold
// time is greater than the hold time, and the hold segment will be
// approximately hold_time minus attack_time. Alternatively, only attack
// and release segments will result from this cascaded design. Furthermore,
// the attack time should still be << than the release time to minimally
// affect the decay.
//
// 4 inputs:
//    AT[n], attack time in seconds;
//    HT[n], hold time in seconds;
//    RT[n], release time in seconds;
//    x[n].
//
// 1 outputs:
//    y[n], envelope profile of x[n].
//
peak_env_AHR_cascade(AT, HT, RT, x) =
      ip.peak_hold(HT, x) : ip.peak_env_AR_cascade(AT, RT);
// ----------------------------------------------------------------------------

// ip.peak_env_AHR_switch(AT[n], HT[n], RT[n], x[n]); ---------------------------
//
// Envelope function with attack, hold, and release times in seconds.
// This design is based on switching filter sections depending on the
// attack or release phases of the input signal, which is determined by
// comparison between the output and the input. When the input is greater
// than the output, the system switches to a one-pole lowpass. Otherwise,
// the system switches to a peak holder (peak_hold) section, and finally
// into a peak envelope (peak_env) section when the hold time has passed and
// no new peaks have been detected. This way, the resulting envelope curve
// will always include attack, hold, and release segments at the specified
// times, where the hold value is dependent on the value that has been reached
```

```
// in the attack section.
//
// 4 inputs:
//    AT[n], attack time in seconds;
//    HT[n], hold time in seconds;
//    RT[n], release time in seconds;
//    x[n].
//
// 1 outputs:
//    y[n], envelope profile of x[n].
//
peak_env_AHR_switch(AT, HT, RT, x) =    loop
                                        ~ _
    with {
        loop(fb) = m2.if(cond1,
                         attack,
                         m2.if(  cond2,
                                 hold,
                                 release))
            with {
                cond1 = abs(x) >= fb;
                cond2 = fi.pole(notcond1, notcond1) <= rint(HT * ma.SR);
                notcond1 = 1 - cond1;
                attack = abs(x) * (1 - AT_coeff) + fb * AT_coeff;
                hold = fb;
                release = RT_coeff * fb;
                AT_coeff = m2.rt55(AT);
                RT_coeff = m2.rt55(RT);
            };
    };
// ------------------------------------------------------------------------------

// ip.peak_env_AR_cascade(AT[n], RT[n], x[n]); -----------------------------
//
// Peak envelope function with dependent attack and release times. The
// function applies two cascaded filter sections regardless of the
// attack or release phases of the input signal: a peak envelope filter
// feeding into a one-pole lowpass filter. The attack time is assumed to
```

```
// be << than the release time for best behaviour. Since the sections are
// cascaded, the effective release time will always be greater than the
// desired one. The difference will be less noticeable when attack and
// release times are far apart. This function provides a smoother
// transition between attack and release phases.
//
// 3 inputs:
//    AT[n], attack time in seconds;
//    RT[n], release time in seconds;
//    x[n].
//
// 1 outputs:
//    y[n], peak envelope of x[n].
//
peak_env_AR_cascade(AT, RT, x) = ip.peak_env(RT, x) : si.smooth(m2.rt55(AT));
// ------------------------------------------------------------------------------

// ip.peak_env_AR_switch(AT[n], RT[n], x[n]); ------------------------------------
//
// Peak envelope function with independent attack and release times. The
// function applies two independent filter sections based on whether
// the input is in the attack or release phase, which is determined by
// comparing the absolute value of the input signal with the output.
// The attack-smoothing filter is a one-pole lowpass: a leaky integrator
// whose input is scaled down with the complement of the pole position.
// The release-smoothing filter, on the other hand, is simply an
// exponential decay of the detected peak.
//
// Ref: (2008) Udo Z lzer     Digital Audio Signal Processing 2nd Edition.
//
// 3 inputs:
//    AT[n], attack time in seconds;
//    RT[n], release time in seconds;
//    x[n].
//
// 1 outputs:
//    y[n], peak envelope of x[n].
//
```

```
peak_env_AR_switch(AT, RT, x) =     loop
                                    ~ _
    with {
        loop(fb) = m2.if(abs(x) > fb, attack, release)
            with {
                attack = abs(x) * (1 - AT_coeff) + fb * AT_coeff;
                release = fb * RT_coeff;
                AT_coeff = m2.rt55(AT);
                RT_coeff = m2.rt55(RT);
            };
    };
// ----------------------------------------------------------------------------

// ip.peak_hold(HT[n], x[n]); ------------------------------------------------
//
// Peak holder: it holds the peak of the absolute value of the input
// signal for a time specified in seconds. If no new peak occurs before
// the specified time, the peak will reset to whatever the absolute value
// of the input is. A new peak is detected if the input is greater or
// equal to the output.
//
// Note that the countdown for the hold time starts after a peak
// disappears. For example, if the function is tested with a step response
// of 20 samples, if the hold time is set to 48 samples (.001 seconds at
// SR = 48 kHz), we will have 1s from sample  n = 0 to n = 19 from the step
// function, 1s from n = 20 to n = 67 from the hold time, and the output will
// then be 0 from sample n = 68.
//
// 2 inputs:
//    HT[n], hold time in seconds (resulting in the closest integer
//        number of samples representing that length);
//    x[n].
//
// 1 outputs:
//    y[n], high peak of |x[n]|.
//
peak_hold(HT, x) = ip.peak_hold_H(HT, abs(x));
// ----------------------------------------------------------------------------
```

```
// ip.peak_hold_H(HT[n], x[n]); ----------------------------------------------------
//
// High peak holder: it holds the highest peak of the input signal for a time
// specified in seconds. If no new peak occurs before the specified time,
// the peak will reset to whatever the value of the input is. A new peak is
// detected if the input is greater or equal to the output.
//
// Note that the countdown for the hold time starts after a peak
// disappears. For example, if the function is tested with a step response
// of 20 samples, if the hold time is set to 48 samples (.001 seconds at
// SR = 48 kHz), we will have 1s from sample  n = 0 to n = 19 from the step
// function, 1s from n = 20 to n = 67 from the hold time, and the output will
// then be 0 from sample n = 68.
//
// 2 inputs:
//    HT[n], hold time in seconds (resulting in the closest integer
//          number of samples representing that length);
//    x[n].
//
// 1 outputs:
//    y[n], high peak of x[n].
//
peak_hold_H(HT, x) =    loop_peak
                        ~ _
    with {
        l = rint(HT * ma.SR);
        loop_peak(fb_p) = m2.if(cond1 | notcond2, x, fb_p)
            with {
                cond1 = x >= fb_p;
                cond2 = loop_timer <= l
                    ~ _
                    with {
                        loop_timer(fb_t) = notcond1 & fb_t <: fi.pole;
                    };
                notcond1 = 1 - cond1;
                notcond2 = 1 - cond2;
```

```
            };
        };
// -----------------------------------------------------------------------------

// ip.peak_hold_L(HT[n], x[n]); ------------------------------------------------
//
// Low peak holder: it holds the lowest peak of the input signal for a time
// specified in seconds. If no new peak occurs before the specified time,
// the peak will reset to whatever the value of the input is. A new peak is
// detected if the input is less or equal to the output.
//
// Note that the countdown for the hold time starts after a peak
// disappears. For example, if the function is tested with a step response
// of 20 samples, if the hold time is set to 48 samples (.001 seconds at
// SR = 48 kHz), we will have 1s from sample  n = 0 to n = 19 from the step
// function, 1s from n = 20 to n = 67 from the hold time, and the output will
// then be 0 from sample n = 68.
//
// 2 inputs:
//      HT[n], hold time in seconds (resulting in the closest integer
//          number of samples representing that length);
//      x[n].
//
// 1 outputs:
//      y[n], low peak of x[n].
//
peak_hold_L(HT, x) =    loop_peak
                        ~ _
    with {
        l = rint(HT * ma.SR);
        loop_peak(fb_p) = m2.if(cond1 | notcond2, x, fb_p)
            with {
                cond1 = x <= fb_p;
                cond2 = loop_timer <= l
                        ~ _
                    with {
                        loop_timer(fb_t) = notcond1 & fb_t <: fi.pole;
                    };
```

```
            notcond1 = 1 - cond1;
            notcond2 = 1 - cond2;

        };
    };

// ----------------------------------------------------------------------------

// ip.peak_hold_LH(HT[n], x[n]); ---------------------------------------------
//
// It detects lower and upper peaks and, if no new peaks are detected,
// it holds them for a specified time in seconds before resetting the peaks
// to the new ones given by the incoming input.
//
// 2 inputs:
//    HT[n], hold time in seconds (resulting in the closest integer
//         number of samples representing that length);
//    x[n].
//
// 2 outputs:
//    y1[n], lower peak in x[n],
//    y2[n], upper peak in x[n].
//
peak_hold_LH(HT, x) =  peak_hold_L(HT, x) ,
                       peak_hold_H(HT, x);
// ----------------------------------------------------------------------------

// ip.recurrence(L[n], H[n], T[n], x[n]); -------------------------------------
//
// It detects and integrates occurrences of signals within a specified
// range using a leaky integrator with a specified decay time in seconds.
//
// 4 inputs:
//    L[n], lower bound (not including the edge for the detection);
//    H[n], upper bound (including the edge for the detection);
//    T[n], decay period in seconds;
//    x[n].
//
```

```
// 1 outputs:
//    y[n], recurrence of the signal x[n] within the specified range
//          ]lower; upper].
//
recurrence(lower, upper, t, in) =  in > lower ,
                                   in <= upper : & : fi.pole(m2.rt55(t));
// ------------------------------------------------------------------------

// ip.rms(R[n], x[n]); ----------------------------------------------------
//
// Root mean square measurement using one-pole lowpass for averaging.
//
// 2 inputs:
//    R[n], analysis rate in Hz;
//    x[n].
//
// 1 outputs:
//    y[n], RMS of x[n].
//
rms(window, in) = in <: * : f2.lp1p(window) : sqrt;
// ------------------------------------------------------------------------

// ip.rms4(R[n], x[n]); ---------------------------------------------------
//
// RMS with four cascaded one-pole lowpass for averaging.
//
// 2 inouts:
//    R[n], analysis rate in Hz;
//    x[n].
//
// 1 outputs:
//    y[n], RMS of x[n].
//
rms4(window, in) = in <: * : seq(i, 4, f2.lp1p(window)) : sqrt;
// ------------------------------------------------------------------------

// ip.roughness(R[n], x[n]); ----------------------------------------------
//
```

```
// The roughness measurement is based on amplitude transients in the
// 15-75 Hz range. Theoretically, roughness decreases below 15Hz,
// peaks around 15-75 Hz, and starts descresing above 75Hz to
// disappear around 150 Hz and above. The analysis rate parameter may affect
// the behaviour in the 0-15Hz range. An analysis rate of 1 Hz seems acceptable.
// References: [Helmholtz 2013; Vassilakis and Kendall 2010; Terhardt
// 1974; Molla and Torr sani 2004; Moore 1995].
//
// 2 inputs:
//     R[n], responsiveness in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], roughness index of x[n] in the range [0; 1].
//
roughness(window, in) =
      instant_amp(in) <: m2.div(m2.delta(.001, .001), rms(1)) :
            rms4(75) : m2.delta(1 / 150, 1 / 150) : rms4(window) / .807242;
// ---------------------------------------------------------------------------

// ip.spec_bal(CF[n], R[n], x[n]); -------------------------------------------
//
// Spectral power (RMS) difference at a splitting point, in Hz, of the
// spectrum of a signal. The input signal is filtered through a one-pole
// crossover; the difference of the RMS of the high and low spectra is the
// output.
//
// 3 inputs:
//     CF[n], crossover cut-off frequency in Hz;
//     R[n], RMS analysis rate in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], RMS difference of the crossover outputs.
//
spec_bal(cf, window, in) = f2.xover1p1z(cf, in) :  rms(window) ,
                                                    rms(window) :
      ro.cross(2) : -;
```

```
// ------------------------------------------------------------------------

// ip.spec_balN(N, CF[n], R[n], x[n]); ------------------------------------
//
// Spectral power (RMS) difference at a splitting point, in Hz, of the
// spectrum of a signal. The input signal is filtered through an
// Nth-order Butterworth crossover; the difference of the RMS of the high and
// low spectra is the output.
//
// 3 inputs:
//    CF[n], crossover cut-off frequency in Hz;
//    R[n], RMS analysis rate in Hz;
//    x[n].
//
// 1 outputs:
//    y[n], RMS difference of the crossover outputs.
//
// 1 compile-time arguments:
//    N, (integer) order of the crossover.
//
spec_balN(N, cf, window, in) = f2.xover_butt(N, cf, in) : rms(window) ,
                                                          rms(window) :
    ro.cross(2) : -;
// ------------------------------------------------------------------------

// ip.spec_peakN(N, R[n], x[n]); ------------------------------------------
//
// Spectral peak with Nth-order spectral tendency calculation and
// 2nd-order normalised BP filter (BLTI). It works by recursively
// removing components on the sides of the spectral tendency frequency.
// Currently a prototype as it still requires a mechanism to detect new
// appearing signals that happen to fall within the filtered band.
// One possibility is to drive the Q through the complement of the
// derivative of a spectral tendency measure at the top of the chain. Still
// a work in progress but it works well to isolate the frequency of pure
// tones in a noisy background.
//
// 2 inputs:
```

```
//     R[n], responsiveness in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], spectral peak in Hz.
//
// 1 compile-time arguments:
//     N, (integer) order of the crossover in the spectral centroid
//           analysis.
spec_peakN(N, window, in) = ((max(10), 1, in) : f2.bp2blti :
      spec_tenN(N, window))
                             ~ * (m2.ny);
// ----------------------------------------------------------------------------

// ip.spec_sprN(N, Q[n], R[n], x[n]); ---------------------------------------
//
// Spectral spread: complement of the ratio between the RMS of the
// spectral-centroid-centered bandpassed input and the input RMS. Nth-order
// spectral tendency, arbitrary Q-factor for the bandpass to tune-in
// specific applications.
//
// 3 inputs:
//     Q[n], Q-factor in the bandpass filter;
//     R[n], RMS analysis rate in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], spectral spread index for x[n] in the range [0; 1] (0 no
//           spread; 1 maximum spread).
//
// 1 compile-time arguments:
//     N, (integer) order of the Butterworth crossover in the spectral
//           centroid analysis.
//
spec_sprN(N, q, window, in) = (    spec_tenN(N, window, in) * m2.ny ,
                                   q ,
                                   in : f2.bpbi :  ip.rms(window)) ,
                                ip.rms(window, in) : m2.div : m2.complement;
```

```
// ------------------------------------------------------------------------------------------

// ip.spec_ten(R[n], x[n]); ----------------------------------------------------------------
//
// Spectral tendency (we can call it centroid as it finds a balancing
// point) equal-power spectral split-point. The adaptive algorithm finds
// the cut-off frequency of a crossover where the RMS difference of its
// outputs is 0 through a negative feedback mechanism. To be precise, the
// algorithm minimally oscillates, hence it is in dynamical equilibrium,
// around the equilibrium point. The input is filtered through a
// one-pole-one-zero crossover; the difference of the RMS of the outputs of the
// filter is integrated; the result is squared and clipped for stability, mapped
// over Nyquist, and fed back to drive the frequency of the crossover.
//
// 2 inputs:
//    R[n], RMS analysis rate in Hz (responsiveness);
//    x[n].
//
// 1 outputs:
//    y[n], spectral centroid of x[n] as an index in the [0; 1] range.
//
spec_ten(window, in) = (    (    _ ,
                                 window ,
                                 in : spec_bal) ,
                            (    window ,
                                 in : rms) : m2.div * window :
      f2.int_clip(0, 1) ^ 2)
                        ~ * (m2.ny);
// ------------------------------------------------------------------------------------------

// ip.spec_tenN(N, R[n], x[n]); ------------------------------------------------------------
//
// Spectral tendency (we can call it centroid as it finds a balancing
// point) equal-power spectral split-point. The adaptive algorithm finds
// the cut-off frequency of a crossover where the RMS difference of its
// outputs is 0 through a negative feedback mechanism. To be precise, the
// algorithm minimally oscillates, hence it is in dynamical equilibrium,
// around the equilibrium point. The input is filtered through an Nth-order
```

```
// butterworth crossover; the difference of the RMS of the outputs of the
// filter is integrated; the result is squared and clipped for stability, mapped
// over Nyquist, and fed back to drive the frequency of the crossover.
// Spectral tendency (centroid) as equal-power spectral split-point with nth-order
// This design is improved as it implements a cubic nonlinear function
// that increases stability and accuracy.
//
// 2 inputs:
//     R[n], RMS analysis rate in Hz (responsiveness);
//     x[n].
//
// 1 outputs
//     y[n], spectral centroid of x[n] as an index in the [0; 1] range.
//
// 1 compile-time arguments:
//     N, (integer) order of the crossover.
//
spec_tenN(N, window, in) = ((    _ ,
                                 window ,
                                 in : spec_balN(N)) ,
                              (  window ,
                                 in : rms) : m2.div ^ 3 :
      f2.int_eu_clip(0, 1, window) ^ 2)
                            ~ * (m2.ny);
// ------------------------------------------------------------------------------

// ip.spec_ten_lite(x[n]); -------------------------------------------------------
//
// Lite spectral tendency algorithm: fixed responsiveness parameter; less
// accurate (especially at low frequencies).
//
// 1 inputs:
//     x[n].
//
// 1 outputs:
//     y[n], spectral centroid of x[n] in the range [0; 1].
//
spec_ten_lite(in) = (   _,
```

```
                    in : f2.xover1p1zraw : ro.cross(2) : balance :
        (_ ,
         (in : rms) : m2.div) : f2.integrator <: *)
                    ~ _
        with {
            a0 = 100 / ma.SR;
            b1 = 1 - a0;
            rms(in) = in <: * : * (a0) :     +
                                             ~ * (b1) : sqrt;
            balance =   rms ,
                        rms : -;
        };
// ---------------------------------------------------------------------------

// ip.zc(x[n]); ---------------------------------------------------------------
//
// Zero-crossing (ZC) indicator function: it returns 1 if a ZC occurs,
// 0 otherwise.
//
// 1 inputs:
//     x[n].
//
// 1 outputs:
//     y[n], ZC indication.
//
zc(x) = x * x' < 0;
// ---------------------------------------------------------------------------

// ip.zcr(R[n], x[n]); --------------------------------------------------------
//
// Zero-crossing rate (ZCR) using a one-pole lowpass filter for averaging.
// The ZCR correlates with the noisiness of signals, although it is
// effective only in specific cases, for example when comparing voiced and
// unvoiced sounds, or percussive and non-percussive ones. The ZCR also
// correlates with the spectral centroid of a signal. For sinusoidal
// signals, the ZCR output of this function can be mapped over Nyquist and
// can be used effectively as a frequency detector.
// References: Gouyon et al. 2000; Herrera-Boyer et al. 2006;
```

```
// Peeters et al. 2011.
//
// 2 inputs:
//     R[n], analysis rate in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], ZCR as an index in the range [0; 1].
//
zcr(window, x) = f2.lp1p(window, zc(x));
// ----------------------------------------------------------------------------

// ip.zcr4(R[n], x[n]); -----------------------------------------------------
//
// Zero-crossing rate with four cascaded one-pole lowpass filters for averaging.
//
// 2 inputs:
//     R[n], analysis rate in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], ZCR as an index in the range [0; 1].
//
zcr4(window, x) = zc(x) : seq(i, 4, f2.lp1p(window));
// ----------------------------------------------------------------------------
```

A.7 maths2.lib

```
// ============================================================================
// ========== maths2.lib ======================================================
// ============================================================================
//
// Math library containing functions for statistics, linear and nonlinear
// fuzzy logic, interpolators, network topologies, matrixes, linear and
// nonlinear mapping, windowing functions, hysteresis, angular frequency,
// and several time constants for exponential decays in one-pole systems.
//
// The environment prefix is "m2".
```

```
//
// List of functions:
//
//     aad,
//     asinc_bi,
//     asinc_uni,
//     avg_ari,
//     avg_ari_w,
//     avg_geo,
//     avg_geo_w,
//     avg_harm,
//     avg_harm_w,
//     avg_pow,
//     avg_quad,
//     bip,
//     complement,
//     dec2bin,
//     delta,
//     delta2,
//     diff,
//     div,
//     factorial,
//     hp_and,
//     hp_imp,
//     hp_nand,
//     hp_nimp,
//     hp_nor,
//     hp_nxr,
//     hp_or,
//     hp_xor,
//     interpolate_mn,
//     if,
//     ifN,
//     inv,
//     line,
//     line_reset,
//     map_lin,
//     map_par,
```

```
//      map_pcw,
//      map_log,
//      map_pow,
//      matrix,
//      maxN,
//      minN,
//      ny,
//      ph,
//      primes,
//      prime_base_pow,
//      relay_hysteron,
//      round_pow2,
//      rt9,
//      rt19,
//      rt55,
//      rt60,
//      seq_catalan,
//      seq_fibonacci,
//      seq_hexagonal,
//      seq_lazy_caterer,
//      seq_magic_number,
//      seq_pentagonal,
//      seq_square,
//      seq_triangular
//      sd,
//      sp,
//      topologies,
//      top_anti_diag,
//      top_diagonal,
//      top_diag_shift,
//      top_full,
//      top_tri_low,
//      top_tri_up,
//      twopi
//      uni,
//      unit_log,
//      var,
//      w,
```

```
//      window_hann,
//      window_sine,
//      wrap,
//      y_and,
//      y_or,
//      zeropad_up,
//      zeropad_down,
//      z_and,
//      z_imp,
//      z_nand,
//      z_nimp,
//      z_nor,
//      z_nxr,
//      z_or,
//      z_xor.
//
// Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario at gmail dot com>
// All rights reserved.

declare name "Math Library";
declare author "Dario Sanfilippo";
declare copyright "Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario
      at gmail dot com>";
declare version "1.0.0";
declare license "GPLv2.0";

ba = library("basics.lib");
de = library("delays.lib");
d2 = library("delays2.lib");
ma = library("maths.lib");
m2 = library("maths2.lib");
ro = library("routes.lib");
si = library("signals.lib");

// m2.aad(N); ----------------------------------------------------------------
//
// Normalised average absolute deviation around the mean.
//
```

```
// N inputs:
//     x1[n];
//     ...
//     xN-1[n];
//     xN[n].
//
// 1 outputs:
//     y[n], average absolute deviation in the range [0; 1] of the input signals.
//
// 1 compile-time arguments:
//     N, (integer) number of input signals.
//
aad(N) = si.bus(N) <: (si.bus(N) ,
                      (si.bus(N) :> / (N) <: si.bus(N)) : ro.interleave(N, 2) :
    par(i, N, (- : abs)) :> _) ,
    (si.bus(N) :> / (N)) : m2.div : / ((N - 1) * 2);
// ----------------------------------------------------------------------------

// m2.asinc_bi(H[n], ph[n]); -------------------------------------------------
//
// asinc_bp[n] = sin(2H[n]2 *ph[n])/(sin(2 *ph[n])*2H[n]).
//
// Bipolar periodic sinc function with arbitrary number of harmonics for
// the generation of band-limited impulse trains (BLIT).
// The bipolar BLIT has no DC component and the number of harmonics include the
// fundamental frequency and its odd multiples.
//
// The amplitude of the function, regardless of the harmonics, is normalised to
// unit-amplitude peaks.
//
// The bipolar sinc has a period of 2 , although the phase input of the
// function is normalised to the range [0; 1] for a full cycle.
//
// Unlike the technique described in [Stilson and Smith 1996], the
// bipolar sinc function is realised by having an even ratio between the
// frequencies of the sine functions, which results in a precise harmonic
// content throughout the entire frequency range when deployed in BLITs.
//
```

```
// 2 inputs:
//    H[n], (integer) number of harmonics including the fundamental
//          frequency and the odd harmonics.
//    ph[n], phase of the sinc function in the range [0; 1]. Values
//          outside of that range are allowed as they are wrapped around.
//
// 1 outputs:
//    y[n], bipolar sinc function.
//
asinc_bi(M, x) = m2.if(phase < ma.EPSILON,
                       1,
                       m2.if(   abs(.5 - phase) < ma.EPSILON,
                                -1,
                                sin(M1 * m2.twopi * phase) /
                                    (sin(m2.twopi * phase) * M1)))
      with {
           M1 = rint(M) * 2;
           phase = ma.frac(x);
      };
// ------------------------------------------------------------------------

// m2.asinc_uni(H[n], ph[n]); ---------------------------------------------
//
// asinc_uni[n] = sin((2H[n]+1)*2  *ph[n])/(sin(2  *ph[n])*(2H[n]+1)).
//
// Unipolar periodic sinc function with arbitrary number of harmonics
// for the generation of unipolar band-limited impulse trains.
// The unipolar BLIT has a DC component and the number of harmonics include the
// fundamental frequency and its multiples (both even and odd).
//
// The amplitude of the function, regardless of the harmonics,
// is normalised to unit-amplitude peaks.
//
// The technique described here is based on the paper
// [Stilson and Smith 1996]:
//
// https://ccrma.stanford.edu/~stilti/papers/blit.pdf.
//
```

```
// The unipolar sinc has a period of   , although the phase input of the
// function is normalised to the range [0; 1] for a full cycle.
//
// 2 inputs:
//    H[n], (integer) number of harmonics including the fundamental
//          frequency and both even and odd multiples.
//    ph[n], phase of the sinc function in the range [0; 1]. Values
//          outside of that range are allowed as they are wrapped around.
//
// 1 outputs:
//    y[n], unipolar sinc function.
//
asinc_uni(M, x) = m2.if(phase < ma.EPSILON,
                        1,
                        sin(M1 * ma.PI * phase) /
                           (sin(ma.PI * phase) * M1))
    with {
       M1 = rint(M) * 2 + 1;
       phase = ma.frac(x);
    };
// ------------------------------------------------------------------------

// m2.avg_ari(N); ---------------------------------------------------------
//
// Arithmetic mean.
//
// N inputs:
//    x1[n];
//    ...
//    xN-1[n];
//    xN[n].
//
// 1 outputs:
//    y[n], arithmetic mean of the input signals.
//
// 1 compile-time arguments:
//    N, (integer) number of input signals.
//
```

```
avg_ari(N) = si.bus(N) :> / (N);
// ----------------------------------------------------------------------------

// m2.avg_ari_w(N); -----------------------------------------------------------
//
// Weighted arithmetic mean.
//
// 2N inputs:
//     x1[n];
//     ...
//     xN-1[n];
//     xN[n], input signals;
//     w1[n];
//     ...
//     wN-1[n];
//     wN[n], corresponding weighting signals.
//
// 1 outputs:
//     y[n], weighted arithmetic mean of the input signals.
//
// 1 compile-time arguments:
//     N, (integer) number of input signals and corresponding weighting
//          signals.
//
avg_ari_w(N) = si.bus(N) ,
               (si.bus(N) <:   si.bus(N) ,
                               (si.bus(N) :> _)) :
     ro.interleave(N, 2) ,
     _ :  (par(i, N, *) :> _) / _;
// ----------------------------------------------------------------------------

// m2.avg_geo(N); -------------------------------------------------------------
//
// Geometric mean.
//
// N inputs:
//     x1[n];
//     ...
```

```
//      xN-1[n];
//      xN[n].
//
// 1 outputs:
//      y[n], geometric mean of the input signals.
//
// 1 compile-time arguments:
//      N, (integer) number of input signals.
//
avg_geo(N) = prod(i, N, _) : pow(1 / N);
// ----------------------------------------------------------------------------

// m2.avg_geo_w(N); -----------------------------------------------------------
//
// Weighted geometric mean.
//
// 2N inputs:
//      x1[n];
//      ...
//      xN-1[n];
//      xN[n], input signals;
//      w1[n];
//      ...
//      wN-1[n];
//      wN[n], corresponding weighting signals.
//
// 1 outputs:
//      y[n], weighted geometric mean of the input signals.
//
// 1 compile-time arguments:
//      N, (integer) number of input signals and corresponding weighting
//           signals.
//
avg_geo_w(N) = (si.bus(N) : par(i, N, log)) ,
               (si.bus(N) <:   si.bus(N) ,
                               (si.bus(N) :> _)) :
      ro.interleave(N, 2) ,
      _ :  (par(i, N, *) :> _) / _ : exp;
```

```
// ---------------------------------------------------------------------------

// m2.avg_harm(N); ----------------------------------------------------------
//
// Harmonic mean.
//
// N inputs:
//     x1[n];
//     ...
//     xN-1[n];
//     xN[n].
//
// 1 outputs:
//     y[n], harmonic mean of the input signals.
//
// 1 compile-time arguments:
//     N, (integer) number of input signals.
//
avg_harm(N) = si.bus(N) : par(i, N, ma.inv) :> N / _;

// m2.avg_harm_w(N); --------------------------------------------------------
//
// Weighted harmonic mean.
//
// 2N inputs:
//     x1[n];
//     ...
//     xN-1[n];
//     xN[n], input signals;
//     w1[n];
//     ...
//     wN-1[n];
//     wN[n], corresponding weighting signals.
//
// 1 outputs:
//     y[n], weighted harmonic mean of the input signals.
//
// 1 compile-time arguments:
```

```
//     N, (integer) number of input signals and corresponding weighting
//           signals.
//
avg_harm_w(N) =     (si.bus(N) : par(i, N, ma.inv)) ,
                    (si.bus(N) <:    si.bus(N) ,
                                     (si.bus(N) :> _)) :
       ro.interleave(N, 2) ,
       _ :  (par(i, N, *) :> _) / _ : ma.inv;
// ------------------------------------------------------------------------

// m2.avg_pow(N, E[n]); --------------------------------------------------
//
// Generalised mean.
//
// N+1 inputs:
//     E[n], exponent for each element in the set;
//     x1[n];
//      ...
//     xN-1[n];
//     xN[n].
//
// 1 outputs:
//     y[n], generalised mean of a set of N elements.
//
// 1 compile-time arguments:
//     N, (integere) number of input signals.
//
avg_pow(N, m) = si.bus(N) : par(i, N, pow(m)) :> / (N) : pow(1 / m);
// ------------------------------------------------------------------------

// m2.avg_quad(N); -------------------------------------------------------
//
// Quadratic mean.
//
// N inputs:
//     x1[n];
//      ...
//     xN-1[n];
```

```
//     xN[n].
//
// 1 outputs:
//     y[n], quadratic mean of the input signals.
//
// 1 compile-time arguments:
//     N, (integer) number of input signals.
//
avg_quad(N) = si.bus(N) : par(i, N, _ <: *) :> / (N) : sqrt;
// ---------------------------------------------------------------

// m2.bip(x[n]); ------------------------------------------------
//
// Unipolar to bipolar signal conversion: [0; 1] range to [-1; 1] range.
//
// 1 inputs:
//     x[n].
//
// 1 outputs:
//     y[n], bipolarised input.
//
bip(x) = x * 2 - 1;
// ---------------------------------------------------------------

// m2.complement(x[n]); -----------------------------------------
//
// Complement.
//
// 1 inputs:
//     x[n].
//
// 1 outputs:
//     y[n], 1 - x[n].
//
complement(x) = 1 - x;
// ---------------------------------------------------------------

// m2.dec2bin(N); -----------------------------------------------
```

```
//
// It converts a decimal integer (N) into a Faust list containing the binary
// digits, that is, the binary digits in parallel.
//
// 0 inputs.
//
// M = ceil(ma.log2(N)) outputs:
//     y1[n],
//     y2[n],
//     ...,
//     yM[n], binary digits representing N.
//
// 1 compile-time arguments:
//     N, decimal integer number.
//
dec2bin(0) = 0:!;
dec2bin(N) =   dec2bin(int(N / 2)) ,
               N % 2;
// ----------------------------------------------------------------------------

// m2.delta(S, dt[n], x[n]); --------------------------------------------------
//
// First derivative using linear interpolation delay lines, hence
// allowing fractional differentiation periods.
//
// 2 inputs:
//     dt[n], differentiation period in seconds;
//     x[n].
//
// 1 outputs:
//     y[n], first derivative of x[n].
//
// 1 compile-time arguments:
//     S, maximum differentiation period in seconds.
//
delta(s, t, in) =  in - d2.del_lin(s, t, in);
// ----------------------------------------------------------------------------
```

```
// m2.delta2(S, dt[n], x[n]); ------------------------------------------------
//
// Second derivative using linear interpolation delay lines, hence
// allowing fractional differentiation periods.
//
// 2 inputs:
//    dt[n], differentiation period in seconds;
//    x[n].
//
// 1 outputs:
//    y[n], second derivative of x[n].
//
// 1 compile-time arguments:
//    S, maximum differentiation period in seconds.
//
delta2(s, t, in) = delta(s, t, in) : delta(s, t);
// ---------------------------------------------------------------------------

// m2.diff(x[n]); ------------------------------------------------------------
//
// Differentiator: special case of a delta-function where the
// differentiation period is 1 / SR.
//
// 1 inputs:
//    x[n].
//
// 1 outputs:
//    y[n], first derivative of x[n].
//
diff(x) = x - x';
// ---------------------------------------------------------------------------

// m2.div(x1[n], x2[n]); -----------------------------------------------------
//
// Divider that avoids divide-by-zero by limiting the denominator to the
// smallest representable value. Essentially, the values of the
// denominator between -epsilon and epsilon are excluded. The corner
// case of x2 = 0 is clipped to positive epsilon.
```

```
//
// For efficiency, for positive denominators the divide-by-zero can
// simply be dealt with as:
//     x1 / max(ma.EPSILON, x2);
//
// while for negative denominators we can use:
//
//     x1 / min(ma.EPSILON * -1, x2);
//
// m2.div covers both cases.
//
// 2 inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n], division safe from divide-by-zero.
//
div(x1, x2) = x1 / m2.if(  x2 < 0,
                           min(ma.EPSILON * -1, x2),
                           max(ma.EPSILON, x2));
// ----------------------------------------------------------------------------

// m2.factorial(N); -----------------------------------------------------------
//
// N!      factorial of N.
//
// 0 inputs.
//
// 1 outputs:
//     y[n], N!.
//
// 1 compile-time arguments:
//     N, positive integer, number to factorialise.
factorial(0) = 0;
factorial(1) = 1;
factorial(N) = N * factorial(N - 1);
// ----------------------------------------------------------------------------
```

```
// m2.hp_and(x1[n], x2[n]); ----------------------------------------
//
// Hyperbolic paraboloid AND.
//
// 2 inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n].
//
hp_and(x, y) = x * y;
// ------------------------------------------------------------------

// m2.hp_imp(x1[n], x2[n]); ----------------------------------------
//
// Hyperbolic paraboloid IMPLIES.
//
// 2 inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n].
//
hp_imp(x, y) = 1 - x + x * y;
// ------------------------------------------------------------------

// m2.hp_nand(x1[n], x2[n]); ---------------------------------------
//
// Hyperbolic paraboloid NOT AND.
//
// 2 inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
```

```
//      y[n].
//
hp_nand(x, y) = 1 - x * y;
// ----------------------------------------------------------------------------

// m2.hp_nimp(x1[n], x2[n]); --------------------------------------------------
//
// Hyperbolic paraboloid NOT IMPLIES.
//
// 2 inputs:
//      x1[n];
//      x2[n].
//
// 1 outputs:
//      y[n].
//
hp_nimp(x, y) = x * (1 - y);
// ----------------------------------------------------------------------------

// m2.hp_nor(x1[n], x2[n]); ---------------------------------------------------
//
// Hyperbolic paraboloid NOT OR.
//
// 2 inputs:
//      x1[n];
//      x2[n].
//
// 1 outputs:
//      y[n].
//
hp_nor(x, y) = 1 - x - y + x * y;
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------
// hyperbolic paraboloid NOT excluding OR
// m2.hp_nxr(x1[n], x2[n]); ---------------------------------------------------
//
// Hyperbolic paraboloid NOT excluding OR.
```

```
//
// 2 inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n].
//
hp_nxr(x, y) = 1 - x - y + 2 * x * y;
// ------------------------------------------------------------------------

// m2.hp_or(x1[n], x2[n]); ------------------------------------------------
//
// Hyperbolic paraboloid OR.
//
// 2 inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n].
//
hp_or(x, y) = x + y - x * y;
// ------------------------------------------------------------------------

// m2.hp_xor(x1[n], x2[n]); -----------------------------------------------
//
// Hyperbolic paraboloid excluding OR.
//
// 2 inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n].
//
hp_xor(x, y) = x + y - 2 * x * y;
// ------------------------------------------------------------------------
```

```
// m2.interpolate_mn(M, N, I[n]); ————————————————————————————————
//
// Linear interpolation to circularly transition between between M blocks of
// N signals ("spat" by Romain Michon). The function shifts through the
// M blocks based on an index in the range [0; 1], representing the
// beginning and end of an interpolation cycle. For example, if we have four
// blocks, the index from 0 to .25 is the transition between block one and
// two, .25 to .5 is the transition between two and three, .5 to .75 is the
// transition between three and four, and .75 to 1 is the transition
// between four and one.
//
// M*N+1 inputs:
//     I[n], interpolation index in the [0; 1] range performing a
//           complete interpolation cycle among the M blocks.
//     x11[n];
//     ...
//     x1N-1[n];
//     x1N[n];
//     ...
//     xM-11[n];
//     ...
//     xM-1N-1[n];
//     xM-1N[n];
//     ...
//     xMN[n].
//
// N outputs:
//     y1[n];
//     ...
//     yN-1[n];
//     yN[n], interpolated N-signal block.
//
// 2 compile-time arguments:
//     M, (integer) number of blocks;
//     N, integer number of signals in each block.
//
interpolate_mn(m, n, index) =  spat(m, index, 1) ,
```

```
                              ro.interleave(n, m) :
    ro.interleave(m, n + 1) :     par(i, m, (_ <: si.bus(n)) ,
                                  si.bus(n) : ro.interleave(n, 2) :
        par(i, n, *)) :> si.bus(n)
    with {
        spat(n, a, d) = par(i, n, (scaler(i, n, a, d) ))
            with {
                scaler(i, n, a, d) = (d / 2.0 + .5) * max(.0, 1.0 -
                    abs(fmod(a + .5 + float(n - i) / n, 1.0) - .5) * n * d);
            };
    };
// ---------------------------------------------------------------------------

// m2.if(C[n], T[n], E[n]); ------------------------------------------------
//
// If-then-else    unlike Faust's if-then-else, the condition is true for
// any value != than 0, whereas Faust gives false for |fractions| < 1.
//
// 3 inputs:
//    C[n], condition;
//    T[n], "then" signal;
//    E[n], "else" signal.
//
// 1 outputs:
//    y[n], which is either T[n] or E[n] depending on C[n].
//
if(cond, then, else) = cond != 0 ,
                       else ,
                       then : select2;
// ---------------------------------------------------------------------------

// m2.ifN((C1[n], T1[n], ..., CN-1[n], TN-1[n], CN[n], TN[n], E[n])); ---------
//
// If-then-else-if-... with arbitrary number of conditions.
// It takes pairs of IF-THEN couples plus a final ELSE.
// These should be given as a list, hence within parentheses.
// Note that it only works if outputting single values for each
// condition due to Faust's limitations in list processing.
```

```
// Code by Oleg Nesterov.
//
// N*2+1 inputs:
//     C1[n];
//     T1[n];
//     ...;
//     CN-1[n];
//     TN-1[n];
//     CN[n];
//     TN[n];
//     E[n].     (Note that the number of inputs depends on the specific
//               functions of each statement.)
//
// 1 outputs:
//     y[n], one of the T-signals or the E-signal.
//
ifN((c, t, e)) = m2.if(c, t, ifN(e));
ifN(e) = e;
// ---------------------------------------------------------------------------

// m2.inv(x[n]); ------------------------------------------------------------
//
// INF-safe inverse.
//
// 1 inputs:
//     x[n].
//
// 1 outputs:
//     1 / x[n].
//
inv(x) = div(1, x);
// ---------------------------------------------------------------------------

// m2.line(S[n]); -----------------------------------------------------------
//
// Line function. Note that the function does not start from 0. If you
// require a precise behaviour, you can use the following:
//     m2.line(ba.select2(1', 0, S[n]));
```

```
//
// 1 inputs:
//    S[n], the slope of the line (y/x ratio).
//
// 1 outputs:
//    y[n], the line function.
//
line(s) = +(s / ma.SR)
          ~ _;
// ------------------------------------------------------------------------------

// m2.line_reset(S[n], reset[n]); ----------------------------------------------
//
// Line function with reset input. Note that the function does not start
// from 0 until it is reset. If you require a precise behaviour from the very
// beginning, you can use the following:
//    m2.line_reset(ba.select2(1', 0, S[n]), reset[n]);
//
// 1 inputs:
//    S[n], the slope of the line (y/x ratio);
//    reset[n], it resets the line to 0 if its value is non-zero.
//
// 1 outputs:
//    y[n], the line function.
//
line_reset(rate, reset) = +(rate / ma.SR * r)
                          ~ *(r)
    with {
        r = 1 - (reset != 0);
    };
// ------------------------------------------------------------------------------

// m2.map_lin(L[n], H[n], x[n]); -----------------------------------------------
//
// Linear mapping of an input signal in the range [0; 1] on to an arbitrary
// range determined by signals.
//
// 3 inputs:
```

```
//     L[n], lower edge of the range;
//     H[n], upper edge of the range;
//     x[n], input [0; 1].
//
// 1 outputs:
//     y[n], mapped x[n].
//
map_lin(lower, upper, x) = x * (upper - lower) + lower;
// ----------------------------------------------------------------------------

// m2.map_par(T[n], V[n], S[n], x[n]); ----------------------------------------
//
// Parabolic mapping of an input signal in the range [0; 1] on to a
// parabola whose curvature, vertex, and sides are determined by signals.
//
// 4 inputs:
//     T[n], tension parameter (curvature), the exponent, which is
//               constrained to be an even integer;
//     V[n], vertex of the parabola;
//     S[n], value of the sides of the parabola at the minimum and
//               maximum values of the input signal (0 and 1).
//     x[n], input [0; 1].
//
// 1 outputs:
//     y[n], mapped x[n].
//
map_par(t, vertex, sides, x) = x * 2 - 1 : pow(t_1) : m2.map_lin(vertex, sides)
    with {
        t_1 = int(t) * 2;
    };
// ----------------------------------------------------------------------------

// m2.map_pcw(S[n], E1[n], L1[n], H1[n], E2[n], L2[n], H2[n], x[n]); ----------
//
// Piece-wise mapping of an input signal in the range [0; 1] on to two
// segments with independent curves and ranges determined by signals.
//
// 8 inputs:
```

```
//     S[n], split-point up until the characteristics of the first
//               segment apply, where as those from the second segment
//               apply if the signal is above it;
//     E1[n], exponent for the first segment;
//     L1[n], lower edge of the first segment;
//     H1[n], upper edge of the first segment;
//     E2[n], exponent of the second segment;
//     L2[n], lower edge of the second segment;
//     H2[n], upper edge of the second segment;
//     x[n], input in the range [0; 1].
//
// 1 outputs:
//     y[n], mapped x[n].
//
map_pcw(split, exp1, l1, u1, exp2, l2, u2, x) =
      m2.if(    x <= split,
                map_pow(exp1, l1, u1, x),
                m2.map_pow(exp2, l2, u2, x));
// ------------------------------------------------------------------------

// m2.map_log(T[n], L[n], H[n], x[n]); ------------------------------------
//
// Mapping of an input signal in the range [0; 1] on to a logarithmic
// curve with arbitrary ranges and curvature determined by signals.
//
// 4 inputs:
//     T[n], 'tension' parameter      curvature of the log function;
//     L[n], lower edge of the function;
//     H[n], upper edge of the function;
//     x[n], input in the range [0; 1].
//
// 1 outputs:
//     y[n], mapped x[n].
//
map_log(t, lower, upper, x) = log(x * (t - 1) + 1) / log(t) :
      m2.map_lin(lower, upper);
// ------------------------------------------------------------------------
```

```
// m2.map_pow(T[n], L[n], H[n], x[n]); ----------------------------------------
//
// Mapping of an input signal in the range [0; 1] on to a power curve
// with arbitrary curvature and ranges determined by signals.
//
// 4 inputs:
//     T[n], `tension` parameter     curvature of the power function;
//     L[n], lower edge of the function;
//     H[n], upper edge of the function;
//     x[n], input in the range [0; 1].
//
// 1 outputs:
//     y[n], mapped x[n].
//
map_pow(t, lower, upper, x) = pow(x, t) : m2.map_lin(lower, upper);
// -----------------------------------------------------------------------------

// m2.matrix(R, C); ------------------------------------------------------------
//
// R-input, C-output matrix:
//
// a11 a12      a1C
// a21 a22      a2C
//
// aR1 aR2      aRC
//
// R+R*C inputs:
//     R, input signals to be distributed through the C outputs;
//     R*C, coefficients as shown in the diagram above;
//
// C outputs.
//
// 2 compile-time arguments:
//     R, (integer) number of rows;
//     C, (integer) number columns.
//
matrix(r, c) = (si.bus(r), ro.interleave(c, r)) : ro.interleave(r, c + 1) :
       par(i, r, (_ <: si.bus(c)) ,
```

```
    si.bus(c) : ro.interleave(c, 2) : par(i, c, *)) :> si.bus(c);
// ----------------------------------------------------------------------------

// m2.maxN(N); ----------------------------------------------------------------
//
// It returns the max value between a specified number of input signals.
//
// N inputs.
//
// 1 outputs:
//    y[n], max value between the N signals.
//
// 1 compile-time arguments:
//    N, (integer) number of input signals.
//
maxN(1) = _;
maxN(2) = max;
maxN(N) = max(maxN(N - 1));
// ----------------------------------------------------------------------------

// m2.minN(N); ----------------------------------------------------------------
//
// It returns the min value between a specified number of input signals.
//
// N inputs.
//
// 1 outputs:
//    y[n], min value between the N signals.
//
// 1 compile-time arguments:
//    N, (integer) number of input signals.
//
minN(1) = _;
minN(2) = min;
minN(N) = min(minN(N - 1));
// ----------------------------------------------------------------------------

// m2.ny; ---------------------------------------------------------------------
```

```
//
// Nyquist or half the samplerate.
//
// 0 inputs.
//
// 1 outputs:
//     y[n], SR/2.
//
ny = ma.SR / 2;
// ------------------------------------------------------------

// m2.ph(F[n], reset[n]); -------------------------------------
//
// Phasor with phase reset.
//
// 2 inputs:
//     F[n], frequency in Hz;
//     reset[n], it resets the phasor to 0 for values different than 0.
//
// 1 outputs:
//     y[n], phasor oscillator.
//
ph(freq, reset) = freq / ma.SR * r : (+ : ma.decimal)
                                     ~ * (r)
    with {
        r = 1 - (reset != 0);
    };
// ------------------------------------------------------------

// m2.primes; -------------------------------------------------
//
// First 64 prime numbers.
//
// 0 inputs.
//
// 64 outputs:
//     2;
//     3;
```

```
//      5;
//      ...;
//      311.
//
primes = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149,
151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233,
239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311);
// ---------------------------------------------------------------------------

// m2.prime_base_pow(N, E[n]); -----------------------------------------------
//
// N-th prime number raised to a power.
//
// 1 inputs:
//     E[n], exponent.
//
// 1 outputs:
//     y[n], N-th prime number raised to a power.
//
// 1 compile-time arguments:
//     N, (integer) N-th prime number up to 64.
//
prime_base_pow(N, exponent) = pow(ba.take(N, primes), exponent);
// ---------------------------------------------------------------------------

// m2.relay_hysteron(A[n], B[n], x[n]); --------------------------------------
//
// Relay hysteron      the basic building block of the Preisach model.
//
// 3 inputs:
//     A[n], lower edge;
//     B[n], upper edge;
//     x[n].
//
// 1 outputs:
//     y[n], relay output, 0 or 1 (non-active or active).
//
```

```
relay_hysteron(alpha, beta, x) =   loop
                                   ~ _
with {
    loop(fb) = m2.if(   x <= alpha,
                        0, m2.if(   x >= beta,
                                    1,
                                    fb));
};
// ----------------------------------------------------------------------------

// m2.round_pow2(x[n]); ------------------------------------------------------
//
// Closest power-of-two to the input signal.
//
// 1 inputs:
//    x[n].
//
// 1 outputs:
//    y[n], closest power-of-two to x[n].
//
round_pow2(x) = 2 ^ (rint(ma.log2(x)));
// ----------------------------------------------------------------------------

// m2.rt9(T[n]); -------------------------------------------------------------
//
// One-pole coefficient for ~9 dB exponential decay (e^-1 factor) in a desired
// time specified in seconds.
//
// 1 inputs:
//    T[n], decay time in seconds.
//
// 1 outputs:
//    y[n], pole position.
//
rt9(t) = ba.tau2pole(t);
// ----------------------------------------------------------------------------

// m2.rt19(T[n]); ------------------------------------------------------------
```

```
//
// One-pole coefficient for ~19 dB exponential decay (e^-2.2 factor factor)
// in a desired time specified in seconds. This decay factor is used in
// [Z lzer 2008] for the calculation of RMS with 1-pole filters.
//
// 1 inputs:
//    T[n], decay time in seconds.
//
// 1 outputs:
//    y[n], pole position.
//
rt19(t) = ba.tau2pole(t / 2.2);
// ----------------------------------------------------------------

// m2.rt55(T[n]); ------------------------------------------------
//
// One-pole coefficient for ~55 dB exponential decay (e^(-2pi) factor) in a
// desired time specified in seconds.
// This coefficient calculation is found in [Chamberlin 1985] for the
// implementation of one-pole lowpass filters that simulate the dis/charging
// behaviours of a capacitor.
//
// 1 inputs:
//    T[n], decay time in seconds.
//
// 1 outputs:
//    y[n], pole position.
//
rt55(t) = ba.tau2pole(t / m2.twopi);
// ----------------------------------------------------------------

// ----------------------------------------------------------------
// 1-pole coeff. for ~60 dB decay (e^-log(1000) factor) in a desired time "t".
// m2.rt60(T[n]); ------------------------------------------------
//
// One-pole coefficient for ~60 dB exponential decay (e^-log(1000) factor) in a
// desired time specified in seconds.
// This coefficient calculation is commonly seen in the design of
```

```
// artificial reverberation.
//
// 1 inputs:
//     T[n], decay time in seconds.
//
// 1 outputs:
//     y[n], pole position.
//
rt60(t) = ba.tau2pole(t / 6.907755279);
// ------------------------------------------------------------------------------

// m2.sd(N); --------------------------------------------------------------------
//
// Stanrdard deviation.
//
// N inputs.
//
// 1 outputs:
//     y[n], standard deviation between the N input signals.
//
// 1 compile-time arguments:
//     N, input signals.
//
sd(N) = sqrt(var(N));
// ------------------------------------------------------------------------------

// m2.seq_catalan(N); -----------------------------------------------------------
//
// First N numbers in the Catalan sequence:
//     1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...
//
// Note that, in single-precision, the function is correct only up to
// the 7th element.
//
// 0 inputs.
//
// N outputs:
//     y1[n];
```

```
//     y2[n];
//     ...;
//     yN-1[n];
//     y[N], the first N numbers in the Catalan sequence.
//
// 1 compile-time arguments:
//     N, the first N numbers in the Catalan sequence.
//
seq_catalan(1) = 1;
seq_catalan(N) =   1 ,
                    par(i, N - 1, factorial(2 * (i + 1)) /
                        (factorial(i + 1 + 1) * factorial(i + 1)));
// ----------------------------------------------------------------------------

// m2.seq_fibonacci(N); ------------------------------------------------------
//
// First N numbers of the Fibonacci sequence:
//     0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
//
// 0 inputs.
//
// N outputs:
//     y1[n];
//     y2[n];
//     ...;
//     yN-1[n];
//     y[N], the first N numbers in the Fibonacci sequence.
//
// 1 compile-time arguments:
//     N, the first N numbers in the Fibonacci sequence.
//
seq_fibonacci(1) = 0;
seq_fibonacci(2) = 0 ,
                   1;
seq_fibonacci(3) = 0 ,
                   1 ,
                   1;
seq_fibonacci(N) = seq_fibonacci(N - 1) <: si.bus(N - 1) ,
```

```
                                       (   par(i, N - 3, !) ,
                                           si.bus(2) :> _);
// ----------------------------------------------------------------------------

// m2.seq_hexagonal(N); -------------------------------------------------------
//
// First N numbers of the hexagonal sequence:
//    1, 6, 15, 28, 45, 66, 91, 120, 153, 190, ...
//
// 0 inputs.
//
// N outputs:
//    y1[n];
//    y2[n];
//    ...;
//    yN-1[n];
//    y[N], the first N numbers in the hexagonal sequence.
//
// 1 compile-time arguments:
//    N, the first N numbers in the hexagonal sequence.
//
seq_hexagonal(N) = par(i, N, 2 * (i + 1) * (2 * (i + 1) - 1) / 2);
// ----------------------------------------------------------------------------

// m2.seq_lazy_caterer(N); ----------------------------------------------------
//
// First N numbers of the Lazy Caterer sequence:
//    1, 2, 4, 7, 11, 16, 22, 29, 37, 46, ...
//
// 0 inputs.
//
// N outputs:
//    y1[n];
//    y2[n];
//    ...;
//    yN-1[n];
//    y[N], the first N numbers in the Lazy Caterer sequence.
//
```

```
// 1 compile-time arguments:
//     N, the first N numbers in the Lazy Caterer sequence.
//
seq_lazy_caterer(N) = par(i, N, (i ^ 2 + i + 2) / 2);
// ------------------------------------------------------------------------

// m2.seq_magic_number(N); ------------------------------------------------
//
// First N numbers of the Magic Number sequence:
//     15, 34, 65, 111, 175, 260, 369, 505, 671, 870, ...
//
// The Magic Number is a constant for an Mth-order square matrix,
// assuming M > 2, of integer values, with the max value being M^2,
// where the constant corresponds to the sum of the numbers in any
// of its rows, columns, or diagonals.
//
// Example of order-3 magic square (constant 15):
//
//     4 9 2
//     3 5 7
//     8 1 6
//
// 0 inputs.
//
// N outputs:
//     y1[n];
//     y2[n];
//     ...;
//     yN-1[n];
//     y[N], the first N numbers in the Magic Number sequence.
//
// 1 compile-time arguments:
//     N, first N numbers in the Magic Number sequence.
//
seq_magic_number(N) = par(i, N, (i + 3) * ((i + 3) ^ 2 + 1) / 2);
// ------------------------------------------------------------------------

// m2.seq_pentagonal(N); --------------------------------------------------
```

```
//
// First N numbers of the pentagonal sequence:
//     1, 5, 12, 22, 35, 51, 70, 92, 117, 145, ...
//
// 0 inputs.
//
// N outputs:
//     y1[n];
//     y2[n];
//     ...;
//     yN-1[n];
//     y[N], the first N numbers in the pentagonal sequence.
//
// 1 compile-time arguments:
//     N, the first N numbers in the pentagonal sequence.
//
seq_pentagonal(N) = par(i, N, (3 * (i + 1) ^ 2 - (i + 1)) / 2);
// ----------------------------------------------------------------

// m2.seq_square(N); ----------------------------------------------
//
// First N numbers of the square sequence:
//     1, 4, 9, 16, 25, 36, 49, 64, 81, 100, ...
//
// 0 inputs.
//
// N outputs:
//     y1[n];
//     y2[n];
//     ...;
//     yN-1[n];
//     y[N], the first N numbers in the square sequence.
//
// 1 compile-time arguments:
//     N, the first N numbers in the square sequence.
//
seq_square(N) = par(i, N, (i + 1) ^ 2);
// ----------------------------------------------------------------
```

```
// m2.seq_triangular(N); ————————————————————————————————————
//
// First N numbers of the triangular sequence:
//    1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...
//
// 0 inputs.
//
// N outputs:
//    y1[n];
//    y2[n];
//    ...;
//    yN-1[n];
//    y[N], the first N numbers in the triangular sequence.
//
// 1 compile-time arguments:
//    N, the first N numbers in the triangular sequence.
//
seq_triangular(N) = par(i, N, (i + 1) * (i + 1 + 1) / 2);
// ————————————————————————————————————————————————————————————

// m2.sp; ———————————————————————————————————————————————————————
//
// Sampling period.
//
// 0 inputs.
//
// 1 outputs:
//    y[n], sampling period, 1 / SR.
//
sp = 1 / ma.SR;
// ————————————————————————————————————————————————————————————

// m2.topologies(R, C, I[n]); ————————————————————————————————
//
// Linear interpolation to circularly shift through matrix topologies
// according to an index in the range [0; 1]. There are four topologies
// with arbitrary rows and columns: full, anti-diagonal, diagonal, and
```

```
// right-shifted diagonal. The ranges to transition among the four topologies
// are 0 to .25 to shift from the first to the second topology, .25 to
// .5 from the second to the third, .5 to .75 from the third to the fourth,
// and .75 to 1 from the fourth and back to the first one.
//
// The output of this function can be used as the coefficients for an
// m2.matrix function that has the same number of rows and columns.
//
// 1 inputs:
//     I[n], interpolation index in the range [0; 1].
//
// R*C outputs:
//     y11 y12      y1C
//     y21 y22      y2C
//
//     yR1 yR2      yRC
//
// 2 compile-time arguments:
//     R, (integer) number of rows;
//     C, (integer) number columns.
//
topologies(r, c, type) =    top_full(r, c) ,
                            top_anti_diag(r, c) ,
                            top_diagonal(r, c) ,
                            top_diag_shift(r, c, 1) :
     interpolate_mn(4, r * c, type);
// ------------------------------------------------------------------------

// m2.top_anti_diag(R, C); ------------------------------------------------
//
// Anti-diagonal matrix topology.
//
// Example for a 4x4 matrix:
//
//     0 1 1 1
//     1 0 1 1
//     1 1 0 1
//     1 1 1 0
```

```
//
// 0 inputs.
//
// R*C outputs:
//     y11 y12      y1C
//     y21 y22      y2C
//
//     yR1 yR2      yRC
//
// 2 compile-time arguments:
//     R, (integer) number of rows;
//     C, (integer) number columns.
//
top_anti_diag(r, c) = par(i, r, par(j, c, (j % r) != (i % c)));
// ----------------------------------------------------------------------------

// m2.top_diagonal(R, C); ----------------------------------------------------
//
// Diagonal matrix topology.
//
// Example for a 4x4 matrix:
//
//     1 0 0 0
//     0 1 0 0
//     0 0 1 0
//     0 0 0 1
//
// 0 inputs.
//
// R*C outputs:
//     y11 y12      y1C
//     y21 y22      y2C
//
//     yR1 yR2      yRC
//
// 2 compile-time arguments:
//     R, (integer) number of rows;
//     C, (integer) number columns.
```

```
//
top_diagonal(r, c) = par(i, r, par(j, c, (j % r) == (i % c)));
// ---------------------------------------------------------------------------

// m2.top_diag_shift(R, C, S[n]); ---------------------------------------
//
// Shifted diagonal matrix topology.
//
// Example for a 4x4 matrix and a one-position-shift:
//
//     0 1 0 0
//     0 0 1 0
//     0 0 0 1
//     1 0 0 0
//
// 1 inputs:
//     S[n], cast to integer, it determines the shift amount.
//
// R*C outputs:
//     y11 y12      y1C
//     y21 y22      y2C
//
//     yR1 yR2      yRC
//
// 2 compile-time arguments:
//     R, (integer) number of rows;
//     C, (integer) number columns.
//
top_diag_shift(r, c, s) =
     par(i, r, par(j, c, (j  % r) == ((i + int(s)) % c)));
// ---------------------------------------------------------------------------

// m2.top_full(R, C); ---------------------------------------------------
//
// Fully-connected matrix topology. Essentially, the function outputs as
// many 1s in parallel as the product between the specified rows and columns.
//
// Example for a 4x4 matrix:
```

```
//
//     1 1 1 1
//     1 1 1 1
//     1 1 1 1
//     1 1 1 1
//
// 0 inputs.
//
// R*C outputs:
//     y11 y12       y1C
//     y21 y22       y2C
//
//     yR1 yR2       yRC
//
// 2 compile-time arguments:
//     R, (integer) number of rows;
//     C, (integer) number columns.
//
top_full(r, c) = 1 <: si.bus(r * c);
// ----------------------------------------------------------------------------

// m2.top_tri_low(R, C); -----------------------------------------------------
//
// Lower-triangle matrix topology.
//
// Example for a 4x4 matrix:
//
//     1 0 0 0
//     1 1 0 0
//     1 1 1 0
//     1 1 1 1
//
// 0 inputs.
//
// R*C outputs:
//     y11 y12       y1C
//     y21 y22       y2C
//
```

```
//     yR1 yR2      yRC
//
// 2 compile-time arguments:
//     R, (integer) number of rows;
//     C, (integer) number columns.
//
top_tri_low(r, c) = par(i, r, par(j, c, (j % r) <= (i % c)));
// ----------------------------------------------------------------------------

// m2.top_tri_up(R, C); -----000----------------------------------------------
//
// Upper-triangle matrix topology.
//
// Example for a 4x4 matrix:
//
//     1 1 1 1
//     0 1 1 1
//     0 0 1 1
//     0 0 0 1
//
// 0 inputs.
//
// R*C outputs:
//     y11 y12      y1C
//     y21 y22      y2C
//
//     yR1 yR2      yRC
//
// 2 compile-time arguments:
//     R, (integer) number of rows;
//     C, (integer) number columns.
//
top_tri_up(r, c) = par(i, r, par(j, c, (j % r) >= (i % c)));
// ----------------------------------------------------------------------------

// m2.uni(x[n]); -------------------------------------------------------------
//
// Function to convert a bipolar signal to unipolar, assuming an input
```

```
// in the range [-1; 1].
//
// 1 inputs:
//     x[n], input signal assumed in the range [-1; 1].
//
// 1 outputs:
//     y[n], output in the range [0; 1].
//
uni(x) = (x + 1) / 2;
// ----------------------------------------------------------------------------

// m2.twopi -------------------------------------------------------------------
//
// 2    constant.
//
// 0 inputs.
//
// 1 outputs:
//     y[n], two times    .
//
twopi = 2 * ma.PI;
// ----------------------------------------------------------------------------

// m2.unit_log(T[n], x[n]); ---------------------------------------------------
//
// Logarithmic mapping of an input signal in the range [0; 1] into an
// output in the same range and arbitrary curvature (tension factor).
//
// 2 inputs:
//     T[n], curvature, tensione factor;
//     x[n], input in the range [0; 1].
//
// 1 outputs:
//     y[n], logarithmically-mapped x[n] in the range [0; 1].
//
unit_log(t, x) = log(x * (t - 1) + 1) / log(t);
// ----------------------------------------------------------------------------
```

```
// m2.var(N); ------------------------------------------------------------------
//
// Variance of a set of N elements.
//
// N inputs.
//
// 1 outputs:
//     y[n], variance of the N input signals.
//
// 1 compile-time arguments:
//     N, (integer) number of input signals.
//
var(N) = si.bus(N) <:  si.bus(N) ,
                        (si.bus(N) :> / (N) <: si.bus(N)) :
       ro.interleave(N, 2) : par(i, N, - <: *) :> / (N);
// -----------------------------------------------------------------------------

// m2.w(F[n]); -----------------------------------------------------------------
//
// Angular frequency.
//
// 1 inputs:
//     F[n], frequency in Hz.
//
// 1 outputs:
//     y[n], angular frequency between 0 and   , assuming an input between
//          0 and Nyquist.
//
w(x) = x * m2.twopi / ma.SR;
// -----------------------------------------------------------------------------

// m2.window_hann(ph[n]); ------------------------------------------------------
//
// Hann window.
//
// 1 inputs:
//     ph[n], phase of the function where a full cycle is in the range [0; 1].
//
```

```
// 1 outputs:
//     y[n], Hann function.
//
window_hann(x) = sin(ma.PI * x) <: *;
// ----------------------------------------------------------------------------

// m2.window_sine(ph[n]); ----------------------------------------------------
//
// Sine window.
//
// 1 inputs:
//     ph[n], phase of the function where a full cycle is in the range [0; 1].
//
// 1 outputs:
//     y[n], Sine function.
//
window_sine(x) = sin(ma.PI * x);
// ----------------------------------------------------------------------------

// m2.wrap(L[n], H[n], x[n]); ------------------------------------------------
//
// Wrapping function.
//
// 3 inputs:
//     L[n], lower edge;
//     H[n], upper edge;
//     x[n].
//
// 1 outputs:
//     y[n], wrapped-up x[n].
//
wrap(lower, upper, x) =
    (x - lower) / (upper - lower) : ma.decimal * (upper - lower) + lower;
// ----------------------------------------------------------------------------

// m2.y_and(x1[n], x2[n]); ---------------------------------------------------
//
// Yager AND.
```

```
//
// inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n], x1[n] AND x2[n].
//
y_and(x, y) = 1 - min(1, sqrt((1 - x) ^ 2 + (1 - y) ^ 2));
// ----------------------------------------------------------------------------

// m2.y_or(x1[n], x2[n]); -----------------------------------------------------
//
// Yager OR.
//
// inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n], x1[n] OR x2[n].
//
y_or(x, y) = min(1, (x ^ 2 + y ^ 2) ^ 2);
// ----------------------------------------------------------------------------

// m2.zeropad_up(N, list); ----------------------------------------------------
//
// This function adds N zeros at the beginning of a list.
//
// 0 inputs.
//
// N+ba.count(list) outputs.
//
// 1 compile-time arguments:
//     N, number of zeros to be added.
//
zeropad_up(0, x) = x;
zeropad_up(N, x) = par(i, N, 0), x;
```

```
// ---------------------------------------------------------------------------

// m2.zeropad_down(N, list); ------------------------------------------------
//
// This function adds N zeros at the end of a list.
//
// 0 inputs.
//
// N+ba.count(list) outputs.
//
// 1 compile-time arguments:
//     N, number of zeros to be added.
//
zeropad_down(0, x) = x;
zeropad_down(N, x) = par(i, N, 0) , x;
// ---------------------------------------------------------------------------

// m2.z_and(x1[n], x2[n]); --------------------------------------------------
//
// Zadeh AND.
//
// inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n], x1[n] AND x2[n].
//
z_and(x, y) = min(x, y);
// ---------------------------------------------------------------------------

// m2.z_imp(x1[n], x2[n]); --------------------------------------------------
//
// Zadeh IMPLIES.
//
// inputs:
//     x1[n];
//     x2[n].
```

```
//
// 1 outputs:
//     y[n], x1[n] IMPLIES x2[n].
//
z_imp(x, y) = 1 - min(x, 1 - y);
// ---------------------------------------------------------------

// m2.z_nand(x1[n], x2[n]); --------------------------------------
//
// Zadeh NOT AND.
//
// inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n], x1[n] NAND x2[n].
//
z_nand(x, y) = 1 - min(x, y);
// ---------------------------------------------------------------

// m2.z_nimp(x1[n], x2[n]); --------------------------------------
//
// Zadeh NOT IMPLIES.
//
// inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n], x1[n] NOT IMPLIES x2[n].
//
z_nimp(x, y) = min(x, 1 - y);
// ---------------------------------------------------------------

// m2.z_nor(x1[n], x2[n]); ---------------------------------------
//
// Zadeh NOT OR.
```

```
//
// inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n], x1[n] NOR x2[n].
//
z_nor(x, y) = 1 - max(x, y);
// ----------------------------------------------------------------------

// m2.z_nxr(x1[n], x2[n]); ---------------------------------------------
//
// Zadeh NOT XOR.
//
// inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n], x1[n] NXR x2[n].
//
z_nxr(x, y) = 1 - x - y + 2 * min(x, y);
// ----------------------------------------------------------------------

// m2.z_or(x1[n], x2[n]); ----------------------------------------------
//
// Zadeh OR.
//
// inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n], x1[n] OR x2[n].
//
z_or(x, y) = max(x, y);
// ----------------------------------------------------------------------
```

```
// m2.z_xor(x1[n], x2[n]); ----------------------------------------------------
//
// Zadeh XOR.
//
// inputs:
//     x1[n];
//     x2[n].
//
// 1 outputs:
//     y[n], x1[n] XOR x2[n].
//
z_xor(x, y) = x + Y - 2 * min(x, y);
// ---------------------------------------------------------------------------
```

A.8 oscillators2.lib

```
// ===========================================================================
// ========== oscillators2.lib ===============================================
// ===========================================================================
//
// This library contains band-limited oscillators with arbitrary harmonic
// content for classic analogue waveforms such as sawtooth, square,
// and triangle waves, as well as band-limited pulse-trains with arbitrary
// duty-cycles. The library includes a quadrature oscillator based on,
// arguably, the best recursive design available in the literature,
// self-oscillating systems based on chaotic functions, and iterative
// systems for complex patterns such as cellular automata.
//
// The environment prefix is "o2".
//
// List of functions:
//
//     blit_bi,
//     blit_bi_duty,
//     blit_uni,
//     eca,
//     lorenz,
```

```
//     osc_quad,
//     ph,
//     pulse_train,
//     saw,
//     square,
//     tri.
//
// Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario at gmail dot com>
// All rights reserved.

declare name "Oscillators Library";
declare author "Dario Sanfilippo";
declare copyright "Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario
      at gmail dot com>";
declare version "1.0.0";
declare license "GPLv2.0";

au = library("auxiliary.lib");
ba = library("basics.lib");
fi = library("filters.lib");
f2 = library("filters2.lib");
ma = library("maths.lib");
m2 = library("maths2.lib");
os = library("oscillators.lib");
o2 = library("oscillators2.lib");

// o2.blit_bi(H[n], F[n]); ------------------------------------------------
//
// Bipolar band-limited impulse train (BLIT) based on period sinc function.
//
// The maximum harmonic number is given by:
//
//     rint(SR/frequency/4).
//
// The bipolar BLIT has no DC component and the number of harmonics include
// the fundamental frequency and its odd multiples. Variations in the harmonic
// content take place at the beginning of each cycle to avoid clicks. Hence,
// changes will take place after a time that is the period of the BLIT or less.
```

```
//
// Unlike the technique described in [Stilson and Smith 1996] where the
// bipolar BLIT is implemented by summation of a unipolar BLIT with its
// delayed and inverted copy, the technique showed here uses an even ratio
// between the frequencies of the sine functions used to generate the sinc,
// which results in a correct harmonic content (odd harmonics) for any given
// BLIT frequencies.
//
// The amplitude of the function, regardless of the harmonics,
// is normalised to unit-amplitude peaks.
//
// 2 inputs:
//    H[n], (rounded to rint inside the asinc function) number of harmonics;
//    F[n], BLIT frequency in Hz.
//
// 1 outputs:
//    y[n], bipolar band-limited impulse train.
//
blit_bi(h, f) = m2.asinc_bi(h1, phase)
      with {
            lim = rint(m2.div(ma.SR, f) / 4);
            h1 = ba.sAndH(trigger, min(lim, h));
            trigger = (ma.signum(f) * (phase - phase') < 0);
            // Add au.dirac to "trigger" if you require the initial value
            // of H[n] to be triggered initially at n = 0.
            phase = os.phasor(1, f);
      };
// ------------------------------------------------------------------------------

// o2.blit_bi_duty(H[n], D[n], F[n]); ---------------------------------------------
//
// Bipolar band-limited impulse train with arbitrary duty cycle
// following the paper by [Stilson and Smith 1996]:
//
// https://ccrma.stanford.edu/~stilti/papers/blit.pdf.
//
// The lowest frequency at which an entire duty cycle can be explored is
// 1 Hz. If lower frequencies are required, the first argument of
```

```
// m2.delta should be changed.
//
// 3 inputs:
//    H[n], (rounded to rint) number of harmonics;
//    D[n], duty cycle in the range [0; 1];
//    F[n], frequency of the BLIT in Hz.
//
// 1 outputs:
//    y[n], bipolar BLIT with arbitrary duty cycle.
//
blit_bi_duty(h, d, f) = m2.delta(1, d1, o2.blit_uni(h, f))
      with {
           d1 = d * m2.div(1, f);
      };
// ----------------------------------------------------------------------------

// o2.blit_uni(H[n], F[n]); --------------------------------------------------
//
// The unipolar BLIT has a DC component and the number of harmonics include the
// fundamental frequency and its multiples (both even and odd).
//
// The technique described here is based on the paper by
// [Stilson and Smith 1996]:
//
// https://ccrma.stanford.edu/~stilti/papers/blit.pdf.
//
// The amplitude of the function, regardless of the harmonics,
// is normalised to unit-amplitude peaks.
//
// 2 inputs:
//    H[n], (rounded to rint inside the asinc function) number of harmonics;
//    F[n], BLIT frequency in Hz.
//
// 1 outputs:
//    y[n], unipolar band-limited impulse train.
//
blit_uni(h, f) = m2.asinc_uni(h1, phase)
      with {
```

```
        lim = floor(m2.div(ma.SR, f) / 2);
        h1 = ba.sAndH(trigger, min(lim, h));
        trigger = (ma.signum(f) * (phase - phase') < 0);
        // Add au.dirac to "trigger" if you require the initial value
        // of H[n] to be triggered initially at n = 0.
        phase = os.phasor(1, f);
    };
// ----------------------------------------------------------------------------

// o2.eca(L, R, I, rate[n]); -------------------------------------------------
//
// One-dimension, two-state, elementary cellular automata with circular
// lattice. The function is defined by the length of the lattice, a rule, and
// an initial condition. Additionally, the function has a "rate" parameter
// that determines the interval between iterations. The rule and the initial
// condition are positive INTs that are converted into binary numbers and
// accordingly zero-padded or limited to reach a binary string of
// appropriate length.
//
// Ref:
//     Wolfram, S. (1984). Cellular automata as models of complexity. Nature,
//     311(5985), 419-424.
//
//     Wolfram, S. (2018). Cellular automata and complexity: collected papers.
//     CRC Press.
//
// 1 inputs:
//     rate[n], iteration rate.
//
// L outputs:
//     y1[n];
//     y2[n];
//     ...;
//     yL[n], states of the cells in the lattice.
//
// 3 compile-time arguments:
//     L, (positive INT) size of the lattice (number of cells);
//     R, (positive INT up to 255) rule applied to the 8 possible cases;
```

```
//     I, (positive INT) initial condition for the cells.
//
eca(L, R, I, rate) =   (    si.bus(L) ,
                            init(I) : ro.interleave(L, 2) : par(i, L, +) :
      iterate : par(i, L, ba.sAndH(trigger)))
                          ~ si.bus(L)
      with {
           trigger = ba.period(ma.SR / max(ma.EPSILON, rate)) == 0;
           wrap(M, N) = int(ma.frac(N / M) * M);
           w_num = m2.zeropad_up(int(8 - ceil(ma.log2(R1))), m2.dec2bin(R1))
               with {
                   R1 = min(255, R);
               };
           init(N) = m2.zeropad_up(int(L - ceil(ma.log2(N1))), m2.dec2bin(N1)) :
               par(i, L, _ <: _ - mem)
               with {
                   N1 = min(N, 2 ^ L - 1);
               };
           rule(x1, x2, x3) =
      ba.if(   c1, w_num : route(8, 1, 1, 1),
           ba.if(  c2, w_num : route(8, 1, 2, 1),
               ba.if(  c3, w_num : route(8, 1, 3, 1),
                   ba.if(  c4, w_num : route(8, 1, 4, 1),
                       ba.if(  c5, w_num : route(8, 1, 5, 1),
                           ba.if(  c6, w_num : route(8, 1, 6, 1),
                               ba.if(  c7, w_num : route(8, 1, 7, 1),
                                   w_num : route(8, 1, 8, 1))))))))
                with {
                    c1 = (x1 == 1) & (x2 == 1) & (x3 == 1);
                    c2 = (x1 == 1) & (x2 == 1) & (x3 == 0);
                    c3 = (x1 == 1) & (x2 == 0) & (x3 == 1);
                    c4 = (x1 == 1) & (x2 == 0) & (x3 == 0);
                    c5 = (x1 == 0) & (x2 == 1) & (x3 == 1);
                    c6 = (x1 == 0) & (x2 == 1) & (x3 == 0);
                    c7 = (x1 == 0) & (x2 == 0) & (x3 == 1);
                    c8 = (x1 == 0) & (x2 == 0) & (x3 == 0);
                };
        iterate = si.bus(L) <:
```

```
            par(i, L, route(L, 3,    wrap(L, i - 1) + 1, 1,
                                     i + 1, 2,
                                     wrap(L, i + 1) + 1, 3) : int(rule));
      };
// ------------------------------------------------------------------------

// o2.lorenz(x0, y0, z0, a[n], b[n], r[n], dt[n]); ------------------------
//
// Lorenz system: chaotic recursive system of differential equations.
//
// Ref: https://ijpam.eu/contents/2013-83-1/9/9.pdf.
//
// Try process = o2.lorenz(1.2, 1.3, 1.6, 10, 8/3, 28, .005); for a strange
// attractor (way out of the [-1; 1] range).
//
// 7 inputs:
//    x0, initial condition for the first equation (0 for n != 0);
//    y0, initial condition for the second equation (0 for n != 0);
//    z0, initial condition for the third equation (0 for n != 0);
//    a[n], coefficient in the first equation;
//    b[n], coefficient in the third equation;
//    r[n], coefficient in the second equation;
//    dt[n], discrete time interval.
//
// 3 output:
//    y1[n], first equation;
//    y2[n], second equation;
//    y3[n], third equation.
//
lorenz(x0, y0, z0, a, b, r, dt) =   iterate
                                    ~ ( _ ,
                                        _ ,
                                        _)
      with {
          iterate(x, y, z) =  x1 + a * (y1 - x1) * dt,
                              y1 + (r * x1 - y1 - x1 * z1) * dt,
                              z1 + (x1 * y1 - b * z1) * dt
          with {
```

```
            x1 = x + x0 - x0';
            y1 = y + y0 - y0';
            z1 = z + z0 - z0';
        };
    };
// ------------------------------------------------------------------------------

// o2.osc_quad(F[n]); -----------------------------------------------------------
//
// Recursive quadrature oscillator by Martin Vicanek. This design is
// arguably the best recursive quadrature oscillator available in the
// literature. The system shows long-term stability as well as accuracy at
// low frequencies.
//
// Ref: https://vicanek.de/articles/QuadOsc.pdf.
//
// 1 inputs:
//    F[n], oscillator frequency in Hz.
//
// 2 outputs:
//    y1[n], cosine (real part);
//    y2[n], sine (imaginary part).
//
osc_quad(f) =  tick
            ~ ( _ ,
                _)
    with {
        k1 = tan(ma.PI * f / ma.SR);
        k2 = 2 * k1 / (1 + k1 * k1);
        tick(u, v) =    omega - k1 * (v + k2 * omega) ,
                        v + k2 * omega
        with {
            omega = (u + au.dirac) - k1 * v;
        };
    };
// ------------------------------------------------------------------------------

// o2.ph(F[n], R[n]); -----------------------------------------------------------
```

```
//
// Phasor with reset input.
//
// Note: the arguments of the function should be inverted for
// consistency, but that requires checking for backward compatibility.
//
// 2 inputs:
//     F[n], frequency in Hz;
//     R[n], reset phasor to zero if R[n] != 0.
//
// 1 outputs:
//     y[n], phasor output.
//
ph(freq, reset) = (+ (freq / ma.SR * r) : ma.decimal)
                  ~ * (r)
    with {
        r = reset == 0;
    };
// ------------------------------------------------------------------------

// o2.pulse_train(H[n], D[n], F[n]); --------------------------------------
//
// BLIT-based variable width pulse train. Implemented following
// [Stilson and Smith 1996]:
//
// https://ccrma.stanford.edu/~stilti/papers/blit.pdf.
//
// 3 inputs:
//     H[n], number of harmonics, both even and odd, cast to the closest
//             INT and phase-locked to the beginning of each cycle. The harmonic
//             content is affected by the duty cycle;
//     D[n], duty cycle in the range [0; 1];
//     F[n], pulse train frequency in Hz.
//
// 1 outputs:
//     y[n], band-limited pulse train.
//
pulse_train(h, d, f) =
```

```
    blit_bi_duty(h, d, f) : m2.div(f2.leaky(.1 / m2.twopi), scale) + d
    with {
        lim = rint(m2.div(ma.SR, f) / 2);
        scale = m2.div(lim, min(lim, h)) : si.smooth(ba.tau2pole(.1));
    };
// ------------------------------------------------------------------------------

// o2.saw(H[n], F[n]); ------------------------------------------------------
//
// BLIT-based band-limited sawtooth oscillator.
//
// This function is based on [Stilson and Smith 1996], although the
// scaling factor for a unit-amplitude normalisation has been determined by
// the author.
//
// Ref: https://ccrma.stanford.edu/~stilti/papers/blit.pdf.
//
// 2 inputs:
//     H[n], number of harmonics (both even and odd), cast to the closest INT;
//     F[n], frequency of the oscillator in Hz.
//
// 1 outputs:
//     y[n], band-limited sawtooth oscillator.
//
saw(h, f) = blit_uni(h, f) <: _ - f2.lp1p(f / 100) :
    m2.div(f2.leaky(.1 / m2.twopi), scale) : fi.highpass(1, 20)
    with {
        lim = floor(m2.div(ma.SR, f) / 2);
        scale = m2.div(lim, (2 * min(lim, h))) : si.smooth(ba.tau2pole(.1));
    };
// ------------------------------------------------------------------------------

// o2.square(H[n], F[n]); --------------------------------------------------
//
// BLIT-based band-limited square oscillator. This technique implements
// the integration of a bipolar BLIT. The bipolar BLIT is based on a sinc
// function with even ratios between the sine functions used in sinc, which
// result in a more precise harmonic content throughout the entire
```

```
// frequency range. The scaling factor normalises the output to unit
// amplitude for all frequencies and harmonics.
//
// 2 inputs:
//    H[n], number of harmonics (odd), cast to the closest
//              INT and phase-locked to the beginning of each cycle.
//    F[n], frequency of the oscillator in Hz.
//
// 1 outputs:
//    y[n], band-limited square oscillator.
//
square(h, f) = blit_bi(h, f) : m2.div(f2.leaky(.1 / m2.twopi), scale) :
      fi.highpass(1, 20)
      with {
          lim = rint(m2.div(ma.SR, f) / 4);
          scale = m2.div(lim, (2 * min(lim, h))) : si.smooth(ba.tau2pole(.1));
      };
// --------------------------------------------------------------------------------

// o2.triangle(H[n], F[n]); --------------------------------------------------------
//
// BLIT-based band-limited triangle oscillator. This technique implements
// the integration of a BLIT-based square wave. The bipolar BLIT in the square
// wave is based on a sinc function with even ratios between the sine functions
// used in sinc, which result in a more precise harmonic content throughout
// the entire frequency range. The scaling factor normalises the output to unit
// amplitude for all frequencies and harmonics
//
// 2 inputs:
//    H[n], number of harmonics (odd), cast to the closest
//              INT and phase-locked to the beginning of each cycle.
//    F[n], frequency of the oscillator in Hz.
//
// 1 outputs:
//    y[n], band-limited triangle oscillator.
//
triangle(h, f) = square(h, f) : m2.div(f2.leaky(.1 / m2.twopi), scale) :
      fi.highpass(1, 20)
```

```
    with {
        scale = rint(m2.div(ma.SR, f) / 4) : si.smooth(ba.tau2pole(.1));
    };
// ----------------------------------------------------------------------------
```

A.9 outformation.lib

```
// ============================================================================
// ========== outformation.lib ================================================
// ============================================================================
//
// Library of functions for the transformation of audio signals. The
// library includes standard techniques such as frequency shifting, artificial
// reverberators with different delay line schemes, and other modulations, as
// well as original techniques such as windowless granular processing based on
// zero-crossing detection.
//
// The environment prefix is "op".
//
// List of functions:
//
//    grains_dl_nhw,
//    grains_dl_zc,
//    grains_zc,
//    pitch_shift,
//    pole_mod,
//    rev_fdn_smo,
//    rev_fdn_pol,
//    sampler,
//    ssbm,
//    time_stretch,
//    tvtf.
//
// Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario at gmail dot com>
// All rights reserved.

declare name "Outformation Library";
declare author "Dario Sanfilippo";
```

```
declare copyright "Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario
      at gmail dot com>";
declare version "1.0.0";
declare license "GPLv2.0";

au = library("auxiliary.lib");
ba = library("basics.lib");
de = library("delays.lib");
d2 = library("delays2.lib");
fi = library("filters.lib");
f2 = library("filters2.lib");
ip = library("information.lib");
op = library("outformation.lib");
os = library("oscillators.lib");
o2 = library("oscillators2.lib");
ma = library("maths.lib");
m2 = library("maths2.lib");
ro = library("routes.lib");
si = library("signals.lib");
st = library("stability.lib");

// op.grains_dl_nhw(S, P[n], R[n], pos[n], E[n], x[n]); ------------------------
//
// Granulator based on delay lines with overlap-add to 1 and non-homogeneous
// windowing and transposition. Hence, for nonlinear factors other than 1
// (the exponent), the windowing function is asymmetrical and the
// reading of each grain includes a pitch modulation.
//
// 5 inputs:
//    P[n], linear pitch factor (1 for no transposition; 2 for an octave
//          up; .5 for an octave down);
//    R[n], amount of grains per second;
//    pos[n], position of the grain in the buffer in the range [0; S],
//          where "S" is the size of the buffer in seconds;
//    E[n], exponent, nonlinearity for the windowing and pitch modulation;
//    x[n].
//
// 1 ouputs:
```

```
//     y[n], granulated x[n].
//
// 1 compile-time arguments:
//     S, size of the buffer in seconds, which is converted into the
//          closest power-of-two samples that represent such length.
//
grains_dl_nhw(size, pitch, rate, position, exponent, x) = head1 + head2
    with {
        s = size * ma.SR : m2.round_pow2 / ma.SR;
        sah(t, in) = ba.sAndH(m2.diff(t) < 0, in);
        ph0 = o2.ph(rate, 0);
        ph1 = ma.decimal(pow(ph0, exponent));
        ph2 = ma.decimal(ph1 + .5);
        w1 = m2.window_hann(ph1);
        w2 = m2.window_hann(ph2);
        head1 = d2.del_pol(s, del1, x) * w1
            with {
                del1 = sah(ph1, position) + shift1 : m2.wrap(0, s);
                shift1 = (1 - sah(ph1, pitch)) *
                    m2.div(1, sah(ph1, rate)) * ph1;
            };
        head2 = d2.del_pol(s, del2, x) * w2
            with {
                del2 = sah(ph2, position) + shift2 : m2.wrap(0, s);
                shift2 = (1 - sah(ph2, pitch)) *
                    m2.div(1, sah(ph2, rate)) * ph2;
            };
    };
// ---------------------------------------------------------------------------

// op.grains_dl_zc(V, S, P[n], R[n], pos[n], x[n]); --------------------------
//
// Delay-line-based windowless (rectangular window) granulator that
// handles discontinuities through zero-crossing detection.
//
// Ref:     https://tmblr.co/Zhtq9xYAy2bPee00;
//          https://tmblr.co/Zhtq9x2i76aPG.
//
```

```
// 4 inputs:
//     P[n], linear pitch factor (1 for no transposition; 2 for an octave
//           up; .5 for an octave down);
//     R[n], amount of grains per second;
//     pos[n], position of the grain in the buffer in the range [0; S],
//           where "S" is the size of the buffer in seconds;
//     x[n].
//
// 1 outputs:
//     y[n], granulated x[n].
//
// 2 compile-time arguments:
//     V, number of voices;
//     S, size of the buffer in seconds, which is converted into the
//           closest power-of-two samples that represent such length.
//
grains_dl_zc(voices, size1) = par(i, voices,     loop
                                                  ~ _) :> / (voices)
      with {
           loop(out, pitch1, rate1, position1, input) =
               (ba.sAndH(trigger(out), zc_index(position, input, out))
                  + shift(trigger(out))) : m2.wrap(0, size) - 1 ,
               input : grain
           with {
               trigger(y) =     loop
                                ~ _
               with {
                   loop(ready) =
                       ip.zc(y) ,
                       (m2.line_reset(ba.sAndH(au.dirac + ready, rate),
                                               ready) >= 1) : &;
               };
               shift(reset) = m2.div(1 - pitch, rate) *
                   m2.line_reset(rate, reset) * ma.SR;
               zc_index(recall, x, y) =
                   index - m2.if(m2.diff(y) >= 0, zc_up, zc_down) :
                       m2.wrap(0, size)
                   with {
```

```
                        zc_up = ba.sAndH(store, index), recall : dl
                            with {
                                store = ip.zc(x) ,
                                        (m2.diff(x) > 0) : &;
                            };
                        zc_down = ba.sAndH(store, index), recall : dl
                            with {
                                store = ip.zc(x) ,
                                        (m2.diff(x) < 0) : &;
                            };
                    };
            size = size1 * ma.SR : m2.round_pow2;
            rate = abs(rate1);
            pitch = ba.sAndH(trigger(out), pitch1);
            position = position1 * ma.SR : m2.wrap(0, size);
            index = ba.period(size);
            grain(del, in) = de.fdelayltv(4, size, del, in);
            dl(in, del) = de.delay(size, del, in);
    };
};
// ---------------------------------------------------------------------------

// op.grains_zc(pos[n], size[n], x[n]); ---------------------------------------
//
// Table-based windowless (rectangular window) granulator that
// handles discontinuities through zero-crossing detection.
//
// Ref:     https://tmblr.co/Zhtq9xYAy2bPee00;
//          https://tmblr.co/Zhtq9x2i76aPG.
//
// 3 inputs:
//    pos[n], position of the grain in the buffer in the range [0; S],
//        where "S" is the size of the buffer in seconds;
//    size[n], size of grains in seconds;
//    x[n].
//
// 1 outputs:
//    y[n], granulated x[n].
```

```
//
grains_zc(position, g_size, x) =    grains
                                    ~ _
    with {
        s = 768000;
        l = g_size * ma.SR;
        p = position * ma.SR;
        input = x;
        rec_index = ba.period(s);
        grains(fb) =    int(s) ,
                        0.0 ,
                        int(rec_index) ,
                        input ,
                        int(read_frame(fb) % s) : rwtable
        with {
            sel_zc(x) =
                ba.if(m2.diff(x) > 0, zc_up_index, zc_down_index);
            frame(start) =  % (dur)
                            ~ + (1)
            with {
                dur = zc_index(start + 1) - start : max(2);
            };
            read_frame(out) = (frame <: _ ,
                                        (== (0) ,
                                        sel_zc(out) : ba.sAndH))
                                        ~ ( ! ,
                                            _) : +;
            zc_up = (ip.zc(input) ,
                    (m2.diff(input) > 0) : &);
            zc_down =   (ip.zc(input) ,
                        (m2.diff(input) < 0) : &);
            zc_index(x) =   int(s) ,
                            0.0 ,
                            int(rec_index) ,
                            (   ip.zc(input) ,
                                rec_index : ba.sAndH) ,
                            int(x % s) : rwtable;
            zc_up_index =   int(s) ,
```

```
                                0.0 ,
                                int(rec_index) ,
                                (   zc_up ,
                                    rec_index : ba.sAndH) ,
                                int(p % s) : rwtable;
                zc_down_index = int(s) ,
                                0.0 ,
                                int(rec_index) ,
                                (   zc_down ,
                                    rec_index : ba.sAndH) ,
                                int(p % s) : rwtable;
            };
        };
// ------------------------------------------------------------------------------

// op.pitch_shift(S, P[n], size[n], x[n]); ------------------------------------
//
// Real-time pitch-shifter using 4th-order Lagrange polynomial fractional
// delay lines.
//
// 3 inputs:
//    P[n], linear pitch factor (1 for no transposition; 2 for an octave
//         up; .5 for an octave down);
//    size[n], size of frames in seconds;
//    x[n].
//
// 1 outputs:
//    y[n], pitch-shifted x[n].
//
// 1 compile-time arguments:
//    S, size of the buffer in seconds, which is converted into the
//         closest power-of-two samples that represent such length.
//
pitch_shift(buff_size, factor, frame, x) =
    d2.del_pol(buff_size, del1, x) * w1 ,
    d2.del_pol(buff_size, del2, x) * w2 :> _
    with {
        frame_1 = abs(frame);
```

```
        rate = m2.div(1, frame_1);
        shift = (1 - factor) * frame_1;
        offset = m2.if(shift < 0, -shift, 0);
        limit = m2.round_pow2(buff_size * ma.SR) / ma.SR;
        ph1 = m2.ph(rate, 0);
        ph2 = ma.decimal(ph1 + .5);
        w1 = m2.window_hann(ph1);
        w2 = m2.window_hann(ph2);
        del1 = shift * ph1 + offset : m2.wrap(0, limit);
        del2 = shift * ph2 + offset : m2.wrap(0, limit);
    };
// ------------------------------------------------------------------------------

// op.pole_mod(R[n], E[n], x[n]); ------------------------------------------------
//
// Pole modulation of normalised one-pole system, hence oscillating
// between lowpass and highpass. The modulator has a shaping parameter
// going from -1 to 1 where the we have squarewave at -1, a sinewave at 0,
// and impulses at 1.
//
// 3 inputs:
//    R[n], modulation rate in Hz;
//    E[n], shaping parameter in the range [-1; 1];
//    x[n].
//
// 1 outputs:
//    y[n], pole-modulated x[n].
//
pole_mod(rate, shaping, x) = x * norm : fi.pole(mod)
    with {
        norm = 1 - abs(mod);
        mod = os.osc(rate) <: ma.signum * (abs : pow(shaping1))
        with {
            shaping1 = pow(1000, st.clip(-1, 1, shaping));
        };
    };
// ------------------------------------------------------------------------------
```

```
// op.rev_fdn_smo(N, S, IT[n], size[n], FB[n], CF[n], x[n]); ————————————————
//
// Elementary Nth-order feedback delay network reverb with non-transposing
// variable delay lines.
//
// 5 inputs:
//     IT[n], interpolation time in seconds to transition between
//          different delays;
//     size[n], exponent for as many prime numbers as the order of the
//          network, the result of which determines the length of the
//          delay lines in seconds;
//     FB[n], feedback coefficient, whose magnitude should be less or
//          equal to 1 for stability;
//     CF[n], cut-off frequency in Hz, of lowpass filters within the
//          feedback loop that model the dampening of high freqiencies.
//
// 1 outputs:
//     y[n], normalised sum of the N signals in the network.
//
// 2 compile-time arguments:
//     N, order of the network (INT);
//     S, max size of the delay lines in seconds.
//
rev_fdn_smo(N, max_size, it, size, fb_coeff, cf, in) =
    (summing : delays : filters : matrix : fb)
    ~ si.bus(N) :> / (N)
    with {
        st = 1 / sqrt(N);
        summing = par(i, N, + (in));
        delays = par(i, N,   max_size ,
                             it ,
                             (    size : m2.prime_base_pow(i + 1)) ,
                                  _) : par(i, N, d2.del_smo);
        filters = par(i, N, f2.lp1p(cf));
        matrix = ro.hadamard(N);
        fb = par(i, N, * (fb_coeff * st));
    };
// ————————————————————————————————————————————————————————————
```

```
// ------------------------------------------------------------------------------------------
// Feedback delay network reverb with variable DL;
// n must be a power of 2; FB coeffients are stable up to a magnitude of 1
// op.rev_fdn_pol(N, S, size[n], FB[n], CF[n], x[n]); ---------------------------
//
// Elementary Nth-order feedback delay network reverb with transposing
// variable delay lines (4th-order Lagrange interpolation).
//
// 5 inputs:
//     size[n], exponent for as many prime numbers as the order of the
//          network, the result of which determines the length of the
//          delay lines in seconds;
//     FB[n], feedback coefficient, whose magnitude should be less or
//          equal to 1 for stability;
//     CF[n], cut-off frequency in Hz, of lowpass filters within the
//          feedback loop that model the dampening of high freqiencies.
//
// 1 outputs:
//     y[n], normalised sum of the N signals in the network.
//
// 2 compile-time arguments:
//     N, order of the network (INT);
//     S, max size of the delay lines in seconds.
//
rev_fdn_pol(n, max_size, size, fb_coeff, cf, in) =
    (summing : delays : filters : matrix : fb)
    ~ si.bus(n) :> /(n)
    with {
        st = 1 / sqrt(n);
        summing = par(i, n, +(in));
        delays = par(i, n,  max_size ,
                          (   size : m2.prime_base_pow(i + 1)),
                              _) : par(i, n, d2.del_pol);
        filters = par(i, n, f2.lp1p(cf));
        matrix = ro.hadamard(n);
        fb = par(i, n, *(fb_coeff * st));
    };
```

```
// ----------------------------------------------------------------------------

// op.sampler(S, size[n], pos[n], P[n], x[n]); --------------------------------
//
// Sampler with pitch, frame size, and buffer position control.
//
// 4 inputs:
//     size[n], frame size in seconds;
//     pos[n], position of the frame in the buffer in the range [0; S],
//         where S is the size of the buffer in seconds;
//     P[n], pitch factor;
//     x[n].
//
// 1 outputs:
//     y[n], sampled x[n].
//
// 1 compile-time arguments:
//     S, size of the buffer in seconds, which is converted into the
//         closest power-of-two samples that represent such length.
//
sampler(buff_size, frame, position, factor, x) = d2.del_pol(buff_size, del, x)
    with {
        frame_1 = abs(frame) : f2.lp1p(20);
        position_1 = position : f2.lp1p(20);
        rate = m2.div(1, frame_1);
        shift = (1 - factor) * frame_1;
        offset = m2.if(shift < 0, -shift, 0);
        limit = buff_size * ma.SR : m2.round_pow2 / ma.SR;
        ph = m2.ph(rate, 0);
        del = shift * ph + offset+position_1 : m2.wrap(0, limit);
    };
// ----------------------------------------------------------------------------

// op.ssbm(F[n], x[n]); -------------------------------------------------------
//
// Single-sideband modulation (positive side).
//
// 2 inputs:
```

```
//     F[n], frequency shift in Hz;
//     x[n].
//
// 1 outputs:
//     y[n], frequency-shifted x[n].
//
ssbm(shift, in) = f2.analytic(in) ,
                  o2.osc_quad(shift) : si.cmul : _ ,
                                                  !;
// ------------------------------------------------------------------------------

// op.time_stretch(S, size[n], T[n], x[n]); ----------------------------------------
//
// Real-time time stretcher with delay lines.
//
// 3 inputs:
//     size[n], frame size in seconds;
//     T[n], time stretching factor;
//     x[n].
//
// 1 outputs:
//     y[n], time-stretched x[n].
//
// 1 compile-time arguments:
//     S, size of the buffer in seconds, which is converted into the
//          closest power-of-two samples that represent such length.
//
time_stretch(buff_size, frame, factor, x) =
     d2.del_pol(buff_size, del1, x) * w1 ,
     d2.del_pol(buff_size, del2, x) * w2 :> _
     with {
          buff = buff_size * ma.SR : m2.round_pow2 / ma.SR;
          position = m2.ph((1 - factor) / buff, 0) * buff;
          frame_1 = abs(frame);
          rate = m2.div(1, frame_1);
          ph1 = m2.ph(rate, 0);
          ph2 = ma.decimal(ph1 + .5);
          w1 = m2.window_hann(ph1);
```

```
        w2 = m2.window_hann(ph2);
        del1 = position : ba.sAndH(m2.diff(ph1) < 0);
        del2 = position : ba.sAndH(m2.diff(ph2) < 0);
    };
// ------------------------------------------------------------------------------

// op.tvtf(S, ZCR[n], TF[n], x[n]); ---------------------------------------------
//
// Time-variant transfer function: the transfer function is determined
// by an incoming signal. The input signal is wrapped around in the
// range [-1; 1]; -1 corresponds to the beginning of the transfer
// function, 0 is the centre of the buffer, whereas 1 is the upper edge.
//
// The input signal that determines the transfer function is lowpassed
// to control the number of zero-crossings in the transfer function, which
// correlates to the number added partials, [Roads 1979] and normalised to
// unit-amplitude peaks.
//
// 3 inputs:
//    ZCR[n], (roughly) number of zero-crossings in the transfer
//        function;
//    TF[n], signal writing the transfer function;
//    x[n].
//
// 1 outputs:
//    y[n], x[n] processed through the transfer function.
//
// 1 compile-time arguments:
//    S, size of the buffer in seconds, which is converted into the
//        closest power-of-two samples that represent such length.
//
tvtf(s, zcr, f, in) = d2.del_pol(s, in1, f1)
    with {
        in1 = in : m2.wrap(-1, 1) : m2.uni * s1;
        s1 = s * ma.SR : m2.round_pow2 / ma.SR;
        f1 = f : seq(i, 4, f2.lp1p(zcr / s1)) : st.dyn_norm_peak(1 / s1, 1);
    };
// ------------------------------------------------------------------------------
```

A.10 stability.lib

```
// =============================================================================
// ========== stability.lib ====================================================
// =============================================================================
//
// This library module includes a set of functions for stability processing
// that can be deployed in self-oscillating systems or any other systems
// that require control over the boundaries of amplitude values. Depending
// on the specific applications, it is possible to use different designs
// ranging from bounded saturators, lookahead limiters, and adaptive
// self-regulating dynamic processing.
//
// The filters for the amplitude analysis are based on a time constant
// that is tau * 2  .
//
// The bounded saturators are taken from [Zavalishin 2012], "The art of
// VA filter design".
//
// The environment prefix is "st".
//
// List of functions:
//
//     clip,
//     cubic,
//     dyn_comp_peak,
//     dyn_comp_rms,
//     dyn_norm_peak,
//     dyn_norm_rms,
//     hyperbolic,
//     limiter,
//     limiter_lookahead,
//     limiter_lookaheadN,
//     parabolic,
//     sinatan,
//     tanh.
//
// Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario at gmail dot com>
```

```
// All rights reserved.

declare name "Stability Processing Library";
declare author "Dario Sanfilippo";
declare copyright "Copyright (c) 2019-2020, Dario Sanfilippo <sanfilippo.dario
    at gmail dot com>";
declare version "1.0.0";
declare license "GPLv2.0";

ba = library("basics.lib");
d2 = library("delays2.lib");
f2 = library("filters2.lib");
ip = library("information.lib");
ma = library("maths.lib");
m2 = library("maths2.lib");
ro = library("routes.lib");
si = library("signals.lib");
st = library("stability.lib");

// st.clip(L[n], H[n], x[n]); ------------------------------------------------
//
// Hard clipping function.
//
// 3 inputs:
//     L[n], lower limit;
//     H[n], upper limit;
//     x[n].
//
// 1 outputs:
//     y[n], hard-limited x[n].
//
clip(lower, upper, in) = min(max(lower, in), upper);
// ---------------------------------------------------------------------------

// st.cubic(x[n]); -----------------------------------------------------------
//
// Cubic saturator.
//
```

```
// 1 inputs:
//     x[n].
//
// 1 outputs:
//     y[n], soft-clipped x[n] in the range [-2/3; 2/3].
//
cubic(x) = select3(cond, -2 / 3, x - x^3 / 3, 2 / 3)
       with {
            cond = ((x > -1) ,
                      (x < 1) : &) + (x >= 1) * 2;
       };
// ----------------------------------------------------------------------------

// st.dyn_comp_peak(R[n], E[n], x[n]); ----------------------------------------
//
// Adaptive compression based on peak envelope analysis.
//
// 3 inputs:
//     R[n], release time in seconds for the peak envelope analysis;
//     E[n], exponential curve, exponent to the complement of the peak
//           envelope curve: higher gain reductions for exponents > 1,
//           lower gain reductions for exponents between 0 and 1;
//     x[n].
//
// 1 outputs:
//     y[n], compressed x[n].
//
dyn_comp_peak(release, curve, in) = in * agc
       with {
            agc = max(0, 1 - min(ip.peak_env(release, in), 1)) : pow(curve);
       };
// ----------------------------------------------------------------------------

// st.dyn_comp_rms(R[n], E[n], x[n]); -----------------------------------------
//
// Adaptive compression based on RMS analysis.
//
// 3 inputs:
```

```
//    R[n], response time in seconds for the RMS envelope analysis;
//    E[n], exponential curve, exponent to the complement of the peak
//          envelope curve: higher gain reductions for exponents > 1,
//          lower gain reductions for exponents between 0 and 1;
//    x[n].
//
// 1 outputs:
//    y[n], compressed x[n].
//
dyn_comp_rms(window, curve, in) = in * agc
    with {
        agc = max(0, 1 - min(ip.rms(window, in), 1)) : pow(curve);
    };
// -----------------------------------------------------------------------------

// st.dyn_norm_peak(R[n], T[n], x[n]); ----------------------------------------
//
// Adaptive normalisation based on peak envelope analysis.
//
// 3 inputs:
//    R[n], release time in seconds for the peak envelope analysis;
//    T[n], target linear amplitude for the normalisation process;
//    x[n].
//
// 1 outputs:
//    y[n], normalised x[n].
//
dyn_norm_peak(release, target, input) = input * agc
    with {
        agc =   ip.peak_env(release, target) ,
                ip.peak_env(release, input) : m2.div;
    };
// -----------------------------------------------------------------------------

// st.dyn_norm_rms(R[n], T[n], x[n]); -----------------------------------------
//
// Adaptive normalisation based on RMS analysis.
//
```

```
// 3 inputs:
//     R[n], response time in seconds for the RMS analysis;
//     T[n], target linear amplitude for the normalisation process;
//     x[n].
//
// 1 outputs:
//     y[n], normalised x[n].
//
dyn_norm_rms(window, target, input) = input * agc
      with {
           agc =   ip.rms(window, target) ,
                   ip.rms(window, input) : m2.div;
      };
// ------------------------------------------------------------------------

// st.hyperbolic(L[n], x[n]); ----------------------------------------------
//
// Hyperbolic saturator.
//
// 2 inputs:
//     L[n], saturation limit.
//     x[n].
//
// 1 outputs:
//     y[n], soft-clipped x[n] in the range [-L[n]; L[n]].
//
hyperbolic(l, x1) = l * (x / (1 + abs(x)))
      with {
           x = m2.div(x1, l);
      };
// ------------------------------------------------------------------------

// st.limiter(L[n], x[n]); -------------------------------------------------
//
// Mono lookahead limiter. Special case of st.limiter_lookahead. (See below.)
//
// 2 inputs:
//     L[n], linear amplitude limiting threshold;
```

```
//     x[n].
//
// 1 outputs:
//     lookahead-limited x[n] in the range [-L[n]; L[n]].
//
limiter(lim, in) = limiter_lookahead(.002, lim, .002, .1, 1, in);
// ----------------------------------------------------------------

// st.limiter_lookahead(D, L[n], A[n], H[n], R[n], x[n]); ------------
//
// Mono lookahead limiter inspired by IOhannes Zm lnig post, which is in
// turn based on the thesis by Peter Falkner "Entwicklung eines digitalen
// Stereo-Limiters mit Hilfe des Signalprozessors DSP56001".
//
// http://iem.at/~zmoelnig/publications/limiter/.
//
// This version of the limiter uses a peak-holder with smoothed
// attack and release based on tau * 2   time constant filters.
// This time constant allows for the amplitude profile to reach
// 1 - e^(-2pi) of the final peak after the attack time. The input path
// can be delayed by the same amount as the attack time to synchronise input
// and amplitude profile, or by any other lookahead time specified by the user.
//
// Note that rather than using two switching filter sections for the
// attack and release smoothing, two independent filters are cascaded, a
// one-pole lowpass to smooth out the attack, and a peak envelope to smooth
// out the release. Since the filters are cascaded, the release time is
// slightly delayed by the lowpass filter, although that will also smooth
// out the attack-release transition knee resulting in a cleaner signal.
//
// 5 inputs:
//     L[n], linear amplitude limiting threshold;
//     A[n], attack time in seconds;
//     H[n], hold time in seconds;
//     R[n], release time in seconds;
//     x[n].
//
// 1 outputs:
```

```
//     y[n], lookahead-limited x[n] in the range [-L[n]; L[n]].
//
// 1 compile-time arguments:
//     D, lookahead delay in seconds.
//
limiter_lookahead(lag, threshold, attack, hold, release, x) =
       x @ (lag * ma.SR) * agc
       with {
            agc = m2.div(threshold, amp_profile) : min(1);
            amp_profile = ip.peak_hold(hold, x) : att_smooth(attack) :
                rel_smooth(release);
            att_smooth(time, in) = f2.lp1p(m2.div(1, time), in);
            rel_smooth(time, in) = ip.peak_env(time, in);
       };
// ----------------------------------------------------------------------------

// st.limiter_lookaheadN(N, D, L[n], A[n], H[n], R[n]); ----------------------
//
// N-channel limiter based on the mono lookahead limiter. See above for a full
// description of the algorithm.
//
// The amplitude profile is calculated based on the peak between all of
// the signals and the same scaling factor is applied to all of the
// channels to preserve their amplitude ratios.
//
// N+4 inputs:
//     L[n], linear amplitude limiting threshold;
//     A[n], attack time in seconds;
//     H[n], hold time in seconds;
//     R[n], release time in seconds;
//     x1[n];
//      ...;
//     xN-1[n];
//     xN[n], input channels.
//
// N outputs:
//     y1[n];
//      ...;
```

```
//     yN−1[n];
//     yN[n], lookahead−limited input channels in the range [−L[n]; L[n]].
//
// 2 compile−time arguments:
//     N, (integer) number of input channels;
//     D, lookahead delay in seconds.
//
limiter_lookaheadN(N, lag, threshold, attack, hold, release) =
      si.bus(N) <: par(i, N, @ (lag * ma.SR)) ,
                  (agc <: si.bus(N)) : ro.interleave(N, 2) : par(i, N, *)
      with {
            agc = m2.div(threshold, amp_profile) : min(1);
            amp_profile = par(i, N, abs) : m2.maxN(N) : ip.peak_hold(hold) :
                  att_smooth(attack) : rel_smooth(release);
            att_smooth(time, in) = f2.lp1p(m2.div(1, time), in);
            rel_smooth(time, in) = ip.peak_env(time, in);
      };
// −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

// st.parabolic(L[n], x[n]); −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
//
// Parabolic saturator.
//
// 2 inputs:
//     L[n], saturation limit;
//     x[n].
//
// 1 outputs:
//     y[n], soft−clipped x[n] in the range [−L[n]; L[n]].
//
parabolic(l, x1) = l * (m2.if(abs(x) >= 2, ma.signum(x), x * (1 − abs(x / 4))))
      with {
            x = m2.div(x1, l);
      };
// −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

// st.sinatan(L[n], x[n]); −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
//
```

```
// Sin(arctan(x)) saturator.
//
// 2 inputs:
//     L[n], saturation limit;
//     x[n].
//
// 1 outputs:
//     y[n], soft-clipped x[n] in the range [-L[n]; L[n]].
//
sinatan(l, x1) = l * (x / sqrt(1 + x * x))
       with {
             x = m2.div(x1, l);
       };
// ------------------------------------------------------------------------

// st.tanh(L[n], x[n]); ------------------------------------------------------
//
// Hyperbolic tangent.
//
// 2 inputs:
//     L[n], saturation limit;
//     x[n].
//
// 1 outputs:
//     y[n], soft-clipped x[n] in the range [-L[n]; L[n]].
//
tanh(l, x1) = l * ((exp(2 * x) - 1) / (exp(2 * x) + 1))
       with {
             x = m2.div(x1, l);
       };
// ------------------------------------------------------------------------
```

Bibliography

[Ashby, 1957] Ashby, W. R. (1957). An introduction to cybernetics. 2.1, 2.1.1, 2.4.1, 6

[Ashby, 1968] Ashby, W. R. (1968). Principles of the self-organizing system. *Modern systems research for the behavioral scientist*, pages 108–118. 2.3.1, 6

[Ashby, 1991] Ashby, W. R. (1991). Principles of the self-organizing system. In *Facets of systems science*, pages 521–536. Springer. 2.1.1, 6

[Bachu et al., 2008] Bachu, R., Kopparthi, S., Adapa, B., and Barkana, B. (2008). Separation of voiced and unvoiced using zero crossing rate and energy of the speech signal. In *American Society for Engineering Education (ASEE) Zone Conference Proceedings*, pages 1–7. 3.3.1

[Barandiaran and Moreno, 2006] Barandiaran, X. and Moreno, A. (2006). On what makes certain dynamical systems cognitive: A minimally cognitive organization program. *Adaptive Behavior*, 14(2):171–185. 1.1

[Baranger, 2000] Baranger, M. (2000). Chaos, complexity, and entropy. *New England Complex Systems Institute, Cambridge*. 2.5

[Bateson, 1979] Bateson, G. (1979). *Mind and nature: A necessary unity*, volume 255. Bantam Books New York. 3.1, 3.2.1

[Beer, 1966] Beer, S. (1966). Decision and control: The meaning of operational research and management cybernetics. (001.424 B4). 2.3.1

[Benkirane et al., 2002] Benkirane, R., Morin, E., Prigogine, I., and Varela, F. (2002). *La complexité, vertiges et promesses: 18 histoires de sciences*. Le pommier. 2.4

[Boccaletti et al., 2006] Boccaletti, S., Latora, V., Moreno, Y., Chavez, M., and Hwang, D.-U. (2006). Complex networks: Structure and dynamics. *Physics reports*, 424(4-5):175–308. 3.2

[Bonabeau and Dessalles, 1997] Bonabeau, E. and Dessalles, J.-L. (1997). Detection and emergence. *Intellectica*, 25(2):85–94. 2.3.1, 2.5, 2.5

[Boner and Boner, 1966] Boner, C. P. and Boner, C. (1966). Behavior of sound system response immediately below feedback. *Journal of the Audio Engineering Society*, 14(3):200–203. 1.1.1

[Booker et al., 2005] Booker, L., Forrest, S., Mitchell, M., and Riolo, R. (2005). *Perspectives on adaptation in natural and artificial systems*, volume 8. Oxford University Press on Demand. 2.5

[Bowers, 2002] Bowers, J. (2002). *Improvising machines: Ethnographically informed design for improvised electro-acoustic music*. University of East Anglia. 1.3.2

[Brand, 1976] Brand, S. (1976). For gods sake, Margaret: conversation with Gregory Bateson and Margaret Mead. *CoEvolution Quarterly*, 10:32–44. 1.1

[Bregman, 1994] Bregman, A. S. (1994). *Auditory scene analysis: The perceptual organization of sound*. MIT press. 2.5, 3.3.2

[Brent, 2010] Brent, W. (2010). A timbre analysis and classification toolkit for Pure Data. In *Proceedings of the international computer music conference*. International Computer Music Association. 3.3.1

[Bridda, 2004] Bridda, E. (2004). Maree digitali. [Online; accessed 20-June-2019]. 1.3.1

[Bunimovich, 1979] Bunimovich, L. A. (1979). On the ergodic properties of nowhere dispersing billiards. *Communications in Mathematical Physics*, 65(3):295–312. 2.2

[Camazine et al., 2003] Camazine, S., Deneubourg, J.-L., Franks, N. R., Sneyd, J., Bonabeau, E., and Theraula, G. (2003). *Self-organization in biological systems*, volume 7. Princeton university press. 2.1.1

[Cardew, 1971] Cardew, C. (1971). Towards an ethic of improvisation. *Treatise handbook*, pages 17–20. 1.5

[Cariani, 1991] Cariani, P. (1991). Adaptivity and emergence in organisms and devices. *World Futures: Journal of General Evolution*, 32(2-3):111–132. 2.3.1

[Cascone, 2000] Cascone, K. (2000). The aesthetics of failure:post-digital tendencies in contemporary computer music. *Computer Music Journal*, 24(4):12–18. 1.3.1

[Cavaliere and Piccialli, 1997] Cavaliere, S. and Piccialli, A. (1997). Granular synthesis of musical signals. *Musical Signal Processing*, pages 155–186. 4.1.1

[Chamberlin, 1985] Chamberlin, H. (1985). *Musical Applications of Microprocessor*. Sams. 3.3.1

[Cilliers, 2002] Cilliers, P. (2002). *Complexity and postmodernism: Understanding complex systems*. routledge. 2.5

[Cook, 1992] Cook, P. R. (1992). A meta-wind-instrument physical model, and a meta-controller for real-time performance control. In *Proceedings of the international computer music conference*, pages 273–273. International Computer Music Association. 1.3.3, 4.1.5

[Corning, 2002] Corning, P. A. (2002). The re-emergence of emergence: A venerable concept in search of a theory. *Complexity*, 7(6):18–30. 1.3.1

[Dahlstedt et al., 2015] Dahlstedt, P., Nilsson, P. A., and Robair, G. (2015). The bucket system-a computer mediated signalling system for group improvisation. In *NIME*, pages 317–318. 1.3.2, 1.4, 5.1

[Davis, 2013] Davis, P. J. (2013). *Circulant matrices*. American Mathematical Soc. 3.1.3

[De Sena et al., 2015] De Sena, E., Hacıhabiboğlu, H., Cvetković, Z., and Smith, J. O. (2015). Efficient synthesis of room acoustics via scattering delay networks. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, 23(9):1478–1492. 4.1.5

[Demany and Semal, 2008] Demany, L. and Semal, C. (2008). The role of memory in auditory perception. In *Auditory perception of sound sources*, pages 77–113. Springer. 3.3.2

[Di Scipio, 1994] Di Scipio, A. (1994). Formal processes of timbre composition challenging the dualistic paradigm of computer music. In *Proceedings of the International Computer Music Conference*, pages 202–202. International Computer Music Association. 1.2, 1.3.1

[Di Scipio, 1997] Di Scipio, A. (1997). Musica tra determinismo e indeterminismo tecnologico. *Musica/Realta*, 53. 1.3.1

[Di Scipio, 1998] Di Scipio, A. (1998). Questions concerning music technology. *Angelaki: journal of the theoretical humanities*, 3(2):31–40. 1.3.1

[Di Scipio, 1999] Di Scipio, A. (1999). Synthesis of environmental sound textures by iterated nonlinear functions. In *Proceedings of the 2nd COST G-6 Workshop on Digital Audio Effects (DAFx99)*, pages 109–117. 2.2

[Di Scipio, 2000] Di Scipio, A. (2000). Tecnologia dell'esperienza musicale nel Novecento. *Rivista Italiana di Musicologia*, 35(1/2):211–245. 1.3.1

[Di Scipio, 2003] Di Scipio, A. (2003). 'sound is the interface': from interactive to ecosystemic signal processing. *Organised Sound*, 8(3):269–277. 1.2, 1.3.3, 3.4, 5.4

[Di Scipio, 2016] Di Scipio, A. (2016). I quanta acustici di Gabor nelle tecnologie del suono e della musica. *Musica/Tecnologia*, 10:17–42. 4.1.1

[Di Scipio and Sanfilippo, 2019] Di Scipio, A. and Sanfilippo, D. (2019). Defining ecosystemic agency in live performance. The Machine Milieu project as practice-based research. *Array Journal*, 12:28–43. 1.2

[Eldridge, 2007] Eldridge, A. (2007). *Collaborating with the behaving machine: simple adaptive dynamical systems for generative and interactive music*. PhD thesis. 1.1.1

[Eldridge et al., 2008] Eldridge, A., Dorin, A., and McCormack, J. (2008). Manipulating artificial ecosystems. In *Workshops on Applications of Evolutionary Computation*, pages 392–401. Springer. 1.1.1

[Eldridge and Kiefer, 2017] Eldridge, A. and Kiefer, C. (2017). The self-resonating feedback cello: interfacing gestural and generative processes in improvised performance. *Proceedings of New Interfaces for Music Expression 2017*, 2017:25–29. 1.1.1

[Etxeberria et al., 1994] Etxeberria, A., Julian Merelo, J., and Moreno, A. (1994). Studying organisms with basic cognitive capacities in artificial worlds. *Intellectica*, 18(1):45–69. 1.1

[Farmer, 2002] Farmer, J. D. (2002). Market force, ecology and evolution. *Industrial and Corporate Change*, 11(5):895–953. 1.1

[Farmer and Lafond, 2016] Farmer, J. D. and Lafond, F. (2016). How predictable is technological progress? *Research Policy*, 45(3):647–665. 1.1

[Feenberg, 1992] Feenberg, A. (1992). Subversive rationalization: Technology, power, and democracy. *Inquiry*, 35(3-4):301–322. 1.3.1

[Forrest et al., 1997] Forrest, S., Hofmeyr, S. A., and Somayaji, A. (1997). Computer immunology. *Communications of the ACM*, 40(10):88–97. 1.1

[Gabor, 1946] Gabor, D. (1946). Theory of communication. part 1: The analysis of information. *Journal of the Institution of Electrical Engineers-Part III: Radio and Communication Engineering*, 93(26):429–441. 3.3.1, 4.1.1

[Gabor, 1947] Gabor, D. (1947). Acoustical quanta and the theory of hearing. *Nature*, 159.4044:591–594. 4.1.1

[Garner and Jordanous, 2016] Garner, T. and Jordanous, A. (2016). Emergent perception and video games that listen: Applying sonic virtuality for creative and intelligent npc behaviours. In *2nd Computational Creativity and Games Workshop*. 6.1

[Gaver, 1993] Gaver, W. W. (1993). What in the world do we hear?: An ecological approach to auditory event perception. *Ecological psychology*, 5(1):1–29. 2.2

[Gell-Mann, 1995] Gell-Mann, M. (1995). *The Quark and the Jaguar: Adventures in the Simple and the Complex*. Macmillan. 3.1

[Gershenson, 2007] Gershenson, C. (2007). *Design and Control of Self-organizing Systems*. PhD thesis, Vrije Universiteit Brussel. 2.1, 2.1.1, 2.3

[Gershenson and Heylighen, 2005] Gershenson, C. and Heylighen, F. (2005). How can we think the complex. *Managing organizational complexity: philosophy, theory and application*, 3:47–62. 1.3.1

[Giannakopoulos and (Auth.), 2014] Giannakopoulos, T. and (Auth.), A. P. (2014). *Introduction to Audio Analysis. A MATLAB Approach.* Academic Press, 1 edition. 3.3.1

[Giannoulis et al., 2012] Giannoulis, D., Massberg, M., and Reiss, J. D. (2012). Digital dynamic range compressor design – a tutorial and analysis. *Journal of the Audio Engineering Society*, 60(6):399–408. 4.2.2

[Gleick, 2011] Gleick, J. (2011). *Chaos: Making a new science*. Open Road Media. 1.3.2, 2.2.1, 3.3.1, 6

[Gouyon et al., 2000] Gouyon, F., Pachet, F., Delerue, O., et al. (2000). On the use of zero-crossing rate for an application of classification of percussive sounds. In *Proceedings of the COST G-6 conference on Digital Audio Effects (DAFX-00), Verona, Italy*, pages 26–31. 3.3.1

[Grey and Gordon, 1978] Grey, J. M. and Gordon, J. W. (1978). Perceptual effects of spectral modifications on musical timbres. *The Journal of the Acoustical Society of America*, 63(5):1493–1500. 3.3.1

[Hamman, 2002] Hamman, M. (2002). From technical to technological: The imperative of technology in experimental music composition. *Perspectives of New Music*, pages 92–120. 1.3.1

[Heisenberg, 1985] Heisenberg, W. (1985). Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik. In *Original Scientific Papers Wissenschaftliche Originalarbeiten*, pages 478–504. Springer. 4.1.1

[Helmholtz, 2013] Helmholtz, H. (2013). *On the sensations of tone*. Courier Corporation. 3.3.1

[Heylighen et al., 2001] Heylighen, F. et al. (2001). The science of self-organization and adaptivity. *The encyclopedia of life support systems*, 5(3):253–280. 2.1, 2.1.1

[Heylighen and Gershenson, 2003] Heylighen, F. and Gershenson, C. (2003). The meaning of self-organization in computing. *IEEE Intelligent Systems*, 18(4). 2.1.1

[Heylighen and Joslyn, 2001] Heylighen, F. and Joslyn, C. (2001). Cybernetics and second-order cybernetics. *Encyclopedia of physical science & technology*, 4:155–170. 1.3.2, 1.3.3, 2.1, 2.1, 5.1, 6

[Holland, 1995] Holland, J. H. (1995). *Hidden order: how adaptation builds complexity*. Number 003.7 H6. 6

[Holland, 2014] Holland, J. H. (2014). *Complexity: A very short introduction*. OUP Oxford. 2.5, 3.1.1, 3.1.2, 6

[Holland et al., 1992] Holland, J. H. et al. (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press. 6

[Holland John, 1975] Holland John, H. (1975). Adaptation in natural and artificial systems. *Ann Arbor: University of Michigan Press*. 1.1

[Jot and Chaigne, 1991] Jot, J.-M. and Chaigne, A. (1991). Digital delay networks for designing artificial reverberators. In *Audio Engineering Society Convention 90*. Audio Engineering Society. 4.1.5

[Karplus and Strong, 1983] Karplus, K. and Strong, A. (1983). Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7(2):43–55. 1.3.3

[Kauffman et al., 2008] Kauffman, S., Logan, R. K., Este, R., Goebel, R., Hobill, D., and Shmulevich, I. (2008). Propagating organization: An enquiry. *Biology & Philosophy*, 23(1):27–45. 3.1

[Kauffman, 1969] Kauffman, S. A. (1969). Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of theoretical biology*, 22(3):437–467. 6.1

[Kauffman, 1984] Kauffman, S. A. (1984). Emergent properties in random complex automata. *Physica D: Nonlinear Phenomena*, 10(1-2):145–156. 1.1

[Kauffman, 1993] Kauffman, S. A. (1993). *The origins of order: Self-organization and selection in evolution*. Oxford University Press, USA. 6.1

[Kellert, 2009] Kellert, S. H. (2009). *Borrowed knowledge: Chaos theory and the challenge of learning across disciplines*. University of Chicago Press. 2.2

[Kitto, 2006] Kitto, K. J. (2006). *Modelling and generating complex emergent behaviour*. Flinders University, School of Chemistry, Physics and Earth Sciences. 1.3.1

[Kohonen, 1990] Kohonen, T. (1990). The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480. 2.1.1

[Krakauer, 2011] Krakauer, D. C. (2011). Darwinian demons, evolutionary complexity, and information maximization. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 21(3):037110. 1.1

[Lagi et al., 2011] Lagi, M., Bertrand, K. Z., and Bar-Yam, Y. (2011). The food crises and political instability in north africa and the middle east. *Available at SSRN 1910031*. 1.1

[Langton, 1986] Langton, C. G. (1986). Studying artificial life with cellular automata. *Physica D: Nonlinear Phenomena*, 22(1-3):120–149. 1.1

[Le Brun, 1979] Le Brun, M. (1979). Digital waveshaping synthesis. *Journal of the Audio Engineering Society*, 27(4):250–266. 4.1.3

[Lendaris, 1964] Lendaris, G. (1964). On the definition of self-organizing systems. *Proceedings of the IEEE*, 52(3):324–325. 2.1.1

[Levien and Tan, 1993] Levien, R. and Tan, S. (1993). Double pendulum: An experiment in chaos. *American Journal of Physics*, 61(11):1038–1044. 2.2

[Lewes, 1874] Lewes, G. (1874). Emergence. Dictionnaire de la langue philosophique. 2.3.1

[Logan, 2014] Logan, R. K. (2014). *What is information?: Propagating organization in the biosphere, symbolosphere, technosphere and econosphere*. Demo Publishing. 3.1

[Lorenz, 1963] Lorenz, E. N. (1963). Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141. 2.2.1

[Lorenz, 1964] Lorenz, E. N. (1964). The problem of deducing the climate from the governing equations. *Tellus*, 16(1):1–11. 2.2

[Mach, 1914] Mach, E. (1914). *The analysis of sensations, and the relation of the physical to the psychical*. Open Court Publishing Company. 4.1.1

[Maes, 1993] Maes, P. (1993). Modeling adaptive autonomous agents. *Artificial life*, 1(1_2):135–162. 2.5

[Mamei et al., 2006] Mamei, M., Menezes, R., Tolksdorf, R., and Zambonelli, F. (2006). Case studies for self-organization in computer science. *Journal of Systems Architecture*, 52(8_9):443–460. 2.1.1

[Matsumoto, 1984] Matsumoto, T. (1984). A chaotic attractor from Chua's circuit. *IEEE Transactions on Circuits and Systems*, 31(12):1055–1058. 2.2

[Maturana, 2002] Maturana, H. (2002). Autopoiesis, structural coupling and cognition: a history of these and other notions in the biology of cognition. *Cybernetics & human knowing*, 9(3-4):5–34. 2.4, 2.4.1, 2.5, 3.1

[Maturana and Varela, 1980] Maturana, H. R. and Varela, F. J. (1980). Autopoiesis: The organization of the living. *Autopoiesis and cognition: The realization of the living*, 42:59–138. 1.1, 1.3.3, 2.4, 2.4.1

[May, 1976] May, R. (1976). Special mathematical models with very complicated dynamics. *Nature*. 2.2

[Miranda and Brouse, 2005] Miranda, E. R. and Brouse, A. (2005). Interfacing the brain directly with musical systems: on developing systems for making music with brain signals. *Leonardo*, 38(4):331–336. 6.1

[Mitchell, 2006] Mitchell, M. (2006). Complex systems: Network thinking. *Artificial Intelligence*, 170(18):1194–1212. 2.2, 2.3, 2.5, 6

[Mitchell, 2009] Mitchell, M. (2009). *Complexity: A guided tour*. Oxford University Press. 2.5, 5.4, 6

[Mizumoto and Tanaka, 1981] Mizumoto, M. and Tanaka, K. (1981). Fuzzy sets and their operations. *Information and Control*, 48(1):30–48. 3.1.2

[Mocenni et al., 2011] Mocenni, C., Facchini, A., and Vicino, A. (2011). Comparison of recurrence quantification methods for the analysis of temporal and spatial chaos. *Mathematical and Computer Modelling*, 53(7-8):1535–1545. 3.3.2

[Molla and Torrésani, 2004] Molla, S. and Torrésani, B. (2004). Determining local transientness of audio signals. *IEEE signal processing letters*, 11(7):625–628. 3.3.1

[Moore, 1995] Moore, B. C. (1995). *Hearing*. Academic Press. 3.3.1

[Morin, 1977] Morin, E. (1977). *La nature de la nature*, volume 123. Seuil Paris. 1.1, 1.3.3

[Morin, 1992] Morin, E. (1992). From the concept of system to the paradigm of complexity. *Journal of social and evolutionary systems*, 15(4):371–385. 2.3

[Morin, 2007] Morin, E. (2007). Restricted complexity, general complexity. *Science and us: Philosophy and Complexity. Singapore: World Scientific*, pages 1–25. 1.3.1

[Mudd, 2017] Mudd, T. (2017). *Nonlinear dynamics in musical interactions*. PhD thesis, The Open University. 2.2

[Mumma, 1967] Mumma, G. (1967). Creative aspects of live-performance electronic music technology. In *Audio Engineering Society Convention 33*. Audio Engineering Society. 1.1.1

[Murray-Rust and Smaill, 2011] Murray-Rust, D. and Smaill, A. (2011). Towards a model of musical interaction and communication. *Artificial Intelligence*, 175(9-10):1697–1721. 5.1

[Orlarey et al., 2004] Orlarey, Y., Fober, D., and Letz, S. (2004). Syntactical and semantical aspects of faust. *Soft Computing*, 8(9):623–632. 4.3

[Orlarey et al., 2009] Orlarey, Y., Fober, D., and Letz, S. (2009). *FAUST : an Efficient Functional Approach to DSP Programming*, pages 65–96. 4.3

[Patteson, 2012] Patteson, T. (2012). The time of Roland Kayns cybernetic music. *Travelling Time, Sonic Acts XIV*, pages 47–67. 1.1.1

[Peeters et al., 2011] Peeters, G., Giordano, B. L., Susini, P., Misdariis, N., and McAdams, S. (2011). The timbre toolbox: Extracting audio descriptors from musical signals. *The Journal of the Acoustical Society of America*, 130(5):2902–2916. 3.3.1, 3.3.1

[Pressing, 1984] Pressing, J. (1984). Cognitive processes in improvisation. In *Advances in Psychology*, volume 19, pages 345–363. Elsevier. 1.3.2

[Pressing, 1988] Pressing, J. (1988). Improvisation: Methods and models. *John A. Sloboda (Hg.): Generative processes in music, Oxford*, pages 129–178. 5.1

[Prigogine, 1978] Prigogine, I. (1978). Time, structure, and fluctuations. *Science*, 201(4358):777–785. 1.5, 5.4, 5.4.1

[Prigogine and Nicolis, 1985] Prigogine, I. and Nicolis, G. (1985). Self-organisation in nonequilibrium systems: towards a dynamics of complexity. In *Bifurcation analysis*, pages 3–12. Springer. 1.1, 2.1.1

[Puckette, 1997] Puckette, M. (1997). Pure data. In *Proceedings of the international computer music conference*, pages 224–227. International Computer Music Association. 4.4

[Puckette, 2007] Puckette, M. (2007). *The theory and technique of electronic music.* World Scientific Publishing Company. 4.1.1

[Puckette et al., 1998] Puckette, M. S., Ucsd, M. S. P., Apel, T., et al. (1998). Real-time audio analysis tools for pd and msp. 3.3.1

[Rimoldi and Manzolli, 2016] Rimoldi, G. and Manzolli, J. (2016). Medidas de quantificaçõ recorrência: uma proposta de análise para Audible Ecosystems de Agostino Di Scipio. In *Proceedings of the XXVI Congresso da Assoc. Nacional de Pesquisa e Pós-Grad. em Musica - Belo Horizonte*. 3.3.2

[Roads, 1979] Roads, C. (1979). A tutorial on non-linear distortion or waveshaping synthesis. *Computer Music Journal*, pages 29–34. 4.1.3

[Roads, 1988] Roads, C. (1988). Introduction to granular synthesis. *Computer Music Journal*, 12(2):11–13. 4.1.1

[Rocchesso, 1997] Rocchesso, D. (1997). Maximally diffusive yet efficient feedback delay networks for artificial reverberation. *IEEE Signal Processing Letters*, 4(9):252–255. 4.1.5

[Rocchesso, 2003] Rocchesso, D. (2003). *Introduction to sound processing.* Mondo estremo. 4.1.1

[Rocchesso and Smith, 1997] Rocchesso, D. and Smith, J. O. (1997). Circulant and elliptic feedback delay networks for artificial reverberation. *IEEE Transactions on Speech and Audio Processing*, 5(1):51–63. 1.3.3, 4.1.5

[Rognoni, 1966] Rognoni, L. (1966). La musica "elettronica" e il problema della tecnica. In *Fenomenologia della musica radicale*. Laterza. 1.3.1

[Rosen, 1978] Rosen, R. (1978). *Fundamentals of measurement and representation of natural systems*, volume 1. North Holland. 2.3.1

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386. 1.1

[Sanfilippo, 2012a] Sanfilippo, D. (2012a). Lies (distance/incidence) 1.0: a human-machine interaction performance. In *Proceedings of the 19th Colloquium on Music Informatics*, pages 198–199. 5.4.1

[Sanfilippo, 2012b] Sanfilippo, D. (2012b). Osservare la macchina performante e intonare l'ambiente: Microphone di David Tudor. *Le Arti del Suono*, 1(6):70–82. 5.4

[Sanfilippo, 2013] Sanfilippo, D. (2013). Turning perturbation into emergent sound, and sound into perturbation. *Interference: A Journal of Audio Culture*, 3. 1.3.1

[Sanfilippo, 2018] Sanfilippo, D. (2018). Time-variant infrastructures and dynamical adaptivity for higher degrees of complexity in autonomous music feedback systems: the Order from Noise (2017) project. *Musica/Tecnologia*, 12(1):119–129. 3.2

[Sanfilippo and Di Scipio, 2017] Sanfilippo, D. and Di Scipio, A. (2017). Environment-mediated coupling of autonomous sound-generating systems in live performance: An overview of the Machine Milieu project. In *Proceedings of the 14th Sound and Music Computing Conference, Espoo, Finland*, pages 5–8. 1.2, 5.1

[Sanfilippo and Valle, 2013] Sanfilippo, D. and Valle, A. (2013). Feedback systems: An analytical framework. *Computer Music Journal*, 37(2):12–27. 1.1.1, 5.4

[Schumann and Smarandache, 2007] Schumann, A. and Smarandache, F. (2007). *Neutrality and many-valued logics*. Infinite Study. 3.1.2, 3.1.3

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423. 3.1

[Siedenburg and McAdams, 2017] Siedenburg, K. and McAdams, S. (2017). The role of long-term familiarity and attentional maintenance in short-term memory for timbre. *Memory*, 25(4):550–564. 3.3.2

[Snyder and Snyder, 2000] Snyder, B. and Snyder, R. (2000). *Music and memory: An introduction*. MIT press. 3.3.2, 3.3.2

[Song and Liang, 2013] Song, W. and Liang, J. (2013). Difference equation of Lorenz system. *International Journal of Pure and Applied Mathematics*, 83(1):101–110. 2.2.1

[Stautner and Puckette, 1982] Stautner, J. and Puckette, M. (1982). Designing multi-channel reverberators. *Computer Music Journal*, 6(1):52–65. 4.1.5

[Steels, 1997] Steels, L. (1997). The synthetic modeling of language origins. *Evolution of communication*, 1(1):1–34. 1.1

[Steels, 2003] Steels, L. (2003). Evolving grounded communication for robots. *Trends in cognitive sciences*, 7(7):308–312. 1.1

[Streich et al., 2006] Streich, S. et al. (2006). *Music complexity: a multi-faceted description of audio content*. Universitat Pompeu Fabra Barcelona, Spain. 3.3.2

[Sullivan, 1990] Sullivan, C. R. (1990). Extending the Karplus-Strong algorithm to synthesize electric guitar timbres with distortion and feedback. *Computer Music Journal*, 14(3):26–37. 4.2.1

[Terhardt, 1974] Terhardt, E. (1974). On the perception of periodic sound fluctuations (roughness). *Acta Acustica united with Acustica*, 30(4):201–213. 3.3.1

[Thompson et al., 2002] Thompson, J. M. T., Thompson, M., and Stewart, H. B. (2002). *Nonlinear dynamics and chaos*. John Wiley & Sons. 2.2

[Tindale et al., 2004] Tindale, A. R., Kapur, A., and Fujinaga, I. (2004). Towards timbre recognition of percussive sounds. In *Proceedings of the international computer music conference*, pages 592–595. 3.3.1

[Truax, 1988] Truax, B. (1988). Real-time granular synthesis with a digital signal processor. *Computer Music Journal*, 12(2):14–26. 4.1.1

[Tsirimokou et al., 2017] Tsirimokou, G., Psychalinos, C., and Elwakil, A. (2017). Procedure for designing fractional-order filters. In *Design of CMOS Analog Integrated Fractional-Order Circuits*, pages 13–39. Springer. 3.3.1

[Vassilakis and Kendall, 2010] Vassilakis, P. N. and Kendall, R. A. (2010). Psychoacoustic and cognitive aspects of auditory roughness: definitions, models, and applications. In *Human Vision and Electronic Imaging XV*, volume 7527, page 75270O. International Society for Optics and Photonics. 3.1.2, 3.3.1, 3.3.2

[Vemuri, 2014] Vemuri, V. (2014). *Modeling of complex systems: an introduction*. Academic Press. 1.1

[Von Foerster, 1952] Von Foerster, H. (1952). Cybernetics; circular causal and feedback mechanisms in biological and social systems. 2.1.1

[Von Foerster, 2003a] Von Foerster, H. (2003a). Cybernetics of cybernetics. In *Understanding understanding*, pages 283–286. Springer. 1.1, 1.3.1, 6

[Von Foerster, 2003b] Von Foerster, H. (2003b). On constructing a reality. In *Understanding understanding*, pages 211–227. Springer. 3.2.2

[Von Foerster, 2003c] Von Foerster, H. (2003c). On self-organizing systems and their environments. In *Understanding Understanding*, pages 1–19. Springer. 2.1.1, 6

[von Glasersfeld, 1979] von Glasersfeld, E. (1979). Cybernetics, experience, and the concept of self. *A cybernetic approach to the assessment of children: Toward a more humane use of human beings*, pages 67–113. 1.1, 3.2.2

[Wagner and Altenberg, 1996] Wagner, G. P. and Altenberg, L. (1996). Perspective: complex adaptations and the evolution of evolvability. *Evolution*, 50(3):967–976. 3.2, 6

[Waldrop, 1993] Waldrop, M. M. (1993). *Complexity: The emerging science at the edge of order and chaos*. Simon and Schuster. 2.5

[Wiener, 1948] Wiener, N. (1948). *Cybernetics or Control and Communication in the Animal and the Machine*. Technology Press. 1.1, 2.1

[Wilensky and Rand, 2015] Wilensky, U. and Rand, W. (2015). *An introduction to agent-based modeling: modeling natural, social, and engineered complex systems with NetLogo*. MIT Press. 1.1

[Zavalishin, 2012] Zavalishin, V. (2012). The art of VA filter design. *Native Instruments*. 3.3.1, 4.1.6, 4.2.1

[Zölzer, 2008] Zölzer, U. (2008). *Digital audio signal processing*. John Wiley & Sons. 3.3.1