INTERPOLATIONS : ÉCRITURE DE CONTRAINTES RÉACTIVES POUR IMPROVISATIONS PIANISTIQUES (DÉMO)

Simon Archipoff[†], *Jaime Arias*[†], *Edwin Buger*[†‡], *David Janin*[†]

LaBRI, CNRS UMR 5800
INRIA Bordeaux Sud-Ouest
Bordeaux INP
Université de Bordeaux,
F-33405 Talence

contact : janin@labri.fr

Résumé

Dans le domaine des systèmes informatiques d'aide à la musique improvisée, nous proposons une série d'expérimentations dans lesquelles le musicien doit s'adapter aux retours automatiquement générés par l'ordinateur.

Dans notre approche, ces « feedbacks » sont décrits à l'aide de fonctions algorithmiques de complexités croissantes, allant de l'ajout automatique de notes, à l'enregistrement et à la restitution transformé de séquences jouées par la musicien.

Apparemment opposé aux approches par apprentissages qui tentent de « comprendre » le style du musicien pour lui proposer un feedback cohérent avec son jeu, nos expérimentations montrent que, sans doute dans les deux cas, le musicien arrive à adapter son jeu en s'appropriant les retours de l'ordinateur pour, in fine, faire de la musique.

1. LE PROJET « INTERPOLATIONS »

À mi-chemin entre modélisation algébrique et musique improvisée, le projet *Interpolations* vise à *explorer* la capacité d'un musicien à improviser dans un environnement augmenté de façon algorithmique via un mécanisme retroaction musicale (voir figure 1). Ce projet sert aussi de prétexte pour *expérimenter* dans un contexte performatif le langage T-score [1], un langage de programmation réactif et temps réel pour de captation, de combinaisons et de transformations temps réel d'objets musicaux.

Ces deux aspects sont explorés à travers l'écriture et la réalisation d'une série de miniatures, les « interpolations ». Dans chaque pièce, le piano devient réactif. En temps réel, le musicien fait face à des figures venues d'un autre monde, celui du scientifique, de l'algèbre à la programmation. Il doit improviser avec des notes, du rythme ou de l'harmonie, mécaniques et arbitraires, qui pourraient lui échapper.

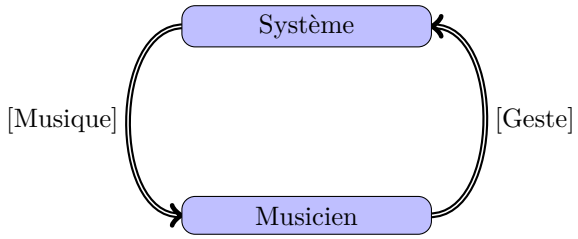

Figure 1: Architecture fonctionnelle : la rétroaction musicale.

On constate à travers les enregistrements disponibles en ligne [1] , comme on pourra sans doute l'observer à nouveau lors du congrès, que le musicien s'adapte en quelques secondes, en recréant un univers esthétique parfois inattendu mais qui lui appartient toujours.

Malgré l'arbitraire algorithmique, le résultat reste profondément humain.

2. SON ARCHITECTURE

L'architecture matériel du projet est décrites figure 2. Elle se compose essentiellement d'un piano MIDI qui assure la capture des gestes du musiciens et la synthèse musicale [2] , et d'un ordinateur sur lequel sont exécutés des modules de traitement des gestes du musiciens et de commande de la rétraction musicale.

† soutenu par l'AMI du programme Arts et Science de l'IdEx de Bordeaux.

‡ Musicien, intermittent du spectacle.

1 . voir http://www.labri.fr/perso/janin/interpolations.html

2 . En toute généralité, ces fonctions de capture et de synthèse pourraient être dissociées en connectant l'installation à deux modules MIDI.

Plus précisément, le comportement de cette architecture est le suivant. Les gestes du musiciens produisent des évènements MIDI qui sont assemblés en notes par le module d'analyse rédiger en UISF/Haskell [3, 2]. Ces notes sont ensuite transmises au module de réaction qui en assure l'*interpolation*.

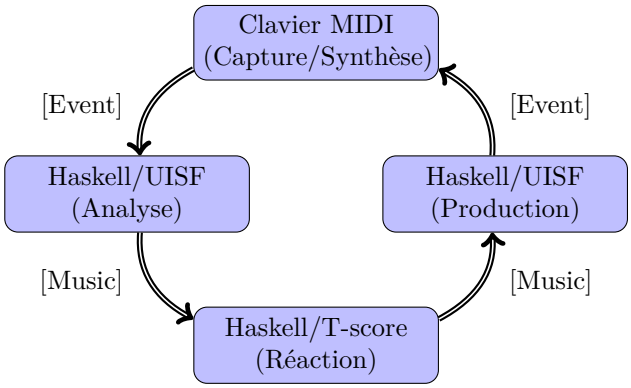

Figure 2: Architecture des composants.

Autrement dit, les notes reçues sont combinées, transformées, enrichies, dupliquées, au gré des fonctions de réactions qui sont décrites dans le langage T-score [1]. Les notes produites par ces réactions sont ensuite transformées à nouveau en successions d'évènements MIDI par le module de synthèse avant d'être jouées par le piano.

Lorsque la fonction de jeu local est désactivée sur le clavier, les notes entendues sont exclusivement les notes produites par l'ordinateur.

3. DU BON USAGE DES ÉVÈNEMENTS *ON* ET *OFF*

Lors de la capture des gestes du musiciens, le clavier envoie des évènements *On* et *Off* correspondant aux enfoncements et aux relâchements des touches du clavier. Ces évènements sont bien appariés. Tout évènement *On* est naturellement suivi d'un évènement *Off*.

Dans le dispositif proposé, cet appariement peut être altéré par le module de réaction qui, par interpolation des gestes du musiciens, peut superposer plusieurs occurrences plus ou moins décalées de la même note.

La norme MIDI n'est pas faites pour gérer une telle situation. Bien que pour une note donnée, le nombre de messages *On* va continuer à correspondre au nombre de message *Off*, le premier message *Off* reçu coupera le son émis par cette note, rendant obsolète tout les messages *Off* suivants. Musicalement, ce n'est pas souhaitable.

La reconversion de notes en évènements MIDI passe donc par un calcul d'accumulation des évènements *On* d'une même note pour désactiver tous les évènements *Off* associés sauf le dernier reçu. Cette gestion des évènements *Off* semble être la seule musicalement cohérente.

Par contre, pour la gestion des évènements *On*, il reste deux modes de jeux possibles. Un premier mode laisse passer tous les évènements *On* dont on peut entendre les effets à travers les attaques de notes associées. Un autre mode consiste à ne transmettre que le premier de ces évènements *On*. Ces deux modes semblant avoir du sens en musique, ils sont laissés en option dans notre dispositif.

Nous listons ci-dessous quelques unes des interpolations qui ont été expérimentées.

4. INTERPOLATIONS « SANS MÉMOIRES »

Une interpolation sans mémoire est une interpolation qui s'applique systématiquement à toute note entrée au clavier. Elle se présente donc sous la forme d'une fonction $f(n)$ où n est la note reçue, décrite par sa fréquence, sa vélocité et sa durée.

Variations sur la copie.

Une première séries d'interpolation tourne autour de la copie. Par exemple, le code

$$c_1\ n = n$$

ne fait que reproduire la note d'entrée en sortie. La fonction

$$c_2\ n = 3 \times \text{fixD}\ \ n$$

la recopie en forçant sa durée à 3 battements grâce à la fonction intermédiaire fixD qui reproduit une note en forçant ça durée.
Dernier exemple, la fonction

$$c3\ n = 3 \times n$$

va reproduire la note jouée en entrée mais en l'étirant d'un facteur 3 : la note reproduite durera trois fois plus longtemps.

Transpositions.

Un mode d'interpolation classique : chaque note jouée est harmonisée d'un intervalle choisi. Deux fonctions d'harmonisations permettent de faire cela, il s'agit des fonctions d'harmonisation chromatique (trpC) et diatonique (trpD).

Par exemple, pour une harmonisation de 4 demi-ton supérieur, la fonction d'interpolation s'écrit simplement :

$$f_1\ n = \text{re}\ \ n + \text{trpC}\ \ n\ 4$$

Pour une harmonisation à la tierce [3] en do majeur, la fonction d'interpolation s'écrit au contraire :

$$f_2\ n = \text{re}\ \ n + \text{trpD}\ \ n\ C\ 2$$

La fonction *re* (*reset*) appliquée à la note n permet de rejouer la note n tout en lançant, *en même temps*, sa transposition. Attention : l'intervalle est noté ici en nombre de degré de déplacement dans la gamme, soit 2 pour une tierce montante.

Cycles.

Bien entendu, les deux modes précédents peuvent être combinées. Par exemple, la fonction

$$f_3\ n = n + \text{trpD}\ \ n\ C\ (-3)$$

permet de rejouer chaque note n en la *faisant suivre* de sa transposition.

Plus généralement, à chaque note appuyée par le musicien, un cycle de transpositions est jouée en arpège autour de la note initiale. On donne ici le code pour exécuter un cycle de transposition chromatique dont les intervalles sont stockés dans la liste cycle.

$f_4\ n =$ pathC n cycle **where**
cycle $= [-3, -4, 7, 4, 3, -4, 3, 5, -6]$
pathC $a\ [\,] = a$
pathC $a\ (x : xs) =$
let $b =$ trpC $a\ x$
in $a +$ pathD $b\ xs$

Un code analogue permet de créer des cycles diatoniques en do majeur.

$f_5\ n =$ pathD n cycle **where**
cycle $= [-3, -4, 7, 4, 3, -4, 3, 5, -6]$
pathD $a\ [\,] = a$
pathD $a\ (x : xs) =$
let $b =$ trpD $a\ C\ x$
in $a +$ pathD $b\ xs$

On remarque que la durée de chaque notes de ces cycles est d'une durée égale à celle de la note initiale. Pour faire des cycles de longueur fixe, on peut insérer des silences entre les reset des mêmes notes. Dans le cas des cycles chromatiques, le code devient :

$f_6\ n =$ pathC n cycle **where**
cycle $= [-3, -4, 7, 4, 3, -4, 3, 5, -6]$
pathC $a\ [\,] = a$
pathC $a\ (x : xs) =$
let $b =$ trpC $a\ x$
in (re a) $+ 1 +$ pathD $b\ xs$

3 . majeure ou mineur selon la note jouée et de la tonalité choisie, une note altérée étant transposés de la même façon que la note de la gamme juste en dessous.

Miroirs.

Avec la fonction pitch qui renvoie le pitch (MIDI) d'une note, on peut écrire une interpolation qui joue en miroir toute note jouée par le musiciens. Elle est réalisée par la fonction :

$f_7\ n =$ **let** $d = (125 - (\text{pitch}\ \ n))$
in re $\ n + \text{trpC}\ \ n\ d$

Une mémoire avec délais fixe entre la note jouée et son miroir peut être codé par la fonction :

$f_8\ n =$ **let** $d = (125 - (\text{pitch}\ \ n))$
in re $\ n + 1 + \text{trpC}\ \ n\ d$

5. INTERPOLATIONS « AVEC MÉMOIRE »

Une interpolation avec mémoire est une interpolation qui s'applique aux notes les unes après les autres en pouvant en outre mettre à jour des paramètres additionnels. Le paramètre d'entrée n'est donc plus la note seule, mais une séquence l de paire de note et de durée (n, d). Attention, dans ces notations, d désigne non pas la durée de la note n mais le silence qui s'écoulera entre la note n et la note qui la suivra.

La manipulation par programme de ces séquences nécessite donc de pouvoir accéder à ces pairs les unes après les autres. On dispose donc de deux fonctions auxiliaires *headS* et *tailS* qui, appliquée à une séquence $l = (n, d) : l1$ vont respectivement valoir

$$headS\ l = (n, d) \text{ et } tailS\ l = l1$$

Une interpolation avec mémoire est alors une fonction de la forme $g\ m\ l$ ou m est la mémoire de g.

Copycat.

Par exemple, la fonction *copycat* ne fait que reproduire en sortie les notes entrées. Dans le langage T-score, qui hérite de la syntaxe Haskell, elle s'écrit simplement de la façon suivante :

$copyCat\ l =$ **let** $(n, d) = headS\ l$
in re $\ n + d + rec\ copyCat\ (tailS\ l)$

Dans ce code, un opérateur de récursion explicite *rec* doit apparaître afin de contrôler l'exécution réactive de cette fonction. En effet, lorsque la note n arrive, la durée d à la note suivante est inconnue.

Remarquons que cette fonction de *copyCat* permet aussi de reproduire les accords, les délais séparant les notes d'un même accord étant de durée nulle.

PlayF.

Un second exemple, tout aussi simple, consiste à appliquer, via la fonction *playF* une fonction f sur toute les notes jouées, c'est à dire une fonction d'interpolation sans mémoire.

En T-score, cette fonction s'écrit :

$$
\begin{aligned}
&\mathit{playF}\ f\ l = \textbf{let}\ (n, d) = \mathit{headS}\ l\\
&\qquad\qquad \textbf{in}\ \text{re}\ (f\ n) + d\\
&\qquad\qquad\quad + \mathit{rec}\ (\mathit{play}\ f)\ (\mathit{tailS}\ l)
\end{aligned}
$$

Bien entendu, ces deux exemples ne manipulent pas à proprement parler de mémoire.

Dodécaphonique.

Une première interpolation qui utilise vraiement de la mémoire est la fonction *dodecaphonique* qui change la classe de pitchs des notes jouées au piano en la remplaçant par une classe de pitch prise dans une liste qui parcours de façon cyclique les douze classes de pitch possible.

Le code de la fonction *dodecaphonique* est donnée par :

$$
\begin{aligned}
&\mathit{cyclePitch} = [2, 3, 4, 7, 9, 11, 1, 5, 0, 6, 10, 8]\\
&\qquad\qquad +\!\!+\ \mathit{cyclePitch}\\
&\mathit{playDod}\ \text{cycle}\ l\\
&\quad = \textbf{let}\ (p : \mathit{cycle1}) = \text{cycle}\\
&\qquad\quad (n, d) = \mathit{headS}\ l\\
&\qquad \textbf{in}\ \text{re}\ (\mathit{setPitchClass}\ p\ n) + d\\
&\qquad\quad + \mathit{rec}\ \mathit{playDod}\ \mathit{cycle1}\ (\mathit{tailS}\ l)\\
&\mathit{dodecaphonique}\ l\\
&\quad = \mathit{playDod}\ \mathit{cyclePitch}\ l
\end{aligned}
$$

Dans ce code, la fonction *playDod* permettant de parcourir une a une les classes de pitchs décrites dans la liste *cyclePitch*. La fonction *setPitchClass p n* permet quant à elle de changer la classe de pitch de la note n tout en préservant son octave, sa durée et sa vélocité.

Plus généralement, une fonction de tirage aléatoire permet de choisir arbitrairement la séquence de classe de pitch a suivre.

Copy and replay.

Dernier exemple, une fonction de copy and replay permet d'enregistrer les notes jouées pendant une durée $d1$ pour les reproduire ensuite, au tempo double, dans un sens puis dans l'autre. Quoique complexe dans ses fonctionnalités, le code de cette fonction s'écrit simplement :

$$
\begin{aligned}
&\mathit{copyAndRep}\ d_1\ (m, l)\\
&\quad = \textbf{let}\ (n, d) = \mathit{headS}\ l\\
&\qquad\quad h = \text{re}\ \ n + d\\
&\qquad\quad b = d_1 \leqslant \mathit{duration}\ (m + h)\\
&\qquad \textbf{in case}\ (b)\ \textbf{of}\\
&\qquad\quad \mathit{True} \rightarrow\ \text{re}\ (1/2 \times m + 1/2 \times (\mathit{reverse}\ m))\\
&\qquad\qquad\qquad + h\\
&\qquad\qquad\qquad + \mathit{rec}\ (\mathit{copyAndRep}\ d_1)\ (h, \mathit{tailS}\ l)\\
&\qquad\quad \mathit{False} \rightarrow h\\
&\qquad\qquad\qquad + \mathit{rec}\ (\mathit{copyAndRep}\ d_1)\ (m + h, \mathit{tailS}\ l)\\
&\mathit{copyAndReplay}\ d_1\ l = \mathit{copyAndRep}\ d_1\ 0\ l
\end{aligned}
$$

Dans la fonction *copyAndRep*, le paramètre m sert à mémoriser la musique reçut. Le booléen b permet de tester lorsque la durée de cette mémorisation vas dépasser d_1.

On peut noter dans les paramètres de l'appel récursif, la recopie systématique de la note et du délais suivant (h). Selon la valeur du booléen b, la mélodie mémorisé m est jouée puis omise de l'appel récursif (cas *True*) ou bien juste accumulé lors de l'appel récursif (cas *False*).

Dans ce code, la fonction auxiliaire *reverse* permet de jouer une mélodie à l'envers. La multiplication d'une mélodie m par $1/2$ dans $1/2 \times m$ permet de jouer la mélodie deux fois plus vite.

6. BILAN DU PROJET

Les premières interpolations, écrites en quelques lignes par le scientifique, ont permis de tester les primitives atomiques du langage de programmation : répétitions, transpositions, composition. Le musicien a transformé cet arbitraire « technologique » en un univers musical cohérent et souvent inattendu pour lui-même. En retour, il a proposé des développements supplémentaires ou, parfois même, d'autres pistes d'explorations.

La liste des interpolations s'est complétée des conditionnelles, des appels récursifs, de la mémorisation offert par le langage de programmation. Ce dernier s'est tout à la fois enrichi et simplifié au fur et à mesure des séances de travail.

Même si elles sont parfois déjà inscrites dans l'histoire musicale, ce projet nous a ouvert et nous ouvre encore l'accès à un espace d'expérimentations musicales infini. Des vidéos de ces expérimentations sont disponibles sur le net [4] .

Musicalement, nous pouvons retenir que ce projet nous a permis d'expérimenter à travers ces séances d'improvisations, des procédés de constructions musicales algorithmiques (répétitions, transpositions, miroirs tonale, répétitions, miroirs temporels, etc...) qui apparaissent somme toute depuis longtemps comme processus de composition musicale. La programmation en T-score permet donc de les mettre en œuvre et de les expérimenter.

Cependant, dans les performances improvisées que nous avons produites, le sens de ces processus de transformations devient différent de leur utilisation en composition. Sur scène, en temps réel, ils participent à la création des rétroactions musicales ; des rétroactions qui sont immédiatement prises en compte et intégrées par le musicien improvisateur.

La qualité esthétique des pièces obtenues semble paradoxalement démontrer qu'on ne peut espérer une validation esthétiques des processus de rétroactions eux-mêmes : le musicien semble pouvoir s'adapter à *n'importe quelle* retroaction, fut-elle construite sur un algorithme simple d'interpolation, ou des outils sta-

4 . voir http://www.labri.fr/perso/janin/interpolations.html

tistiques complexes d'analyse et de restitution « à la manière de ».

Remerciement

Les séances visibles sur internet n'auraient pu voir le jour sans l'aide de Julia Hanadi Al Abed et de Pierre Cochard pour la préparation de la salle Hémicyclia et les prises de son, et l'aide de Christian Faurens pour l'enregistrement et le montage des vidéos.

7. REFERENCES

[1] S. Archipoff and D. Janin. Vers une programmation reactive structureée. Technical report, LaBRI, Université de Bordeaux, 2016.

[2] P. Hudak, J. Hugues, S. Peyton Jones, and P. Wadler. A history of Haskell : Being lazy with class. In *Third ACM SIGPLAN History of Programming Languages (HOPL)*. ACM Press, 2007.

[3] P. Hudak, D. Quick, M. Santolucito, and D. Winograd-Cort. Real-time interactive music in haskell. In *Work. on Functional Art, Music, Modeling and Design (FARM)*, pages 15–16. ACM, 2015.