Music Notation with SuperCollider

Bernardo Barros

May, 2013

Outline

1. About myself
2. Predecessors: PatchWork/OpenMusic/PWGL family
3. LilyPond, FOMUS
4. SuperFomus use case: Sieves and set operations (Xenakis)
5. LilyCollider use case: Rhythm Trees & List Comprehensions
6. Future work: LilyCollider 2

About myself

- I spend most of my time composing (instrumental and electronic works) and hacking my live-electronics setup for performances.
- I code as I need the features for my own work.
- When it makes sense I share useful tools with others.

Motivation

- I was a OpenMusic user.
- It stopped working with MacIntels, and in the meanwhile I moved to Linux.
- I didn't find a replacement for it.
- The programming language I knew best was SuperCollider, since I've been using it a lot for my electronic work.
- I slowly started to try ways to visualize music notation with SuperCollider

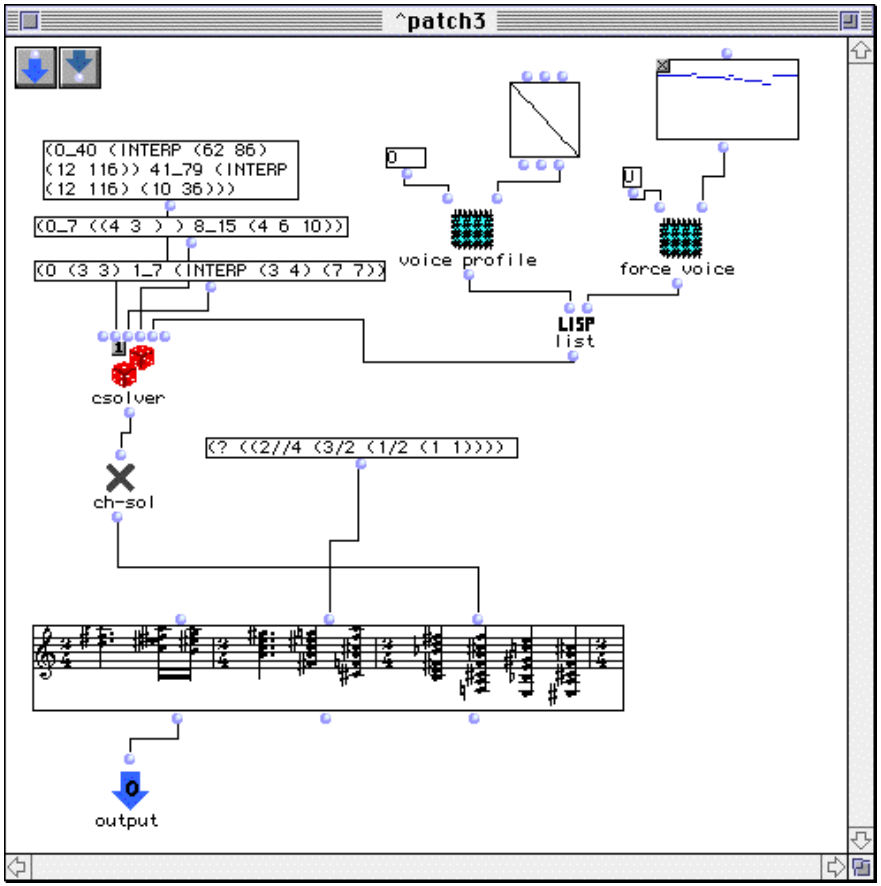
OpenMusic (1)

So what is OpenMusic?

- Same family of PatchWork and PWGL.
- Common Lisp.
- Those are visual programming languages very convenient for algorithmic music composition.
- Various classes implementing musical data are provided.
- They are associated with graphical editors to visualize music materials.

OpenMusic (2)

All the programming part are already implemented or can be easily implemented in SuperCollider. The missing part is the graphical element used to visualize music materials:



How to do it?

- How to make a similar tool for score visualization in SuperCollider?
- My guess is that LilyPond is the right tool.
- There are other alternatives, with advantages and disadvantages.
- For instance, INScore[1] would be faster, but has no automated music engraving, maybe it would be better for live performances.

[1] http://inscore.sourceforge.net/

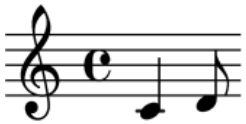
LilyPond

- LilyPond[2] is probably the best candidate to handle automatic music notation.
- It's cross-platform (Linux, Windows, Mac), just as SuperCollider.
- It's Free Software (GNU/GPL)
- The input is text-based, easy to be algorithmically generated.
- Drawback: rendering time, you have to wait until the score is engraved (not much of a problem for most cases).

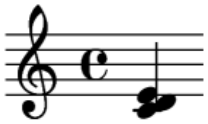
[2] http://lilypond.org

LilyPond Syntax

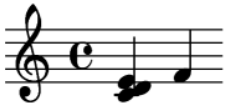
- sequential elements: { c'4 d'8 }



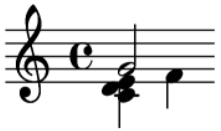
- simultaneously elements: <<c4 d4 e4>>



- combined: { <<c4 d4 e4>> f4 }



- combined: << { <<c4 d4 e4>> f4 } g2 >>



LilyPond + SuperCollider

3 different projects:

- SuperFomus
- LilyCollider 1 (it did the job for what I needed at the time)
- LilyCollider 2 (being written from scratch, the idea is to be more flexible and general)

SuperFomus: automatic quantization

- Use FOMUS by David Psenicka [3]
- FOMUS works with a list of music events with duration values.
- It does the metric quantization for you.
- SuperFomus works with List of Events and Patterns.
- It also exports MusicXML (readable by MuseScore, Finale, Sibelius and others) and MIDI.
- Not the tool if you want to work with metric structures!

[3] http://fomus.sourceforge.net/

SuperFomus: simple example (1)

The most simple and straightforward example:

```
a = [
  ('midinote': 60,   'dur': 0.5),
  ('midinote': 66.5, 'dur': 1.0),
  ('midinote': 70,   'dur': 0.25)
];
f = Fomus();
f.put(a);
f.ly;
f.xml;
```

example: superfomus1.scd

SuperFomus: Microtonal Chords (2)

Another one:

```
a = 12.collect({|i|
  ( 'midinote': (
    (63 + rrand(-6,6.5) + [0,8,13]) ++
    (60 + rrand(-6,6.5) + [0,5,6,9])),
    'dur': 1 )});
f = Fomus();
f.put(a);
f.ly;
```

example: superfomus2.scd

SuperFomus: Patterns (3)

With patterns:

```
p = Pbind(
  \midinote, Prand((60,60.5..80), inf),
  \dur, Prand([0.125, 0.25, 0.5], inf));
f = Fomus(p.asStream, 30);
f.ly;
```

example: superfomus3.scd

SuperFomus: Patterns (4)

Adapted from James Harkins' "A Practical Guide to Patterns":

```
p = Pbind(
  \midinote, Pif(Pwhite(0.0, 1.0, inf)
      < 0.7, Pwhite(60, 80.5, inf),
      Pwhite(45, 55.5, inf)),
  \dur, Prand([0.125, 0.25, 0.5], inf) );
p.play;
f.put(p.asStream, 40);
f.ly; // LilyPond
f.midi; // MIDI file
f.xml; // MusicXML
```

example: superfomus4.scd

SuperFomus: Sieves (Xenakis)(5)

"every well-ordered set that can be represented as points on a line, if it is given a reference point for the origin and a length u for the unit distance, and this is a sieve"[4].
Sieves were first discussed in the final section of "Towards a Metamusic".

Sieves are based on modulus operation:

Example

$3_2 = 2, 5, 8, 11, 14, \ldots$
$4_2 = 2, 6, 10, 14, \ldots$

[4] Xenakis, Formalized Music, 1992, p. 268

SuperFomus: Sieves (6)

If we treat it as sets, we can apply logical operations upon those sets:

+ *union*
* *intersection*
- *complementation*

Example

$3_2 + 4_2 = 2, 5, 6, 8, 10, 11, 14, \ldots$
$3_2 * 4_2 = 2, 14, \ldots$

SuperFomus: Sieves (7)

In SuperCollider our example $3_2 + 4_2$ can be expressed as a *list comprehension:*

```
a = {: x+2, x <- inf, (x%4==0) || (x%3==0) };
a.nextN(20);
```

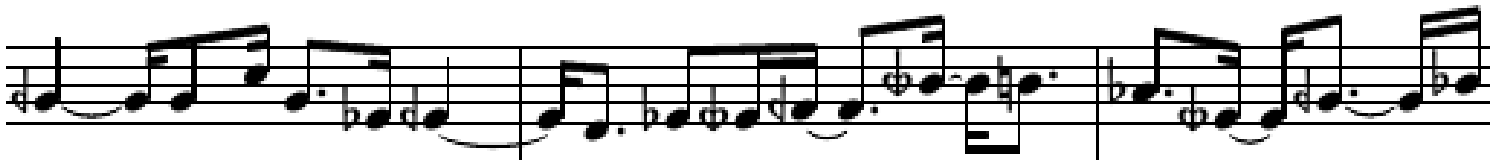
And also with *Set* operations:

```
a = all {: x+2, x <- (0..40), (x%4)==0 || (x%3)==0 };
b = all {: x+2, x <- (0..40), (x%3)==0 || (x%5)==0 };
x = a.asSet | b.asSet; x.asArray.sort;
=> [ 5, 7, 8, 11, 12, 17, 20, 22, 23, 27, 29, 32, 35, 37, 41, 42 ]
y = a.asSet - b.asSet; y.asArray.sort;
=> [ 5, 8, 11, 17, 20, 23, 29, 32, 35, 41 ]
```

SuperFomus: Sieves (8)

This Sieve can be very easily visualized in traditional western music notation using SuperFomus:

```
z = x.asArray.sort.differentiate;
a = z.collect({|i| ('midinote': rrand(60,72.5), 'dur': i/4 )});
f = Fomus();
f.put(a);
f.ly;
```



example: superfomus5.scd

SuperFomus (9)

- SuperFomus can be a solution for some cases.
- It has limitations, specially with more advanced rhythm algorithms
- It knows very little about music metric.

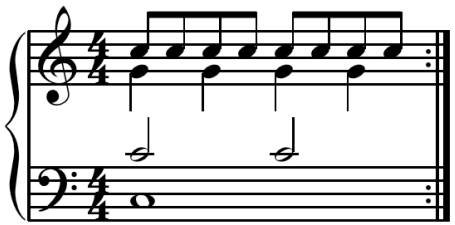
Division and Addition models to Rhythm (1)

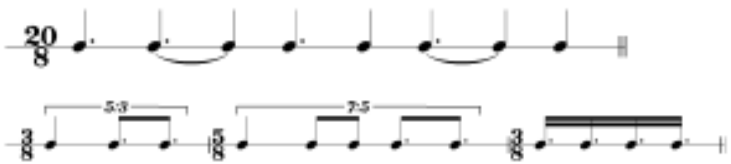
- Paul Nauert talks[5] about those two methods of rhythm construction: *division*- and *addition*-based models to rhythm construction.
- The *sieve* example can be characterized as a purely addition-based rhythm construction, we're defining points on a line.

[5] "Division- and Addition-Based Models of Rhythm in a Computer-Assisted Composition System," Computer Music Journal 31.4 (2007): 56-70

Division and Addition models to Rhythm (2)

- Division-based models are also possible.
- The principle is also simple: a larger period of time is divided into smaller rhythmic units (equal or not).





Division and Addition models to Rhythm (3)

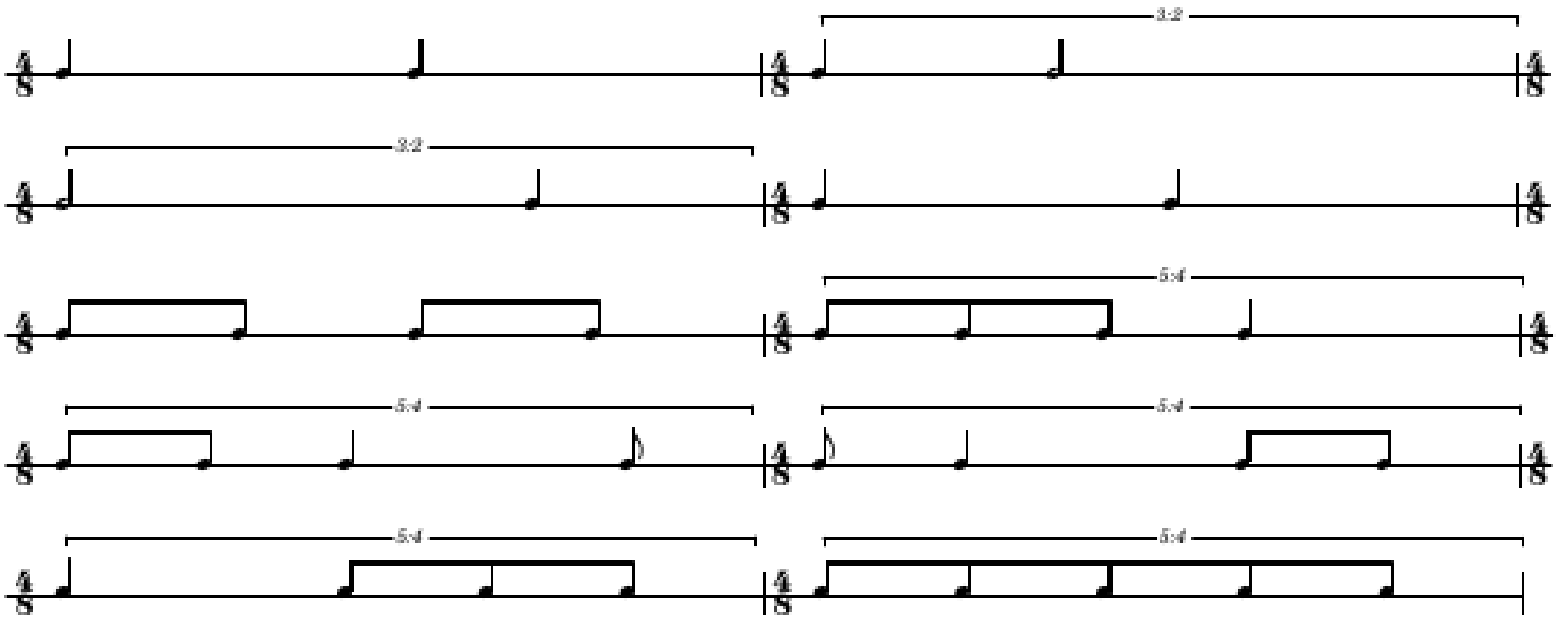
- There are more adequate data types to represent this kind of construction.
- For instance the one created by Mikael Laurson for PatchWork: *Rhythm Trees.*
- *Rhythm Trees* can efficiently represent rhythms based on divisions of a pulse according to proportions.
- It allows most traditional Western music notation structures (not all, they don't allow partial tuplets used by some composers such as John Cage in *Music of Changes*, Michael Finnissy and others).

Rhythm Trees: definition

- So, what is a *rhythm tree*?
- RT is a array in the form $[x, y]$ where:
  - x is the "beat-counter" (a duration)
  - y is the "rtm-list", a list of proportions that takes place within the duration x
- This structure can be arbitrarily nested.

Rhythm Trees: example 1

- Example: the proportions `[ [ 1, 1 ], [ 1, 2 ], [ 2, 1 ], [ 2, 2 ], [ 1, 1, 1, 1 ], [ 1, 1, 1, 2 ], [ 1, 1, 2, 1 ], [ 1, 2, 1, 1 ], [ 2, 1, 1, 1 ], [ 1, 1, 1, 1, 1 ] ]` applied to 4/8 bars, will look like this:



LilyCollider 1

- *LilyCollider 1* uses the *rhythm trees* as the only way to construct rhythms.
- It's a work in progress, and I haven't touched it recently, but it does the job.
- I've used it moslty for rhythm sketches for my instrumental compositions.
- It also implements pitch material, but SuperFomus can be a better option for this at this point, since it handles polyphony and cross-staff material somewhat better.
- I'm working on a second version from scratch, hopefully with a better design.

Rhythm Comprehensions: comprehensions applied towards divisive-based approach

List comprehension that generates all possibilities given:
a. proportions allowed *b.* number of elements *c.* sum of elements

```
~allCells = { |allowed, n, sum|
  var result;
  sum.do { |j|
    n.do { |i|
      result = result ++ all {:x,
        x <- (allowed!i).allTuples,
        x.sum == j }}}
  result };
~allCells.([1,2], [3, 4], [4])
=> [ [ 2, 1, 1 ], [ 1, 1, 2 ], [ 1, 1, 1, 1 ], [ 1, 2, 1 ] ]
```

Constructing Rhythm Trees with List Comprehensions (2)

```
~cells = ~allCells.(
  allowed: [1,2],
  n: [1, 2, 4],
  sum: [3, 4, 5] );
```

Result:

```
[ [ 1, 2 ], [ 2, 1 ], [ 2, 2 ], [ 1, 1, 1, 1 ],
  [ 1, 1, 1, 2 ], [ 1, 1, 2, 1 ], [ 1, 2, 1, 1 ],
  [ 2, 1, 1, 1 ] ]
```

example: rt1.scd

Constructing Rhythm Trees with List Comprehensions (3)

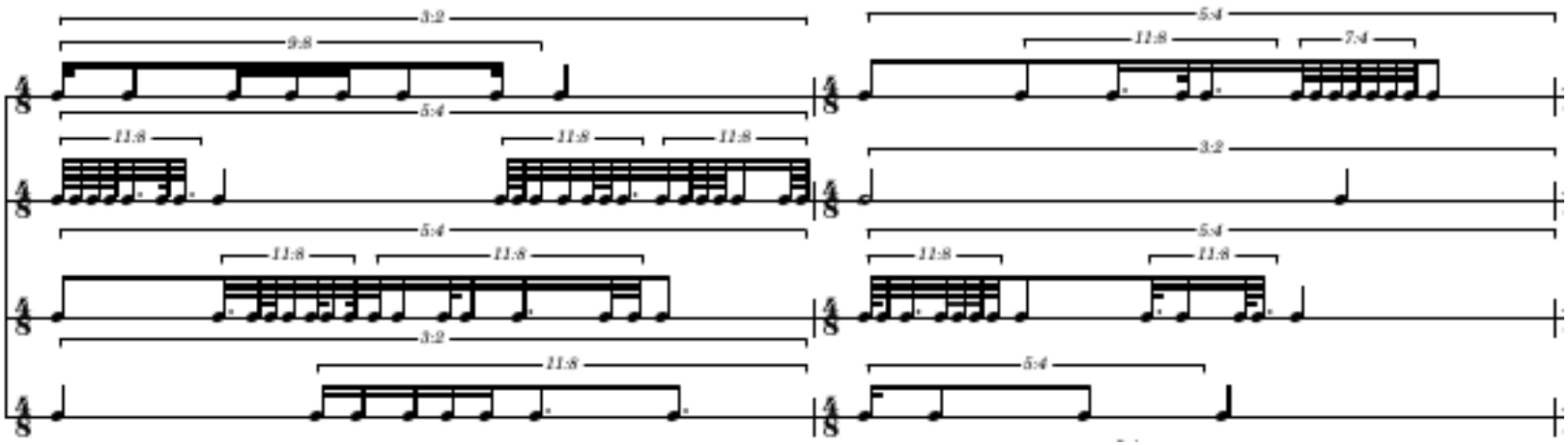
We can also define secondary subdivitions (nested rhythm trees):

```
~subCells = ~allCells.(
  allowed: [1,2, 3, 4],
  n: [3, 4, 7],
  sum: [3, 4, 5, 7, 9, 11] );
```

Substitution function:

```
~proportions = { ~cells.collect { |i|
  i.collect { |j|
    ~prob.coin.if {
      [j, ~subCells.choose]
    } {
      j
    }
}}};
```

Constructing Rhythm Trees with List Comprehensions (4)



example: rt2.scd

Constructing Rhythm Trees with List Comprehensions (5)

Primary partition with a simetri shape:

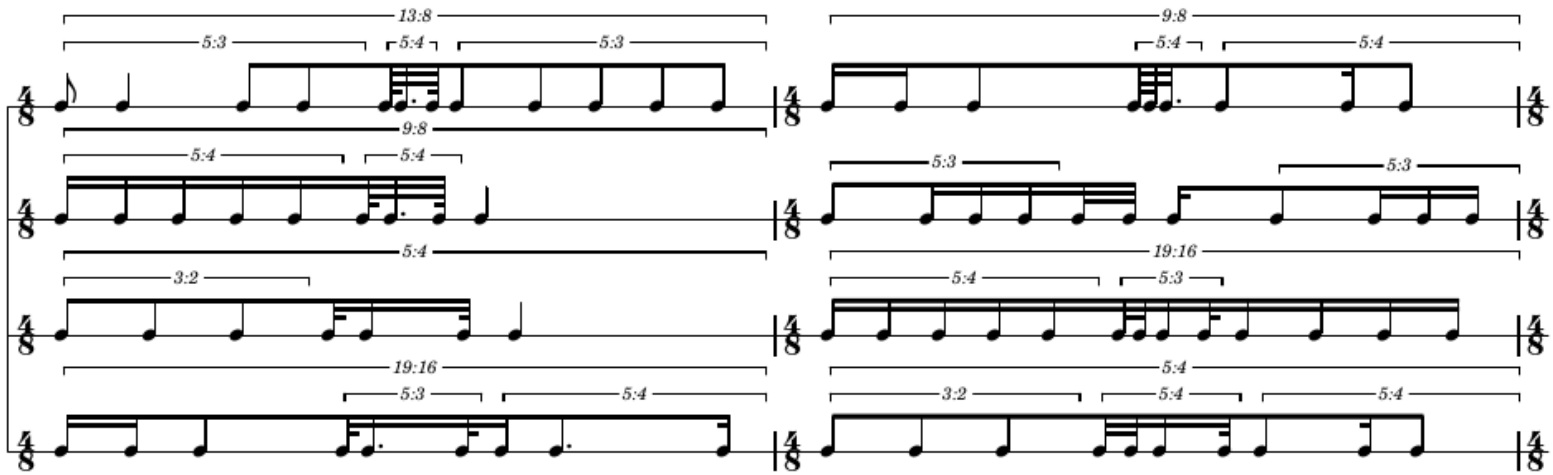
```
~cells = all {: [x,y,x],
  x <- [2,3,4,6,8],
  y <- (1..x),
  y != 5,
  (x+y).isPrime
};
```

The substitutive cells are the result of probabilities (zero order Markov chain)

```
a = [
  [ [ 1, 1, 1 ], 5 ],
  [ [ 1, 1, 2 ], 3 ],
  [ [ 1, 2, 1 ], 3 ],
  [ [ 2, 1, 1 ], 1 ],
  etc...
];
```

example: rf3.scd

Constructing Rhythm Trees with List Comprehensions (6)

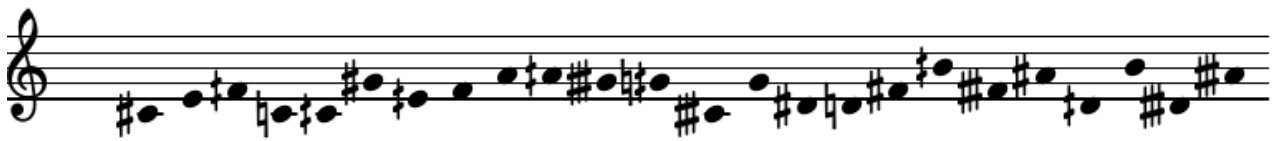


example: rf3.scd

LilyCollider 1: Pitch (4)

LilyCollider also does pitch material.

```
a = LilyPitch(0).plot;
a.template = "just-pitches";
a.plot;
a.lily_("fih"");
a.template = "doc";
a.plot;
a = LilyPitchSeq(Array.series(24, 0, 0.5));
a.template = "just-pitches";
a.scramble.plot;
```

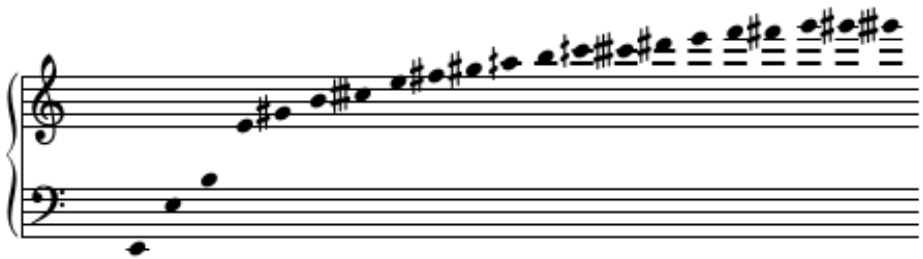


example: pitch1.scd

LilyCollider 1: Pitch (5)

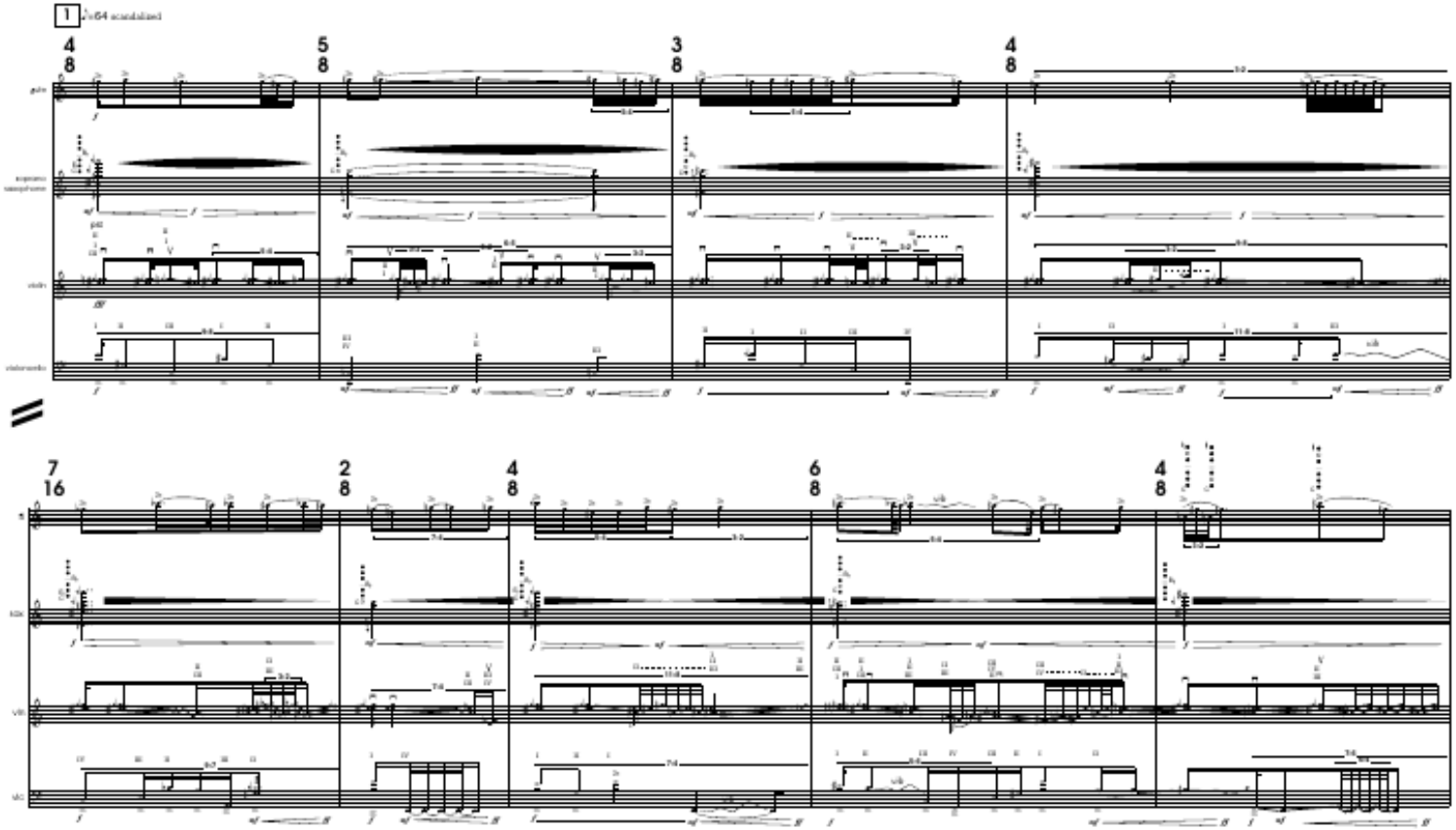
Handy methods:

```
a = LilyPitch(-20).makeHarmonicSeries(0, 24, 0.5);
a.asPitchSeq.plot // as pitch sequence
a.playMidi;
```

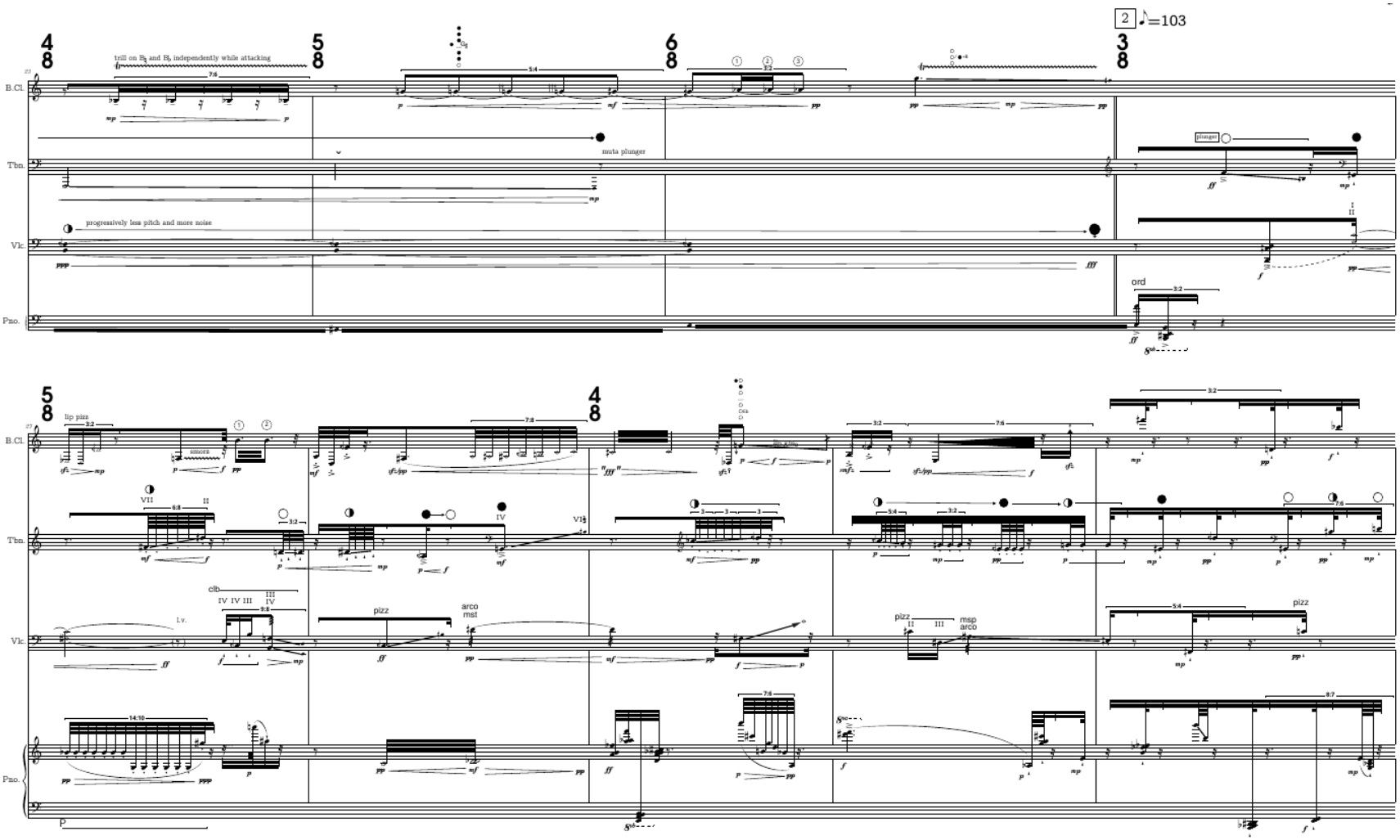


example: pitch2.scd

Compositions: resíduo (2012)



Compositions: catastrophe (2012)



LilyCollider 2 (1)

- LilyCollider 1 is frozen for many months. It did the job for what I needed.
- LilyCollider 2 will have a more direct mapping of LilyPond elements to classes/types in SuperCollider, instead of rendering only few high level objects directly to complex strings (bad design decision).

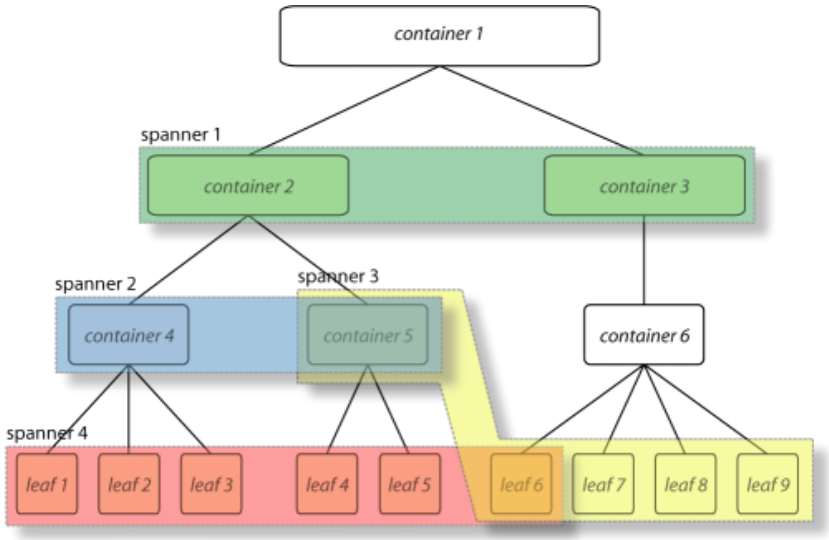
LilyCollider 2 (2)

- All durations will be rational numbers.
- Eighth-tone (48ET) and 12th-tone (72ET) microtonal intervals.
- Better LilyPond parsing, ideally aware of as much LilyPond elements as possible.

```
Lily("<c' des' gih'>") // chords
Lily("4. 8 8 8") // just the rhythm
Lily("3/2{ 8 8 8 }") // tuplets
Lily("fff mp f pp") // just dynamics
etc.
```

LilyCollider 2 (4)

- Containers and Spanners, inspired by Abjad[6]:
- Good way to implement expressions, crescendi, slurs etc.



[6] http://www.projectabjad.org/

Call for Help

- It would be good to have contributions to the second version.
- And also to receive feedback if an generic interface would be possible or preferable, as opposed to local and idiosyncratic solutions specific to each composer.

repos I

- *SuperFomus:* `https://github.com/smoge/superfomus`
- *LilyCollider (both branches):* `https://github.com/smoge/lilycollider`

Thank you!