Elements of Algorithmic Composition

Oscar Riveros

June 2, 2013

Introduccion

The algorithmic composition is the result of automating tedious, repetitive and complex traditional musical composition process, For example suppose we have a cantus firmus, and want to find a counterpoint of some kind, we can do it by hand, but the question is: does my solution or my counterpoint is one of few possible?, ie, we lose all the wealth of artistic taste, that to write all that counterpoints hand, we would not have life, then taking techniques such as constraint programming, we can model a meta-structure of our counterpoint, giving an intention background beyond of simple causality and human limitations, so we could ask for example apart from following the rules of strict counterpoint, there was some sort of stronger relationship between the material generated, say that the climax is in some hubicacion or under certain proportions, as the golden ratio, we could choose some notes that represent a more generic reason, and finally generated counterpoint would be a deliberate interpolation on our sketch, the only limit on algorithmic composition, is our knowledge.

Thus algorithmic composition differs from traditional that the first is limited to the human condition, and the second occurs at a very different transhuman plane where a target structure is modeled in technical terms is pure information, materialized in Music, in the traditional sense.

Largely an algorithmic composer is a fusion of a Master Composer Traditional to an Expert in Computer Science, so that our programs are such in the usual sense, but instead of having compiled code, we meta-music, either this MIDI, MusicXML, LilyPond, fomus, etc.

The techniques used transcend technologies and programming languages, and Algorithmic Composition is a disipline itself, giving rise to metdos results and not applicable in the traditional composition.

Getting Started

First we declare the top of our composition, in which we include the libraries to use, it all depends on what we do or have in mind, always good to have a preconceived idea, but if not possible, it is good to try new ideas in sketches , at some point in the process, have a good material to work with.

One of the things I want to stress upon and want to be categorical about it, NEVER USED FOR YOUR COPOSICIONES RANDOMNESS, the brain recognizes patterns and everything in music from the basic wave ring of an instrument are patterns, noise unlike music, that the noise is random, and the sound of music or some upper structure as a musical composition, it just depends on their quality, their dominant pattern, meaning you can create very complex music, but always be deterministic, in the sense of being rebuilt, is the only way that the brain considers an upper structure, and not simply a nice ambient sound.

```
# palette of libraries

from IPython.external import mathjax
from sympy import *
from sympy.utilities.iterables import variations
from sympy.plotting import plot3d
from music21 import *
from itertools import cycle

x, y, z, r, s, t                    = symbols('x y z r s t')
m, n, i, j, k                       = symbols('m n i j k', integer=True)
f, g, h, eq, plus, minus, div, times = symbols('f g h eq plus minus div times', cls=Function)
```

The following is define or subject with which we work, a motif is a melodic and rhythmic contour, but not for a particular group of notes, and depends on the context in which it develops, for example the same motif would behave differently, a chromatic context, that tonal context, but the ear would recognize both layers as realizations of the same motif.

In our implementation of the motif, we first define some basic functions that operate on the notes of the lattice, which we will define later, the framework are the threads that weave music and motif is the design that we put together with those threads.

```
# motif definition

eq    = Lambda((x, y), x)
plus  = Lambda((x, y), x + y)
minus = Lambda((x, y), x - y)
times = Lambda((x, y), x * y)
div   = Lambda((x, y), x / y)

motive_male =
[(eq, 0), (plus, 2), (plus, 2), (minus, 5), (plus, 3), (plus, 6), (plus, 7), (minus, 8)]
rhythm_male =
cycle([1./2, 1./2, 1./2, 1./2, 1./2, 1./1, 1./1, 1./1] * 2)

motive_female =
[(eq, 0), (plus, 3), (plus, 6), (minus, 7), (plus, 5), (plus, 2), (plus, 5), (minus, 7)]
rhythm_female =
cycle([1./2, 1./2, 1./2, 1./2, 1./2, 1./1, 1./1, 1./1][::-1] * 2)
```

Now we have the motif, we need an expander, which develops the motif on the network, it would like our knitter head that takes our basic design and extends along the threads defined by the network.

```
# motif expander

def framework(start, end, intervals):
    material = [start]
    tmp = start

    for index in range(start, end):
        if tmp < end:
            tmp = tmp + intervals.next()
            material = material + [tmp]

    return material


def expander(motive, length):
    return map(lambda x, y: x[0](x[1], y), motive * length, range(len(motive * length)))
```

Here we implement a generic maker parts, dependent on our chosen music library.

```
def make_part(part, material, material_maping, rhythm, rhythm_mapping, octave):

    map(part.append,
        map(lambda pc:
            note.Note(material_maping(pc),
                      quarterLength = rhythm_mapping(next(rhythm)),
                      octave = octave),
            material))

    return part
```

Now we define some global parameters, such as scales for areas, here is a rudimentary implementation, then create one with greater definition and detail, about the final result.

```
intervalic_pattern = cycle([3, 2, 2, 3, 2] * 2 + [2, 1, 2, 2, 2, 1, 2] * 2)
size               = 12

# global instruments

voice1 = stream.Part()
voice2 = stream.Part()
voice3 = stream.Part()
voice4 = stream.Part()

# global motives development

material        = framework(0, 127, intervalic_pattern)
voice1_material = expander(motive_male         , size)
voice2_material = expander(motive_female       , size)
voice3_material = expander(motive_male[::-1]   , size)
voice4_material = expander(motive_female[::-1], size)
```

We define special characteristics of each of the sections.

```
# Section A

voice1_material_maping = lambda x: list(material)[int(x**1) % len(material)]
voice2_material_maping = lambda x: list(material)[int(x**2) % len(material)]
voice3_material_maping = lambda x: list(material)[int(x**3) % len(material)]
voice4_material_maping = lambda x: list(material)[int(x**4) % len(material)]

voice1_rhythm_maping   = lambda x: float(x*1)
voice2_rhythm_maping   = lambda x: float(x*2)
voice3_rhythm_maping   = lambda x: float(x*3)
voice4_rhythm_maping   = lambda x: float(x*4)

voice1 = make_part(voice1, voice1_material, voice1_material_maping, rhythm_male, voice1_rhythm_maping, 5)
voice2 = make_part(voice2, voice2_material, voice2_material_maping, rhythm_male, voice2_rhythm_maping, 4)
voice3 = make_part(voice3, voice3_material, voice3_material_maping, rhythm_male, voice3_rhythm_maping, 3)
voice4 = make_part(voice4, voice4_material, voice4_material_maping, rhythm_male, voice4_rhythm_maping, 2)

# Section B

voice1_material_maping = lambda x: list(material)[int(x**4) % len(material)]
voice2_material_maping = lambda x: list(material)[int(x**3) % len(material)]
voice3_material_maping = lambda x: list(material)[int(x**2) % len(material)]
voice4_material_maping = lambda x: list(material)[int(x**1) % len(material)]
```

```
voice1_rhythm_maping    = lambda x: float(x*4)
voice2_rhythm_maping    = lambda x: float(x*3)
voice3_rhythm_maping    = lambda x: float(x*2)
voice4_rhythm_maping    = lambda x: float(x*1)

voice1 = make_part(voice1, voice1_material, voice1_material_maping, rhythm_male, voice1_rhythm_maping, 5)
voice2 = make_part(voice2, voice2_material, voice2_material_maping, rhythm_male, voice2_rhythm_maping, 4)
voice3 = make_part(voice3, voice3_material, voice3_material_maping, rhythm_male, voice3_rhythm_maping, 3)
voice4 = make_part(voice4, voice4_material, voice4_material_maping, rhythm_male, voice4_rhythm_maping, 2)
```

and finally materialize our meta-score

```
# Let's Put It All Together

new = stream.Stream()
new.insert(0, voice1)
new.insert(4, voice2)
new.insert(8, voice3)
new.insert(16,voice4)
new.show()
```

Full Source Code: https://github.com/maxtuno/Advanced-Algorithmic-Composition
Materialized Music: https://soundcloud.com/maxtuno/aac-ex102
Blog:
http://oscar-riveros.blogspot.com/
http://mx-clojure.blogspot.com/
Facebook Page: https://www.facebook.com/pages/Oscar-Riveros/207414032634682