Pyo, un module python dédié au traitement de signal audio

Olivier Bélanger
Faculté de Musique, Université de Montréal
olivier.ajaxsoundstudio.com
olivier.belanger@umontreal.ca

Abstract
Pyo is a Python module containing classes for a wide variety of audio signal processing types. With pyo, user will be able to include signal processing chains directly in Python scripts or projects, and to manipulate them in real time through the interpreter. Tools in pyo module offer primitives, like mathematical operations on audio signal, basic signal processing (filters, delays, synthesis generators, etc.), but also complex algorithms to create sound granulation, spectral manipulations and others creative sound compositions. Pyo supports OSC protocol (Open Sound Control), to ease communications between softwares, and MIDI protocol, for generating sound events and controlling process parameters. Pyo allows creation of sophisticated signal processing chains with all the benefits of a mature, and widely used, general programming language. After an introduction to the basic concepts of the library, examples of sound synthesis, signal processing, music composition and software development will be presented to help getting started with audio programming in python.

Résumé
Pyo est un module Python offrant une grande variété de classes dédiées au traitement de signal audio. Avec pyo, il est maintenant possible d'inclure des chaînes de traitement audio à même un programme Python et de manipuler le processus en temps réel via l'interpréteur. Pyo offre des outils de base tels que les manipulations mathématiques sur le signal audio et les traitements sonores courants (filtres, délais, générateurs de synthèse et bien d'autres), mais aussi une gamme d'outils sophistiqués permettant le traitement par granulation, les manipulations spectrales et autres compositions sonores originales. Pyo supporte le protocole OSC (Open Sound Control), afin de faciliter la communication inter-logiciel, ainsi que le protocole MIDI, dédié à la génération d'événements et au contrôle des paramètres. Pyo permet la création de chaînes de traitements audio élaborées tout en bénéficiant des atouts d'un langage de programmation mature et largement utilisé. Après une introduction aux paradigmes de base de la librairie, des exemples de synthèse sonore, de traitement de signal, de composition musicale et de développement de logiciels seront présentés pour faciliter la prise en main de la programmation audio en python.

Keywords
python, audio, traitement de signal, programmation, algorithme, synthèse sonore

Mots-Clés
python, audio, traitement de signal, programmation, algorithme, synthèse sonore

Introduction

Il existe à ce jour plusieurs langages spécialement dédiés à la programmation sonore. Des langages de programmation textuelle tels que Csound [1], créé par Barry Vercoe en 1986, et SuperCollider [2], développé par James McCartney en 1996, jouissent aujourd'hui d'une grande popularité et d'une communauté de développeurs florissante. On y retrouve tous les processeurs audio essentiels à la programmation musicale sous la forme de fonctions (ou objets) connectées entre elles à l'aide de variables. Le paradigme de programmation graphique implémenté dans le logiciel PureData [3], développé par Miller Puckette en 1988 (sous le nom *Patcher*), permet de créer des chaînes de traitement de signal ainsi que des structures de contrôle à l'aide de boîtes connectées entre elles par des fils. Ce type de programmation attire particulièrement les musiciens qui trouvent une certaine aisance à développer une idée musicale de façon graphique. Ces différents environnements ont prouvé depuis longtemps leur valeur dans le monde de la création sonore numérique. La motivation qui a mené à la création d'un nouveau moteur audio, sous la forme d'un module python [4], provient du fait que tous ces logiciels sont principalement dédiés à la programmation multimédia et offrent très peu d'outils pour la programmation générale. Après plusieurs années à développer des logiciels sonores alliant le langage python

pour la gestion de l'interface et Csound comme moteur audio, parmi lesquels figure Ounk [5], l'ancêtre de pyo, le besoin d'un environnement unifié devint manifeste. Un environnement de développement qui intégrerait de façon plus flexible le moteur audio au langage de programmation permettrait de créer des interactions plus sophistiquées entre les structures de contrôle et le processus audio. Un langage de programmation générale, tel que python, offre une panoplie de fonctionnalités, des mathématiques complexes aux librairies d'interface graphique en passant par la programmation de base de données, qui sont généralement absentes dans les environnements dédiés au son. Le module pyo tente de créer le pont entre le langage servant à construire la structure du logiciel, c'est-à-dire les contrôles de paramètres, et la génération des algorithmes sonores. En évoluant à l'intérieur du même environnement de programmation, la communication entre les éléments d'interface, ou les structures génératives, et les processus audio est grandement simplifiée puisque les objets ont un accès direct les uns aux autres. Le moteur audio devient donc un module spécialisé parmi des centaines d'autres modules disponibles, chacun étant optimisé pour effectuer une tâche particulière.

Cet article se divise en quatre sections. D'abord, sera détaillé l'installation des ressources logicielles nécessaires au bon fonctionnement du module audio pyo. Ensuite, dans le chapitre 3, nous effectuerons un survol de la structure de la librairie, de ses modes de communication et des principaux types de générateurs sonores. Le chapitre 4 explorera les différents éléments de langage mis en place pour permettre la création de programmes audio variés, que ce soit pour la simple génération de son, la composition algorithmique ou le développement d'interface graphique de contrôle. Le cinquième et dernier chapitre présentera des exemples concrets de synthèse sonore, de traitement de signal, de composition et de développement de logiciels.

1. Installation des ressources logicielles

Écrire des programmes audio sous python nécessite l'installation d'un certain nombre de dépendances. Ces dépendances serviront notamment à afficher les éléments d'interface graphique, à lire et écrire des fichiers sonores sur le disque dur et à assurer la communication entre le moteur audio et les différentes interfaces externes. Toutes les ressources logicielles requises pour utiliser le module pyo sont gratuites, de sources libres et fonctionnent sous tous les systèmes d'exploitation majeurs.

En premier lieu, une version 2.6 ou 2.7 de python doit être installée sur le système afin de pouvoir utiliser le module pyo. Comme le module contient des éléments d'interface graphique écrits avec wxPython [6], il est fortement recommandé d'installer aussi ce module. Ensuite, il suffit d'installer pyo pour avoir un environnement de travail complet pour l'écriture de programmes audio. Des installeurs binaires de python, wxPython et pyo existent pour les plates-formes Windows et OSX. Les usagers Linux pourront, quant à eux, trouver ces ressources dans le gestionnaire de paquets de leur distribution. La plateforme Github est utilisée pour le développement et la distribution du code source de pyo. On trouvera les installeurs binaires de pyo sur la page web du projet :

`http://ajaxsoundstudio.com/software/pyo`

Pour assurer la communication avec les interfaces externes, pyo utilise les quatre librairies suivantes (qui font partie intégrante de l'installeur de pyo sous Windows et OSX) :

- **Portaudio** : Assure la communication audio avec la carte de son.
- **Portmidi** : Gère les messages MIDI (*Musical Instrument Digital Interface*).
- **Libsndfile** : Permet l'écriture et la lecture de fichiers audio sur le disque.
- **Liblo** : Communication à l'aide du protocole OSC (*Open Sound Control*).

L'installation de *pyo* offre aussi un éditeur de texte, **E-Pyo**, spécialement configuré pour l'écriture de programmes audio avec la librairie. On y retrouve des raccourcis pour l'exécution des programmes, un accès direct à la documentation des fonctionnalités audio, des exemples d'utilisation de la librairie dans divers contextes ainsi que plusieurs automatismes permettant d'accélérer l'écriture du code et la gestion des projets complexes.

La documentation de pyo peut être consultée à même l'application E-Pyo. Ce format offre l'avantage de pouvoir exécuter et modifier l'exemple sonore de chacun des objets sans quitter l'environnement de travail. Une version en format html est aussi disponible en téléchargement et peut être consultée en ligne à l'adresse suivante :

`http://ajaxsoundstudio.com/pyodoc/`

Sous les systèmes à base *unix*, il est aussi possible de compiler pyo à partir des sources, afin de profiter des derniers avancements d'un module en développement actif. La procédure est disponible sur le site internet du manuel.

2. Structure de la librairie

Communications

Les connexions entre les différentes composantes de la librairie sont illustrées sur le diagramme ci-dessous. On y décèle quatre blocs principaux : le serveur audio, la librairie de processeurs (*pyolib*), les contrôleurs externes et les éléments d'interface graphique.

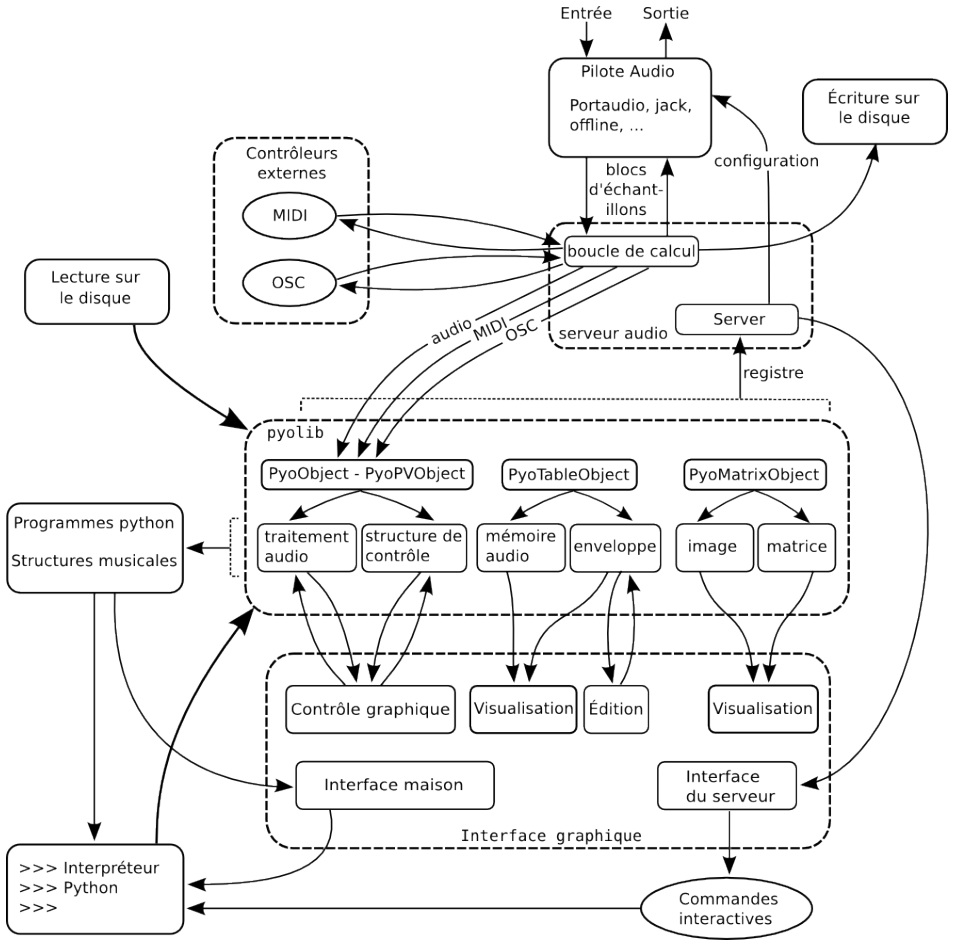

FIGURE 1. Schéma de communication

L'élément central de la librairie est le serveur audio, dont le rôle principal consiste à gérer la boucle de calcul et la communication avec les interfaces externes. Les échantillons audio et le data en provenance des contrôleurs sont acheminés par le serveur au moteur de la librairie, *pyolib*, là où réside les composantes servant à créer les chaînes de traitement de signal. Ces composantes seront agencées selon les instructions fournies dans le programme python afin de mettre en place la structure musicale désirée. Au lancement du programme, les séquences d'instructions seront exécutées par l'interpréteur python et de nouvelles commandes interactives pourront être envoyées soit par le biais de l'interface graphique, soit à l'aide de nouvelles lignes d'instructions données directement à l'interpréteur.

Serveur audio

Le serveur audio, représenté par un objet **Server**, est l'élément central de tout programme avec pyo. Cet objet assure le transfert des échantillons, en entrée et en sortie, entre le processus et le pilote audio. Les arguments donnés à l'initialisation du serveur permettent notamment de spécifier la résolution du signal sonore, c'est-à-dire la fréquence d'échantillonnage, le nombre de canaux et la taille des blocs d'échantillons traités en entrée et en sortie. C'est également via le serveur que l'on choisi le pilote à utiliser ainsi que les interfaces audio et MIDI. En cours de performance, le rôle du serveur consiste à maintenir l'historique de création des objets audio et à lancer le calcul des processus chaque fois que le pilote fait une requête pour un groupe d'échantillons. Un serveur audio doit donc impérativement être créé avant

tout autre objet de la librairie, puisque la première tâche effectuée par ces derniers, au moment de leur création, consiste à s'insérer dans la boucle de calcul du serveur.

Le serveur assume aussi un second rôle qui consiste à garder le programme actif le temps nécessaire pour produire le résultat sonore dans son entièreté. En effet, un programme audio présente la particularité que des échantillons doivent être générés sur une période temps indéterminée, et ce, même après que chacune des lignes de code ait été exécutée. Un programme avec interface graphique, lorsqu'il est lancé, attend que la dernière fenêtre soit fermée avant de quitter. Afin de tirer parti de cette fonctionnalité, l'objet **Server** est doté d'une fenêtre de contrôle, que l'on affiche à l'aide de la méthode *gui()*, permettant de garder le programme actif aussi longtemps que nécessaire. Cette fenêtre contient un vu-mètre, qui rend compte de l'amplitude moyenne du signal de sortie, et une horloge qui tient le décompte du temps écoulé depuis le début de la performance. Elle permet aussi de démarrer ou d'arrêter la boucle de calcul du serveur et d'ajuster le volume global du signal de sortie du programme. Le champs *Interpreter* est une extension de l'interpréteur python et permet à l'utilisateur d'exécuter de nouvelles commandes en cours de performance.

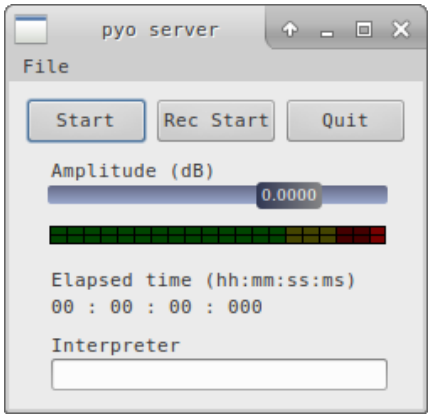

FIGURE 2. Interface graphique du serveur audio

Classes parentes

La structure interne de la librairie pyo repose essentiellement sur quatre classes élémentaires, chacune d'elles ayant à sa charge la gestion d'un type de data particulier. Tous les objets de la librairie sont dérivés de l'une ou l'autre de ces classes.

PyoObject

Le **PyoObject** est la classe en charge de la gestion du data qui représente le signal audio sous la forme d'un vecteur de nombres à virgule-flottante. Ce type de signal, renouvelé à chaque tour d'horloge du serveur, est le plus commun. Il véhicule le son qui sera ultimement acheminé à la sortie audio. Cet objet constitue la classe parente de la grande majorité, plus de deux cents, des générateurs et processeurs audio disponibles dans la librairie. Les fonctionnalités communes aux générateurs audio sont, par conséquent, définies dans la déclaration du **PyoObject** afin que chacun puisse y avoir accès. Certains opérateurs standards du langage, tels que les symboles mathématiques, les opérateurs de comparaison et les techniques d'indiçage ont été remplacés dans la définition du **PyoObject** pour agir sur le signal audio. Ainsi, il est possible d'effectuer des opérations mathématiques sur chacun des éléments du vecteur d'échantillons avec la syntaxe usuelle du langage.

Quelques méthodes essentielles du **PyoObject** :

- *play* : Active le calcul des échantillons de l'objet.
- *out* : Active le calcul des échantillons de l'objet et indique que le signal généré doit être acheminé à la sortie audio. Les arguments de la méthode permettent de contrôler vers quels canaux de sortie les signaux seront envoyés.
- *stop* : Arrête le calcul des échantillons de l'objet, permettant ultimement d'économiser du CPU.
- *ctrl* : Ouvre une fenêtre où des potentiomètres sont assignés aux paramètres de contrôle de l'objet. Elle permet une exploration rapide de l'impact des paramètres sur le signal sonore généré.

PyoPVObject

La classe **PyoPVObject** est en charge des signaux complexes, c'est-à-dire possédant une partie réelle et une partie imaginaire, résultant d'une analyse spectrale effectuée avec la transformée de Fourier rapide [7]. Le vocodeur de phase [8], est un procédé d'analyse-synthèse d'une grande importance dans l'évolution du traitement de signal numérique. Il permet, entre autres, une meilleure compréhension des processus de génération sonore par l'analyse de leur contenu spectral, c'est-à-dire l'énergie présente dans les différents registres du son. Le vocodeur de phase est aussi une technique de traitement très puissante où il est possible de manipuler les différents partiels d'un son de façon indépendante. Les objets qui héritent du **PyoPVObject** offrent un large éventail de manipulations du son dans le domaine spectral. Pyo offre aussi un objet **Spectrum** qui permet l'affichage, grâce à l'analyse de Fourier, du contenu spectral d'un signal audio.

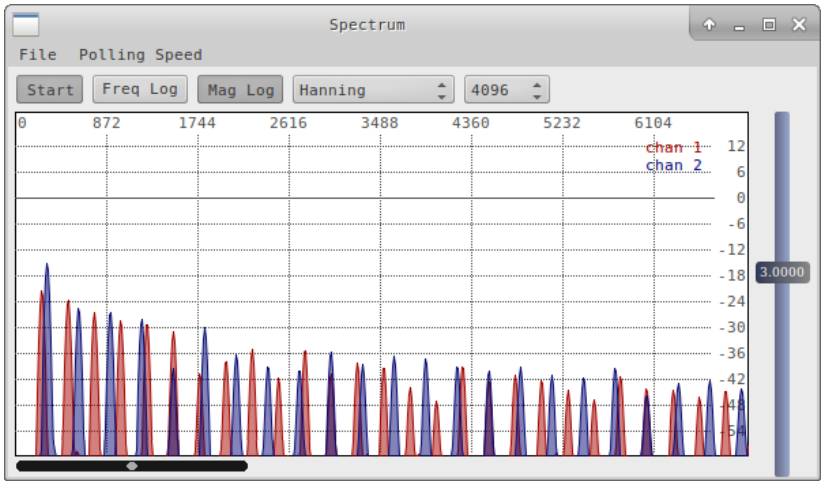

FIGURE 3. Affichage du contenu spectral d'un signal audio

PyoTableObject

Les objets dérivés de la classe **PyoTableObject** permettent de charger des échantillons dans la mémoire vive de l'ordinateur, offrant ainsi aux autres objets un accès rapide au data mémorisé. Le contenu peut-être le signal audio d'un fichier son, une enveloppe destinée au contrôle de paramètre ou des valeurs arbitraires définies à des fins algorithmiques. Ce data reste fixe, quoique renouvelable sur demande, et peut-être lu au besoin par tout objet acceptant une table en argument. Le chargement d'un son dans une table permet, entre autres, l'application de traitements sophistiqués, tels que la granulation ou la lecture à boucles variables, nécessitant un accès rapide à chacun des échantillons du signal. Les méthodes *write* et *read* permettent respectivement de sauvegarder et de lire le contenu d'une table, en format texte, sur le disque dur. Il est possible, à tout moment, de visualiser, ou d'éditer, le contenu d'une table à l'aide des méthodes *view* et *graph*.

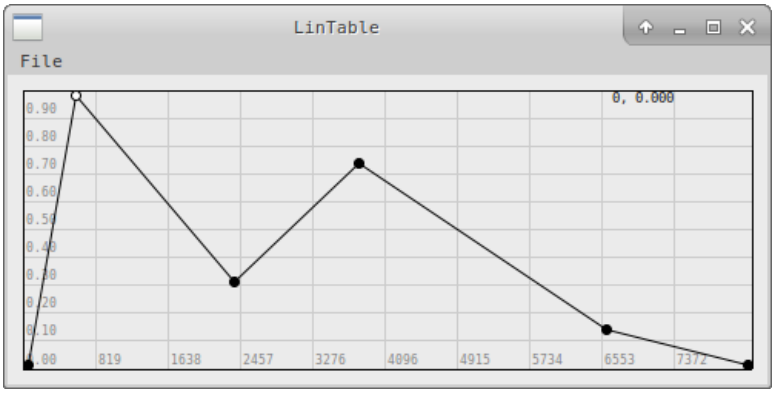

FIGURE 4. Fenêtre d'édition d'une ligne segmentée

PyoMatrixObject

Les objets dérivés de la classe **PyoMatrixObject** permettent de charger du son ou tout autre forme de data dans un espace mémoire à deux dimensions. Les matrices appliquent les mêmes concepts que les tables mais sur deux dimensions plutôt qu'une seule. Elles permettent notamment de mémoriser des images et d'interpréter les pixels en tant qu'échantillons sonores. La lecture, au taux audio, d'une matrice s'effectue à l'aide d'un objet **MatrixPointer**, qui prend deux signaux audio représentant les positions en X et en Y dans la mémoire et interprète les valeurs aux points successifs en tant que signal audio. La lecture de matrices permet notamment l'implémentation de la *wave terrain synthesis* [9], une technique de synthèse où deux oscillateurs créent une forme ovoïde balayant le contenu d'une mémoire à deux dimensions, considérée comme le terrain. Les valeurs lues aux positions données par le croisement des oscillateurs constituent la forme d'onde, c'est-à-dire le signal de sortie.

3. Éléments de langage

Cette section détaillera, étape par étape, les différents éléments de langage nécessaires à la bonne compréhension de la librairie. Les techniques de programmation propres à la génération sonore avec le module pyo seront illustrées à l'aide d'exemples élémentaires.

Lire un fichier son

Le premier exemple consiste simplement à lire un fichier son sur le disque dur et à envoyer le signal à la carte de son. Nous utiliserons l'objet **SfPlayer** pour spécifier le fichier à lire ainsi que la vitesse de lecture et l'amplitude du signal. L'argument *loop* permet d'activer ou de désactiver le mode bouclage de la lecture. Afin d'acheminer le signal à la sortie audio de pyo, la méthode *out()* est appelée à la création de l'objet.

```
from pyo import *

s = Server().boot()

a = SfPlayer("/home/olivier/rfla2016/flute.aif", loop=1, mul=.5).out()

s.gui(locals())
```

Code 1. Lecture d'un fichier son

Le premier argument de l'objet **SfPlayer** est obligatoire car aucune valeur ne lui est assignée par défaut. C'est à l'aide de cet argument (*path*) que l'on indique à l'objet où se trouve le fichier qui doit être lu. On spécifie le chemin à l'aide d'une chaîne de caractères en utilisant la syntaxe en vigueur sur les systèmes *unix*, c'est-à-dire en séparant les éléments hiérarchiques par le symbole "/".

"/home/olivier/rfla2016/flute.aif"

Sur la plateforme Windows, python se chargera de convertir la chaîne de caractères dans le bon format avant d'aller chercher le son sur le disque. Nous y écrierions donc :

"C :/Users/olivier/rfla2016/flute.aif"

Chaînes de traitement

Tous les objets servant à modifier le signal (par exemple les filtres, les réverbérations ou les distorsions) possèdent un premier argument nommé *input*. Un objet audio de type **PyoObject** doit impérativement être donné à ce paramètre. L'objet modificateur récupérera le signal de l'objet donné en entrée et lui appliquera un algorithme de transformation afin de générer un nouveau signal sonore. Ce signal pourra ensuite être envoyé soit à la sortie audio, soit à un autre objet de transformation. Voici un exemple d'une chaîne de traitements simple, une distorsion suivi d'une réverbération, appliquée à la lecture d'un fichier son :

```
from pyo import *

s = Server().boot()

# Lecture d'un son --> distorsion --> reverberation
a = SfPlayer("/home/olivier/rfla2016/flute.aif", loop=True, mul=0.3)
```

```
b = Disto(a, drive=0.9, slope=0.8, mul=.1)
c = WGVerb(b, feedback=0.8, cutoff=3000, bal=0.25).out()

s.gui(locals())
```

Code 2. Appliquer les traitements en série

Contrôle des processus

Une méthode rapide et efficace pour explorer le contrôle des paramètres du son consiste à utiliser la fenêtre de potentiomètres disponible pour tous les objets dérivés de la classe **PyoObject**. On affiche la fenêtre en appelant la méthode *ctrl()* sur l'objet dont on désire manipuler les paramètres. Les potentiomètres prendront alors la valeur courante des arguments auxquels ils sont associés et les valeurs définies de façon graphique resteront en place à la fermeture de la fenêtre. L'appel *a.ctrl()*, ajouté au script précédent, ferait apparaître la fenêtre de contrôle du lecteur de fichier sonore, permettant notamment de modifier la vitesse de lecture du son.

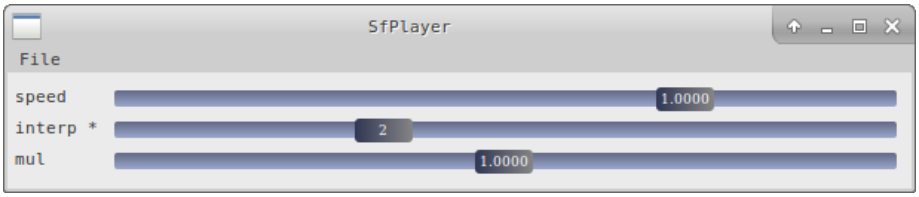

FIGURE 5. Fenêtre de contrôle de l'objet **SfPlayer**

Variations continues des paramètres du son

Un son naturel est rarement statique, il est plutôt en constante évolution. Que ce soit par de légères variations de fréquence ou d'amplitude, le timbre d'un son varie au cours du temps. Afin de créer des signaux sonores dynamiques, il est nécessaire de conférer une évolution temporelle aux différents paramètres des objets composant le processus sonore. Des objets pyo peuvent être utilisés pour créer des trajectoires de contrôle, c'est-à-dire des signaux audio non pas destinés à l'envoi vers la carte de son mais plutôt à créer des variations de paramètres dans le temps. Tout argument, si le manuel spécifie qu'il accepte un *float* ou un **PyoObject**, peut être soumis à un objet pyo préalablement créé. L'exemple suivant utilise deux oscillateurs à basse fréquence pour faire varier la fréquence et l'amplitude d'une sinusoïde de façon cyclique :

```
from pyo import *

s = Server().boot()

freq = LFO(freq=10, mul=100, add=300)
amp = LFO(freq=10.1, type=3, mul=0.3, add=0.3)
synth = Sine(freq=freq, mul=amp).out()

s.gui(locals())
```

Code 3. Variations continues des paramètres

Gestion de la polyphonie

La gestion de la polyphonie constitue un des dilemmes les plus complexes à résoudre dans le développement d'un moteur audio. Le défi consiste à établir des règles intuitives, mais suffisamment flexibles, quant à la distribution des canaux audio à l'intérieur de l'environnement. D'une part, les différentes composantes, ou objets, doivent être en mesure d'échanger des signaux audio de différentes natures entre eux, qu'ils représentent un vecteur d'échantillons ou qu'ils proviennent du data en mémoire dans une table ou une matrice. D'autre part, le langage doit fournir un mécanisme flexible pour la gestion des signaux qui doivent être acheminés vers les sorties audio. L'élément audio à la base de la structure interne de pyo est l'objet **Stream**.

L'object *Stream*

Un objet **Stream** est une structure interne au module permettant la gestion d'un vecteur de son monophonique. Tout signal audio est rattaché à un objet **Stream**, ce qui permet aux différents objets de la librairie de s'échanger des groupes d'échantillons entre eux, établissant ainsi la chaîne de traitements sonores. Un objet de type **PyoObject** générera autant de *streams* que nécessaire afin de créer le processus sonore demandé. Un exemple concret consiste en la lecture d'un fichier son avec l'objet **SfPlayer**. Si le fichier est stéréo, l'objet créera deux *streams* monophoniques, un pour chaque canal. Les objets de la librairie ont été conçus pour répondre à certaines méthodes associées aux conteneurs python. Ainsi, on peut questionner un objet sur le nombre de *streams* qui lui sont associés avec la fonction *len()*, ou bien récupérer le signal de l'un ou l'autre de ces *streams* avec les techniques d'indiçage usuelles du langage python. L'exemple suivant illustre ces deux procédés.

```
from pyo import *

s = Server().boot()

# Lecture d'un fichier stereo
sf = SfPlayer("/home/olivier/rfla2016/voice.aif", loop=1, mul=0.5)
# Affiche le nombre de streams
print len(sf)

# Les frequences graves a gauche
lp = Biquad(sf[0], freq=1000, type=0).out()
# Les frequences aigues a droite
hp = Biquad(sf[1], freq=1000, type=1).out(1)

s.gui(locals())
```

Code 4. Gestion des canaux audio

En ce qui concerne la distribution des *streams* d'un objet vers les sorties audio, le comportement par défaut (modifiable via les arguments de la méthode *out()* du **PyoObject**) est d'alterner les sorties, en fonction du nombre disponibles, à chaque nouveau *stream*. En débutant à 0, un son stéréo sera réparti sur les sorties 0 et 1, c'est-à-dire le premier *stream* à gauche et le second à droite.

L'expansion multi-canal

Un concept très puissant en programmation, appelé *multi-channel expansion*, introduit par James McCartney dans le langage de programmation musicale SuperCollider, consiste à donner une liste comme valeur à un argument d'un objet pour créer plusieurs *streams* audio en une seule instruction. Ce concept est généralisé à tous les paramètres des objets pyo, afin de décupler les processus sonores sans avoir à répéter sans cesse les mêmes lignes de code. Ainsi, pour créer un décalage progressif de plusieurs lectures d'un son, on exécuterait le code suivant :

```
from pyo import *

s = Server().boot()

# 5 lectures d'un son stereo = 10 streams audio
a = SfPlayer("/home/olivier/rfla2016/voice.aif", loop=True, mul=.2,
             speed=[1,0.999,1.003,.995,1.006]).out()

s.gui(locals())
```

Code 5. Expansion des processus avec les listes

Un objet qui reçoit une liste comme valeur d'un de ses arguments générera le nombre de canaux nécessaires afin de produire le rendu sonore désiré. Si, pour un même objet, plusieurs arguments reçoivent une liste en entrée, la plus longue liste sera utilisée pour déterminer le nombre de *streams* gérés par l'objet. Les valeurs des listes plus courtes seront utilisées dans l'ordre, avec retour au début lorsque la fin de la liste est atteinte. Utiliser des listes de différentes longueurs aux arguments d'un objet produira un résultat sonore riche et varié puisque les combinaisons de paramètres seront différentes pour chaque instance du processus sonore.

```
from pyo import *

s = Server().boot()

a = SumOsc(freq=[80,160.2,200.5,240.7], ratio=[.501,.753,1.255],
```

```
            index=[.3,.4,.5,.6,.7,.4,.5,.3,.6,.7,.3,.5], mul=.02).out()

s.gui(locals())
```

Code 6. Croisement des paramètres

Gestion temporelle des événements

Le concept de trigger

Un *trigger*, ou *trig*, est une impulsion, c'est-à-dire un signal audio dans lequel un échantillon ayant une valeur de 1 est entouré d'échantillons de valeurs 0. La raison d'être de ce type de signal est de pourvoir pyo d'un système de gestion des événements à haute résolution temporelle. Un *trigger* étant un signal audio, il possible de créer des traitements parallèles avec une synchronisation à l'échantillon près.

Séquence d'événements

Une séquence d'événements peut être produite à l'aide d'un objet de génération de *triggers*, un métronome par exemple, activant de façon périodique le processus d'une chaîne de traitement de signal. L'exemple suivant illustre le déclenchement d'une enveloppe d'amplitude appliquée à un synthétiseur dont la fréquence est choisie au hasard, parmi un groupe de valeurs prédéterminées, au début de chaque note.

```
from pyo import *

s = Server().boot()

# Enveloppe
env = LinTable([(0,0), (100,1), (500,.5), (5000,.5), (8191,0)])
# Generation de triggers
met = Metro(time=.125).play()
# Pige une frequence au hasard
fr = TrigChoice(met, choice=[100,150,200,250,300,350], port=.005)
# Active la lecture de l'enveloppe
amp = TrigEnv(met, table=env, dur=.1, mul=.5)
# Synth stereo
a = SumOsc(freq=fr, ratio=[.245,.255], index=0.7, mul=amp).out()

s.gui(locals())
```

Code 7. Génération d'événements synchronisés

Les protocoles de contrôle

La manipulation des processus audio via des contrôleurs externes est très efficace pour produire des variations de paramètres aux contours naturels. Pyo supporte deux protocoles de communication, soit les protocoles MIDI et OSC.

MIDI - Musical Instrument Digital Interface

Le protocole MIDI est très présent, encore aujourd'hui, dans les studios de production musicale. Que ce soit pour la génération d'événements, à l'aide du clavier, ou pour la manipulation de paramètres en continu, à l'aide de potentiomètres, le MIDI offre une interaction simple et directe avec le processus musical. L'interface désirée doit être spécifiée au serveur audio, avant initialisation (c'est-à-dire avant l'appel de la méthode *boot*), afin de pouvoir récupérer les signaux dans le programme.

```
from pyo import *

# Recupere l'interface MIDI par defaut
idev = pm_get_default_input()

# Creation du serveur
s = Server()
# Configuration du MIDI
s.setMidiInputDevice(idev)
# Initialisation du serveur
s.boot()

# Pitch bend et note MIDI
bend = Bendin(brange=2, scale=1)
```

```
note = Notein(poly=8, scale=1, first=0, last=127)
fr = note['pitch'] * bend
# Enveloppe d'amplitude
amp = MidiAdsr(note['velocity'], release=1, mul=.3)
# Synth stereo
a = SumOsc(freq=fr, ratio=[.245,.255], index=0.7, mul=amp).out()

s.gui(locals())
```

Code 8. Synthétiseur MIDI

OSC - Open Sound Control

Le protocole OSC [10], développé au CNMAT en 1997, permet d'échanger des données de contrôle entre différents environnements, logiciels ou matériels, via des envois réseaux. Le protocole peut transmettre plusieurs flux de données sur un même port d'écoute, en identifiant les canaux de data via un système d'adresses. Les adresses, spécifiées à l'aide de chaînes de caractères, utilisent la syntaxe des chemins d'accès sous les systèmes à base *unix*, avec la barre oblique (/) comme séparateur. Voici un exemple où un procédé de granulation sonore est contrôlé à l'aide d'une tablette graphique. La position du pointeur en abscisse donne la position de lecture dans le fichier sonore tandis que la position en ordonnée contrôle la hauteur de lecture entendue.

```
from pyo import *

s = Server().boot()

table = SndTable("/home/olivier/rfla2016/voice.aif")
env = HannTable()

# Observe le port 9000, position X, Y du crayon
rec = OscReceive(port=9000, address=['/wacom/pen/x', '/wacom/pen/y'])

# rec['/wacom/pen/x'] -> position de lecture
# rec['/wacom/pen/y'] -> hauteur du son
pos = Port(rec['/wacom/pen/x'], .05, .05, mul=table.getDur()*44100.)
pit = Port(rec['/wacom/pen/y'], .05, .05, mul=2, add=-1)

# legeres variations de la duree des grains
dur = Noise(mul=0.002, add=.1)
grain = Granulator( table=table,    # table contenant les echantillons
                    env=env,        # enveloppe des grains
                    pitch=pit,      # hauteur globale des grains
                    pos=pos,        # position de lecture
                    dur=dur,        # duree des grains en secondes
                    grains=32,      # nombre de grains
                    basedur=.1,     # duree de reference
                    mul=.1).out()

s.gui(locals())
```

Code 9. Contrôle via le protocole OSC

Création d'une interface graphique

Les composantes audio de pyo étant des objets python standards, la communication avec une interface graphique peut s'effectuer sans intermédiaire car les deux modules existent dans le même espace mémoire d'une application unique. L'exemple suivant illustre l'utilisation de la librairie wxPython pour créer une interface graphique (image ci-dessous) et communiquer les données de contrôle aux objets audio. Comme le serveur et le synthétiseur audio sont créés à l'intérieur de la même classe que le bouton et le potentiomètre, les fonctions appelées par ces derniers n'ont rien d'autre à faire que de transférer directement les valeurs aux variables *self.server* et *self.synth* pour opérer des changements dynamiques du résultat sonore.

```
import wx
from pyo import *

class MyFrame(wx.Frame):
    def __init__(self, parent, title, pos, size):
        wx.Frame.__init__(self, parent, -1, title, pos, size)
```

```
        self.server = Server().boot()
        self.synth = SuperSaw(freq=[330,331], mul=.2).out()

        audio = wx.ToggleButton(self, -1, "on / off", (90,10))
        audio.Bind(wx.EVT_TOGGLEBUTTON, self.handleAudio)
        label = wx.StaticText(self, -1, "Freq", (13,62))
        fr = wx.Slider(self,-1,330,50,600,(50,45),(200,-1),wx.SL_LABELS)
        fr.Bind(wx.EVT_SLIDER, self.changeFreq)

        self.Show()

def handleAudio(self, evt):
    if evt.GetInt() == 1: self.server.start()
    else: self.server.stop()

def changeFreq(self, evt):
    self.synth.freq = [evt.GetInt(), evt.GetInt()+1]

app = wx.App(False)
frame = MyFrame(None, 'Super Saw Emu', (25,25), (270,150))
app.MainLoop()
```

Code 10. Création d'une interface graphique de contrôle

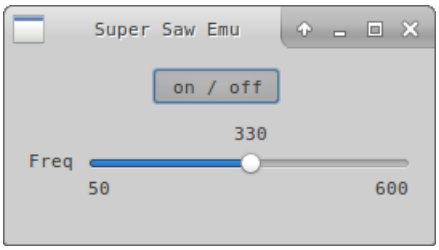

FIGURE 6. Interface graphique maison.

4. Exemples concrets

Dans cette section seront illustrés des exemples concrets de synthèse sonore, de traitements appliqués au signal audio, de composition algorithmique et de développement de logiciels.

Synthèse sonore

Le programme ci-dessous implémente un type de synthèse appelé *Bucket-Brigade Device* [11]. Le procédé, initialement conçu à l'aide de circuits analogues, consiste à établir une boucle entre deux lignes de délai, la sortie de la première ligne alimentant l'entrée de la seconde et vice versa. Une onde sinusoïdale, de fréquence variable, servira de matière première pour tout le circuit. Cette forme de synthèse peut être développée de différentes façons, selon le type de processus appliqué au signal à chacun des tours de boucle initiés par les lignes de délai. Dans cet exemple, une modulation en anneaux suivi d'un filtrage passe-bas constituent les variations dynamiques qui seront appliquées, en un cycle récursif, sur le signal de départ.

```
from pyo import *

s = Server(sr=48000, nchnls=2, buffersize=512, duplex=0).boot()

# Source – onde sinusoidale
t = HarmTable(size=32768)
src = Osc(t, 100)
src.ctrl(title="Input oscillator controls")
in_src = src * 0.025

# Amplitude de reinjection
feed = .8
cross_feed = .99
```

```
# Premiere ligne de delai + modulation + filtrage
del_1 = Delay(in_src, delay=Sine(.005, 0, .05, .25))
sine_1 = Osc(t, Sine(.007, 0, 50, 250))
ring_1 = del_1 * sine_1
filt_1 = Biquad(ring_1, 3000)

# Seconde ligne de delai
del_2 = Delay(in_src, delay=Sine(.003, 0, .08, .3))
sine_2 = Osc(t, Sine(.008, 0, 40, 200))
ring_2 = del_2 * sine_2
filt_2 = Biquad(ring_2, 3000)

# Reinjections croisees
cross_1 = filt_2 * cross_feed
cross_2 = filt_1 * cross_feed

# Re-assignation des entrees
del_1.setInput(filt_1 * feed + cross_1 + in_src)
del_2.setInput(filt_2 * feed + cross_2 + in_src)

# Sortie audio
mix = Mix([filt_1, filt_2], voices=2)

s.gui(locals())
```

Code 11. Circuit de synthèse *Bucket-Brigade Device*

Traitement du signal audio

Cet exemple illustre l'implémentation, à l'aide d'outils de base, d'une réverbération digitale selon un des algorithmes de Schroeder [9]. Manfred Schroeder, des Laboratoires Bell, est le pionnier des réverbérateurs artificiels. Ses algorithmes proposent l'utilisation de filtres récursifs pour simuler l'accumulation des milliers d'échos produits par les réflexions du son sur les obstacles d'une salle. Dans ce programme, le signal source passe tout d'abord par quatre filtres en peigne en parallèle, dont les sorties sont additionnées. La somme des filtres en peigne est ensuite envoyée dans une série de deux filtres passe-tout afin de décupler le nombre d'échos entendus, formant le nuage de son très dense de la réverbération.

```
from pyo import *

s = Server(duplex=0).boot()

# Source audio
snd = SNDS_PATH + "/transparent.aif"
a = SfPlayer(path=snd, loop=True, mul=0.3).mix(2).out()

# Quatre filtres en peigne en parallele
comb1 = Delay(a, delay=[0.0297,0.0277], feedback=0.65)
comb2 = Delay(a, delay=[0.0371,0.0393], feedback=0.51)
comb3 = Delay(a, delay=[0.0411,0.0409], feedback=0.5)
comb4 = Delay(a, delay=[0.0137,0.0155], feedback=0.73)

# Somme des filtres en peigne
combsum = a + comb1 + comb2 + comb3 + comb4

# Deux filtres passe-tout en serie
all1 = Allpass(combsum, delay=[.005,.00507], feedback=0.75)
all2 = Allpass(all1, delay=[.0117,.0123], feedback=0.61)
# Filtre passe-bas et sortie audio
lowp = Tone(all2, freq=3500, mul=.2).out()

s.gui(locals())
```

Code 12. Réverbération digitale selon un modèle de Schroeder

Composition algorithmique

L'exemple de composition algorithmique ci-dessous illustre la création d'une boîte à rythmes. Quatre sons sont d'abord chargés en mémoire dans des tables. La lecture des tables est activée à l'aide de *trigs* générés par un objet **Beat**.

Cet objet construit des séquences rythmiques aléatoires en fonction des probabilités, données en argument, pour les temps forts (*w1*), les temps intermédiaires (*w2*) et les temps faibles (*w3*). Le dernier temps de chaque mesure appelle la fonction *change*, dont le rôle est de renouveler le matériel rythmique après quatre itérations des séquences courantes. Une réverbération est finalement appliquée aux lectures de sons avant l'envoi à la sortie audio.

```
from pyo import *
s = Server(sr=44100, nchnls=2, buffersize=512, duplex=0).boot()

# Chargement des sons en memoire
sons = ['/home/olivier/rfla2016/alum1.wav', '/home/olivier/rfla2016/alum2.wav',
        '/home/olivier/rfla2016/alum3.wav', '/home/olivier/rfla2016/alum4.wav']
tabs = SndTable(sons)
# Generation algorithmique de sequences rythmiques
seq = Beat(time=.12, w1=[90,30,30,20],
           w2=[30,90,50,40], w3=[0,30,30,40]).play()
# Lecture des sons sur reception des trigs
tap = TrigEnv(seq, table=tabs, dur=seq['dur']*2, mul=seq['amp']*0.5)
# Mix et reverberation
rev = WGVerb(tap.mix(2), feedback=.65, cutoff=3500, bal=.2).out()

x = 0
def change():
    global x
    x += 1
    if (x % 4) == 0:
        seq.new() # Nouvelles sequences apres 4 repetitions

# Chaque fin de mesure appelle la fonction "change"
call = TrigFunc(seq['end'][0], function=change)
s.gui(locals())
```

Code 13. Boîte à rythmes

Développement de logiciel

Le développement de logiciels est une des raisons principales de l'existence de pyo en tant que module Python. Un environnement unifié, où toutes les composantes nécessaires au fonctionnement du logiciel sont intégrées au même langage, offre au programmeur la possibilité de se concentrer sur le développement des algorithmes, des structures de contrôle et des processus musicaux, plutôt que sur la transmission de données, c'est-à-dire la communication entre les différentes parties de l'application. L'addition du module pyo permet une intégration simple et flexible de processus audio dans un programme écrit avec le langage python. Pour un aperçu exhaustif des capacités de la combinaison python et pyo à produire des applications sonores autonomes et originales, le lecteur est invité à visiter les logiciels suivants, développés par l'auteur. Tous ces logiciels sont gratuits, libres de sources et fonctionnent sous tous les systèmes d'exploitation majeurs.

Soundgrain - http ://ajaxsoundstudio/software/soundgrain/

Soundgrain est un logiciel de granulation sonore où l'utilisateur est invité à contrôler le processus en dessinant des trajectoires sur une surface de jeu. Plusieurs opérations sur les trajectoires sont possibles en cours de jeu : copier/coller, édition, déplacement, gel de la lecture, vitesse variable et bien d'autres.

Zyne - http ://ajaxsoundstudio/software/zyne/

Zyne est un synthétiseur modulaire contrôlable en MIDI. L'atout principal du logiciel est la possibilité pour l'utilisateur d'écrire ses propres modules de synthèse sous la forme de classes python. Plusieurs algorithmes de traitement par lots (*batch processing*) sont offerts pour permettre la création de banques de sons originales.

Cecilia5 - http ://ajaxsoundstudio/software/cecilia/

Cecilia5 est un logiciel de traitement sonore destiné à la production audio en studio. L'application offre une multitude de modules de traitement du son originaux. Un éditeur graphique de courbes permet de contrôler la trajectoire de chacun des paramètres au cours du temps. Les automations en provenance d'interfaces externes, MIDI ou OSC, peuvent être enregistrées, et éditées, à même l'éditeur graphique.

Conclusion

L'ajout du module pyo à un langage de programmation largement utilisé fait de Python un environnement idéal pour le développement de projets audio, de la génération de synthèse sonore au logiciel de traitement du son, en passant par la composition de musiques algorithmiques. Avec le langage Python, le programmeur a accès à de multiples modules externes permettant d'accomplir certaines tâches connexes au développement d'un algorithme musical. Des tâches comme la construction d'interfaces graphiques, la résolution de calcul vectoriel ou la communication avec le système d'exploitation. Le langage python jouit d'une grande communauté de développeurs, ce qui en fait un environnement de choix pour le développement d'une multitude d'algorithmes spécialisés. L'addition du module pyo à python permet aujourd'hui de créer des applications musicales complètes à l'aide d'un seul langage de programmation, favorisant ainsi une interaction simplifiée entre le moteur audio et les différentes composantes du programme. La librairie est disponible en téléchargement pour toutes les plates-formes majeures, en format source ou via des installeurs binaires.

Références

[1] Barry Vercoe and Dan Ellis. Real-time csound : Software synthesis with sensing and control. In *ICMC'90 Conference Proceedings*, pages 209–211. International Computer Music Association, 1990.

[2] James McCartney. Supercollider : a new real time synthesis language. In *ICMC'96 Conference Proceedings*, pages 257–258. International Computer Music Association, 1996.

[3] Miller Puckette. Pure data : another integrated computer music environment. In *Conference Proceedings*, pages 37–41. Second Intercollege Computer Music Concerts, 1996.

[4] Python Software Foundation. Python programming language – official website. `http://python.org/`, 1990-2013.

[5] Olivier Bélanger. Ounk - an audio scripting environment for signal processing and music composition. In *ICMC'08 Conference Proceedings*, pages 399–402. International Computer Music Association, 2008.

[6] Robin Dunn and Noel Rappin. *WxPython in Action*. Manning publications, Greenwich, CT, 2006.

[7] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90) :297–301, 1965.

[8] Richard Boulanger and Victor Lazzarini. *The Audio Programming Book*. MIT Press, Cambridge, Massachussets, 2010.

[9] Curtis Roads. *The Computer Music Tutorial*. MIT Press, Cambridge, Massachussets, 1996.

[10] Matthew Wright, Adrian Freed, and Ali Momeni. Opensound control : State of the art. In *NIME'03 Conference Proceedings*. Conference on New Interfaces for Musical Expression, 2003.

[11] Colin Raffel and Julius O Smith. Practical modeling of bucket-brigade device circuits. In *DAFx'10 Conference Proceedings*, Graz, Austria, 2010. Conference on Digital Audio Effects.