PLISP: A PlayLISt Processing Language

Austin Franklin
Mälardalen University
austin.franklin@mdu.se

Rikard Lindell
Dalarna University
rli@du.se

ABSTRACT

This paper introduces a text-based, domain-specific language (DSL) for audio sample manipulation and playlist-based granular synthesis, embedded within the Cycling '74 Max `v8ui` *environment using the mgraphics system, supporting real-time control and live coding. The interface enables command-line interaction with large-scale local audio corpora organised on a 2D self-organising map (SOM). It employs LISP-inspired prefix syntax and nested expressions, enabling performers to navigate and manipulate audio material through gesture-based control. The system emphasises modularity and customisation, enabling each musical gesture to be independently timed, chained, or sequenced, whilst supporting interaction models based on path-finding, spectral feature and metadata attribute filtering, and stochastic and probabilistic variation. This paper outlines the system's architecture and influences, describes its core syntax and behaviors, and discusses implications for future developments in live coding design.*

1. INTRODUCTION

Navigating large-scale local audio repositories—some containing over 50,000 files—poses a serious challenge for composers and sound artists working with sample-based media. Conventional folder hierarchies and filename-based search often proves insufficient for meaningful interaction with such massive collections. This paper introduces *PLISP* (short for PlayLISt Processing), a text-based DSL designed to make this process expressive and fast.

At the heart of *PLISP* is the *playlist*—a mutable, compositional object. Playlists are containers for curated subsets of samples from the SOM, enabling performers to structure sound material beyond individual trigger events. Through a LISP-inspired syntax that blends declarative expressions with first-class functions, users can create, modify, persist, and load playlists as part of an exploratory workflow for real-time sonic manipulation. *PLISP*'s syntax facilitates the definition of custom functions and complex expressions, enabling users to extend the DSLs capabilities beyond its built-in commands to tailor interaction patterns for creative coding workflows. The granular synthesis engine, implemented in Gen~, is driven exclusively by the contents of the active playlist, providing a flexible sound generation mechanism that emphasises texture, timbral variation, and formal development.

Copyright: ©2025 First Author et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License 3.0 Unported, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

As part of its core functionality, *PLISP* incorporates filtering and selection subsystems that allow precise querying based on metadata attributes such as duration, directory, file extension, file creation date, and spectral features. These filtering capabilities, in conjunction with selection algorithms grounded in spatial similarity and stochastic processes, shape how sonic material is selected, organized, and transformed during music performance. The inclusion of a comprehensive logging system enhances transparency and reproducibility of performance, enabling on-the-fly inspection of metadata, playlist contents, and audio parameters with a read–eval–print loop (REPL) command-line.

The interface includes the SOM visual display and the REPL (Figure 1). During live coding, users enter commands that control a selector (red dot) on the display. Once a playlist is loaded, all currently loaded samples glow yellow and persist as filters are activated. Filtered samples are weighted by density at a given cell location. The brighter the cell, the more samples exist there, meaning many are perceptually similar. All of the selection-based commands operate on the filtered samples, meaning the same set of selection commands can be run numerous times with different filter states and yield vastly different results.

2. BACKGROUND

The design of DSLs for musical performance represents a rich intersection of computer science, music composition, and human-computer interaction. The language outlined in this paper–designed for interaction with a self-organising map (SOM), playlist-based granular synthesis engine, and live performance workflow—builds on decades of experimentation with systems that enable musicians to interact with digital sound in real time. It also reflects a growing interest in the design of modular digital instruments, a field concerned with creating flexible, composable systems for artistic exploration [1].

Historically, one of the earliest formal attempts to develop DSLs for computer music was Max Mathews's Music-N languages, beginning with MUSIC I in 1957 [2]. These early languages were not real-time, but they pioneered concepts such as the unit generator–a building block for synthesis that remains foundational to modern environments like SuperCollider [3], Max [4], and Pure Data [5].

LISP has played a foundational role in the history of computer music, with its influence extending across decades of creative and technical development. Early landmark systems include Common Lisp Music [6], developed by Bill Schottstaedt in the late 1980s, followed by Heinrich Taube's Common Music in 1991 [7], and Roger Dannenberg's Nyquist [8], introduced in 1997. In the mid-2000s, Andrew Sorensen's Impromptu–later reimagined as Extem-

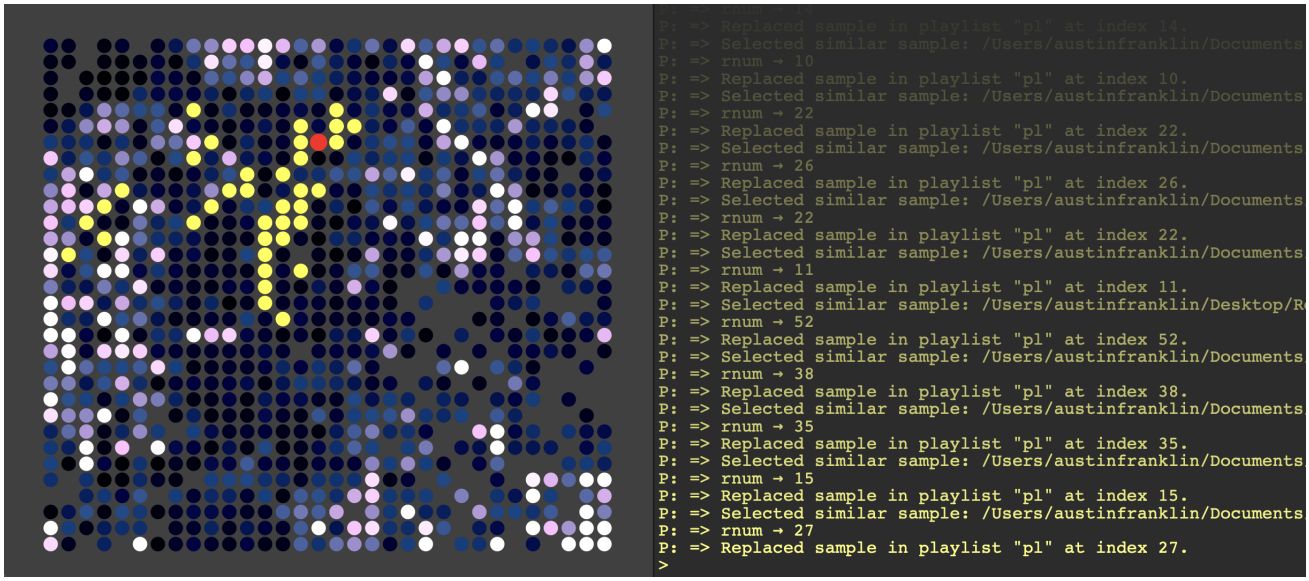

Figure 1. Self-Organising Map and REPL Interface

pore [9]–brought Lisp-style live coding to real-time audio-visual performance. These environments exemplify how various Lisp dialects have been tailored for algorithmic composition, synthesis, and interactive control. The tradition continues today with several Lisp-family languages providing interfaces to platforms like SuperCollider, including Overtone (Clojure), rsc3 (Scheme), and cl-collider (Common Lisp) [10].

Several efforts have explored integrating Lisp with Max through external objects. One such project is MaxLispJ, developed by Brad Garton, which embeds a Common Lisp interpreter (Armed Bear Common Lisp) via the Java runtime, enabling Lisp code execution within Max using the mxj object [11]. An alternative strategy is MOZ'Lib by Julien Vincenot, which connects Max to a separately running Steel Bank Common Lisp (SBCL) process [12]. This method is conceptually similar to how Cycling '74's Node for Max object interfaces with Node.js, allowing extended computations to occur outside of Max's real-time thread, thus avoiding potential execution bottlenecks.

Live coding as a cultural practice has become increasingly prominent, with communities such as TOPLAP advocating for "code as live performance" when writing code becomes a performative act [13]. These systems are frequently used in "algoraves," where audiences watch artists generate dance music live on stage via code. This emphasis on immediacy, feedback, and liveness is central to the design goals of the *PLISP*: providing performers with quick, expressive tools for navigating large databases of audio material. Like ChucK and Orca, *PLISP* prioritises immediacy and mnemonic clarity over abstract generality [14].

PLISP also draws from developments in music information retrieval (MIR) and machine learning (ML) for creative applications. SOMs, first introduced by Teuvo Kohonen in 1982, have been widely used for organising large sound repositories according to perceptual similarity [15]. Projects like SOMeJB have shown how MIR features–such as spectral centroid and onset density–can be used to position sounds in a navigable 2D space [16]. More recently, applications such as AudioStellar allow for music creation with samples visualised using a SOM [17]. In Max specifically, the FluCoMa package provides a set of externals for corpus navigation, and has been used in spatial and virtual corpus exploration projects [18].

The playlist abstraction introduced in *PLISP* adds a crucial layer of modularity and temporal control. Research on modular instrument design–such as Magnusson's concept of Instrumentality–argues that digital instruments are defined not by fixed physical structures, but by configurations of software, data, and interaction [1]. The playlist mechanism can be understood as a modular container: a semantic grouping of sonic material from which musical form can emerge. By loading, saving, manipulating, and recalling playlists as discrete structures, the language provides a clear compositional and performative scaffold.

Granular synthesis, the primary audio engine in this system, has long been associated with textural, emergent sonic structures. Curtis Roads (2001) discusses the "microsonic" scale of granular sound design as fertile ground for algorithmic and generative music. *PLISP*'s control over parameters like grain density, length, and pitch aligns well with research into gesture-based control and mapping strategies in live electronic performance [19].

3. DOCUMENTATION

3.1 System Architecture

PLISP is structured around several interconnected subsystems. At its core lies the interpreter, written in Max's v8 engine, that exposes the Javascript language (ECMAScript 6+) and some Max specific extensions. This parses bracketed expressions into abstract syntax trees and evaluates them within the global environment. The global maintains the overall state and library of commands that support SOM interaction, playlist creation, logging, and audio control, while injecting Max context bindings to enable integration between DSL commands and Max JavaScript APIs. The state engine manages application-wide data such as filter settings, playlists, sample selections, spatial co-

ordinates, and playback parameters to ensure consistent, real-time responsiveness. The execution environment is provided by the Max runtime, handling task scheduling and user interface rendering with the mgraphics system, while the audio engine, powered by Gen~, is responsible for audio synthesis. Together, these subsystems form a cohesive framework that supports responsiveness and expressive control.

3.1.1 Syntax and Execution Model

PLISP employs a simple, uniform syntax that can be easily extended to implement abstractions with minimal extraneous syntax. Every expression is composed of an operator and zero or more operands. Expressions are written in prefix notation, in which operators precede their operands.

```
+ 2 3 ; -> 5
```

Listing 1. Prefix syntax

An operand may be an atom–a constant or variable–or another expression, allowing for nested structures. Nested structures in *PLISP* are written using "[]" rather than LISP's "()" for quicker access on EN-US keyboard layouts. Unlike traditional LISP syntax, the outermost parenthesis are omitted for brevity. Expressions are parsed into nested list structures and evaluated recursively. The evaluation strategy processes function arguments from left to right before invoking the function itself.

```
* 3 [+ 2 3] ; -> 15
```

Listing 2. Nested structure evaluation

Similar to LISP, *PLISP*'s program syntax mirrors the syntax of one of its fundamental data structures–the list (LISP, short for LISt Processing). Programs themselves are represented internally as lists, enabling a seamless integration between data and code. This uniform representation allows functions to be treated as first-class values–meaning functions can be created, passed, and invoked dynamically–facilitating a highly expressive and flexible programming model. This approach aligns with Lisp's philosophy of building complex behavior from simple, well-defined building blocks, without the need for separate syntactic categories or special cases [20].

3.1.2 Special Forms

Function calls are resolved dynamically at runtime. The system supports special forms: `quote`, `lambda`, `def`, `deflib`, `eval`, `let`, `if`, `repeat`, and `for`.

3.1.3 Example Expressions

```
quote [+ 2 3] ; -> ["+", "2", "3"]
'[+ 2 3] ; -> ["+", "2", "3"]
+ 2 3 ; -> 5
```

Listing 3. Quoting

```
def square [lambda [x] [* x x]]
[square 5] ; -> 25
deflib square [lambda [x] [* x x]] ; saves
    function to library
```

Listing 4. Function definition and application

```
let [[a 1] [b 2]] [+ a b] ; -> 3
```

Listing 5. Scoped bindings with `let`

```
if [> 5 3] [1] [0] ; -> 1
```

Listing 6. Conditionals

```
repeat 3 [print tick] ; prints "tick" 3x
for [i 0 5] [print i] ; prints 0 through 4
```

Listing 7. Iteration

3.1.4 Built-in Arithmetic and Logic

The interpreter provides native arithmetic and comparison operations, enabling expressive numeric computation:

`+`	Sum of all arguments
`-`	Subtraction or negation
`*`	Product of all arguments
`/`	Division
`mod`	Modulo
`floor`	Floor
`ceil`	Ceiling
`round`	Rounding
`=`	Equality
`> < >= <=`	Numeric comparisons

```
* 3 [floor 3.9] ; -> 9
```

Listing 8. Operation

3.2 Command Overview

The following commands represent the core building blocks of *PLISP*. These provide direct access to filtering, sample selection, logging, playlist manipulation, audio parameters, and temporal controls.

3.2.1 Filtering

Filter commands restrict which samples are visible or eligible for interaction by setting constraints on metadata and spectral attributes. Each filter is set using its corresponding command with "on" or "off." The `date` and `dir` filters use the currently selected sample on the map for input.

```
dur 10 ; Filter for samples 10 seconds long
date ; Filter for same creation date
dir ; Filter for same directory
ext wav ; Filter for wave files
spec flux 0.8 0.1 ; Use feature "flux",
    value 0.8, tolerance 0.1
```

Listing 9. Filter commands

The `spec` command includes the following 14 features: centroid, spread, skewness, flatness, kurtosis, entropy, crest, flux, slope, rolloff, mean_volume, max_volume, lufs, loudness_range. The filtering is handled via a shell script and `ffmpeg` that combs the local disk, identifies every audio file with specifc codecs (wav, aiff, mp3), analyses them using the aforementioned features, and builds javascript dictionaries that are referenced by `v8`. Filters may be combined to construct compound constraints:

```
[dur on] [dur 30]
[date on] [date]
[ext on] [ext wav]
[spec on] [spec spread 0.5 0.2] ; Displays samples using all filters
```

Listing 10. Aggregating filters

3.2.2 *Sample Selection Functions*

These commands let you choose or traverse samples:

```
cell 3 5 ; Select sample at map position (3, 5)
sel drums 2 ; Select sample index 1 in playlist "drums"
filename 808kick.wav ; Select sample by filename
rndm ; Select random sample from filtered selection
sim 8 ; Selects nearby sample within 8 cells
```

Listing 11. Sample selection commands

3.2.3 *Playlist Management*

Create, edit, and store playlists for compositional reuse:

```
create pl ; Create playlists named "pl"
add pl ; Add selected sample to playlist "pl"
load pl ; Load playlist "pl" into audio engine
addlayer pl ; Add all samples at cell location x, y to playlists "pl" (if exists)
addall pl ; Add all currently filtered samples to playlists "pl"
replace pl 5 ; Replace index 5 in playlist "pl" with selected sample
empty pl ; Remove all entried from playlists "pl"
save pl ; Save playlist "pl" to disk
merge drums bass drumNbass ; Merge two playlists
```

Listing 12. Playlist management commands

3.2.4 *Instantiation*

This is a quick way to hear sound:

```
create pl ; Create playlists named "pl"
rndm ; select random sample from filtered selection
add pl ; Add selected sample to playlist "pl"
load pl ; Load playlist "pl" into audio engine
```

Listing 13. Initialising audio playback

3.2.5 *Audio Playback Control*

These commands modify parameters for loaded playlist:

```
rate 0.8 ; Playback rate for grains (per second)
length 750 ; Length of grains in ms
volume 0.5 ; Amplitude (0-1)
speed 0.5 ; Playback rate of file (pitch ratio)
pan ; Panning width (0-1)
feedback 0.5 ; Feedback amount (0-1)
ahead 250 ; Look ahead time in ms
```

Listing 14. Audio parameters

3.2.6 *Time-Based Scheduling*

These commands schedule delayed or repeated actions:

```
after 5000 '[load drums] ; Load playlist "drums" after 5000 ms
every 10 250 '[sim 5] '[add synths] ; Select similar sample and add it to playlist "synths" 250 times at 10 ms intervals
loop gliss 100 '[speed [count 1 16]] ; Creates a loop named "gliss" that varies grain speed at 100ms intervals
stop gliss ; Stops the loop named "gliss"
```

Listing 15. Temporal controls

3.2.7 *Logging and Querying Information*

The `Info` command provides insight into the system state. The name, dir, dur, date, ext, and spec arguments use the currently selected sample on the map for input.

```
info lib ; Print functions saved to library
info layer ; Print number of samples at current map locations
info name ; Print file name
info dir ; Print file directory
info dur ; Print file duration in seconds
info date ; Print file creation date
info ext ; Print file extension
info spec ; Print spectral feature data
info playlists ; Print all known and saved playlists
info load ; Print currently loaded playlists
info filters ; Print active filter states
```

```
info myPlaylist 2 ; Print spectral and metadata for index 2 in "myPlaylist"
info volume ; Print current volume
info rate ; Print current grain rate
info speed ; Print current grain pitch
info length ; Print current length
info pan ; Print current pan amount
info feedback ; Print current feedback factor
```

Listing 16. Logging with `info`

The system also includes built in documentation with `doc` that can be used for a specific command:

```
doc dur ; -> "dur on|off|int - duration filter in seconds"
doc sim ; -> "sim int - Select nearby sample (distance <= int)"
```

Listing 17. Command help with `doc`

3.3 Error Handling

In a live performance context, error handling is crucial for preserving artistic momentum. *PLISP* implements multiple layers of protection–from parsing and syntax checks in the LISP interpreter to safeguards in visual rendering, sample selection, and audio playback. Rather than throwing fatal errors, it prints messages to the REPL, allowing execution to continue uninterrupted. For example, if a playlist isn't found or a syntax issue occurs, the interpreter logs the error and proceeds. The `info` command complements this by offering real-time insights into the system state, helping performers diagnose and correct issues mid-performance.

Custom functions are designed with defensive checks to ensure that playlists, paths, and indices are valid before manipulation occurs. Additionally, the system employs `try...catch` blocks–particularly around the core evaluation loop–to gracefully handle runtime errors during expression parsing and execution; this ensures that malformed or unexpected user input doesn't crash the interpreter but instead generates informative feedback in the REPL. To maintain speed and responsiveness, especially under high DSP load, the system caches frequently accessed data–such as filtered sample indices–to avoid expensive recomputation. This architecture supports both reliability and fluid improvisation, making the environment resilient even during complex or rapid live interactions.

4. INTEGRATION WITH MAX, V8, AND GEN~

PLISP is designed to fully leverage Max's environment. The language's architecture is deeply embedded within the v8 engine, facilitating live-coded interactions, dynamic data structures, and expressive control logic. This tight coupling between *PLISP* and Max's runtime offers scripting flexibility and real-time audiovisual feedback. However, the DSL does not support the creation, deletion, or direct manipulation of Max objects or patcher structures via the Max JS API; its scope is limited to the set of commands and behaviors explicitly defined within the interpreter and library.

Max's `v8ui` object enables modern ECMAScript 6+ features, allowing for functional programming idioms, closures, and higher-order functions–constructs that map onto *PLISP*'s syntax. Unlike standalone live coding languages, *PLISP* benefits from Max's visual programming context: text-based expressions can interoperate with Max objects, UI components, and signal routing in real time. Additionally, the `dict` object in Max affords the ability to read and write structured data (a dictionary) associated with a name directly in the patcher and `v8ui`.

At the signal level, the Gen~environment is used to implement *PLISP*'s granular synthesis engine. Gen~offers low-latency, sample-accurate digital signal processing (DSP) capabilities within Max, written in a specialised functional language that compiles to efficient machine code. This engine consumes playlists generated through the *PLISP* interpreter and renders them into complex, evolving textures. Because Gen~patches are compiled, they maintain the computational efficiency of C while remaining tightly integrated into the Max patching environment–enabling real-time modulation of synthesis parameters based on the evolving state of the interpreter.

For sample management and storage, the `polybuffer~` object is used asynchronously. Instead of loading an entire playlist–which may contain thousands of samples–into memory all at once, each file is loaded into `polybuffer~` the first time a grain is requested from it. Once loaded, it remains available for reuse until explicitly removed or replaced. This approach significantly reduces CPU overhead while maintaining fast lookup times, addressing the performance issues typically associated with large-scale uses of `polybuffer~`.

By aligning *PLISP*'s textual DSL with Max's event-driven architecture, the system supports a holistic model of interaction that spans multiple timescales: symbolic manipulation of large audio datasets, high-quality audio synthesis, and millisecond-level scheduling. This makes it possible to write composable, reactive code that can alter both high-level structure and low-level sound parameters on-the-fly. Max, in this context, is more than a patching environment: it is the operating system of the instrument.

4.0.1 Execution Entry Point

Text is evaluated via the `anything` handler in Javascript:

```
function anything() {
    const code = arrayfromargs(messagename, arguments).join(" ");
    try {
        const parsed = parseLisp(code);
        const result = evalLisp(parsed, globalEnv);
        if (result !== undefined) print("=> " + result + "\n");
    } catch (e) {
        print("Error: " + e.message + "\n");
    }
}
```

Listing 18. CL text evaluation with anything()

4.0.2 Global Environment and Function Injection

The interpreter initialises a global environment and injects all *PLISP* functions for access:

```
const globalEnv = {};
this.scope = globalEnv;

function injectNativeFunctions(context, env) {
    for (const key in context) {
        if (typeof context[key] === "function" && !(key in env)) {
            env[key] = context[key].bind(context);
        }
    }
}
```

Listing 19. Injecting PLISP commands

4.0.3 Integration with PLISP

By embedding a fully programmable interpreter, *PLISP* enables users to construct custom behaviors for playlist generation, sample filtering, spatial traversal, and real-time performance. This LISP-inspired DSL complements the visual workflow of Max with text-based generative structures, supporting rapid prototyping and personalised musical systems. Users may define reusable commands and variables, iterate over samples, and construct higher-order logic for navigating large-scale sound corpora—entirely from within the REPL. Here are some examples:

```
repeat 50 [create [count 0 49]]
```

Listing 20. Create 50 playlists

```
def rloop [lambda [] '[count 0 [rnum 10]]]
```

Listing 21. Define a random step counter

```
repeat 50 [merge [rnum 50] [rnum 50]
    [create [+ 100 [count 0 49]]]]
```

Listing 22. Mangle 50 playlists

4.1 Time-Based Scheduling with Max v8 Tasks

The `Task` class provides a mechanism to schedule repeated or delayed execution of JavaScript functions without blocking the main Max scheduler thread. By encapsulating callback functions in `Task` instances, *PLISP* can orchestrate periodic updates, timed bursts, and algorithmic traversals of the SOM and playlists. This approach enables asynchronous behaviors integral to live coding and real-time performance contexts:

```
loop walk 100 '[sim 1] '[replace pl [rnum 10]]
```

Listing 23. Define a "wanderer" on the map

```
after 60000 '[stop walk]
```

Listing 24. Stop loop after one minute

```
every 50 100 '[speed [* 0.1 [count 0 99]]]
```

Listing 25. Create a glissando effect

5. USER INTERFACE AND EVALUATION

A heuristic evaluation of the *PLISP* interface was conducted using Benyon's 12 principles of user interface design [21]. This was carried out collaboratively and conversationally by the authors, through "over-the-shoulder" observation and annotation rather than a formal, documented process. The method revealed key strengths and limitations in how *PLISP* supports sonic exploration through live coding. For example, in response to the *Visibility* principle, we modified the SOMs color gradient to better convey node density. High-density regions are now more discernible, and color cues–yellow for loaded samples and red for the current selection–stand out against the deep grey background.

One of the clearest strengths aligned with Benyon's principle of *Consistency*: commands consistently begin with a function name followed by operands, and results appear in the REPL. Our evaluation also found *Feedback* to be a strong point–commands like `info` return clear, immediate responses. However, the principle of *Familiarity* presented mixed results. While the syntax is intentionally minimal and modeled after LISP, the bracketed expressions and prefix notation may be unfamiliar to users without programming experience. To mitigate this, we added inline documentation with the `doc` command to help new users acclimate quickly.

Principles such as *Control* raised concerns. While the system is resilient to errors, it does little to prevent them. For instance, users can issue `load` or `add` commands with misspelled playlist names, with no autocomplete or preview mechanism. These small mistakes are common in live performance contexts. Our evaluation highlighted opportunities for improvements such as command completion or fuzzy matching to reduce cognitive load.

The *Recovery* principle was strengthened by the use of `try...catch` blocks and non-blocking task scheduling, which ensure that the system degrades gracefully without halting execution. *Flexibility* was another strength: the same commands could be used for algorithmic composition or fine-grained sample selection. Additionally, the inclusion of the `save` command lets users build their own custom tools to be used in music performance.

6. CONCLUSION AND FUTURE WORK

PLISP is a domain-specific language (DSL) for live coding of a self organising map (SOM). By embedding a LISP-inspired interpreter within Max's v8 engine and coupling it to a performant Gen˜synthesis engine, *PLISP* bridges live coding, file finding, and real-time signal processing in a way that is both versatile and musically immediate.

Future work will involve a formal user evaluation with distributable software, along with expanding the vocabulary of behaviors, refining the ergonomics of the interpreter, and exploring how this paradigm might evolve in collaborative or distributed contexts. How might different interpreters or syntax influence interaction? What role might user feedback play in shaping traversals? These questions extend the scope of *PLISP* from an individual tool to a broader speculative framework for algorithmic performance.

Acknowledgments

Information Retrieval in Embedded Systems for Audiovisual Artistic Processes (IRESAP) is supported by The Knowledge Foundation (KKS).

7. REFERENCES

[1] T. Magnusson, "Designing constraints: Composing and performing with digital musical systems," *Computer music journal*, vol. 34, no. 4, pp. 62–73, 2010.

[2] M. V. Mathews, "The Digital Computer as a Musical Instrument: A computer can be programmed to play" instrumental" music, to aid the composer, or to compose unaided." *Science*, vol. 142, no. 3592, pp. 553–557, 1963.

[3] J. McCartney, "Rethinking the computer music language: Super collider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[4] Cycling '74, "Max/MSP," 2024. [Online]. Available: https://cycling74.com/products/max

[5] M. Puckette *et al.*, "Pure data: another integrated computer music environment," *Proceedings of the second intercollege computer music concerts*, pp. 37–41, 1996.

[6] G. Wang, "A History of Programming and Music," in *The Cambridge Companion to Electronic Music*, N. Collins and J. D'Escrivan, Eds. Cambridge, UK: Cambridge University Press, 2017, ch. 16, pp. 55–85. [Online]. Available: https://doi.org/10.1017/9781316459874.006

[7] H. Taube, "Common Music: A Music Composition Language in Common Lisp and CLOS," *Computer Music Journal*, vol. 15, no. 2, pp. 21–32, 2002. [Online]. Available: https://doi.org/10.2307/3680913

[8] R. B. Dannenberg, "The Implementation of Nyquist, A Sound Synthesis Language," *Computer Music Journal*, vol. 21, no. 3, pp. 71–82, 1997.

[9] A. Sorensen and H. Gardner, "Programming with Time: Cyber-Physical Programming with Impromptu," *ACM SIGPLAN Notices*, vol. 45, no. 10, pp. 822–834, 2010. [Online]. Available: https://doi.org/10.1145/1932682.1869526

[10] I. Duncan, "Scheduling Musical Events in Max/MSP with Scheme For Max," *Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University*, vol. 1, no. 1, 2021.

[11] B. Garton, "maxlispj," 2011, retrieved Jan, 2011. [Online]. Available: http://sites.music.columbia.edu/brad/maxlispj/

[12] J. Vincenot, "LISP in Max: Exploratory Computer-Aided Composition in Real-Time," in *International Computer Music Conference*, vol. 2017. Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 2017, pp. 87–92. [Online]. Available: http://hdl.handle.net/2027/spo.bbp2372.2017.012

[13] N. Collins, A. McLean, J. Rohrhuber, and A. Ward, "Live coding in laptop performance," *Organised sound*, vol. 8, no. 3, pp. 321–330, 2003.

[14] G. Wang, P. R. Cook *et al.*, "Chuck: A concurrent, on-the-fly, audio programming language," in *ICMC*, 2003.

[15] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological cybernetics*, vol. 43, no. 1, pp. 59–69, 1982.

[16] H. Terasawa, M. Slaney, and J. Berger, "Perceptual distance in timbre space," in *Proceedings of the International Conference on Auditory Display (ICAD05)*, 2005, pp. 1–8.

[17] L. Garber, T. Ciccola, and J. C. Amusategui, "AudioStellar, an open source corpus-based musical instrument for latent sound structure discovery and sonic experimentation," in *Proceedings of the International Computer Music Conference*, 2021, pp. 62–67.

[18] M. A. V. Stark, "Developing a 3D interface for sound corpus manipulation in virtual environments," in *2025 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*. IEEE, 2025, pp. 643–646.

[19] D. Wessel and M. Wright, "Problems and prospects for intimate musical control of computers," *Computer music journal*, vol. 26, no. 3, pp. 11–22, 2002.

[20] K. H. Sinclair and D. A. Moon, "The philosophy of LISP," *Communications of the ACM*, vol. 34, no. 9, pp. 40–47, 1991.

[21] D. Benyon, *Designing interactive systems: A comprehensive guide to HCI, UX and interaction design*. Pearson Harlow, 2014, vol. 3.