LAURENT POTTIER
Département de Musicologie - CIEREC
Université Jean Monnet (Lyon-Saint-Etienne)

Pédagogie, recherche et création en musique avec ordinateur au département de musicologie de l'université Jean-Monnet de Saint-Étienne

ATELIERS D'INFORMATIQUE MUSICALE EN MUSICOLOGIE

Tous les ans, à l'université Jean-Monnet (UJM) de Saint-Étienne, les étudiants en deuxième année de licence de musicologie suivent pendant deux semestres, en deux groupes, un enseignement obligatoire intitulé « Atelier d'informatique musicale » de 12 heures TD + 12 heures TP par semestre.

Ils y abordent dans un premier temps les bases d'un langage de programmation spécifique adapté au traitement du son en temps réel ou à la composition assistée par ordinateur (CAO). Selon les années, il peut s'agir des langages **Max, Csound, Pure Data, Faust ou OpenMusic**.

Les étudiants apprennent à maîtriser des éléments de lutherie numérique pour construire des instruments de musique sur ordinateur – **synthétiseurs et processeurs d'effets** – qu'ils peuvent utiliser sur leurs propres ordinateurs ou smartphones. Ils composent collectivement – ou individuellement – une **pièce de musique électronique ou mixte** d'une dizaine de minutes, qu'ils doivent par la suite interpréter en direct, avec une présentation au public et/ou un enregistrement en studio en fin d'année.

>> APPRENTISSAGE DE MAX

L'enseignement du langage Max débute par la présentation de la différence existant entre les données – et les types associés (int, float, list, string) – et les fonctions. L'apprentissage de la **norme MIDI** se fait en parallèle en demandant aux étudiants de programmer un clavier maître – zones de split, changements de programmes, transposition, courbes de vélocité . Nous abordons la question des **algorithmes et des processus** qui permettent de produire de la musique automatiquement. Cela passe par exemple par la combinaison des fonctions metro et counter pour la réalisation d'un **chronomètre** (fig. 1) ou d'une **gamme par tons** (fig. 2). L'enseignement se poursuit par la construction d'un séquenceur rudimentaire ou d'une **boîte à rythme MIDI** (fig. 3).

La deuxième étape consiste à faire découvrir les bases de la synthèse sonore et du traitement du signal. La liaison avec les exercices précédents se fait en reliant le séquenceur à un synthétiseur possédant les fonctionnalités suivantes : plusieurs générateurs, avec des formes d'ondes variées, une ou plusieurs enveloppes, un filtre et un delay (fig. 4). À partir de là, les objets poly et poly~ sont présentés pour permettre de rendre les instruments polyphoniques (fig. 5). Un contrôle du pitch-bend et une modulation sont ajoutés.

Les enseignements se poursuivent en présentant la construction d'un sampleur polyphonique (fig. 6) ainsi qu'un programme de traitement du signal, sorte de multi-effet programmable (fig. 7). Sur un même ordinateur, il est possible d'ouvrir simultanément deux sampleurs indépendants et un multi-effet.

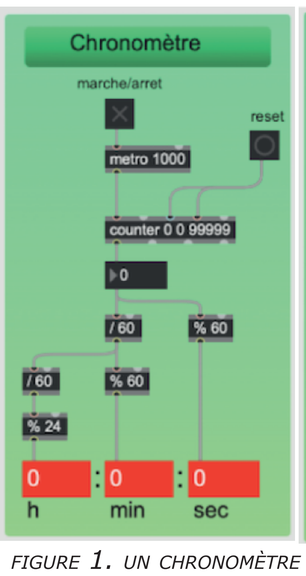

FIGURE 1. UN CHRONOMÈTRE

FIGURE 2. GÉNÉRATEUR DE GAMME

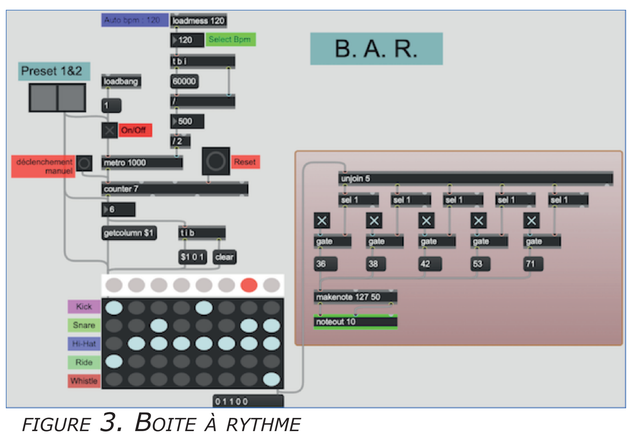

FIGURE 3. BOITE À RYTHME

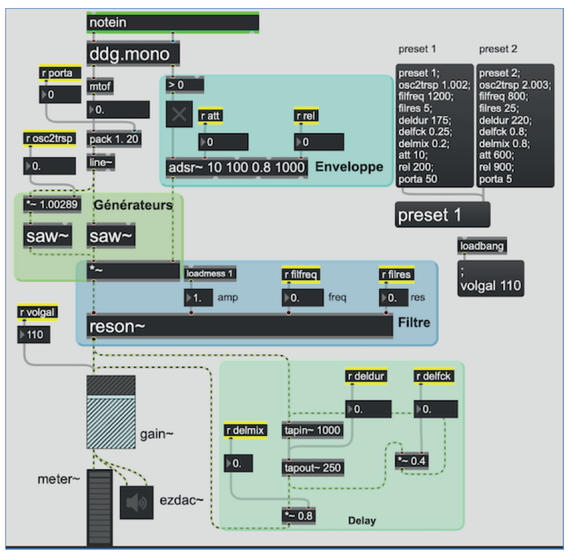

FIGURE 4. SYNTHÉTISEUR MONOPHONIQUE

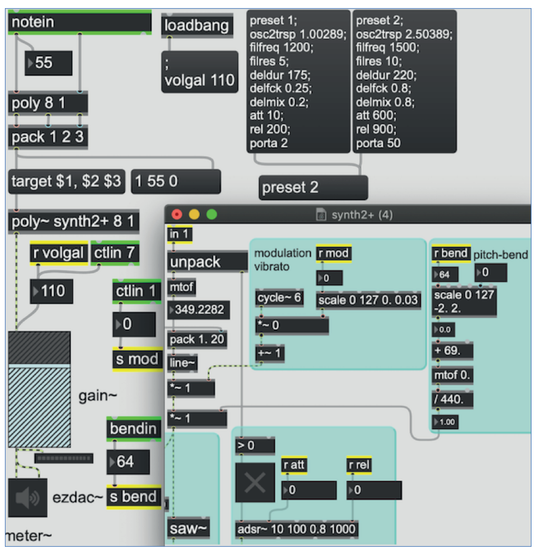

FIGURE 5. SYNTHÉTISEUR POLYPHONIQUE

FIGURE 6. ECHANTILLONNEUR

FIGURE 7. MULTI-EFFET

LES CREATIONS 2019-2020

En 2019-2020, le laboratoire **CIEREC** a bénéficié d'un financement de **l'IDEX de l'Université de Lyon** (UdL) pour inviter deux compositeurs sous la forme d'une résidence à l'UJM, en partenariat avec GRAME. Ces deux compositeurs, Vincent Carinola et Luis Quintana, ont encadré respectivement un groupe d'étudiants en Master 1 et un groupe d'étudiants en Licence 2 de musicologie pour leur faire produire chacun une pièce mixte interactive en vue d'une création lors du festival biennal **Musiques exploratoires** prévue le 22 mars 2020 au CNSMD de Lyon, en première partie d'un concert de l'Ensemble Orchestral Contemporain. En raison de la situation sanitaire, ce concert a été reporté en février 2021.

Le projet présenté dans cette section concerne un troisième groupe d'étudiant, inscrits en Licence 2, que nous avons encadrés nous-même.

La structure de la pièce a été imposée aux étudiants (fig. 8).

L'introduction est jouée avec des smartphones sur lesquels ont été placées deux applications réalisées avec le langage **Faust**. Il s'agit de synthétiseurs produisant un bruit blanc, disposant d'un filtre résonant et d'un système de granulation (fig. 10).

Les **sections A, B et C** sont jouées sur les sampleurs pour construire des textures qui évoluent progressivement. Les notes à jouer sont à prendre dans un réservoir de hauteurs. Les accords des sections A et B proviennent de la pièce *EnTrance* (1996) de Fausto Romitelli. Les accords C ont été calculés avec OM# par interpolation entre deux accords (fig. 9).

La **section D** est une partie mixte avec piano, saxophone et trombone. Les sons des instruments sont envoyés dans le programme de traitement et sont contrôlés en temps réel. Dans la **section E**, les étudiants lisent et transforment des boucle provenant de sons prélevées dans la section D.

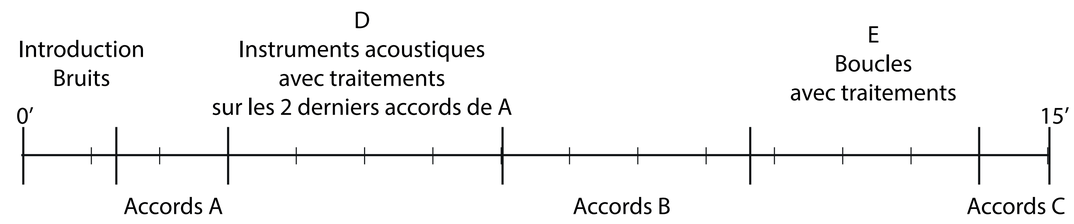

FIGURE 8. STRUCTURE DE LA PIÈCE

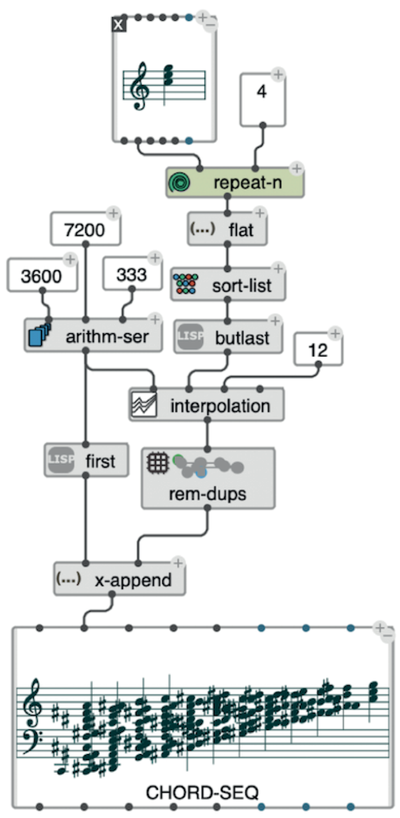

FIGURE 9. LA PROGRESSION HARMONIQUE C CALCULÉE DANS OM#

```
declare name        "WindMgCr3";
declare version     "1.03";
declare author      "LP - granul part adapted from sfIter by C. Lebreton";
declare license     "BSD";
declare copyright   "(c)CIEREC 2017-2020";
//---------------------- WindMgCr2 ------------------------
import("stdfaust.lib");
gain = hslider("[1]Gain", 0.2, 0, 1, 0.01):si.smooth(0.9997);
freq = hslider("[2]Freq", 440, 20, 5000, 1):si.smooth(0.9997);
res  = hslider("[3]Res", 0.8, 0, 0.995, 0.001):si.smooth(0.9997);
crounch  = hslider("[4]Crounch", 0.0, 0, 1, 0.01):si.smooth(0.9997);
freq2 = hslider("[5]Grain Size", 200,100,2205,1);
speed = hslider ("[6]Speed[unit:Hz]", 10,1,20,0.0001):si.smooth(0.9999);
proba = hslider ("[7]Proba[unit:%]", 70,50,100,1) * (0.01):si.smooth(0.9999);
P = freq2; // fundamental period in samples
Pmax = 4096; // maximum P (for delay-line allocation)
gain22 = 1;
//------------------------ FILTER -------------------------
moog_vcf_wind =  ve.moog_vcf(res,freq) * 0.5; // fix select normalized
//----------------------- NOISEBURST ------------------------
noiseburst = no.noise : *(gate : trigger(P))
    with {
    upfront(x) = (x-x') > 0;
    decay(n,x) = x - (x>0)/n;
    release(n) = + ~ decay(n);
    trigger(n) = upfront : release(n) : > (0.0);
    };
gate = phasor(1) :-(0.001):pulsar;
//------------------------ PHASOR -------------------------
phasor(init) =  (+(float(speed)/float(ma.SR)) : fmod(_,1.0)) ~ *(init);
//------------------------ PULSAR -------------------------
//Le pulsar permet de créer une 'pulsation' plus ou moins aléatoire (proba).
pulsar = _<:((_<(ratio_env)):@(100))*(proba>(_,abs(no.noise):ba.latch));
ratio_env = 0.5;
fade = (0.5); // min > 0 pour eviter division par 0
duree_env = 1/(speed: / (ratio_env*(0.25)*fade));
//------------------------ VU-METER -----------------------
tbar        = 0.1;                  // attack/release time in seconds
gbar        = exp(-1/(ma.SR*tbar)); // corresponding gain factor
envbar      = abs : *(1-gbar) : + ~ *(gbar) : ba.linear2db;
meter(x)    = attach(x, envbar(x) : hbargraph("level", -96, 10));

process= no.noise * (1 - crounch) + noiseburst * crounch * 2 :*(gain) :
    moog_vcf_wind :max(-0.99):min(0.99) : meter <: _,_;
```

FIGURE 10. LE CODE FAUST DU SYNTHÉTISEUR DE BRUIT WINDMGC