Ziffers - Numbered Notation for Algorithmic Composition

Miika Alonen
Department of Computer Science. Aalto University, Finland
miika.alonen@aalto.fi

Raphaël Maurice Forment
ECLLA. Université Jean Monnet, France
raphael.forment@gmail.com

ABSTRACT

Ziffers is an algorithmic number based musical notation system offering a terse syntax for the improvisation and composition of complex melodies and rhythms. Ziffers gradually evolved from empirical experiments aiming at closing the gap between several number-based notational practices spanning from numerical and staff notation to pitch class set and post-tonal theories. Ziffers places a special emphasis on providing support for live algorithmic and generative composition, embedding a toolset of algorithmic processes directly in its syntax model. The system also tries to accommodate these fixed media types of notation to the liveness experienced when using live coding environments. Ziffers has been influenced by *TidalCycles* (McLean and Wiggins 2010) and *Ixi Lang* (Magnusson 2011), two popular live coding libraries designed around the use of a custom domain specific language, called mininotation, allowing the live manipulation and transformation of musical patterns. In this article, we propose an environment agnostic presentation of the Ziffers musical notation system. As a proof of concept, this article also introduces two implementations running in different environments: *Sonic Pi* (Aaron 2016), a live coding platform designed for education and music performance and Musescore, a general purpose scorewriter and score playback software. In doing so, we are hoping to highlight the versatility of our approach, which seems to allow the use of a unified syntax for different contexts of execution and interpretation.

1 INTRODUCTION: ZIFFERS, QUICKLY EDITABLE NUMBER BASED NOTATION

Ziffers — named and inspired by the older *Ziffersystem* (Warkentin 2022) — proposes a system focused on the conciseness and expressiveness of musical notation. Ziffers is designed to enable generation and transformation of musical patterns (McCormack et al. 1996; McLean 2020) in minimal amounts of time and typing. To do so, Ziffers uses a text-based syntax for representing various short-hand symbolic notations used in music theory and concepts in programming languages. The need for concise and succinct notation in the context of live coding performance is a well-known constraint (Charlie Roberts and Wakefield 2018) that gave rise to many practices and techniques aiming to reduce frictions in the conversational feedback loop between the musician and its programming interface (REFERENCE). Domain specific languages (DSLs) and the use of terse mini notations have become an important design pattern in the conception of live coding interfaces (Hoogland 2019; ADD OTHER), and can also be noted to be a key concept in the larger realm of exploratory or conversational open-ended programming where reactivity and fast decision-making is of prime importance to achieve a creative state of flow (Nash 2015; Kery and Myers 2017).

Ziffers is designed as a terse and platform independent syntax capable of embedding algorithmic and generative processes at the notational level. Basic notation tokens denoting pitch, rhythm or expression marks form the base notation, enriched by a set of generative operators and tokens denoting algorithmic transformations of that first layer of notation. Implementation of the Ziffers system is specific to each targeted platform, in conformity with the notation described in this article and the documentation for the prototype implementation. The first prototype of Ziffers has been created in 2018 (Alonen 2018) as an inline parser for *Sonic Pi*. It was thought as an attempt to speed up the process of writing melodic lines on-the-fly, taking advantage of the large number of predefined methods and compositional helpers offered by *Sonic Pi's* rich internal library. Its design aimed at resolving the dissociation between pitch and rhythm imposed by *Sonic Pi* data structure and imperative-oriented programming model. After a few years of evolutionary prototyping, Ziffers 2.0 was released in 2022 (Alonen 2022), including a new parser aiming at extending Ziffers capabilities with new operators, nested structures and better support for stochastic and aleatoric composition.

2 METHOD: DESIGNING BY DRAWING FROM MUSIC NOTATION HISTORY

The Ziffers music notation project initially started as an experiment to confront historical numeric pitch-based music notation practices with the kairotic notations (Cocker 2013) typically found in live coding and algorithmic music performance systems (Blackwell et al. 2022). Therefore, the system has been designed iteratively in conversation with the diverse and complex historical legacy of said practices. Documents pertaining to the systems of numeric pitch notation in common practice can be traced back as early as the European baroque up to the contemporan era through examples as varied as the American Nashville system, the Chinese Jianpu simplified notation (Kaminski 2022; MNMA 2022) or the theoretical pitch-class set numbering of musical structures and chords used in post-tonal music theory (Forte 1973).

Numerical systems for score, harmonic or melodic writing – less commonly encountered than their staff-based counterparts – have historically served as compositional tools for polyphonic contrapuntal notation (Davantès 1560), early theoretical endeavours in algorithmic composition (Kircher 1650) or in pedagogy and teaching (Rousseau 1781). Finding a new breath during the late nineteenth-century through the Galin-Paris-Chevré notation (Dauphin 2012), the practice spread to Asia where it is widely used, known and practiced, sometimes concurrently with standard staff notation. Some specialized applications of numbered notation can also be found in the toolset of contemporary ethnomusicologists and ethnomathematicians that often take advantage of the versatility of symbolic cipher notation to devise notation systems capable of formalizing musical systems or previously non-written musical practices (Chemillier 2002).

We perceive – as a thought experiment – the use of mininotations (Magnusson and McLean 2018) as modern live coding analogues to previously mentioned numerical music notation systems. In that regard, with their conciseness and economy of expression, we observe that older non-computer based numerical notations systems are adequate fits for the domain but have never been fully transposed and applied to the realm of live coding performance. Strikingly, we see them being used by musicians and music theorists alike as pragmatic and practical tools for draft notation and information sharing. In our opinion, the versatility of pitch-based number notations as well as their adequacy to the traditional way of inputting complex data structures (as lists, arrays or ordered data) in the most music programming languages (Roads 1996; Dannenberg 2018) make this type of notation a fruitful exploration domain for live coding and computer-based music notation alike. Generative numeric notation can also mitigate the separation between static and dynamic music notation languages as defined by Dannenberg's typology of the domain. Examples such as Adagio (Dannenberg 1986), ABC (Walshaw 2011), Guido (Hoos et al. 1998), MML (Izumi 2001), MusicXML (Good 2001), Lilypond (Nienhuys and Nieuwenhuizen 2003) or MusicTXT (Li and Li 2021), as static languages, focus on the encoding of musical information on fixed medium while other dynamic languages such as Nyquist (Dannenberg 1997), SuperCollider (McCartney 2002), Sonic Pi (Aaron 2016), TidalCycles (McLean and Wiggins 2010), Open Music (Bresson et al. 2017), OpusModus (Opusmodus 2022), often extends musical notation by blending it with computational models of control flow, sound processing capabilities or binding different means of sonic writing: "The instrument incorporates notational elements, but conversely the notational is becoming increasingly instrumental and systematic" (Magnusson 2019).

3 OBJECTIVES: SPECIFICATION OF A CROSS-PLATFORM NOTATION SYSTEM

Objectives of the Ziffers numerical notation adheres to the cognitive dimensions of music notations as defined by Nash (2015) and to the criterion defined by the Music Notation Project (MNMA 2022). The notation must allow users of different ages and backgrounds to compose music. The Ziffers extension built for Sonic Pi presented in latter sections of this paper builds on the accessibility and pedagogical efforts pursued by Aaron (2016). The development of the numerical notation aims to remove the exclusivity that comes with the music theory and provide a beneficial live coding experience for the uninitiated. Use of numeric notation should make it possible to write melodic sequences and interval movements using numbers regardless of the key and scale. The notation should also be minimal enough to be easily manipulated through any writing tool by relying solely on symbols taken from the ASCII table. To easily conceptualize pitches in time, defining a combination of pitch and duration should be as straightforward as possible. The notation should be able to function as an educational tool for teaching music theory and composition. Furthermore, by making the user aware of the relationship between musical and mathematical notation, the system should also support teaching advanced music theory concepts such as musical set theory and the manipulation of pitch-sets or any complex musical objects.

The number of live coding environments have steadily increased in recent years (Toplap 2022). We argue that there is now a need for some amount of notational interoperability between different live coding environments. We interpret the non coordinated emergence of Tidal-like mininotations in tools such as Gibber (Charles Roberts and Pachon-Puentes 2019), Bacalao (Fraser 2020), Petal (kn1kn1 2017) and Hydra (Jack 2019) as an attempt from live-coders to agree on a common rhythmic pattern language despite the use of different and often highly personalized environments. In a similar fashion, the Ziffers project is hoping to bring some cross-environment support for concise and rich melodic notation. Moreover, this project can also be considered as an attempt to bridge the gap between traditional staff-based notation tools such as Avid's Sibelius, Steinberg's Dorico or Musescore and kairotic pattern notations used by live-coders. As

demonstrated by the Musescore interpreter 2, Ziffers can be used both as a live and fixed-media notation, allowing any musician learning the system to quickly jot down or hear their ideas directly in any Ziffers supported software system capable of audio playback.

The Ziffers system is open for the multiple interpretations of any integer sequence. Depending on context and purpose, numbers can receive multiple interpretations, denoting musical objects such as integer notation (`0-9` and `T=10` and `E=11`), scale degrees (`1-9, 0=rest`), midi notation (`1-127`), MIDI channels (`0-15`) or even roman numerals for chord sequence writing (`i-vii`). The notation also supports commenting and providing supplementary information to clarify the meaning of any number sequence, thus adding support for the definition of custom scales, temperaments and events of any kind. The base syntax has also been designed to support common score-centric notation symbols (articulations, bar lines, repetitions) and extended generative symbols (random numbers, mathematical operations) more typically found in a live coding context. In doing so, the Ziffers system allows for some amount of transduction between different notation mediums and contexts, allowing the displacement of live coding ideas into the realm of score-based music composition. From another perspective, Ziffers also opens the field for quickly bringing musical ideas from staff-based musical notation in to the typically improvised and delayed feedback (Rohrhuber, Dean, and McLean 2018) oriented performing context permitted by live coding environments.

4 SONIC-PI BASED PROTOTYPE IMPLEMENTATION

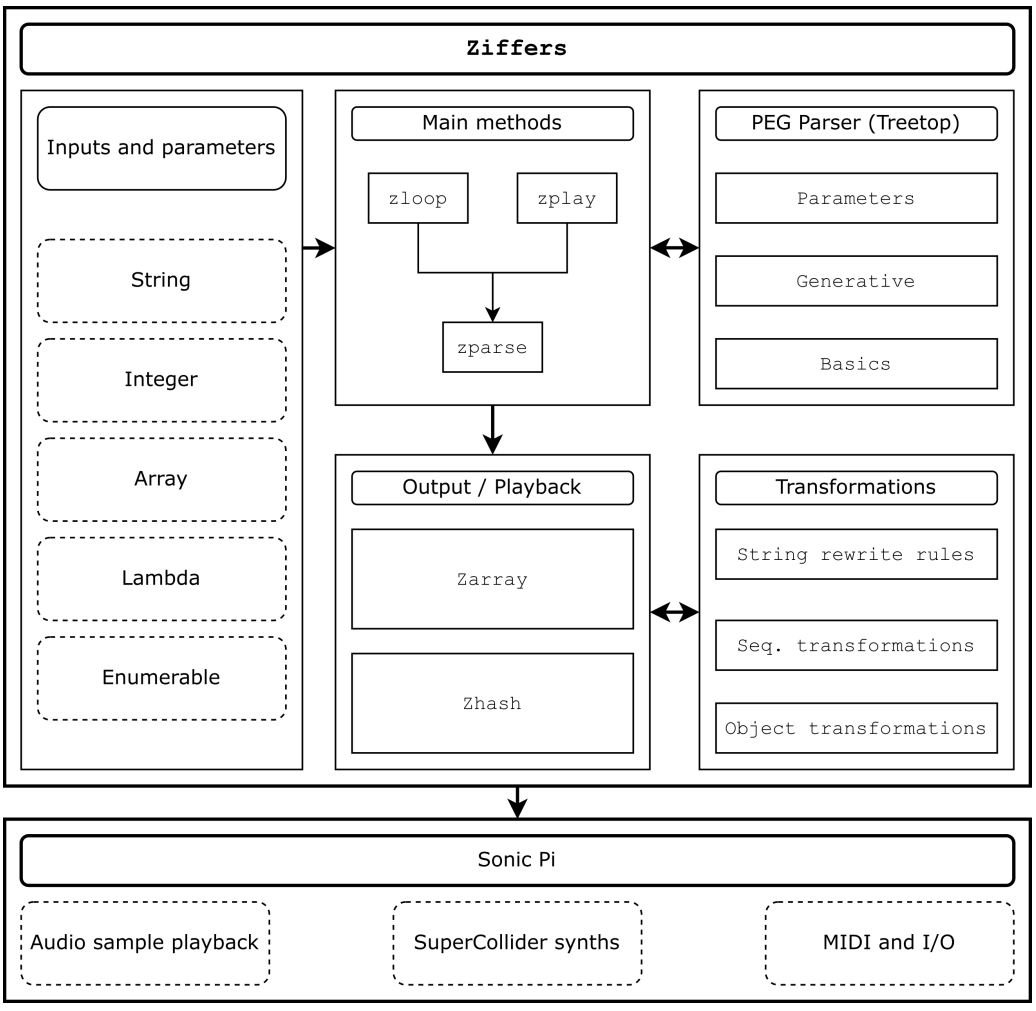

Figure 1: Ziffers architecture for Sonic Pi.

Ziffers has evolved from a simple script embedded in one of many Sonic Pi's text buffers to a fully fledged DSL and framework for computer aided algorithmic composition and live coding. The current software architecture for the Sonic Pi's based Ziffers implementation is presented in the figure above. A *Musescore 3.0* plugin allowing the transformation of staff-based score format to Ziffers notation has also been created and proposed as a proof of concept tool for learning the new numeric notation.

MuseScore plugin can be used to transform traditional scores (from MusicXML, ABC notation or MIDI files) to the

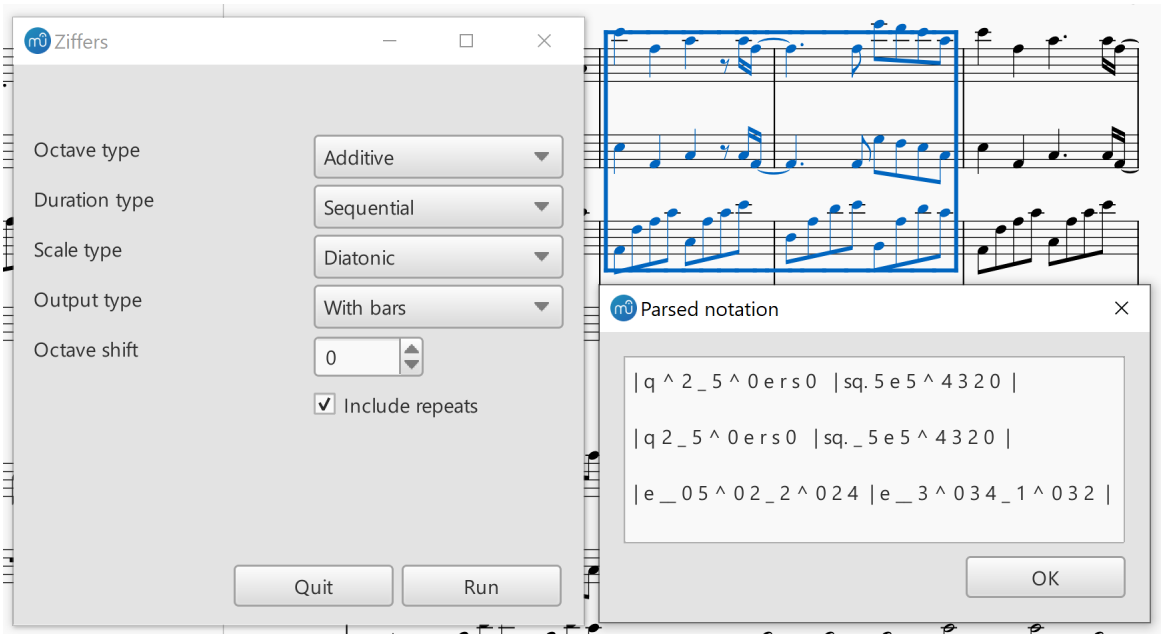

Figure 2: Ziffers plugin for MuseScore exporting fragment from Aphex Twin's Avril 14th score.

numeric notation for analyzing the musical structures from your favorite composers and using the displaced sequences as form of rehersal and inspiration for the live coding. Examples above and below exemplifies this form of learning practice using the extracted sequence from Avril 14th by Aphex Twin.

```
ziffers " // Aphex Twin - Avril 14th with added transformations
/ synth: piano
| q ^ 2 _ 5 ^ 0 e r s 0  | sq. 5 e 5 ^ 4 3 2 0  | / retrograde: [false,true]
| q 2 _ 5 ^ 0 e r s 0 _  | sq. _ 5 e 5 ^ 4 3 2 0  | / octave: [0,1,-1]
| e __ 0 5 ^ 0 2 _ 2 ^ 0 2 4  | e __ 3 ^ 0 3 4 _ 1 ^ 0 3 2 | / inverse: [0,1]"
```

4.1 Main methods

The **zplay**, **zloop** (z0...z20) and **zparse** methods are the main entry points to interact with Ziffers. The multiline methods, **ziff** for single play and **ziffers** for looping, allow vertical writing of polyphonic patterns (as in Aphex Twin example above). A slightly different version, **ztracker**, emulates the behavior of classic music tracker softwares (e.g. Pro Tracker, Renoise) for multiline horizontal writing. Multiple modes of interaction with the patterning system are detailed in the repository hosting the *Sonic Pi* implementation. Each method has its own specificities, and can be adapted to a particular context of execution. For instance, composing will preferably be supported by the non-looping zplay method while improvisation is preferably handled through the looping and automatically synchronized zloop shorthands (each loop synchronizing on z0 unless otherwise stated). There are also various shorthands for manipulating the effects, synth parameters and playback, such as **cycle**, **fade** and **tweak** documented in the effects section](https://github.com/amiika/ziffers/wiki/Effects) of the documentation. Being non intrusive, Ziffers can also adapt to the **live_loop** mechanism defined by Sonic Pi to enable quick playback of different inputs and further enhance musical expressivity. The combined usage of **live_loop** and **zplay** methods is supported for musicians willing to use both patterning paradigms.

```
# Example of combining live_loop and zplay in Sonic Pi
with_fx :reverb, room: 1.0 do
  live_loop :sonic_pi do
    with_synth_defaults divisor: rrand(0.1,0.15), attack: rrand(0.01,0.1) do
      pling = rrand_i(1000,3000)
      4.times do
        zplay pling, synth: :fm, scale: :mixolydian, rhythm: "q.eqe", octave: ->(){rrand_i(-1,1)}
      end
    end
  end
end
```

4.2 Inputs and parameters

All inputs are normalized to a string, type conversion being applied when necessary. The integer '2468' can be parsed to a sequence of notes or a chord, etc... Adjusting the base behavior of the parser can be done by providing additional keyword arguments defining a context for the parser. In a similar fashion, user input can also be specified as a lambda function or as enumerable defined by the *Ruby* language. This is allowing for the definition and use of infinite note sequences such as the Morse-Thue sequence demonstrated below, one that is commonly used in fractal music composition (Kindermann 2001; Gómez and Nasser 2021).

```
# zplay is the fastest way to ziffer
zplay "q 0 236 q.4 e6 3 0 2", key: :g3, scale: :minor, synth: :piano
zplay 2468, key: :c3, synth: :fm, release: (ring 0.125,0.25,1.0)
zplay 2468, key: :c3, synth: :fm, parse_chord: true
zplay [2,4,6,8], key: :E4, scale: :hex_sus, synth: :chiplead, width: 2, pan: ->(){rand}
zplay [[1,0.5],[3,0.25],[0,1.0]], key: 60, scale: :aeolian, synth: :blade, res: 0.1

# z0-z20 are shorthands for looping the sequence
z0 ->(){rrand_i(-9,9)}, scale: :blues_minor, synth: :kalimba, clickiness: ->(){rand}

# Morse-thue sequence using mod 7
z1 (0..Float::INFINITY).lazy.collect{|n|n.to_s(2).split('').count('1')%7},rhythm:spread(7,9)

# Built-in enumerable for playing the digits of pi
z1 pi.take(10).to_a, scale: :blues_minor, synth: :tech_saws, rhythm: 1211, cutoff: tweak(:sine,30,100,10).reflect
```

4.3 Parsers

Parsing is implemented with parsing expression grammars (PEGs) using the Ruby-based Treetop library (Treetop 2022). Ziffers is constituted of three parsers, each one handling specific use cases depending on the nature of the input. The **Parameter parser** is used for multiline parsing where notation can include supplementary information for the parsing, such as the key, scale, synth, MIDI port, MIDI channel and other context or environment specific parameters. **Generative parser** has been specialized to parse random number generators, sequence generators relying on various conditional statements and mathematical operations. Furthermore, an optional string rewrite loop can be used to extend these generative capabilities by transforming the input through recursive substitution of some parts of the initial input. One can think of the Ziffers parsing system as a multi-layered conditional parsing operation. After a first part of context specific or generative parsing operations, the resulting string is parsed again using the **Basic parser**, which includes only the static definitions of pitches, octaves and durations. The relevant data structures for the parsed notation is implemented as subclasses of array and hash which implements all of the necessary methods for the transformations.

4.4 Transformations

The Transformation phase is an optional step for the manipulation of the generated musical patterns as defined by Spiegel (1981), for example retrograde, inversion, substitution or by defining custom transformations that can manipulate the order, pitches, durations or any other parameters in the sequence. Transformations can be done in multiple ways and in different phases of the live composition process.

```
# Transformations applied on a live sequence
z1 "e0 s 1 2 3 4 e5", inverse: [false,1,0,-1], retrograde: [true,false], swap: [0,3,2,4]

# Deferred transformations using 'zparse'
a = zparse "q 0 2 1 4"
b = a.inverse
c = a.retrograde
d = a.inverse(-1).retrograde
zplay (a*2+b+c*2+b)

# Inline variant of the above mentioned methods
z1 "((q 0 e 4 2 5 3)<inverse>(3 0 2 3))<swap>(1 3 0)", key: :g3, scale: :mixolydian
```

Ziffers also implements a **string rewriting system** that is a non-deterministic Turing machine for evaluating axioms using unrestricted, context-sensitive and stochastic grammars (McCormack et al. 1996). Rules for matching can be defined using regular expressions and substituted by using any syntax defined in the Ziffers notation. In contrast to some other common generative grammar systems (Nierhaus 2009), the substitutions made to each generation can be evaluated within a loop. This also allows rules to be added, edited and evaluated on the fly, creating a new form of rule machine that can be used in a live performance.

```
# Simple rules creating a long pattern
zplay "0", rules: {"0"=>"1 2", "1"=>"2 1", "2"=>"1 0"}, gen: 6
# Match using regex, substitute with evaluation and mod by 7
zplay "1 2 3", rules: {/[1-9]/=>"({$*2} [e,q] {$*3})%7"}, gen: 3
# Play and mutate generations on-the-fly
z1 "0", rules: {"0"=>"q 0 1", "1"=>"e (1,4) 0" }
# Change durations using rules
z2 "q0 e2 e1 q4", rules: {"q"=>"e", "e"=>"q"}
# Stochastic rules using Ziffers notation and regular expressions
z1 "q0", rules: {"q0"=>"{%>0.5?q3:q0}", "q3"=>"{%>0.25?q(3,7):q0}", /q[1-9]/=>"[q3,q0]"}
```

Using different input parameters to control the parsing and output can be seen as more implementation specific. Capital characters can be used as place handlers to time different kind of events and implementation specific functions.

```
zplay "[: q K H [K K] H :4]", K: :bd_boom, H: :drum_cymbal_closed
zplay "[: q K H [K K] H :4]", K: 12, H: 90, port: "ext_midi", channel: 1
define :foo do sample :bd_zum end # Define Sonic Pi method
z1 "q F e F F", F: :foo # Time the method using ziffers
```

5 NOTATION

Ziffers notation is divided into basic and generative syntax. Basic notation contains the elementary building blocks for sequential and polyphonic melodies. Generative notation builds on the basic notation and includes syntax for arithmetics, logical operations and transformations. Examples in this section are written in implementation independent form and could be performed using different methods.

5.1 Basic Notation: pitch, rhythm and silence

The Ziffers syntax is supporting many classic musical notation symbols as well as many variations in their writing. For better readability and for the sake of brevity, we have chosen to list and comment only the most commonly accessed tokens, relegating other notation symbols to the example tables. Detailed sections will cover the more advanced and Ziffers specific features.

Basic notation contains all static symbols, which can be parsed without additional processing. By default, Ziffers interprets numbers as in pitch class notation `[0-9TE]`, extending numbers to capital characters `T` for `10` and `E` for `11` to support faster input for 12-tone compositions. Pitch classes higher than `9` or lower than `-9` can also be escaped using curly braces. For example pitch classes `-12` and `24` could be notated as `{-12 23}`, which is then interpreted to the right octave depending on the scale. For example in major scale number `24` becomes pitch class `3` in octave `3`.

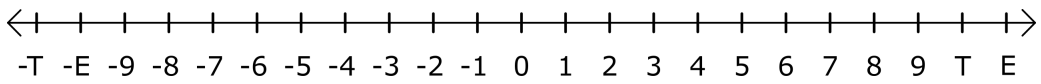

Figure 3: Pitches in integer notation.

In pitch class notation 0 represents the root for the scale. Negative pitch classes can be used to denote pitch classes below the root. Scales can be thus mentally visualized as number lines ranging from negative to positive, which then makes it easier to find the right pitch without any calculation and knowledge of the given scale.

Alternatively degree based interpretation can be used as a separate option, where integers from 1-9 are interpreted as scale degrees. Use of degree based notation can be easier for those who are only familiar with scale degrees and not accustomed to pitch class notation.

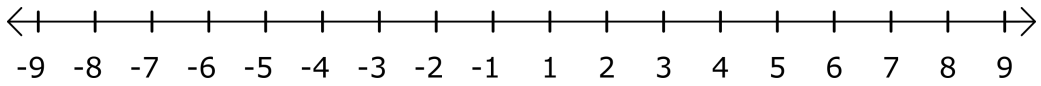

Figure 4: Pitches in degree notation.

In degree based notation degrees range from 1 to 9 (depending on the scale length). Compared to pitch class notation the degree based notation does not have the root as a mirroring axis. Pitches can also be sharpened using **#** or flattened with **b** when using diatonic scales.

Note durations are denoted with lower case characters which are selected from note length names, for example **w** for whole, **h** for half, **q** for quarter, **e** for the eight note etc. In integer notation, silence is defined using the character **r** for rest. In degree based notation, 0 is also treated as silence, as used in the Galin-Paris-Chevé notation (Dauphin 2012). Characters for nearby triplet notes have been selected on the basis of the close proximity on the qwerty keyboard. See the full list of duration characters in the documentation. Alternatively, decimals can be used instead of characters. Dotted lengths are used as in traditional musical notation to increase the note duration. Similarly, decimal notation can be used as an alternative to letter-based durations and dots, especially for venturing outside of the traditional note lengths. For example in children's song row your boat could be notated differently depending on the chosen duration syntax:

```
// Melody using note length characters
| q. 0 0 | q0 e1 q.2 | q2 e1 q2 e3 | h.4 | e 7 7 7 4 4 4 2 2 2 0 0 0 | q4 e3 q2 e1 | h. 0 |
// Melody using decimal notation
| 0.375 0 0 | 0.25 0 0.125 1 0.375 2 | 0.25 2 0.125 1 0.25 2 0.125 3 | 0.75 4 | …
```

Key is determined from given parameters or from inline notation denoted by either MIDI notes, note names or scientific pitch notation. **Scale** is determined from the predefined scale names or by defining custom scales using an array of intervals where 1 represents semitone and 2 tone, or by using decimals for intervals smaller than a semitone. **Octaves** can be changed using special characters _ for lowering and ^ for raising the octave or by using escaped integers <3> to define the exact octave. Octaves can be changed using negative integers which also keeps the relative distance when for example, the scale is changed from minor to minor pentatonic. Consider following pattern both key: `| 0 3 -1 2 | 0 3 _6 2 |` In minor scale both measures are exactly the same, but since minor pentatonic has only 5 scale degrees there is a difference in how the `_6` is interpreted compared to `-1`. **Chords** are typically represented by using number groups. The basic diatonic major chord can thus be represented as 024 in diatonic context, and as 047 in a chromatic context. Pitches in chords can be defined in different octaves combining the notation for example `024^0 _12<1>4`. **Whitespace** acts as an separator for different objects, like the chords in a sequence. **Chord inversions** can be amended using the `%` symbol like `024%2`. Roman numerals are also supported as part of the extended syntax for chord writing based on tonal/diatonic techniques. Definition of **Measures** are optional but can prove necessary if there is a need to access them in different order. They are denoted using the pipe character `|` and each such character always resets the current octave and note length to the default value. This is important for both human and a machine to be able to start from any given measure without having the need to retrace to the beginning of the arrangement.

```
// This is a comment                 // One-line comments in Ziffers notation
/* this is a multiline comment */    // Multiline comments
<d3> 0 2 3 <minor> 0 2 3             // Inline key and scale
| h 0 ^ q. 4 e 2 | q. 4 h 6 e 2 |    // Measures
``0 `0 0 ´0 ´´0                      // Accents (` softer and ´ louder amp.)
0 '0 ''0                             // Staccato (shortened duration)
~<0.3>0123456                        // Glissando/bends (with speed param.)
{decay: 0.5}  1 {port: ext_2} 3      // Hash notation for inline parameters
```

5.2 Generative notation

Repetition is the simplest form of generative syntax also used by the traditional staff notation. **Repeats** are denoted using dotted list notation with optionally nested structure as exemplified here with frere jacques:

```
[: q 0 1 2 0 :] [: q 2 3 h4 :] [: [: e 4 5 4 3 q 2 0 :] [: q 0 _4 h0 :] :]
```

Bracket syntax is alternative notation for durations that subdivides the note lengths using nested brackets. Similar notation is also used by TidalCycles and Alda. Bracket syntax is especially useful for notating triplets or n-tuplets.

```
// Frere Jacques again using bracket syntax
w [: [0 1 2 0] :] [: [[2 3] 4] :] [:[: [[4 5] [4 3] 2 0] :] [: [[0 _4] 0] :]:]
```

Cyclic notation for events has been previously introduced by TidalCycles. Cyclic structures can be defined by enclosing the sequence using angle brackets to optionally nested structure such as `<1 2 <3 <4 5>>>`. Ziffers enables the use of cyclic notation in loops or as part of repeat notation as alternative endings also used in the traditional staff notation. When cycles are defined within a repeat notation, the cycles are evaluated within the context of the repeat, for example `[: 1 2 <3 4> :]`.

Random numbers are the basic building blocks of all aleatoric melodies. There are multiple ways of defining random integers and decimals in the generative syntax and the generated values can be used for different purposes. Number sequences can be generated using range notation inspired by the Ruby syntax which has updated to be more versatile.

```
(0.25,0.5) (0,5)            // Random decimal and pitch between defined range
% ?                         // % = (0.01,1.0) ? = (0,scale length)
[0.125,0.25,0.5] [0,2,4,6]  // Random numbers from selection
-1..1                       // -1 0 1
0..3+2                      // 0 2
0..3*2                      // 0 2 4
?..(3,5)                    // 5..3 => 5 2 3
```

Lists syntax can be used to arrange pitches into operable sequences. Arithmetic operations `+ - * ** / ^ % | & << >>` can be applied to list values in a serial manner. Arithmetic operations can also be applied between to lists creating cartesian operations. To produce long alternating patterns arithmetic operations can also be chained together. There are also 5 special methods that have designated short-hand symbols. Combine **&** for creating a chord out of list, separate **$** for creating the sequence from a chord, unique **!** for removing same pitches from the list, pick random **?** for picking 1-n pitches from the list and suffle **~** for suffling and picking unique random values from the list.

```
(q 0  e 1 2 q 3 5)+1*4%7              // Applying multiple operations to a list
(1 2)+(3 4)                           // Same as: 1+3 1+4 2+3 2+4
(q 0  e 1 2 q 3 5)+(3 0 -2 3)-(2 1 3 4) // Lists and operations can be chained
```

Transformations can also be notated inline using the list syntax and escape notation for built-in methods: `(list)<method>(optional-list)`. Ziffers also implements numerous shorthand notations for useful transformations, like **cyclic zip** for combining values from two lists, notation for **pitch-class set multiplication** [heinemann1998pitch] and **sequence interpolation** inspired by the Thesaurus of scales and melodic patterns (Slonimsky 1986). **Generative repeat** syntax can be used to generate values multiple times, where as normal repeats are used to repeat the generated values and create a sense of repetition. **List functions** are inspired by polynomial functions, and can be used to transform the values using arithmetic expressions.

```
(1 2 3)<retrograde>
(q e e)<>(0..5) // q 0 e 1 q 2 e 3 q 4 # Cyclic zip
(1 2)<+>(3 4 5) // 1 3 1 4 1 5 2 3 2 4 2 5 # Product of two lists
(0 5)<*>(0 3 6 9) //  0 5 3 8 6 {11} 9 {14} # Pitch-class set multiplication
(1 3)<4>(1 3) // 2 4 6 1 3 5 # Interval interpolation
(: (1,4) :3) // Generated 3 different random numbers
q (0..3){(2x-1)(2x^2-4)} // Using function to transform a list
((1,5)){x<2?(x+3):(2x)(x-2)} // Applying functions conditionally
```

Chords can be generated using roman numerals `(i-vii)` and further modified using chord names, inversions and modal interchange. **Chord names** are defined using the caret symbol and the corresponding name, for example: `i^dim7`. Chord names are based on Sonic Pi's definitions and further standardization would be needed to include more chord names and harmonize varying different practices. Chords can also be inverted using `%` symbol, for example `iv%-2`. Tri-chords can also be built by defining number of pitches `{i-vii}+{number of pitches}`, for example `ii+6`. **Borrowed chords** can be created using shorthand characters for modes `(a-g)` combined with the roman numerals, for example: `iib` (locrian), `iiig` (mixolydian). Chords can also be generated from lists using combine **&** shorthand that transforms sequences to chords, for example: `((10,100) (100,1000))&` or `((4..7)*4)&`. **Arpeggios** can be generated by defining sequences of chords and selecting pitches using the ampersand ***-operator, for example: `(i ii iii iv v vi vii)@(e 0 1 2 012)`.

Similarly, a list of pitches can be also accessed horizontally using the `#`-operator, for building new chords or sequences, for example: `(0 2 4 6)#(e 0 1 2 012)`.

Euclidean rhythms have gained popularity among music composers (Morrill 2022) for some years after Godfried Toussaint first presented the idea of using euclidean algorithm to generate rhythms from binary sequences (Toussaint 2005). *Sonic Pi* also implements the euclidean algorithm as a spread method named after the evenly spread boolean. Ziffer's has it's own approach to euclidean patterns and implements an algorithm defined by Thomas Morrill (Morrill 2022) and a novel syntax that can be used to combine both onset and offset values from the binary sequence. Syntax for the euclidean generator is defined as an operator for one or two lists: `(onset)<beats,total,rotate>(offset)`. Values will be selected from onset or offset list according to the binary sequence generated by the euclidean algorithm. Default offset value is a rest, but it can be replaced with a list of alternative offset values. Both onset and offset lists can include any syntax defined in the numeric notation. Values in the list will overflow to the beginning if there are not enough values for the whole cycle. Inner cycles can also be defined using cyclic syntax, to make more complex structures.

```
/ synth: :pretty_bell
s (<0 <3 <5 (-3,3)>>>)<3,6> // Spread nested cycles
/ synth: :kalimba
((q 0 3) (q 4 3))<3,5>((e 3 4) (e 1 4))
/ X: :elec_flip, B: :drum_cowbell
(eX sB)<13,16>(eB sX)
```

Expressions can be **evaluated** using curly braces: `{...}`, which can be used for escaping pitches, arithmetic operations or conditional logic. Using ternary operator syntax `(cond)?(true):(false)` it is possible to define alternative or conditional parts. Generative notation can be stored to **variables** denoted using capital letters to form repetitive random structures from the generated values.

```
{10 11 9+2 3*5}          // Evaluate as pitches
={10 (10,20) 3*5}        // Evaluates as chords
{%>0.5?3}                // Conditional evalution
{(0,9)>4?(1,3):(3,6)}    // Random numbers based on condition
{%>0.5?4 %>0.2?5:3}     // Multiple conditionals

// Assign Ziffers patterns to variables
A=(0 1 (1,6)) B=(0 [-2,6] 3 [-5,5]) A B A B A
A=<1 2 3> B=(1,6) B A B B A
A=(0 3 2 4) B=(A)<retrograde> C=(B)<inverse>(3) A B C
A=(1,6) {A>3?(q1):(e 1 b2)}
A=% {A>0.5?(e 1 2):(q2)} {A<0.5?4:5}
```

6 Conclusion and future work

In this paper, we have presented a novel numeric notation and a pattern language usable in a live coding context. The presented notation is designed as a bridge from the old to the new, linking and facilitating exploration between different forms of computer and staff-based musical notation; from pitch-based staff writing to generative and improvised performance. The Ziffers notation is designed to be platform independent, making it possible to share generative melodies and patterns between different live coding languages and more traditional composition tools. Even though Ziffers as a language is expressive enough to describe arbitrarily complex musical sequences using mathematical and generative operations, profound and meaningful interaction is still to be found in the link between Ziffers and the underlying logic and flow centered operation of live coding interfaces implementing it. Further experimentation is still needed to find the perfect balance between the proposed numbered notation and different live coding languages.
For now, Sonic Pi is the only fully supported platform for Ziffers. In the future such extensions may become obsolete due to a planned change from Ruby to Elixir programming language. By then, the implementation needs to be rewritten, but it has also already served its purpose as an exploratory medium for prototyping numbered notation in live coding. It is also always possible to use Ziffers with the latest *Sonic Pi* version with the Ruby support. As a future work, we intend to develop parsers in other live coding libraries such as Sardine (Forment 2022), SuperCollider and others. An upgrade for the MuseScore plugin is also planned to the upcoming MuseScore version 4.0, as a way to support input of the generative numeric notation directly to traditional sheet music and create a hybrid algorithmic composition environment for both numbered and standard notation. We envision that the proposed notation can also be used as a tool for teaching music analysis and pitch-class set theory and act as a stepping stone between music theory and live coding.

7 Acknowledgment

Authors would like to thank Sam Aaron and the whole Sonic Pi community for the important work on improving the pedagogy and accessibility of programming and for the inspiration to pursue research in the field of live coding. Raphaël would like to thank Laurent Pottier and Alain Bonardi for their support, as well as the doctoral school *3LA* from the University of Lyon for the funding it provided to this research.

References

10 Aaron, Sam. 2016. "Sonic Pi–Performance in Education, Technology and Art." *International Journal of Performance Arts and Digital Media* 12 (2): 171–78.

Alonen, Miika. 2018. *Ziffers: Numbered Notation for Algorithmic Composition* (version 0.1). https://github.com/amiika/ziffers/commit/c12516d2b1604836925cbe829e488e033578405c.

———. 2022. *Ziffers: Numbered notation for algorithmic composition* (version 2.0). https://github.com/amiika/ziffers.

Blackwell, Alan F, Emma Cocker, Geoff Cox, Alex McLean, and Thor Magnusson. 2022. *Live Coding: A User's Manual*. MIT Press.

Bresson, Jean, Dimitri Bouche, Thibaut Carpentier, Diemo Schwarz, and Jérémie Garcia. 2017. "Next-Generation Computer-Aided Composition Environment: A New Implementation of OpenMusic." In *International Computer Music Conference (ICMC'17)*.

Chemillier, Marc. 2002. "Ethnomusicology, Ethnomathematics. The Logic Underlying Orally Transmitted Artistic Practices." In *Mathematics and Music*, 161–83. Springer.

Cocker, Emma. 2013. "Live Notation:–Reflections on a Kairotic Practice." *Performance Research* 18 (5): 69–76.

Dannenberg, Roger B. 1986. "The Cmu Midi Toolkit." In *ICMC*, 86:53–56.

———. 1997. "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis." *Computer Music Journal* 21 (3): 50–60.

———. 2018. "Languages for Computer Music." *Frontiers in Digital Humanities* 5: 26.

Dauphin, Claude. 2012. "Le Devenir Du Système de Notation Musicale de Jean-Jacques Rousseau." *Rousseau En Musique, Orages*, no. 11: 79–98.

Davantès, Pierre. 1560. "Pseaumes de David, Nouvelle Et Facile Methode Pour Chanter Chacun Couplet Des Pseaumes Sans Recour Au Premier." 1560. https://gallica.bnf.fr/ark:/12148/bpt6k3135388/f23.item.

Forte, Allen. 1973. *The Structure of Atonal Music*. Vol. 304. Yale University Press.

Fraser, Glen. 2020. *Balacao - Somewhat Fishy Live Cod(e) Extensions to SuperCollider.* (version 0.1). https://github.com/totalgee/bacalao.

Gómez, Ricardo, and Luis Nasser. 2021. "Symbolic Structures in Music Theory and Composition, Binary Keyboards, and the Thue–Morse Shift." *Journal of Mathematics and Music* 15 (3): 247–66.

Good, Michael. 2001. "MusicXML for Notation and Analysis." *The Virtual Score: Representation, Retrieval, Restoration* 12 (113-124): 160.

Hoos, Holger H, Keith A Hamel, Kai Renz, and Jürgen Kilian. 1998. "The GUIDO Notation Format–a Novel Approach for Adequately Representing Score-Level Music."

Izumi. 2001. "Music Macro Language." 2001. http://www.vgmpf.com/Wiki/index.php/Music_Macro_Language.

Jack, Olivia. 2019. "Hydra: Live Coding Networked Visuals." In *Proceedings of the Fourth International Conference on Live Coding*, 353–54.

Kaminski, Joseph S. 2022. "Jianpu Simplified Notation and the Transnational in Musical Repertoires of New York's Chinatown." In *Material Cultures of Music Notation*, 110–23. Routledge.

Kery, Mary Beth, and Brad A Myers. 2017. "Exploring Exploratory Programming." In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 25–29. IEEE.

Kindermann, Lars. 2001. "MusiNum-the Music in the Numbers." In *In: ER Miranda, Composing with Computers. Music Technology Series, Focal Press, Oxford.*

Kircher, Athanasius. 1650. “Musurgia Universalis Sive Ars Magna Consoni Et Dissoni.” 1650. https://archive.org/details/bub_gb_97xCAAAAcAAJ/mode/2up.

kn1kn1. 2017. *Petal - a Small Language on Sonic Pi with Similar Syntax to TidalCycles*. https://github.com/siaflab/petal.

Li, Kelian, and Wanwan Li. 2021. “MusicTXT: A Text-Based Interface for Music Notation.” In *Proceedings of the 11th Workshop on Ubiquitous Music (UbiMus 2021)*, 62–71. g-ubimus.

Magnusson, Thor. 2011. “The Ixi Lang: A Supercollider Parasite for Live Coding.” In *ICMC*.

———. 2019. *Sonic Writing: Technologies of Material, Symbolic, and Signal Inscriptions*. Bloomsbury Academic.

Magnusson, Thor, and Alex McLean. 2018. “Performing with Patterns of Time.”

McCartney, James. 2002. “Rethinking the Computer Music Language: Super Collider.” *Computer Music Journal* 26 (4): 61–68.

McCormack, Jon et al. 1996. “Grammar Based Music Composition.” *Complex Systems* 96: 321–36.

McLean, Alex. 2020. “Algorithmic Pattern.” In *Proceedings of the 20th Conference on New Interfaces for Musical Expression. Birmingham, UK*.

McLean, Alex, and Geraint Wiggins. 2010. “Tidal–Pattern Language for the Live Coding of Music.” In *Proceedings of the 7th Sound and Music Computing Conference*, 331–34.

MNMA. 2022. “The Music Notation Project.” 2022. https://musicnotation.org/tutorials/numerical-notation-systems/.

Morrill, Thomas. 2022. “On the Euclidean Algorithm: Rhythm Without Recursion.” *Bulletin of the Australian Mathematical Society*, 1–7.

Nash, Chris. 2015. “The Cognitive Dimensions of Music Notations.”

Nienhuys, Han-Wen, and Jan Nieuwenhuizen. 2003. “LilyPond, a System for Automated Music Engraving.” In *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*, 1:167–71. Citeseer.

Nierhaus, Gerhard. 2009. *Algorithmic Composition: Paradigms of Automated Music Generation*. Springer Science & Business Media.

Opusmodus. 2022. “Opusmodus - Structural Composition System.” 2022. https://opusmodus.com/.

Roads, Curtis. 1996. *The Computer Music Tutorial*. MIT press.

Roberts, Charles, and Mariana Pachon-Puentes. 2019. “Bringing the Tidalcycles Mini-Notation to the Browser.” In *Proceedings of the Web Audio Conference*.

Roberts, Charlie, and Graham Wakefield. 2018. “Tensions and Techniques in Live Coding Performance.”

Rohrhuber, Julian, RT Dean, and A McLean. 2018. “Algorithmic Music and the Philosophy of Time.”

Rousseau, Jean-Jacques. 1781. *Projet Concernant de Nouveaux Signes Pour La Musique*. éditeur non identifié.

Slonimsky, Nicolas. 1986. *Thesaurus of Scales and Melodic Patterns*. Schirmer Trade Books.

Spiegel, Laurie. 1981. “Manipulations of Musical Patterns.” In *Proceedings of the Symposium on Small Computers and the Arts*, 19–22.

Toplap. 2022. “Awesome Live Coding - a Curated List of Live Coding Languages and Tools.” https://github.com/toplap/awesome-livecoding.

Toussaint, Godfried. 2005. “The Euclidean Algorithm Generates Traditional Musical Rhythms.” In *Renaissance Banff: Mathematics, Music, Art, Culture*, edited by Reza Sarhangi and Robert V. Moody, 47–56. Southwestern College, Winfield, Kansas: Bridges Conference. http://archive.bridgesmathart.org/2005/bridges2005-47.html.

Walshaw, Christopher. 2011. “The Abc Music Standard 2.1.” *URL: Http://Abcnotation.com/Wiki/Abc: Standard: V2* 1.

Warkentin, Emily Grace. 2022. “The Story of the Ziffersystem and the Russian Mennonites: Counting Blessings.”