ZIFFERS (1/2)

Ziffers is a generative musical notation and library for algorithmic composition inspired by numerical notation, text based notations, and live coding mini notations.

INSTALLATION AND EXAMPLE SETUP

Go to your home folder (or some other location) and clone Ziffers from Github:

```
git clone https://github.com/amiika/ziffers.git
```

Require Ziffers in Sonic Pi buffer or in the init file that located in the home directory: *"~/.sonic-pi/init.rb":*

```
require "~/ziffers/ziffers.rb"
```

(or change ~ to your preferred location)

BASIC SYNTAX

Ziffers notation is written using numbers **0-9** or characters **T=10, E=11.** By default melodies are played in C Major. Examples:

```
zplay "0 1 2 3" # Play 0 1 2 3 sequentially (C D E F)
zplay "0 023" # Play 0 and then chord 023
```

Ziffers supports all the scales defined in Sonic Pi. See list of supported scales in Sonic Pi's help. Custom scales can be generated from the array of integers or floats represeting the steps in the scale.

```
zplay "0 1 2 3 4 5 6 7 8 9 T E", scale: :chromatic
zplay "0 023 3 468", key: :d, scale: :minor
zplay "q 0 1 2 3", scale: :minor
zplay "q 0 1 2 3", scale: :jiao
zplay "q 0 1 2 3", scale: [1,2,1.234,2] # Custom scale
```

NOTE LENGTHS: Note lengths are denoted using lower case letters, **w**=whole, **h**=half, **q**=quarter, **e**=eight note etc.. See full list in the right corner.

RESTS: Use **r** to create musical rest in the melodies. r can be combined with note length:
zplay "q 1 h r 2", key: :d

PITCH AND DEGREE MANIPULATIONS

SHARP AND FLATS: b for flat, **#** for sharp.

OCTAVE CHANGE: Use **^** (up) or _ (down), or numbers **<-1>** to set a specific octave. If using measures "**|**", the octave reseted after every bar.

NEGATIVE DEGREES: Negative degrees can be more intuitive and makes scale changes more pleasant:
zplay "-9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7"

ESCAPING DEGREES: Degrees that require more than one character can denoten using **{}** which will also evaluate a mathematical operation. Ruby string interpolation #{} is also possible:

```
zplay "{-10} {12} {24}" # Single escaped
zplay "0 5 6 {10 12 14}" # Multiple escaped notes
zplay "{2*2} {3*3}" # Eval math
```

MICROTONALITY: custom scales or escaped decimal notation:

```
zplay "q 0 3 2 3 7 2 0", key: 59.34,
scale: [0.3, 0.12, 1.234, 2.2], synth: :piano

zplay "h {0.4 0.6 2.7 2.3 5 4.6}",
key: :D, scale: :minor, synth: :kalimba
```

RHYTHMS

DECIMALS: Note lengths can be notated as decimals. **zplay "0.25 4 2"**

TIES: Tied note lengths are summed up for the next degree: **zplay "q 0 qe 2 3 4 qe 3 4"**

DOTTED NOTES: zplay "q. 0 1 2 q. 0 1 2"

SUBDIVISION: List notation subdivides the previous note length to equal proportions.
zplay "[4 2 4 2] [4 5 [4 2]] [3 [1 [3 1]]]"

TRIPLETS: two different ways:

```
zplay "q 2 6 a 1 3 2 q 5 1 a 4 3 2"
zplay "q 2 6 h [1 3 2] q 5 1 h [4 3 2]"
```

PARAMETERS: Default note length can be changed via a parameter. **zplay "1 2 3", release:0.25, sleep: 0.25**

RHYTHMIC MOTIVE: rhythms can be modified separately from the degree/note sequence using different options:

```
zplay "0 1 2 3", rhythm: [0.25, 0.25, 0.5]
zplay "0 1 2 3", rhythm: "qeeq"
zplay "1 2", rhythm: {1=>"h",2=>"q"}
zplay "1 2 3", rhythm: {hex: 0x0F }
zplay "0 1 2 3", rhythm: {major: 7, minor: 3}
```

EUCLIDIAN RHYTHMS: Use special list syntax: **(onbeat)<pulses,steps,rotate>(offbeat)** where offbeat is optional and defaults to rest:
zplay "q (X)<2,4>", X: :bd_haus

Third parameter can be used to specify the pattern rotation: **zplay "q (X S)<6,9,3>**

Patterns can be alternated using lists:

```
zplay "q (X S)<2,4>", X: :bd_haus, S: :bd_zum
z1 "((e 4 2) (e 0 ?))<6,8>(q6 q(5,7))"
```

Every parameter in euclidean syntax can use generative syntax:

```
z1 "q (X)<[1,2],4>", X: :bd_haus
z2 "q (0 2)<[2,3],[4,5],[0,1,2]>(4 3)"
z3 "q (0 ?)<(1,4),(4,5)>(4 (3,4))"
z4 "(q 3 (2,5))<(1,3),5>([1,4,5] [2,0,5])"
```

PLAYING PATTERNS

ZPLAY: Play strings, integers or arrays:

```
zplay 1; zplay [1, 2, 3]; zplay "w1 h2 q3";
zplay [[1, 1], [2, 0.5], [3, 0.25]]
```

ZLOOP: Loops strings or preparsed patterns:

```
zloop :z1, "q (1..7)" # Named loop
z1 "q (1..7)" # Shorthand for above
```

Loops (**z1-z20**) are synced automatically to first loop **z1**, or can be synced manually using, **cue**, **wait** and **sync:**

```
zloop :foo, "0 1"
zloop :bar, "2 6", sync: :foo
z1 "q 1 2 3 4"
z2 "q 3 r 5" # Syncs to :z1 by default
z4 "0 1", stop: true # Stop loop
z4 "//0 1" # Alternate way to stop loop
```

Cues can be assigned in the middle of a melodic pattern for other loops to wait on:

```
z1 "q 1 3 2 4 C 4 3 2 1", C: {cue: :foo}
z2 "q 6 6 6 6", wait: :foo
```

MIDI: Support for midi and cc messages:

```
z1 "0 1 2", port: "foo", channel: 2
z2 "q 60 70 80", port: "foo", cc: 80
```

PARSING MEASURES, BARS AND REPETITIONS

ZPARSE: Parse patterns using using **zparse**

```
m = zparse "1 2 3", rotate: 2
print m.notes # Print parsed notes
```

MEASURES: The **|** symbol is used to create measures, which also reset the octave and note lengths.

```
m = zparse "| q 0 1 2 3 | q 3 2 e 4 2 4 2 |"
print m.measures[0].pcs # Print pitch classes
```

REPEATS: [: ... :] is used for repetition, and can work recursively. **[: [: 1 2 :] 3 :]**

ALTERNATE SECTIONS: < part ; part > for alternating between two different sections **in repeated parts only.**

CHORDS AND ARPEGGIOS

CHORDS: Use degrees or roman numerals i-iv:

```
zplay "i 1 2 3 ii vi v 026"
zplay "i 1 2 3 ii vi v 026", chord_key: :D
```

CHORD DEGREES: Build large chords with additional degrees: **zplay "i+7"**

CHORD INVERSION: Chords can be inverted using **%**:
zplay "i%2 ii%-2"

CHORD NAMES: chords can also be named and combined with inversion
z1 "i^maj i^min iv^maj9%-4"

ARPEGGIOS: playing chord arpeggios is done using **@()** notation. You can use subset of ziffers notation to denote chord notes and note lengths:

```
z1 "@(q 0 1 e 2 1 0 1) [: i^7 :3] [: iv^dim7%-1 :3]",
key: :d4, scale: :mixolydian
```

PARAMETRIC MODULATION

FADING VOLUME: The fade parameter can be used for amplitude fading (e.g. **zplay "0 1", fade: 1..0**). To fade longer, use **fade_in** or **fade_out** to define how many loops the fading takes, for example:
z1 "q 0 2 0 1 4", fade: 0..0.5, fade_in: 5

The **fader** parameter can help to define the type of fading you want (cubic, linear, etc..):
z1 "0 1", fade: 0..1, fade_in: 3, fader: :expo.

PARAMETRIC CHANGES: Values can be changed by defining arrays, rings, Sonic Pi's methods or using **tweak**, for example:

```
z1 "0 1 2 3", add: [-1,2,0,2]
z2 "q 0 1", pan: tweak(:quint,-1,1,7).ring.mirror
z3 "q 0 3 2 1", synth: :fm, release: 0.1,
pan: range(-1,1,step: 0.1).mirror,
cutoff: range(60,120).mirror
```

LAMBDA FUNCTIONS: Lambda functions can be used for changing any parameter at any time. **z1 "q 0 1 2", release: ->(){rrand_i(0,1)}.** Some parameters (?) cannot be changed yet with lambdas. Post issue to Github if you want something to be working like you want :)

TWEAK: creates an array that can be used for fading in, out, or for changing any parameter. Current options are: **:sine, :quad, :cubic, :quart, :quint, :expo, :circ, :back, :bounce,** by using following pattern: **tweak(<option>, <start>, <end>, <length>)**, for example:
print tweak(:quad, 0.4,1.0,10)

SAMPLES AND CONTROL CHARACTERS

SAMPLES: Samples can be assigned to letters from Sonic pi or loaded from a directory, for example:

```
z1 "[D [D D]]", D: :bass_woodsy_c

samples = {X: :bd_tek, O: :drum_snare_soft}
z2 "q X O e X X q O", use: samples

z3 "h A", A: {sample: "~/samples/kick/kick 01.wav"}
```

ACCENTS: ´ and ` are shorthands for doubling or halving the amp: **zplay "0 `1 ´2 ``1 ´´2".**

STACCATO: ' is a shorthand for halving the release or shortening the sample: **zplay "q 1 '1 A 'A"**

SLIDE: ~ starts the slide. Slide speed can be specified with **<>: zplay "q ~0123 ~<0.1>01910"**

CONTROL CHARACTERS: Upper case characters with escaped values (e.g. **A<0.5>**) are used to control some melodic properties

A: amp, **E:** env_curve, **C:** attack, **P:** pan,
D: decay, **S:** sustain, **R:** release, **Z:** sleep,
X: chordSleep, **I:** pitch, **K:** key, **L:** scale.

```
zplay "q C<0.1> 0 1 2 C<2.0> 0 1 2"
zplay "q 0 3 5 L<minor> 0 3 5"
```

EFFECTS

Ziffers methods like **zplay, zloop, z1,** etc.. can be run with Sonic Pi FX (and some other blocks) by defining run arrays. Benefit of using run compared to normal block is that effects with run can be changed during the the loop. Run works with FX effects and following blocks. See parameters from Sonic Pi documentation: **with_fx, with_bpm, density, with_swing, with_octave.** Examples:

```
zplay "|h 1 2 3 4 | 4 3 2 1|", run: [
  {with_bpm: 120}, {with_fx: :echo},
  {with_fx: :bitcrusher, bits: 5 }]
```

Effects can also be adjusted on the fly using ring or array. Rings can be used to create continuous change and arrays as fade in or fade out. Normally using run the adjusting happens at the start of the loop, alternatively run_each (or apply&run) will adjust the parameters for each degree or sample.

```
# Phase parameter (start of the loop)
z1 "q 1 3 2 4", run: [{with_fx: :ixi_techno,
phase: tweak(:sine, 0.01,1.0,10).ring.mirror}]

# Phase parameter (each degree in the melody)
z2 "e 0 1 2 3 4 5 6 7 8 7 6 5 4 3 2 1 0",
run_each: [{with_fx: :flanger, phase: tweak(:sine,
0.05,1.0,10).ring.mirror}]
```

Effects can also be applied to single samples using run:

```
z1 "B K (B B) K", run: [{with_bpm: 120}],
use: {B: :bd_fat, K: { sample: :drum_snare_soft,
run: [{with_fx: :echo}] }}
```

Or all of the samples (within use-parameter):

```
z1 "B K (B B) K", run: [{with_bpm: 120}],
use: {B: :bd_fat, K: { sample: :drum_snare_soft },
run: [{with_fx: :echo}]}
```

NOTE LENGTH	
m	8
k	⌐3¬
l	4
p	⌐3¬
d	2
c	⌐3¬
w	1
y	⌐3¬
h	0.5
n	⌐3¬
q	0.25
a	⌐3¬
e	0.125
f	⌐3¬
s	0.06
x	⌐3¬
t	0.04
g	⌐3¬
u	0.015
j	⌐3¬
o	0.005
z	0

(approx..)

ZIFFERS (2/2)

Ziffers evaluates generative syntax first and applies changes as string replacements. Final string is parsed after all generative syntax is unrolled before parsing the melody.

BASIC GENERATIVE SYNTAX

RANDOM PITCHES: Use **?** that randomizes notes within one octave: **zplay "? ? ?"**

Random notes can also be randomized from given range: **z1 "q (-4,3)"**

CHOOSE RANDOM FROM ARRAY: Random notes or note lengths can be randomized by random array notation: **zplay "[h,q,e] [1,3,4] [h2,q2]"**

RANDOM PERCENT: Percents (random between (0.0-1.0) can be used as note lengths or in conditional statements:
zplay "% ? % ?"

RECURSIVE SYNTAX: Everything in Ziffers can be combined recursively, for example:
z1 "[q1,e [2 1,4 (0,3)],[q3,q4,e 4 ?]]"

EVALUATION AND CONDITIONALS

EVALUATION: Mathematical equations can be evaluated escaping the notation in **{}**, ex:
z1 "q{(1,4)*2^3}"

CONDITIONALS: Values can be assigned randomly by using conditionals:

```
z1 "{%>0.75?0:1}"
z2 "{(1,6)>3?(1,3):(3,6)}"
z3 "0 {(1,3)==3?6}" # 2nd param is optional
z4 "{%>0.5?{1+2}:3}" # Eval inside condition
```

SEQUENCES AND LISTS

SEQUENCES: Different number sequences can be generated with n..n syntax:

```
zplay "1..7" # Sequence: 1 2 3 4 5 6 7
zplay "q 0..(1,7)" # 0 to random
zplay "0..6+2" # Sequence: 0 2 4 6
zplay "0..6+4" # Sequence: 0 4
zplay "0..6*4" # Sequence: 0 4 8 12 16 20
zplay "0..4**2" # Geometric sequence: 1 2 4 8
zplay "-3..4*3" # Sequence: -3 0 3 6
```

LISTS: Lists can be created to modify and combine lists with other lists using operators. See list of supported operators on the right. Examples:

```
zplay "(0 1 2 3)+2" # Add 2 to each
zplay "q (2..4)+(0..7)" # Product of 2 lists
zplay "q ((1..8)*(1 3 2 4))%7" # Prod & mod
zplay  "((: (3,6) :6)^(0 2 4))" # Loop & xor
```

LIST REPEAT: List repeat differs from normal repeat by generating randomized values for each cycle

```
zplay "(: 0 1 ? 3 :4)" # Repeat list 4 times
zplay "(: 0 {%>0.5?2:3} ? (1,4) :4)"
```

LIST ASSIGNATION: Lists can be assigned to capital letters to repeat the evaluated values multiple times:

```
zplay "A=(0 1 2) B=((1,6) 3 2) A B B A"
```

LIST SHORTHANDS & METHODS

LIST SHORTHANDS: Shorthands for frequently used things, like unique set and transforming between single pitches and chords:

```
zplay "(0 1 1 2)!" # ! Unique set: 0 1 2
zplay "(1 2 3 4)@" # @ Reflect set. Nice for loops
zplay "(123)$" # $ Pitches 123 -> 1 2 3
zplay "(10..12)&" # & Chords 10 11 12
zplay "((1000,5000))&" # Generates a chord: 1243
```

LIST METHODS: All list methods & transformations can be called from the string:

```
zplay "(1 2 3).transpose(4)"
zplay "(1 2 3).rotate(2)"
zplay "(0..7).retrograde(2)"
```

LIST FUNCTIONS

LIST FUNCTIONS: List functions can be used to map values based on given conditions. List values are represented as x:

```
zplay "(-3..4){x**4}"
zplay "(0..7){x%2==0?x+2}"
zplay "(0..7){x%3==0?x-1:x+1}"
zplay "((1,5)){x<2?4}"
zplay "(: (1,6) :5){x>3?x+10}"
zplay "(0..20){x>7?x-(1,7):x+2}"
zplay "(0..10){x>3?x-(1,4):x+(1,4)}"
```

POLYNOMIAL FUNCTIONS: Polynomial functions can include term multiplications like **3x** and **(2x)(3x)**:

```
zplay "q (0 1 2){2x-1}" # 1 term
zplay "q (0 1 2){(2x-1)(2x**2-4)}" # 2 terms
zplay "q (0 1 2){(2x)-(3x)}"
z1 "q (0 1 2){((1,2)x-1)(x-[4,1,-1])}"
z2 "q ((1,4)){(2x-1)+(2x-(4,6))+(x-(1,4))}"
```

Shorthands are useful safeguards for polynomials and other functions:

```
zplay "(0..10){(2x)(3x)}" # Without safeguard
zplay "((0..10){(2x)(3x)})$" # With safeguard
z1 "q ((0..7){x%3==0?x+(1,3):x-3})@" # Mirror
```

COMBINATORICS

Combinatoric patterns are created on the fly using zplay or zloop.

PERMUTATION: Create all possible permutations. It is possible to play very large permutations that would last for years.

```
zplay "e 0 1 2 3 4 5 6 7 8 9 E T", scale:
:chromatic, permutation: 12
```

Combinatoric patterns can be combined with transformations

```
z1 "q 0 1 2 3", permutation: 2,
add: [0,-1,2]
# repeated: true to include same values
```

COMBINATION: Combination creates unique combinations of the given value:

```
z1 "q 0 1 2 3", combination: 2
# repeated: true to include same values
```

ITERATION: Iterate played melody in chunks:

```
z1 "q 0 1 2 3", iteration: 2
z2 "q 0 1 2 3 4", iteration: 3
```

RULES

Rules is a string replacement system that can be used to fractal melodies and recursive patterns with ziffers syntax:

```
zplay "q 1 3 1 2 1",
rules: {"1"=>"1 3","3"=>"3 ?"}, gen: 3
```

REGEXP: Include regular expressions for partial matching:

```
zplay "1",
rules: {
 /(3)1/ => "q ={$1+1} 1 ={$1+2}",
 /[1-7]/ => "e313"
}, gen: 4
```

STOCHASTIC: Stochastic rules using ziffers syntax or lambdas in the values:

```
zplay "q 1 2 3 4",
rules: {"1"=>"{%>0.2?(1..9):6}"}, gen: 9

zplay "q 1 2 3 4",
rules: {/[1-7]/=>"{%>0.3?(0,3):(4,7)}"},
gen: 3
```

TRANSFORMATIONS

TRANSFORMATIONS: Melodic patterns can be parsed to hash arrays using **zparse** and transformed and combined to other patterns, for example:

```
use_synth :piano
a = zparse "q 0 2 1 4"
b = a.inverse
c = a.retrograde
d = a.inverse.retrograde
e = a.transpose -3
zplay (a*2+b*2+c*2+d*2+b+e)*2
```

See full list of transformations in Ziffers wiki. Transforms can also be used with zplay or zloop, for example:

```
zplay "0 024 2 3", retrograde: true
zplay "0 1 2 3", transpose: -2
zplay "0 1 2 3", inverse: 1
zplay "0 1 2 3", swap: 1
zplay "0 1 2 3", mirror: true
zplay "0 1 2 3", reflect: true
zplay "0 1 2", zip: "3 4 5"
z1 "0 1 2 3", store:true, rotate: 1
z1 "q 0 3 6 8", add: [1,2,-1]
z1 "q 0 1 2 3 4", shuffle: 2..3
```

LOOP TRANSFORMATIONS: Transformations can be applied to certain cycles or pitches

```
z1 "q 0 2 1 3",
cycle: [ # Every 2, 3 and 8
 {at: 2, retrograde: true},
 {at: 3, add: -2},
 {at: 8, transpose: ->(){rrand_i(-1,4)}}]

z1 "q 0 2 1 3",
cycle: [ # Between 1..2 in total of 6
{range: 1..2, at: 6, retrograde: true}]

z3 "q 0 3 6 9", apply: {at: [1,3],
harmonize: [2,4]}
```

TONNETZ: Create chord progressions using Neo-Riemannian operations:

```
zplay "i", tonnetz: "p l lr pr lpr rlpl"
z1 "i v", tonnetz: "pl r", store: true
z1 "i", cycle: [{at: 2, tonnetz: "lr" }]
```

ENUMERABLES

Enumerables are something that can be calculated and played on the fly (which is more fun).

INTEGERS AS SEQUENCES: Consider a number 369420. This number can be played sequentially as a melody or concurrently as a chord. By default integers are played sequentially. To play integer as a chord use **parse_chords: true** parameter:

```
zplay rrand_i(1343,1241)
zplay rrand_i(10,20), parse_chords: true
```

INTEGER SEQUENCES: Musically interesting integer sequences can be generated as ruby enumerations and played using **zplay** or **zloop.**

New enumerators can be created using ruby ranges or Enumerator class:

```
m=(1..Float::INFINITY).lazy.collect {|x| x*x}
# To infinity and beyond!
zplay m, sleep: 0.125
```

There are plenty of preprogrammed ranges to play with, for example:

```
zplay pi, rhythm: [0.125,0.125,0.25]
zplay phi, rhythm: "eeqq"*2+"hqq"*2
zplay euler, sleep: 0.25
zplay (collatz 987654321), rhythm: "qqeeee"
z1 reverse_sum(313), rhythm: "qqh"
```

You can also get certain number of values from any sequences using **take**:

```
zplay recaman.take(12), sleep: 0.25
```

CELLULAR AUTOMATA: 2-dimensional cellular automatas can be generated and played using **live_cells or dead_cells** methods. Ziffers plays the integers of the living or the dead cells:

```
zplay live_cells("0101101", 245), sleep: 0.25

zplay dead_cells("1010001", 14),
sleep: 0.25, scale: :minor
```

MARKOV CHAINS: Two types of markov chain generators are implemented. One to create chain from integer or hex string and one for analyzing existing ziffers. Markov generator interprets the input as a chain and generates

```
zplay (markov_generator 12345), sleep: 0.25

zplay markov_generator("034A34F3562G3"),
rhythm: "hqq"*2+"qqeeee"*2
```

Markov chain generator can be used to generate new melodies from parsed melodies. Chain order can be used to determine how similar the generated melody is to the original:

```
m = zparse "q 0 0 4 4 5 5 h4 q 3 3 2 2 1 1 h0"

# (melody, chain order, start index)
zplay markov_analyzer m, 1, 0
```

New melodies can also be created by taking certain number of elements from the enumerables:

```
new_melody = (markov_analyzer(m,1,0).take(10)
```

See **Ruby Enumerator class** for more methods and ideas how to use and create new enumberables.

RANDOM NOTATION	
?	RANDOM INTEGER
%	RANDOM 0.01-1.0
[q4,e2]	RANDOM SELECTION
(1,7)	INTEGER BETWEEN
(1.0,2.0)	DECIMAL BETWEEN
-1..5	RANGE BETWEEN

LIST OPERATORS	
()	LIST
+	ADDITION
-	SUBSTRACTION
*	MULTIPLICATION
/	DIVISION
%	MODULUS
**	EXPONENT
&	AND
|	OR
^	XOR
<<	SHIFT LEFT
>>	SHIFT RIGHT