ECHO

ORPHEUS INSTITUUT

a journal of music, thought and technology

Title: Emergent Behaviors in Sound Synthesis

Subtitle: The RITI Project and Other Systemic Approaches in Electronic Music

Author(s): Luca Spanedda

Issue: #7

Publication date: 2026

Review status:

Article DOI:

Abstract

This paper explores systemic music through the lens of chaotic synthesis, culminating in the development of *RITI: Room Is The Instrument*. By tracing a trajectory from early electronic and cybernetic music to contemporary complex adaptive systems in live electronic performance, the work examines how nonlinear feedback interactions in sound synthesis give rise to emergent sonic behaviors.

RITI is a live electronics composition in which a chaotic, complex adaptive system is performed in real time, with sound generation directly derived from the solution of chaotic equations. The piece transforms the performance space into a resonant, evolving instrument, where environmental interactions continuously shape its sonic behavior.

1. SYSTEMIC THINKING IN ELECTRONIC MUSIC

The idea of composition as a system has long been present in musical thought, and the artistic and scientific revolutions of the 20th century introduced a more precise notion of systemic thinking. With the advent of cybernetics and complexity science, feedback systems gained significant attention in music, taking different forms depending on how composers conceptualized musical structures and the organization of sound. Cybernetic approaches, in particular, became deeply intertwined with early electronic music: Here, the system is defined by formalized rules yet conceived as an interactive model generating emergent relationships within the sonic material.

In Europe, Werner Meyer-Eppler, Robert Beyer, and Herbert Eimert, key participants in the establishment of the WDR Electronic Music Studio in Cologne, were receptive to emerging scientific ideas in systems theory and organizational theory. Their approach to music theory emphasized treating sound as a structured set of parameters (total serialism) rather than following conventional compositional methods, investigating how musical form could emerge from the interactions and relationships between these parameters. The Cologne Studio became quickly the most influential in the world during the 1950s and 1960s, hosting many of the most important contemporary composers such as Franco Evangelisti, Karlheinz Stockhausen, György Ligeti, Roland Kayn, Herbert Brün, and Cornelius Cardew.

Electronic music encompassed the composition of sound itself and the exploration of its internal relationships. At the Cologne Studio, Karlheinz Stockhausen and Franco Evangelisti developed methods in which musical form arose naturally from the properties of the generated sound materials. Their pioneering works, *Studie II* (1954) and *Incontri di Fasce Sonore* (1956), are central for their early exploration of music as a system, where form emerges from the relationships and interactions among sound parameters.

In this context, the work of composer Roland Kayn is particularly significant. His cybernetic music project received its initial impulse in 1953 when, as a young student, he came into contact with the philosopher Max Bense[1]. Immediately after that first encounter with Bense, Kayn met Herbert Eimert at the Cologne studio. Kayn was fascinated by the sonic possibilities of the new technologies, but considered the serialist aesthetic dominant at the studio at the time to be too restrictive. After this experience, over the next decade he focused mainly on instrumental composition and the formal application of cybernetic theories (Hernandez 2017).

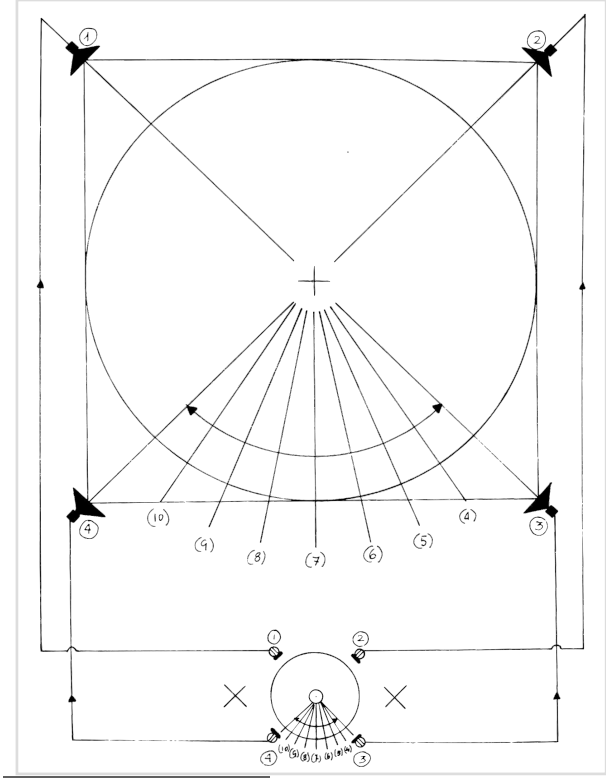
In 1964, Franco Evengelisti, also inspired by the theories of cybernetics, established the Gruppo di Improvvisazione Nuova Consonanza (GINC), a collective of musicians oriented towards timbral and formal experimentation. In the same year Roland Kayn moved to Rome and joined the GINC. Kayn began to experiment intensively with self-regulated cybernetic systems based on feedback loops, no longer simply as formal models for instrumental compositions but also as signal-generating networks of analogue devices. Kayn left in 1968, dissatisfied with the group's reliance upon clichés and inability to properly integrate cybernetic concepts into their improvisatory framework (Hernandez 2017).

[1] Max Bense was among the first European thinkers to integrate the primarily Anglophone disciplines of cybernetics and information theory into artistic and aesthetic practice. Roland Kayn spent three years in Bense's circle at the Technische Universität Stuttgart, where he was deeply influenced by these ideas.

In 1968, Walter Branchi joined the GINC, and the following year he composed *Thin* for cymbal and amplified tam-tam. In this work the system was no longer an abstract model but derived directly from the behavior of the instrument itself, extending Stockhausen's ideas of live electronic music, as developed with *Mikrophonie I* (1964).

In Branchi's notion of *sistema sonoro*, for the first time, composition is no longer a structure imposed from the outside but arises from the nature of the system that produces it[2]. In *Thin*, each sound manifests the physical system with its specific spectral and morphological characteristics. The notation itself is not conceived a priori but emerges from the analysis of the instrument's behavior. The cymbal and the tam-tam, with their spectral peculiarities determine the rules of musical writing creating a conceptual feedback loop between the score and the properties of the instruments themselves.

It is important to note that in those same years, a similar conception of the instrument-space relationship can also be found in the first self-built instruments of David Tudor and Hugh Davies, where the system's internal behavior is the work itself, shaping both the form of the piece and the sonic outcome. This approach could be understood in terms of musical instruments as *epistemic tools* (Magnusson 2019): instruments and technologies are not merely sound-producing tools but embody knowledge and ways of thinking about musical processes. The acoustical instruments of *Thin* themselves function in this manner, structuring compositional decisions and shaping the emergent musical outcome.



In *Thin*, Walter Branchi conceives the spatial setup as an extension of the instruments themselves. In this excerpt from the score, the correspondence between microphones and loudspeakers functions as a conceptual mapping: the diffusion points mirror the listening points, creating the impression of listening from within the cymbal's acoustic space. In this way, the instrument expands centrifugally, projecting its internal space into the room and physically incorporating the listeners into the instrument.

Branchi prescribes a minimum of four microphones placed around the suspended cymbal, each corresponding to one loudspeaker located at the corners of a square. The space acts as both resonant body and reflective medium, situating the listener inside the system's dynamics[3].

[2] Walter Branchi describes the idea of *sistema sonoro* in depth in his seminal writings, including: (Pizzaleo, 2022, Branchi 2023).

[3] For a more detailed examination of Branchi's approach in Thin see Pizzaleo (2022).

While European composers in this period explored systemic relationships through formalized rules and compositional models, American composers such as John Cage developed approaches in which scores are themselves generative systems, emphasizing open processes rather than fixed outcomes.

In his *Imaginary Landscape No. 4* (circa 1949, first performed in 1951), Cage employed twelve radios exploring live electronic generation of sound. Earlier works, like *Imaginary Landscape No. 1* (1939), combined recordings of constant and variable pitch frequencies with conventional percussion instruments. In some of Cage's early works from the 1960s, the score explicitly mentions the term *feedback*, making the generation of feedback an integral part of live performance. For instance, in *Cartridge Music* (1960), or in *Electronic Music for Piano* (1964).

Similarly, David Tudor's performance of Cage's *Variations II* (1961) shows how contact microphones and the free vibration of strings create a complex system of feedback and resonance that cannot be fully predicted (Perloff 2001).

During the 1960s, numerous live electronic performance groups formed, often employing extensive use of feedback. Among these were the *Sonic Arts Union* (SAU, founded in 1966) with Gordon Mumma, Robert Ashley, Alvin Lucier, and David Behrman; *Musica Elettronica Viva* (founded in 1967 in Italy) with Frederic Rzewski, Allan Bryant, and Alvin Curran; and the *Once Group* (Michigan) with Robert Ashley, Gordon Mumma, and others.

In SAU works, particularly those by Gordon Mumma, feedback circuits in live electronic music became central to a musical practice where performance and system architecture are inseparable from the compositional act itself. Mumma describes his compositional process as follows:

> My own electronic music equipment is designed as part of the process of composing my music. I am really like the composer who builds his own instruments, though most of my 'instruments' are inseparable from the compositions themselves [...] My decisions about electronic procedures, circuitry, and configurations are strongly influenced by the requirements of my profession as a music maker. This is one reason why I consider that my designing and building of circuits is really 'composing.' I am simply employing electronic technology in the achievement of my art.
>
> ***(Mumma 1967)***

This approach shifts the focus from the written score to performance and situatedness: perfromers become active agents within a complex network of living relations. For example, in Gordon Mumma's *Hornpipe* (1967) for horn and Cybersonic console, the acoustic behavior of the space—its resonances, reflections, and frequency responses—is a compositional

parameter, transforming the environment into part of the musical performance and dissolving the boundaries between notation, instrument, space, and performer.

With the availability of computers, new frontiers opened up for composers interested in systemic approaches. Particularly with real-time computing and human-computer interaction, the composer is no longer just an organizer of musical events, but becomes now a designer of processes that evolve in real-time. This transformation established systemic thinking as a fundamental paradigm in contemporary music.

1.1 THE DEVELOPMENT OF CHAOTIC SYNTHESIS

Although early insights into chaotic behavior emerged in early twentieth century, chaos theory was formally developed in the 1960s and 1970s, particularly through the work of Edward Norton Lorenz (Gleick 1987). Subsequently, chaos theory emerged as a framework for understanding deterministic yet unpredictable dynamics.

As in electronic music, a significant catalyst for this field was the advent of electronic computers. The ability to computationally model and analyze nonlinear dynamics provided new tools for exploring unpredictable behaviors in innumerable domains, revealed to be useful for artistic explorations too. As a result, scientific inquiry shifted from simply predicting and controlling systems to understanding their dynamics and relationships, allowing for an incredible variety of practical applications.

Towards the late 1980s, the increasing computational power of computers enabled real-time sound synthesis and processing, and initiated a new era in electronic music. This advancement made it possible to use computers not only for offline sound synthesis but also for live electronic music performance. Composers began experimenting with sound synthesis techniques based on iterated nonlinear functions (Di Scipio 1990, Truax 1990, Di Scipio 2001).

Between 1990 and the early 2000s, Agostino Di Scipio and others (Degazio 1993, Choi 1994a, Choi 1994b, Yadegari 2003) explored in depth the application of chaos theory to sound synthesis numerically solving chaotic equations while also enabling real-time control of these processes. More recent research has been conducted by (Pirrò 2017, Mudd 2019, Sanfilippo 2021a), focusing on complex sound generation based on modified chaotic differential equations.

What distinguishes the latter contributions from earlier work is a focus on the autonomy of the system over large time scales. Unlike earlier approaches, which often involved manual or predefined controls to guide signal generating chaotic behaviors, the works presented in the following sections aim at creating in systems unsupervised ways to evolve over longer time spans. In the context of the RITI project, the approach benefits from specially implemented control signal processing units (CSPu) which allow the system to dynamically and coherently adapt to sound signals. Based on that, the RITI process shows a unique capacity to generate emergent sound behaviors while remaining responsive to a partner performer's input.

1.2 RITI (ROOM IS THE INSTRUMENT) AS A RESEARCH PLATFORM

RITI (Room Is The Instrument) is a live electronics composition that builds upon the real-time simulation of chaotic differential equations. At the core of RITI is a feedback delay network (FDN) that enables coupling of complex sound generators (CSGs). The latter (discussed below in more detail) are further conditioned by filter banks modelled after specific sound materials (musical instruments or other), thus allowing the occurrence of modal resonances in the synthesized sound. During performance, control signal processing units extract parameters resampling the oscillators' output, and utilize such data to change the interaction of coupled oscillators, thus modifying their behavior.

Since the composition relies on chaotic equations, it is inherently sensitive to initial conditions. In a performative situation, even the smallest deviations, whether introduced by the performer or by the physical environment, can drastically alter the system's trajectory and future behavior, with the flow of subsequent interactions producing unique outcomes based on minute variations. The interaction between the performer and the system defines the performance process and, ultimately, the global form of the piece. By adjusting the weights in the network, the performer shapes the chaotic dynamics, thus engaging with a fine balance of process determinacy and emergent sonic byproducts. As Pirrò notes (Pirrò 2017), the use of chaotic dynamical systems leads to a fundamentally different process than typically found in traditional sound synthesis: the sonic outcome is largely unknown a priori, and the composer formulates rules of evolution rather than specifying perceptual appearance. This approach resonates with the concept of non-standard synthesis techniques, as discussed by Döbereiner (Pirrò 2017), in which sound is defined by the process generating it rather than by perceptual expectations. An exploration of the behavior space generated by this process is necessary in order to construct the piece.

All signal processing operations in the RITI project is implemented in the FAUST programming language (Orlarey 2004, GRAME FAUST), which compiles into C++ and supports various platforms. This ensures cross-platform compatibility while maintaining the same software infrastructure.

2. COMPLEX SOUND GENERATORS

The concept of complex sound generators refers to sound synthesis modules that numerically solve chaotic equations in a controlled manner. This idea is elaborated and further developed in recent works, particularly by Dario Sanfilippo (Sanfilippo 2021a). Sanfilippo also introduces mathematical constraints to chaotic equations. These constraints allow for the exploration of chaotic equations even in unstable regions. This is done, for example, with a mix of *DC blockers* and *waveshapers*, centering the signal on the x-axis and limiting it into the [-1, +1] range (most chaotic models would, by themselves, provide signal values outside the range). Sanfilippo terms a single module that meets such requirements a Complex Sound Generator (CSG).

In RITI, CSGs are built using Duffing oscillators. In addition to the constraints proposed by Sanfilippo, and following an approach outlined by Tom Mudd (Mudd 2019) bandpass filtering is introduced in the Duffing differential equation. As previously noted, this allows the system to manifest modal resonances reminiscent of a musical instrument's behavior. Overall, that seems a consistent approach, as in fact both physical modeling synthesis and chaotic oscillators require simulating nonlinear complex system equations (Rodet 1999, Mudd 2019).

I started by solving the Duffing equation, as (Mudd 2019). Other methods of numerical solution of the Duffing oscillator can be found in papers by (Degazio 1993, Yadegari 2003). The latter shows how to solve numerically various chaotic oscillators for sound synthesis using the *fexpr~* audio-rate object he and Miller Puckette designed in the Pure Data programming environment. Besides the Duffing oscillator, one may consider the numerical solutions of many other chaotic equations, some of which have already been implemented in the FAUST programming language (Sanfilippo 2021a).

2.1 THE MODIFIED DUFFING OSCILLATOR

The Duffing equation is a second-order nonlinear differential equation that describes the behavior of an oscillator with a cubic nonlinearity. Specifically, it is an **ordinary differential equation (ODE)**, i.e., an equation involving derivatives of a function of a single variable (time):

$$\ddot{x} + \delta\dot{x} + \alpha x + \beta x^3 = \gamma \cos(\omega t) \tag{1}$$

The Duffing equation was first studied by Georg Duffing in the 1910s, as a model for the motion of a mechanical system with a nonlinear spring. It can be found in his original book (Duffing, 1918).

Here the function $x(t)$ in equation (1) represents the displacement of the oscillator as a function of time. The derivatives are:

$$\dot{x}(t) = \frac{dx}{dt}, \quad \ddot{x}(t) = \frac{d^2x}{dt^2} \tag{2}$$

where $\dot{x}(t)$ is the velocity and $\ddot{x}(t)$ is the acceleration. The parameters in equation (2) are:

$$\delta, \quad \alpha, \quad \beta, \quad \gamma, \quad \omega \tag{3}$$

where:

- δ damping coefficient, governing energy loss in the system.
- α linear stiffness coefficient.
- β strength of the non-linear cubic term x^3.
- γ strength of the external driving force.
- ω frequency of the periodic driving force $\gamma \cos(\omega t)$.

This system exhibits various behaviors, including harmonic oscillations and chaotic motion, depending on parameter values. The term βx^3 makes it a nonlinear system, while the presence of $\gamma \cos(\omega t)$ indicates external forcing. A discrete version of equation (1), as found in (Mudd 2019), is given by:

$$X_{n+1} = Y_n \tag{4}$$

$$Y_{n+1} = K Y_n - \alpha f(X_n^3) - B \cos(\omega T) \tag{5}$$

Taken together, equations (4) and (5) represent a discrete mapping of the continuous-time Duffing equation. In FAUST, a implementation with fixed parameters follows the same structure (see Listing 1), solved numerically via Euler's method[4].

The Figures 1 and 2 illustrate the outputs of the Duffing oscillator generated by the FAUST implementation over the first 250 milliseconds at a sampling frequency of 192 kHz. Figure 1 shows the 2D phase space trajectory, in which the position of the oscillator is plotted against its velocity, highlighting the nonlinear dynamics and the chaotic behavior of the system. The trajectory in this plane reveals the structure of the underlying chaotic attractor. Figure 2 displays the corresponding waveforms of the oscillator, one below the other, showing the amplitude variations over time. Together, these visualizations provide insight into both the temporal and phase-space behavior of the oscillator under the chosen parameters (in Listing 1).

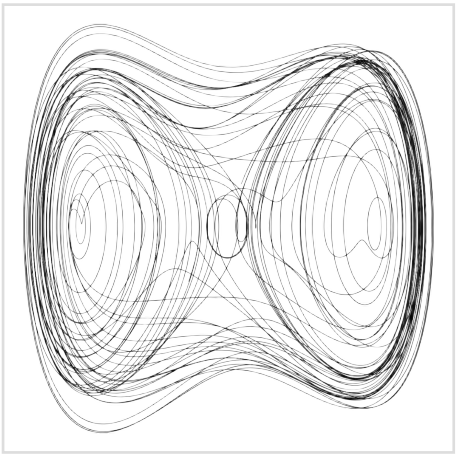


Figure 1. Phase space plot of the Duffing oscillator

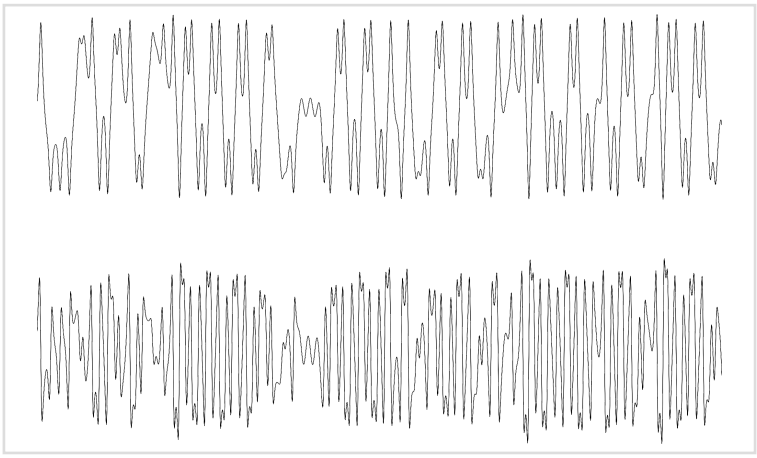


Figure 2. Monodimensional plot of the Duffing oscillator

[4] Numerical integration methods, such as Euler's method, are widely used to approximate solutions of differential equations when analytical solutions are unavailable. In the context of audio DSP and physical modeling, explicit methods like Euler's are simple to implement but may require small time steps for stability, while more advanced methods (e.g., Runge-Kutta) offer improved accuracy and stability but at the cost of computational complexity.

Using FAUST's functional grammar, we can rewrite the Duffing oscillator code (as in Listing 2). This produces the block diagram of the oscillator as in Figure 3. The linear coefficient *alpha* = -1.0 is algebraically simplified by incorporating the negative sign into the subtraction operation, effectively transforming the linear term into a positive contribution that yields the simplified form $x - x^3$. Negative signs are resolved within the signal flow, consolidating the mathematical expression into its essential form. Parameters are renamed for semantic clarity: delta becomes damping (visible in the feedback multiplication from *y*), *gamma* becomes *forcing*, and *omega* becomes *forcing_frequency*, while *dt* (the integration time step) remains explicit. This algebraic simplification eliminates redundant constant multiplications and sign operations, allowing the code structure to directly mirror the signal flow topology. As a result, the implementation appears as the block diagram of the oscillator shown in the Figure 3, where each processing operation corresponds to a distinct circuit block without unnecessary terms (in this context).

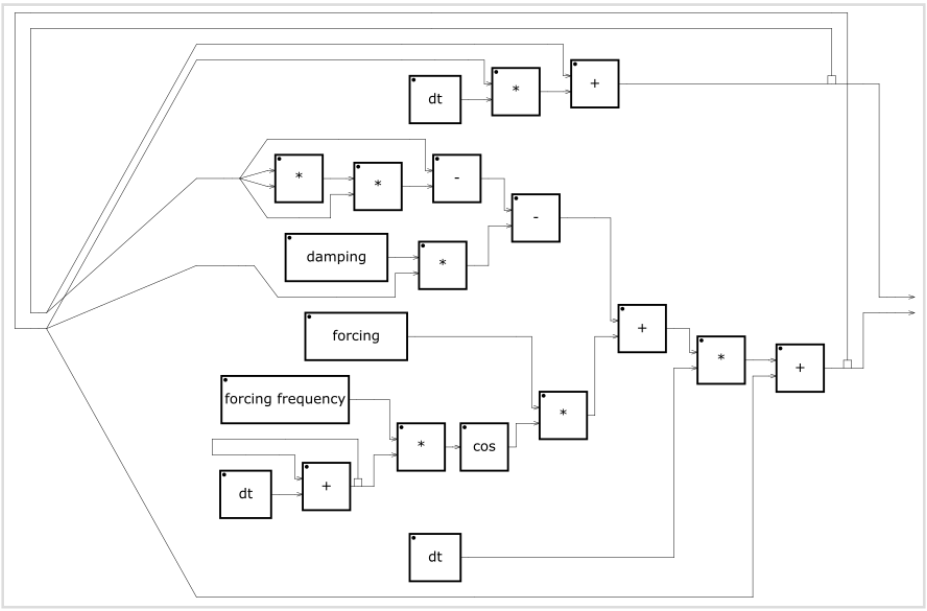


Figure 3. Block diagram in FAUST of the semplified Duffing oscillator

The implementation then introduces the arc tangent function, in order to prevent the signal from exceeding the numerical range and to be able to also explore unstable, extremely chaotic regions. Finally, we add a set of bandpass filters, in order to further constrain the oscillator, and enable it to resonate at a specific frequency of the spectrum (see Listing 3). The latter coupling is achieved by adding 24 parallel bandpass filters to equation (6):

$$X_{n+1} = \sum_{i=1}^{24} BP_i(\arctan(Y_n)) \tag{6}$$

$$Y_{n+1} = KY_n - \alpha f(X_n^3) - B\cos(\omega T) \tag{7}$$

These filters function as constraints on the oscillator, each filter having its own frequency. The set of filters is designed using Robert Bristow-Johnson's biquad filter coefficients (Bristow Johnson Audio EQ, Smith Digital Filters).

Some CSG parameters define the initial conditions for the system (in the code Listing 3 these include *gain*=0.5, *damping*=0.15, *forcing*=0.7, *forcing_frequency*=328.0). These parameters, along with others available in the GUI, can be modified in real-time by the performer. Notice, in Listing 3, the *influence* and *external_Mic* variables: these enable coupling between multiple signal generating processes in the network, as well as the coupling between those and the external environment. The former represents the interaction between multiple CSGs affecting the *damping* parameter, while the latter is a microphone input signal fed directly into the *sin* function of the Duffing oscillator, allowing the system to fold into itself through the external environment via loudspeakers and microphones. The FAUST diagram of the complete CSG is shown in Figure 4.

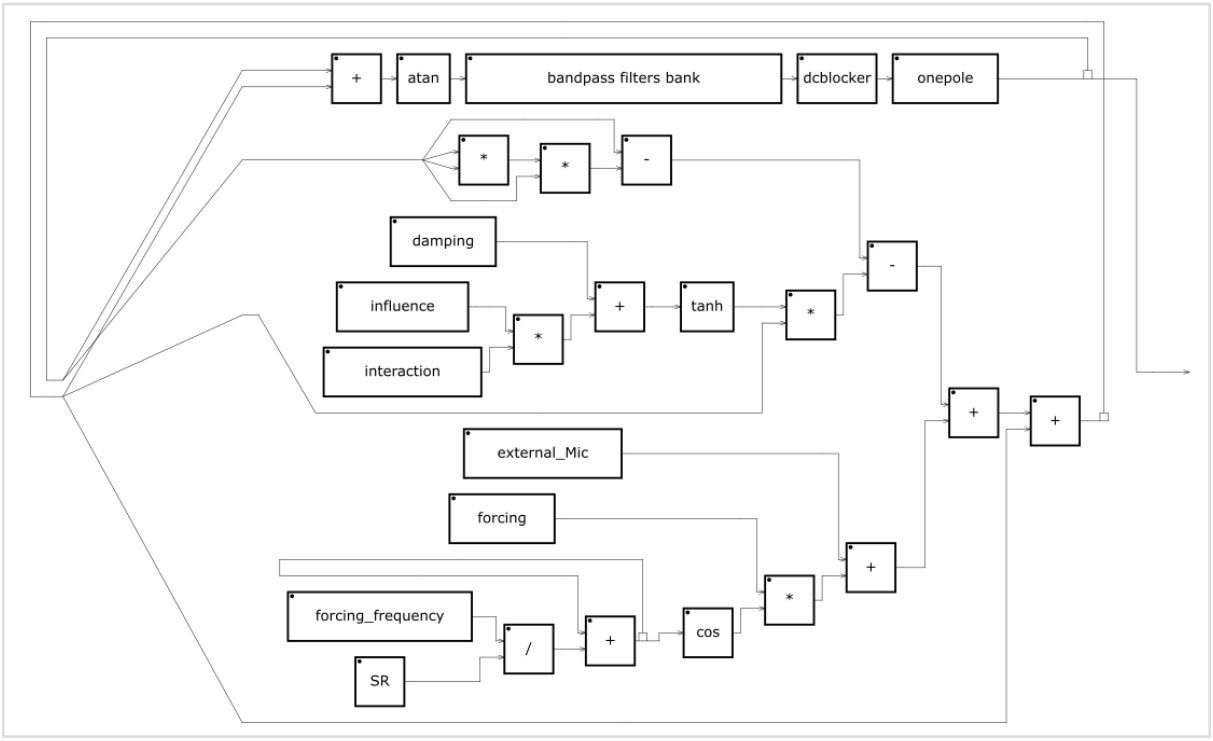


Figure 4. Complex sound generator

Within the network, the CSGs are coupled in a cross-feed configuration. Each CSG affects the damping factor of the other. The system equations describing this interaction are as follows

$$X_1^{n+1} = \sum_{i=1}^{24} BP_i(\arctan(Y_1^n)) \tag{8}$$

$$Y_1^{n+1} = \left(K + X_2^{n-m}\right) Y_1^n - \alpha f(X_1^n)^3 - B\cos(\omega T) \tag{9}$$

$$X_2^{n+1} = \sum_{i=1}^{24} BP_i(\arctan(Y_2^n)) \tag{10}$$

$$Y_2^{n+1} = \left(K + X_1^{n-m}\right) Y_2^n - \alpha f(X_2^n)^3 - B\cos(\omega T) \tag{11}$$

where variable m represents a delay (in samples) in the signal exchanges between the CSGs, with x_1 and x_2 affecting each other in a feedback loop. Equations (8) to (11) are implemented in FAUST using a configuration with 8 CSG as shown in code Listing 4. As discussed below, the mutual influence of CSG's incorporating nonlinear controls can be extended over larger time scales.

3. CONTROL SIGNAL PROCESSING UNITS

In the context of feature extraction, a distinction is made between low-level and high-level audio information processing. Low-level algorithms enable continuous feature measurement and operate with short analysis frames. In contrast, high-level algorithms are original designs informed by both perceptual principles and complexity theory, designed to analyze musically meaningful information (Sanfilippo 2021b).

From this perspective, control signals should be seen as the processing of extracted audio features, and their mapping at a low-frequency generates autonomous dynamical behaviors directly derived from the audio signals of the network. Control signal processing function as a kind of self-regulating mechanism in the network system itself allowing it to evolve autonomously over time (Di Scipio 2003). Di Scipio and Sanfilippo characterize such an approach as follows:

> "autonomous"" is not to be confused with "automated." Automation, implies centralized control. In typical computer music designs, sound events are “automatically” scheduled, or driven, by some formal rules (either a deterministic or indeterministic process), which shape the musical flow in a domain entirely independent of – and fundamentally (in)different to – the medium of sound (be it understood as signal or as a physical and perceptual phenomenon). In our design, [...] we develop larger musical articulations out of the material acoustical environment and [...] situated acoustical events.
>
> ***(Di Scipio and Sanfilippo 2019)***

In RITI, control signal processing units (CSPUs) rely neither purely on audio feature extraction from audio signals nor use automated controls. Instead, they act as nonlinear controllers meant to project the system's dynamics over larger temporal and gestural frames, enabling complex dynamics spanning across the network [5]. Hence, the CSGs network operates as a fully interconnected, large-scale chaotic system, where control signals are intertwined with audio signals.

3.1 CONTROL SIGNALS

The CSPUs in RITI are composed of two main elements: the trigger section and the sampling section. The triggers generate a low-frequency stream of single-sample pulses at uneven time intervals, as shown in Figure 5.

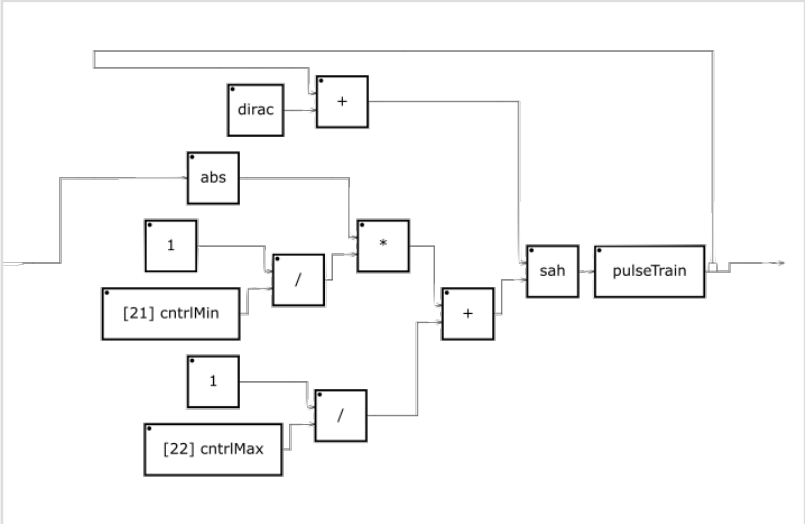


Figure 5. Aperiodic pulse train

These triggers (one for each parameter controlling the CSGs) are activated by a Dirac impulse at run time. The Dirac impulse triggers a sample-and-hold module that receives an external signal, unique to each trigger module. The sample-and-hold output controls the impulse train frequency, and the pulses from the impulse train are fed back into the sample-and-hold to update the train frequency, ensuring that each pulse is output at a different time frame. This creates an aperiodic pulse train mechanism. The input signal to the sample-and-hold is remapped in such as to produce a pulse period in the range of 4 to 20 seconds. The whole process is illustrated in Figure 6, Listing 5 shows a code sample where triggers are started following a noise signal (with a linear congruential generator).

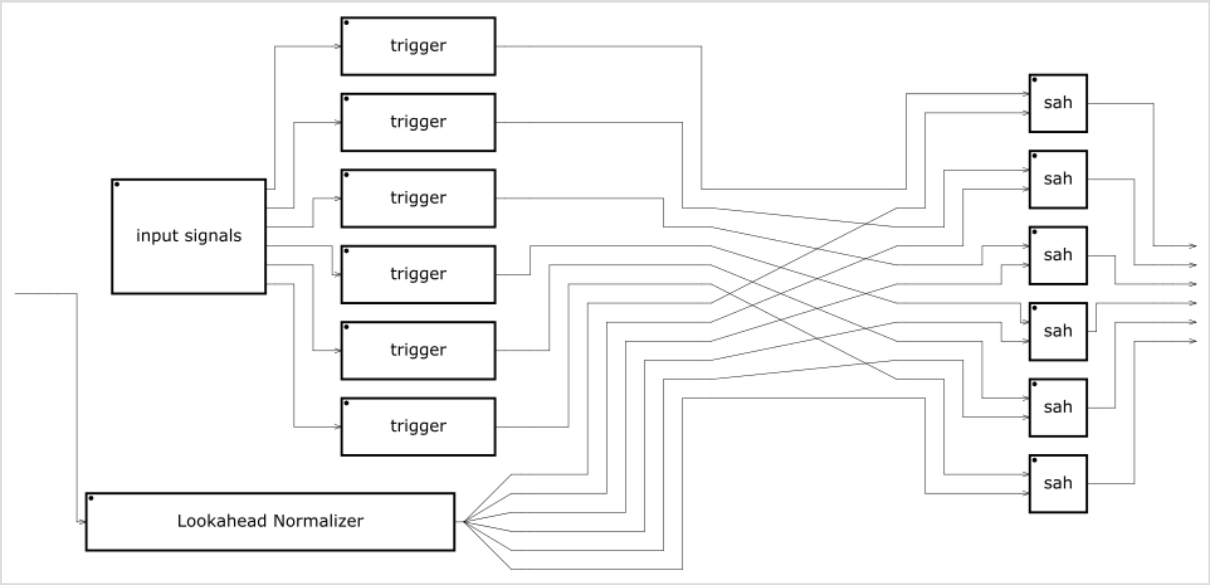


Figure 6. Control signal processing unit

The multiple triggers activate other sample-and-hold modules, each controlling specific CSG parameters. Various methods can be employed to abstract the final six signals (as shown in Figure 6), such as Linear Congruential Generators or other techniques. With this design, an aperiodically sampled sound source (a CSG or a microphone signal) affects the behavior of another CSG. The signals from the network are dynamically normalized with a lookahead limiter to ensure a consistent signal range. For reference, the implementation of the lookahead limiter can be found in (Sanfilippo 2022). Other methods for dynamically normalizing the input signals to the CSP units could be applied.

3.2 LARGE TIME SCALES INTERACTIONS BETWEEN CSG

The coupling between the CSGs can be extended to larger time scales through the use of CSPUs. This allows both the oscillators' interactions and the parameters they control to evolve in time, introducing greater variety in the system's overall behavior. The complete network is shown in Figure 7. Note that extra simple delay lines have been added in the feedback loop (with delay times from $^{n}_{1}$ up to $^{n}_{8}$). This extends to the time scale at which CSG signals fed back into the network, while the control of the CSPUs pathways acting across multiple temporal resolutions — from the second to several minutes — enable the system to achieve a long-term autonomy while remaining sensitive to performer interventions. As a result, sound morphologies emerge, which may retain memory of previous states or behaviors. The FAUST code for the full RITI network is in Listing (6).

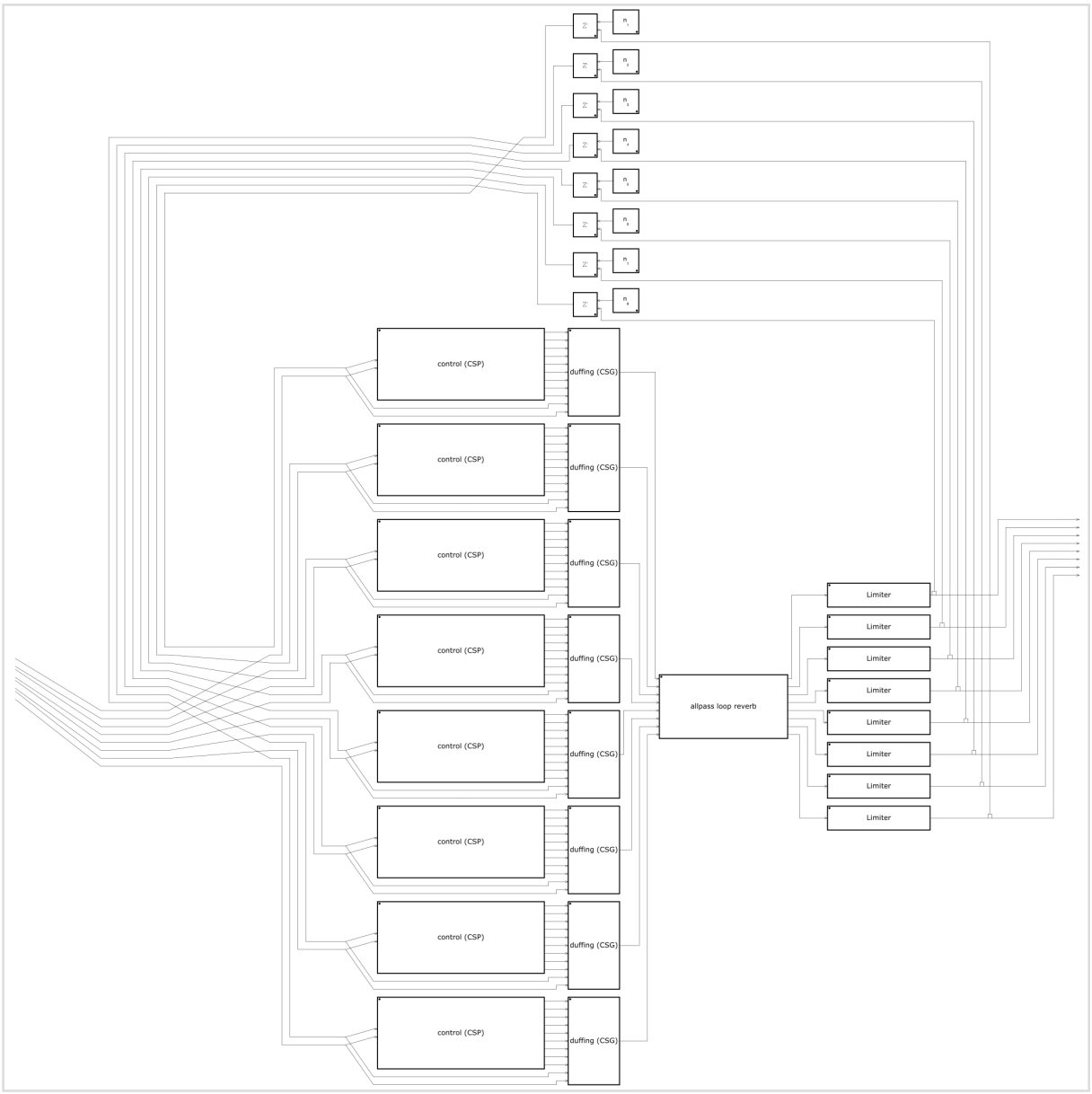


Figure 7. RITI Network.

4. CONCLUSIONS

In continuity with the systemic approaches outlined in the first part of this paper, RITI positions itself within a lineage where musical form emerges from the system dynamics rather than from external structures. This lineage spans from the early cybernetic theoretical models of Roland Kayn and Walter Branchi, through scores as generative systems in the work of John Cage and circuits as scores in the work of Gordon Mumma and David Tudor, to the more recent complex adaptive networks developed by Agostino Di Scipio and Dario Sanfilippo. RITI reexamines the boundaries between instrument, environment, performer, and performance in continuity with this tradition, integrating the study of chaotic dynamics at the

signal-processing level and drawing on contemporary advances in chaos theory and complex adaptive systems studies.

This work was conceived as a preliminary investigation laying the groundwork for developing a compositional methodology and strategy for building complex adaptive systems from scratch in live-electronics music. The project, exploring how chaotic and adaptive behaviors can emerge within self-regulating networks, provided initial insights on structuring future research methodologies and goals.

Future research will formalize the process of defining primitives in creating such systems. This includes designing novel models to obtain emergent behaviors in first-order feedback delay networks (FDNs), comb-like structures and novel chaotic oscillators, and ultimately developing methods for designing complex feedback-delay networks while documenting step by step how emergent behaviors can be obtained. Part of this program will involve a systematic analysis of methodologies developed by other composers in their works. In parallel, further development includes the study and documentation of feature-extraction methods enabling the systems to self-regulate themselves via cybernetic strategies and to allow them to learn and adapt to their environment.

While documenting these strategies and methodologies, new transduction methods beyond loudspeakers and microphones, such as actuators and contact microphones on resonant mechanical materials or other forms of physical systems (such as augmented autonomous instruments via human-machine interaction), will be investigated to extend the performative possibilities of Complex Adaptive Systems. These approaches may open new types of spatial interactions, studies on sound diffusion, and propagation of local systems.

By advancing these ideas, this research contributes to the growing field of complex adaptive systems in music and to understanding chaos, emergent behaviors, and unpredictability not as obstacles to be eliminated, but embraced as fundamental principles of sonic creativity. Beyond technical considerations, RITI invites a reconsideration of what it means to compose within a system, suggesting that composition today could may also involve designing the conditions under which sound systems evolve, adapt, and self-organize in relation to their environments.

Code Listings

(Listing 1) Duffing Oscillator

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

// duffing oscillator - parameters for chaotic regime
duffing = f ~ si.bus(2)
with {
    f(x, y) = x + (y * dt) + initX,
              y + ((-delta * y - alpha * x - beta * x * x * x +
              gamma * cos(omega * time)) * dt) + initY;
    dt = 0.01;
    delta = 0.2;
    alpha = -1.0;
    beta = 1.0;
    gamma = 0.3;
    omega = 1.0;
    time = (+ (dt) ~ _);
    x0 = 0.1;
    y0 = 0.2;
    initX = x0 - mem(x0);
    initY = y0 - mem(y0);
};
process = duffing : par(i, 2, _ * 0.5);
```

(Listing 2) Semplified Duffing Oscillator

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

// single circuit - semplified duffing oscillator
duffing_semplified(dt, damping, forcing, forcing_frequency) =
    ((_, _) <: (_ + _ * dt),
    ((((_, !) <: (_ - (_ * _ * _))) - ((!, _) <: damping * _) +
    (forcing * cos(forcing_frequency * (+ (dt) ~ _)))) * dt + (!, _))) ~
    si.bus(2);

process = (0.01, 0.2, 0.3, 1.0) : duffing_semplified : par(i, 2, _ * 0.5);
```

(Listing 3) Complex Sound Generator

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

// One-pole low-pass
onepole(fc, x) = x : op
with {
    a = exp(-2.0 * ma.PI * fc / ma.SR);
    op = (_ * a + _ * (1 - a)) ~ _;
};

// Bandpass filters bank
bpfiltersbank(N, Program, F, Q, G) = _ <:
    par(i, N, (((FrequenciesTable, 0) : (_, _, _ + i) :
    rdtable) * F, Q, G, _) : bandpassBiquad) :> _
with{
    // Robert Bristow-Johnson's BP Biquad Filter - Direct Form 1
    bandpassBiquad(cf, bw, gf) = biquadFilter * onepole(100, (cf > 0))
    with {
        biquadFilter =
        _ * gf <: _, (mem  <: (_, mem)) : (_ * a0, _ * a1, _ * a2) :> _ :
        ((_, _) :> _) ~ (_ <: (_, mem) : (_ * - b1, _ * - b2) :> _);
        Fx = max(20, min(20000, onepole(10, cf)));
        Qx = max(ma.EPSILON, min(ma.MAX, onepole(10, bw)));
```

```
        K = tan(ma.PI * Fx / ma.SR);
        norm = 1 / (1 + K / Qx + K * K);
        a0 = K / Qx * norm;
        a1 = 0;
        a2 = - a0;
        b1 = 2 * (K * K - 1) * norm;
        b2 = (1 - K / Qx + K * K) * norm;
    };
    // Spectrum - Bandpass filters values
    FrequenciesTable =
    waveform{45.1, 45.8, 46.3, 47.2, 48.6, 49.7, 51.4, 53.2,
          56.8, 59.1, 62.7, 67.3, 71.9, 76.4, 82.1, 89.7,
          97.3, 108.5, 124.8, 147.2, 178.6, 234.1, 312.7, 487.3};
};

// Graphic User Interface
GUI(i) = dampingGUI, forcingGUI, forcingfreqGUI, bpQGUI, bfFshiftGUI,
    bpSmoothGUI, bpGainGUI, bpprogramGUI(i), interactionGUI, _, 0
with{
    modeGUI = checkbox("mode");
    dtVal = 0.01;
    dampingGUI = hslider("Damping", 0.15, 0, 1.0, 0.001) : si.smoo;
    forcingGUI = hslider("Forcing", 0.7, 0, 1.0, 0.001) : si.smoo;
    forcingfreqGUI = hslider("Frequency", 328.0, 0, 400.0, 0.001) : si.smoo;
    bpQGUI = hslider("Bandpass Q", 100, 0.1, 240, 0.001) : si.smoo;
    bfFshiftGUI = 8 ^ hslider("Bandpass F", 0.5, -1, 1.0, 0.001) : si.smoo;
    bpSmoothGUI = hslider("Smooth", 1.0, 0.001, 1.0, 0.001) :
        20 * exp(_ * log(20000/20)) : si.smoo;
    bpGainGUI = hslider("Bandpass G", 0.5, 0.0, 2.0, 0.001) : si.smoo;
    bpprogramGUI(i) = hgroup("BP presets",
        hslider("osc %i[style:knob]", i, 1, 20, 1) - 1);
    interactionGUI = hslider("Interaction", 1, 0, 1, 0.001) : si.smoo;
};

// single circuit - semplified modified duffing oscillator
duffing_riti(damping, forcing, forcing_frequency, bpQ, bpFshift, bpSmooth,
    bpGain, bpProgram, interaction, influence, external_Mic) =
    ((_, _) <: (_ + _ :
        (atan : bpfiltersbank(24, bpProgram, bpFshift, bpQ, bpGain) :
        fi.dcblocker : onepole(bpSmooth))),
    ((((_, !) <: (_ - (_ * _ * _))) - ((!, _) <:
        ma.tanh(damping + influence * interaction) * _) +
        (external_Mic + forcing * cos((+ (forcing_frequency / ma.SR) ~ _)))) +
        (!, _))) ~ si.bus(2) : (_, !);

process = GUI(1) : duffing_riti * 0.5 : ma.tanh <: si.bus(2);
```

(Listing 4) CSGs coupling

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

// One-pole low-pass
onepole(fc, x) = x : op
with {
    a = exp(-2.0 * ma.PI * fc / ma.SR);
    op = (_ * a + _ * (1 - a)) ~ _;
};

// BP filters bank
bpfiltersbank(N, Program, F, Q, G) = _ <:
    par(i, N, (((FrequenciesTable, int(Program * N)) : (_, _, _ + i) :
    rdtable) * F, Q, G, _) : bandpassBiquad) :> _
with{
    // Robert Bristow-Johnson's BP Biquad Filter - Direct Form 1
    bandpassBiquad(cf, bw, gf) = biquadFilter * onepole(100, (cf > 0))
    with {
        biquadFilter =
        _ * gf <: _, (mem  <: (_, mem)) : (_ * a0, _ * a1, _ * a2) :> _ :
        ((_, _) :> _) ~ (_ <: (_, mem) : (_ * - b1, _ * - b2) :> _);
        Fx = max(20, min(20000, onepole(10, cf)));
        Qx = max(ma.EPSILON, min(ma.MAX, onepole(10, bw)));
```

```
        K = tan(ma.PI * Fx / ma.SR);
        norm = 1 / (1 + K / Qx + K * K);
        a0 = K / Qx * norm;
        a1 = 0;
        a2 = - a0;
        b1 = 2 * (K * K - 1) * norm;
        b2 = (1 - K / Qx + K * K) * norm;
    };
    // Spectrum - Bandpass filters values
    FrequenciesTable =
    waveform{
            // frequencies list 1
            48.41, 61.13, 70.6, 73.0, 73.83, 85.07, 90.48, 92.84, 97.9, 130.8,
            143.62, 149.49, 168.63, 170.25, 195.14, 224.31, 389.0, 461.01,
            489.24, 502.87, 601.43, 814.15, 1011.82, 1220.88,
            // frequencies list 2
            48.41, 61.13, 70.6, 73.0, 73.83, 85.07, 90.48, 92.84, 97.9, 130.8,
            143.62, 149.49, 168.63, 170.25, 195.14, 224.31, 389.0, 461.01,
            489.24, 502.87, 601.43, 814.15, 1011.82, 1220.88,
            // frequencies list 3
            51.48, 55.04, 83.62, 98.87, 131.74, 133.77, 142.9, 143.41, 148.7,
            200.26, 203.26, 204.13, 206.54, 268.43, 331.22, 373.86, 379.07,
            395.1, 421.23, 448.44, 490.65, 497.53, 669.08, 933.05,
            // frequencies list 4
            47.36, 50.19, 62.04, 62.54, 81.86, 89.89, 91.5, 102.59, 117.05,
            136.7, 232.73, 268.47, 335.18, 339.65, 371.59, 397.33, 430.64,
            431.43, 465.69, 613.4, 643.34, 836.52, 905.66, 976.14,
            // frequencies list 5
            50.17, 52.94, 53.94, 61.12, 70.98, 75.72, 80.8, 101.67, 104.57,
            104.72, 107.42, 158.96, 159.76, 160.2, 198.74, 249.11, 260.17,
            265.98, 341.42, 413.59, 423.51, 597.14, 827.88, 1315.59,
            // frequencies list 6
            49.56, 50.35, 52.82, 55.13, 59.17, 71.29, 71.46, 72.5, 74.35,
            86.22, 105.52, 113.8, 220.89, 275.22, 276.94, 295.09, 309.73,
            310.24, 394.56, 728.6, 753.56, 1009.26, 1071.31, 1257.69,
            // frequencies list 7
            45.98, 47.52, 49.98, 57.92, 73.67, 87.08, 89.61, 98.17, 122.49,
            135.51, 157.74, 216.09, 237.32, 324.15, 330.72, 331.4, 460.23,
            482.13, 502.96, 542.09, 551.01, 644.91, 1159.58, 1263.72,
            // frequencies list 8
            48.87, 54.63, 55.47, 58.58, 61.58, 65.86, 80.4, 81.3, 103.35,
            115.19, 120.35, 157.47, 172.0, 186.76, 211.46, 246.77, 249.78,
            268.61, 270.8, 303.7, 668.95, 857.52, 1056.37, 1411.06,
            // frequencies list 9
            45.97, 47.55, 64.99, 66.53, 85.83, 90.07, 105.61, 120.06, 131.51,
            131.63, 134.05, 145.82, 157.72, 201.9, 217.95, 243.22, 260.69,
            263.83, 275.88, 278.09, 313.41, 425.92, 469.0, 1166.48,
            // frequencies list 10
            45.27, 46.06, 50.7, 59.53, 60.09, 60.34, 63.26, 67.56, 72.45,
            82.1, 88.75, 124.03, 175.67, 183.77, 213.87, 238.32, 261.5,
            272.36, 298.43, 314.68, 368.84, 520.22, 579.01, 1034.62
    };
};

// Graphic User Interface
GUI(i) = dampingGUI, forcingGUI, forcingfreqGUI, bpQGUI, bfFshiftGUI,
    bpSmoothGUI, bpGainGUI, bpprogramGUI(i) + start(i), interactionGUI, _, 0
with{
    modeGUI = checkbox("mode");
    dtVal = 0.01;
    dampingGUI = hslider("Damping", 0.46, 0, 1.0, 0.001) : si.smoo;
    forcingGUI = hslider("Forcing", 0.95, 0, 1.0, 0.001) : si.smoo;
    forcingfreqGUI = hslider("Frequency", 314.0, 0, 400.0, 0.001) : si.smoo;
    bpQGUI = hslider("Bandpass Q", 17, 0.1, 240, 0.001) : si.smoo;
    bfFshiftGUI = 8 ^ hslider("Bandpass F", 0.1, -1.0, 1.0, 0.001) : si.smoo;
    bpSmoothGUI = hslider("Smooth", 1.0, 0.001, 1.0, 0.001) :
        20 * exp(_ * log(20000/20)) : si.smoo;
    bpGainGUI = hslider("Bandpass G", 0.6, 0.0, 2.0, 0.001) : si.smoo;
    bpprogramGUI(i) = hgroup("BP presets", hslider("osc %i[style:knob]",
        1, 1, 8, 1) - 1);
    start(i) = ((i - 1) * (1 - 1 @ (ma.SR * 0.1)));
    interactionGUI = hslider("Interaction", 1.0, 0, 1, 0.001) : si.smoo;
};
```

```
// single circuit - semplified modified duffing oscillator
duffing_riti(damping, forcing, forcing_frequency, bpQ, bpFshift, bpSmooth,
    bpGain, bpProgram, interaction, influence, external_Mic) =
    ((_, _) <: (_ + _ :
        (atan : bpfiltersbank(24, bpProgram, bpFshift, bpQ, bpGain) :
        fi.dcblocker : onepole(bpSmooth))),
    ((((_, !) <: (_ - (_ * _ * _))) - ((!, _) <:
        ((ma.tanh(damping + influence * interaction) + 1) * 0.5) * _) +
        (external_Mic + forcing * cos((+ (forcing_frequency / ma.SR) ~ _)))) +
        (!, _))) ~ si.bus(2) : (_, !);

// Number of CSGs in the FDN
network_oscillators(N) = par(i, N, (GUI(i + 1)) : duffing_riti) ~
    (par(i, N, _ @ 1999) : ro.crossNM(N - 1, 1));

process = network_oscillators(8) :> par(i, 2, _ * 0.125 : ma.tanh);
};
```

(Listing 5) CSPUs triggers

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

noise(N, initSeed) = ((_ + (initSeed - initSeed') :
    seqN(N, (_ * a + c) % m )) ~ _ : par(i, N, _ / m))
    with{
        // Linear Congruential Generator - constant values
        a = 18446744073709551557; c = 12345; m = 2 ^ 31;
        // Sequential operations
        seqN(N, OP) = _ <: seq(i, N, (OP <: _, _), si.bus(i + 1)) :
            (_, !, si.bus(N - 1), !);
    };

trigger(N, secMax, secMin) = par(i, N, (_ + dirac, abs * (1 / secMin) +
    (1 / secMax) : sah : pulseTrain) ~ _)
with{
    dirac = 1 - 1';
    selector(sel, x, y) = x * (1 - sel) + y * (sel);
    sah(t, x) = selector(t, _, x) ~ _;
    phasor0(f) = (_ <: _ + f, _) ~  _ % ma.SR : (!, _ / ma.SR);
    derivate(x) = x < x';
    pulseTrain(f) = phasor0(f) : derivate;
};
process = noise(2, 4321) : trigger(2, 1.0, 0.1);
```

(Listing 6) RITI Network

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

// RITI Network
network(N, global_damping, global_forcing, global_forcing_freq, global_q,
    global_freq_shift, global_smooth, global_gain, global_interaction,
    global_rev) = si.bus(2) <:
        (ro.interleave(N, 2) : par(i, N, si.bus(2) <:
        ((si.bus(2) : controlMapping(i + 1, global_damping, global_forcing,
        global_forcing_freq, global_q, global_freq_shift, global_smooth,
        global_gain, global_interaction)), si.bus(2)) : duffing_riti) :
        reverb(N, N, 8, 2, 5, 10, global_rev) :
        par(i, N, LookaheadLimiter(0.98))) ~
        (par(i, N, _ @ 1999) : ro.crossNM(N - 1, 1));
```

```
// Output
process = (globalNetworkGUI, ins(2)) : network(8) : outs(8, 2)
with{
    // Mixer
    ins(N) = vgroup("Mixer", hgroup("Channels",
        hgroup("Master", hgroup("inputs",
        par(i, N, * (vslider("G", -60, -60, 20, 0.001) :
        ba.db2linear : si.smoo))))));
    outs(N, O) = vgroup("Mixer", hgroup("Channels",
        hgroup("Master", hgroup("outputs",
        par(i, N, * (vslider("G", -60, -60, 20, 0.001) :
        ba.db2linear : si.smoo : _ * (O / N)))))))) :> si.bus(O);
};

// One-pole low-pass
onepole(fc, x) = x : op
with {
    a = exp(-2.0 * ma.PI * fc / ma.SR);
    op = (_ * a + _ * (1 - a)) ~ _;
};

// BP filters bank
bpfiltersbank(N, Program, F, Q, G) = _ <:
    par(i, N, (((FrequenciesTable, int(Program * N)) : (_, _, _ + i) :
    rdtable) * F, Q, G, _) : bandpassBiquad) :> _
with{
    // Robert Bristow-Johnson's BP Biquad Filter - Direct Form 1
    bandpassBiquad(cf, bw, gf) = biquadFilter * onepole(100, (cf > 0))
    with {
        biquadFilter =
        _ * gf <: _, (mem  <: (_, mem)) : (_ * a0, _ * a1, _ * a2) :> _ :
        ((_, _) :> _) ~ (_ <: (_, mem) : (_ * - b1, _ * - b2) :> _);
        Fx = max(20, min(20000, onepole(10, cf)));
        Qx = max(ma.EPSILON, min(ma.MAX, onepole(10, bw)));
        K = tan(ma.PI * Fx / ma.SR);
        norm = 1 / (1 + K / Qx + K * K);
        a0 = K / Qx * norm;
        a1 = 0;
        a2 = - a0;
        b1 = 2 * (K * K - 1) * norm;
        b2 = (1 - K / Qx + K * K) * norm;
    };
    // Spectrum - BP filters values
    FrequenciesTable =
    waveform{
            // frequencies list 1
            48.41, 61.13, 70.6, 73.0, 73.83, 85.07, 90.48, 92.84, 97.9, 130.8,
            143.62, 149.49, 168.63, 170.25, 195.14, 224.31, 389.0, 461.01,
            489.24, 502.87, 601.43, 814.15, 1011.82, 1220.88,
            // frequencies list 2
            48.41, 61.13, 70.6, 73.0, 73.83, 85.07, 90.48, 92.84, 97.9, 130.8,
            143.62, 149.49, 168.63, 170.25, 195.14, 224.31, 389.0, 461.01,
            489.24, 502.87, 601.43, 814.15, 1011.82, 1220.88,
            // frequencies list 3
            51.48, 55.04, 83.62, 98.87, 131.74, 133.77, 142.9, 143.41, 148.7,
            200.26, 203.26, 204.13, 206.54, 268.43, 331.22, 373.86, 379.07,
            395.1, 421.23, 448.44, 490.65, 497.53, 669.08, 933.05,
            // frequencies list 4
            47.36, 50.19, 62.04, 62.54, 81.86, 89.89, 91.5, 102.59, 117.05,
            136.7, 232.73, 268.47, 335.18, 339.65, 371.59, 397.33, 430.64,
            431.43, 465.69, 613.4, 643.34, 836.52, 905.66, 976.14,
            // frequencies list 5
            50.17, 52.94, 53.94, 61.12, 70.98, 75.72, 80.8, 101.67, 104.57,
            104.72, 107.42, 158.96, 159.76, 160.2, 198.74, 249.11, 260.17,
            265.98, 341.42, 413.59, 423.51, 597.14, 827.88, 1315.59,
            // frequencies list 6
            49.56, 50.35, 52.82, 55.13, 59.17, 71.29, 71.46, 72.5, 74.35,
            86.22, 105.52, 113.8, 220.89, 275.22, 276.94, 295.09, 309.73,
            310.24, 394.56, 728.6, 753.56, 1009.26, 1071.31, 1257.69,
```

```
            // frequencies list 7
            45.98, 47.52, 49.98, 57.92, 73.67, 87.08, 89.61, 98.17, 122.49,
            135.51, 157.74, 216.09, 237.32, 324.15, 330.72, 331.4, 460.23,
            482.13, 502.96, 542.09, 551.01, 644.91, 1159.58, 1263.72,
            // frequencies list 8
            48.87, 54.63, 55.47, 58.58, 61.58, 65.86, 80.4, 81.3, 103.35,
            115.19, 120.35, 157.47, 172.0, 186.76, 211.46, 246.77, 249.78,
            268.61, 270.8, 303.7, 668.95, 857.52, 1056.37, 1411.06,
            // frequencies list 9
            45.97, 47.55, 64.99, 66.53, 85.83, 90.07, 105.61, 120.06, 131.51,
            131.63, 134.05, 145.82, 157.72, 201.9, 217.95, 243.22, 260.69,
            263.83, 275.88, 278.09, 313.41, 425.92, 469.0, 1166.48,
            // frequencies list 10
            45.27, 46.06, 50.7, 59.53, 60.09, 60.34, 63.26, 67.56, 72.45,
            82.1, 88.75, 124.03, 175.67, 183.77, 213.87, 238.32, 261.5,
            272.36, 298.43, 314.68, 368.84, 520.22, 579.01, 1034.62,
            // frequencies list 11
            62.43, 64.06, 67.06, 78.4, 87.19, 92.78, 185.09, 196.33, 197.02,
            226.49, 235.84, 282.56, 295.08, 296.38, 429.52, 437.19, 475.06,
            729.63, 867.91, 932.07, 1031.89, 1128.47, 1522.02, 1546.86,
            // frequencies list 12
            45.21, 52.65, 55.29, 81.21, 88.29, 95.03, 105.06, 108.56, 126.48,
            140.18, 152.01, 153.35, 168.32, 179.7, 194.74, 221.72, 243.62,
            254.15, 267.42, 414.09, 488.11, 489.48, 533.19, 912.04,
            // frequencies list 13
            47.67, 56.08, 58.38, 62.92, 86.91, 96.11, 100.4, 127.59, 140.74,
            152.79, 156.03, 192.88, 193.81, 196.56, 198.66, 262.26, 268.02,
            268.21, 294.43, 303.14, 314.92, 332.51, 466.5, 609.44,
            // frequencies list 14
            45.71, 46.22, 51.56, 56.77, 57.75, 61.54, 82.34, 95.6, 101.1,
            108.56, 137.48, 160.43, 181.05, 187.95, 266.74, 271.39, 488.15,
            557.13, 557.97, 686.48, 773.99, 943.23, 1160.16, 1161.38,
            // frequencies list 15
            45.48, 45.6, 54.44, 58.7, 67.54, 76.74, 88.0, 88.52, 100.89,
            107.66, 112.15, 115.96, 128.52, 177.48, 230.9, 253.19, 267.84,
            293.82, 324.99, 375.01, 416.58, 433.57, 888.01, 1201.83,
            // frequencies list 16
            46.8, 49.37, 49.58, 65.82, 95.22, 117.91, 144.74, 150.64, 202.23,
            230.03, 234.88, 244.03, 294.1, 413.96, 462.41, 528.64, 533.67,
            551.69, 643.0, 643.23, 941.46, 980.19, 1030.22, 1276.16,
            // frequencies list 17
            49.74, 67.17, 67.43, 87.94, 123.9, 129.63, 152.83, 221.31, 230.97,
            273.32, 281.47, 290.65, 293.16, 306.21, 331.87, 335.37, 366.26,
            379.86, 397.83, 429.28, 453.89, 591.65, 717.56, 982.65,
            // frequencies list 18
            52.76, 52.82, 74.47, 88.0, 94.0, 94.29, 115.78, 143.88, 162.75,
            202.96, 203.07, 211.57, 284.24, 306.57, 333.21, 361.2, 384.75,
            410.39, 520.58, 573.72, 600.5, 654.72, 681.8, 801.3,
            // frequencies list 19
            47.87, 49.44, 53.93, 73.81, 77.48, 123.13, 125.01, 139.86, 159.57,
            159.65, 188.04, 189.91, 224.36, 234.72, 280.01, 286.0, 304.4,
            320.65, 325.9, 434.58, 527.12, 760.98, 794.01, 837.07,
            // frequencies list 20
            45.11, 46.26, 48.53, 50.3, 58.21, 61.07, 63.17, 80.85, 88.58,
            91.97, 135.53, 150.17, 157.38, 189.82, 194.84, 224.51, 251.75,
            275.28, 671.18, 703.84, 995.79, 1273.73, 1350.26, 1556.95
    };
};

// Peak Envelope and Peak Envelope Att Rel
peakenvelope(period, x) = loop ~ _
    with {
        loop(y) = max(abs(x), y * coeff);
        twoPIforT = (2.0 * ma.PI) * (1.0 / ma.SR);
        coeff = exp(-twoPIforT / max(ma.EPSILON, period));
    };

peakEnvAttRel(att, rel, x) = peakenvelope(rel - att, x) :
    onepole(1.0 / max(ma.EPSILON, att));
```

```
// Peakholder
peakHolder(t, x) = loop ~ si.bus(2) : ! , _
    with {
        loop(timerState, outState) = timer , output
            with {
                isNewPeak = abs(x) >= outState;
                isTimeOut = timerState >= rint(t * ma.SR);
                bypass = isNewPeak | isTimeOut;
                timer = (1.0 - bypass) * (timerState + 1);
                output = bypass * (abs(x) - outState) + outState;
            };
    };

peakHoldCascade(N, holdTime, x) = x : seq(i, N, peakHolder(holdTime / N));

// Cascaded onepole smoothers with attack and release times
smoother(N, att, rel, x) = loop ~ _
    with {
        loop(fb) = coeff * fb + (1.0 - coeff) * x
            with {
                cutoffCorrection = 1.0 / sqrt(pow(2.0, 1.0 / N) - 1.0);
                coeff = ba.if(x > fb, attCoeff, relCoeff);
                TWOPITC = (2.0 * ma.PI * (1 / ma.SR)) * cutoffCorrection;
                attCoeff = exp(-TWOPITC / att);
                relCoeff = exp(-TWOPITC / rel);
            };
    };
smootherCascade(N, att, rel, x) = x : seq(i, N, smoother(N, att, rel));

// Lookahead limiter
gainAttenuation(th, att, hold, rel, x) =
    th / (max(th, peakHoldCascade(8, att + hold, x)) :
        smootherCascade(4, att, rel));
LookaheadLimiter(th, x_) = xDelayed * gainAttenuation(th, .01, .1, .1, abs(x_))
    with {
        delay = rint((.01 / 8) * ma.SR) * 8;
        xDelayed = x_ @ delay;
    };
LookaheadNormalizer(threshold, holdSec, decaySec) = _ <: 1 /
    max(ma.MIN, (_ : peakHolder(holdSec) :
        onepole(1 / holdSec) : peakenvelope(decaySec))
    ) * _ @ (holdSec * ma.SR);

// RITI GUI
globalNetworkGUI = vgroup("Mixer", hgroup("Global Control",
    (hslider("[1] damping [style:knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
    (hslider("[2] forcing [style:knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
    (hslider("[3] forcing freq [style:knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
    (hslider("[4] q [style:knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
    (hslider("[5] freq shift [style:knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
    (hslider("[6] smooth [style:knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
    (hslider("[7] gain [style:knob]", 1, -1, 1, 0.001) * 4 : si.smoo),
    (hslider("[9] interaction [style:knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
    (hslider("[10] reverberation [style:knob]", 0, 0, 1, 0.001) : si.smoo)
));
// Control Signals based on Input Signal (SAH)
ControlSignalPath(N, seed, secMin, secMax, secMaxSmooth) = si.bus(2) :> _ :
    ((noise(N, seed) :
    par(i, N, randomtrigger)), (LookaheadNormalizer(1.0, 0.1, 10) <: si.bus(N))
    ) : ro.interleave(N, 2) : par(i, N, sah : si.smoo)
with{
    selector(sel, x, y) = x * (1 - sel) + y * (sel);
    sah(t, x) = selector(t, _, x) ~ _;
    // Noise - Linear Congruential Generator - Multiple Outputs
    noise(N, initSeed) = ((_ + (initSeed - initSeed') :
    seqN(N, (_ * a + c) % m )) ~ _ : par(i, N, _ / m))
    with{
        a = 1844674407370955157; c = 12345; m = 2 ^ 31;
    };
```

```
    // Sequential operations
    seqN(N, OP) = _ <: seq(i, N, (OP <: _, _), si.bus(i + 1)) :
    (_, !, si.bus(N - 1), !);
    phasor0(f) = (_ <: _ + f, _) ~  _ % ma.SR : (!, _ / ma.SR);
    derivate(x) = x < x';
    pulseTrain(f) = phasor0(f) : derivate;
    dirac = 1 - 1';
    randomtrigger = (_ + dirac, abs * (1 / secMin) +
        (1 / secMax) : sah : pulseTrain) ~ _;
};
// Mapping control signals and GUI
controlMapping(i, global_damping, global_forcing, global_forcing_freq, global_q,
    global_freq_shift, global_smooth, global_gain, global_interaction) =
hgroup("voice %i",
    // control signal generation
    (ControlSignalPath(9, 4321 * (i + 1), 10, 40, 60) :
    (si.bus(7), (_ * 0), _)),
    // local GUI control for each voice
    hgroup("individual control",
        (vslider("[1] damping [style: knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
        (vslider("[2] forcing [style: knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
        (vslider("[3] forcing F [style: knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
        (vslider("[4] Q [style: knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
        (vslider("[5] freq shift [style: knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
        (vslider("[6] smooth [style: knob]", 0, -1, 1, 0.001) * 4 : si.smoo),
        1,
        (vslider("[8] preset [style:knob]", i, 1, 20, 1) - 1),
        (vslider("[9] interact [style: knob]", 0, -1, 1, 0.001) * 4 : si.smoo)
    ) :>
    // receive GUI global control for voices
    (
        (_ + global_damping),
        (_ + global_forcing),
        (_ + global_forcing_freq),
        (_ + global_q),
        (_ + global_freq_shift),
        (_ + global_smooth),
        (_ + global_gain),
        (_ + 0),
        (_ + global_interaction)
    ) :
    // compress the signals into a bipolar activation function
    // conver bipolar signals to unipolar signals
    (par(k, 7, ma.tanh : bitouni), _, (ma.tanh : bitouni)) :
    // mapping for the signals
    hgroup("inspectors",
        hgroup("damping", map(i, 0, 1)),
        hgroup("forcing", map(i, 0.6, 1)),
        hgroup("forcing freq", map(i, 1, 400)),
        hgroup("bp Q", map(i, 0.1, 240)),
        8 ^ hgroup("bp Fshift", map(i, -1, 1)),
        hgroup("bp smooth", map(i, 1, 20000)),
        hgroup("gain", map(i, 0, 1)),
        _,
        hgroup("interaction", map(i, 0, 1))
    )
) :
(si.bus(6), (_ : mixerVoiceG(i)), si.bus(2))
with{
    // mapping function
    map(i, minL, maxL) = _ * (maxL - minL) : _ + minL <:
        _, vbargraph("%i [style:numerical]", minL, maxL) : attach;
    // bipolar to unipolar function
    bitouni = (_ + 1) / 2;
    // Mixer
    mixerVoiceG(i) =
        vgroup("Mixer", hgroup("Channels", vgroup("%i", vol)))
    with{
        vol = * (vslider("CSG [midi:ctrl %i]", 0.001, 0.001, 2.0, 0.001));
    };
};
```

```
// Reverb (Keith Barr Allpass Loop)
reverb(I, O, N, SEQ, parspace, start, KRT) = si.bus(I) <:
(InputStage :
    (ro.interleave(N, N/(N/2)) : par(i, N, si.bus(2) :> _) :
        par(i, N,
        seq(k, SEQ,
            // cleanfunc
            ModAPF(
                (primenumbers((k + start) + (i * parspace)) : MS2T),
                (primenumbers((k + start) + (i * parspace)) / 100 : MS2T),
                (0.5 / primenumbers((k + start) + (i * parspace)) : MS2T)
            )
        ) :
        _ * KRT
    )   : ro.crossNM(N - 1, 1)) ~
    si.bus(N) :>
OutputStage), OutGain :> par(i, O, _)
with{
    // In / Out - Network
    InputStage = par(i, N, _);
    OutputStage = par(i, O, _);
    OutGain = par(i, O, _ * (1 - KRT));
    // conversion : milliseconds T to samples
    MS2T(t) = (t / 1000) * ma.SR;
    // index of the primes numbers
    primenumbers(index) = ba.take(index , list)
    with{
        list =  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, 313, 317, 331,
                337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401,
                409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467,
                479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557;
    };
    // function for test prime numbers
    cleanfunc(x, y, z) = _ * (x + y + z) ;
    // AP Coefficents
    COEFF = .65;
    // Modulated Allpass filter
    ModAPF(delsamples, samplesmod, freqmod) = ( + : _ <:
        delayMod(delsamples, samplesmod, freqmod),
        * (COEFF))~ * (-COEFF) : mem, _ : + : _
        with{
            delayMod(samples, samplesMod, freqMod, x) = delay
            with{
                modulation(f, samples) = ((os.osc(f) + 1) / 2) * samples;
                delay = x : de.fdelay(samples,
                                samples - modulation(freqMod, samplesMod));
            };
        };
};
```

References

- Branchi, Walter. 2023. “Una musica per ascoltare il mondo: pensieri e riflessioni sulla musica della complessità, scritti (2018-2022), più sette colloqui immaginari con Thalia, colei che è festiva“. Rome: Fondazione Isabella Scelsi.
- Bristow-Johnson, Robert, “Cookbook formulae for audio equalizer biquad filter coefficients” Web resource, Available at: https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html. Accessed September 14.
- Choi, Insook. 1994a. “Interactive Exploration of a Chaotic Oscillator for Generating Musical Signals in Real-time Concert Performance.” Journal of the Franklin Institute 331 (6): 785–818.
- Choi, Insook. 1994b. “Sound synthesis and composition applying time scaling to observing chaotic systems.” In Proceedings of the Second International Conference on Auditory Display, Santa Fe, NM, 79–107.
- Degazio, Bruno. 1993. “Towards a chaotic musical instrument.” In Proceedings of the International Computer Music Conference (ICMC).
- Di Scipio, Agostino. 1990. “Composition by exploration of non-linear dynamic systems.” In Proceedings of the International Computer Music Conference, 324–327.
- Di Scipio, Agostino. 2001. “Iterated Nonlinear Functions as a Sound-Generating Engine.” Leonardo 34 (3): 249–254.
- Di Scipio, Agostino. 2003. “‘Sound is the interface’: From interactive to ecosystemic signal processing.” Organised Sound 8 (3): 269–277.
- Di Scipio, Agostino, and Sanfilippo, Dario. 2019. “Defining Ecosystemic Agency in Live Performance. The Machine Milieu Project as Practice-Based Research.” array. the journal of the ICMA: 28–43.
- Duffing, Georg. 1918. “Erzwungene Schwingungen bei veränderlicher Eigenfrequenz und ihre technische Bedeutung“. Braunschweig: F. Vieweg & Sohn.
- Gleick, James. 1987. “Chaos: Making a New Science“. New York: Viking Penguin.
- GRAME Centre National de Création Musicale, “Faust: Functional audio stream,” Web resource, Available at: https://faust.grame.fr/. Accessed September 14.
- Hernandez, David Roberto. 2017. “Music in the Age of Communication and Control“. PhD diss., University of California, Santa Cruz.
- Magnusson, Thor. 2019. Sonic Writing: Technologies of Material, Symbolic, and Signal Inscriptions. London: Bloomsbury Academic.
- Mumma, Gordon. 1967. “Creative Aspects of Live-Performance Electronic Music Technology“. 33rd Convention of the Audio Engineering Society, October 1967.
- Mudd, Tom. 2019. “Between chaotic synthesis and physical modelling: Instrumentalising with gutter synthesis.” In Proceedings of the 7th Conference on Computation, Communication, Aesthetics & X (xCoAx 2019), 217–228.
- Orlarey, Yves, and Fober Dominique, and Letz Stéphane. 2004. “Syntactical and Semantical Aspects of Faust.” Soft Computing 8 (9): 623–632.
- Perloff, Nancy. 2001. “The Art of David Tudor: Audio and Video – Variations II (1961)“. Getty Research Institute. Updated 2010. Accessed September 14, 2025. https://www.getty.edu/research/tools/guides_bibliographies/david_tudor/av/variations.html.

- Pirrò, David. 2017. Composing Interactions. PhD diss., University of Music and Performing Arts Graz, Institute of Electronic Music and Acoustics.
- Pizzaleo, Luigi. 2022. "Walter Branchi, Dal segno alla circostanza". Rome: La Bussola.
- Rodet, Xavier, and Christophe Vergez. 1999. "Nonlinear Dynamics in Physical Models: Simple Feedback-Loop Systems and Properties." Computer Music Journal 23 (3): 18–34.
- Sanfilippo, Dario. 2021a. "Constrained differential equations as complex sound generators." In Proceedings of the 18th Sound and Music Computing Conference (SMC 2021), 150–157.
- Sanfilippo, Dario. 2021b. "Time-Domain Adaptive Algorithms for Low- and High-Level Audio Information Processing." Computer Music Journal 45 (1): 24–38.
- Sanfilippo, Dario. 2022. "Envelope following via cascaded exponential smoothers for low-distortion peak limiting and maximisation." In Proceedings of the 3rd International Faust Conference (SMC/IFC-22), 167.
- Smith III, Julius O. 2007. Introduction to Digital Filters: With Audio Applications, W3K Publishing, 2007, Available at: https://ccrma.stanford.edu/~jos/filters/. Accessed September 14.
- Truax, Barry. 1990. "Chaotic Non-linear Systems and Digital Synthesis: An Exploratory Study." In Proceedings of the International Computer Music Conference (ICMC 1990), Glasgow, Scotland, 50–56.
- Yadegari, Shahrokh. 2003. "Chaotic signal synthesis with real-time control: solving differential equations in PD, MAX/MSP, and JMAX." In Proceedings of the 6th International Conference on Digital Audio Effects (DAFx-03), 1–5.