

MUS-2321 - Algorithmie et Effets Audionumériques 1

Olivier Bélanger

9 septembre 2018

Table des matières

1	Introduction à Pure Data	7
1.1	Quelques ressources internet	7
1.2	Pure Data - bref historique	7
1.3	Configuration de Pure Data	8
1.3.1	Path...	8
1.3.2	Startup...	9
1.3.3	Audio Settings...	9
1.3.4	Midi Settings...	11
1.4	L'environnement Pure Data	11
1.4.1	La fenêtre Pd	11
1.4.2	La fenêtre Patcher	11
1.4.3	La palette d'objets	12
1.4.4	Les connexions	14
1.4.5	Entrées chaudes versus entrées froides	15
1.5	Les types de messages	15
1.5.1	Bang	15
1.5.2	Nombres	16
1.5.3	Symbole	16
1.5.4	Liste	16
1.5.5	Table ou <i>array</i>	17
1.6	La hiérarchie des messages	17
1.7	Documentation des objets	17
1.8	Objets à étudier	19
1.9	Exercices	19
1.9.1	Génération d'une ligne mélodique aléatoire	19
1.9.2	Lectures	19
2	Boucles, décisions et automatismes	21
2.1	Lecture de segments en boucle	21
2.1.1	Incrémenteur	22
2.1.2	Espace mémoire	22
2.1.3	Registre	23
2.2	Décisions	24
2.3	Automatismes	25

2.3.1	Implantation d'algorithmes simples opérant des automatismes	25
2.3.2	Encapsulation	27
2.4	Objets à étudier	27
2.5	Exercices	28
2.5.1	Construction d'un métronome maison	28
2.5.2	Lecture de mélodies en boucle	28
2.5.3	Lectures	28
3	Introduction au traitement de signal audio	29
3.1	Lecture et écriture de fichiers sons sur le disque dur	29
3.1.1	Lecture sur le disque dur	29
3.1.2	Écriture sur le disque dur	30
3.1.3	Construction dynamique des noms de fichier	31
3.2	Lecture et écriture de fichiers sons en RAM	33
3.2.1	Lecture en RAM	33
3.2.2	Algorithmes de lecture en RAM	35
3.2.3	Écriture en RAM	37
3.3	Objets à étudier	38
3.4	Exercices	38
3.4.1	Lecture d'un fichier son par segmentation aléatoire	38
3.4.2	Lectures	38
4	Résolution de problèmes et aiguillage des données	39
4.1	Aiguillage des données	39
4.2	Aiguillage et contrôle	40
4.3	Gestion de la polyphonie	40
4.4	Éléments de hasard et contrôleurs MIDI	42
4.4.1	Variations mélodiques	42
4.4.2	Variations rythmiques	43
4.4.3	Contrôle des variations	43
4.5	Objets à étudier	45
4.6	Exercices	45
4.6.1	Exercices de révision	45
4.6.2	Lectures	45
5	La synthèse soustractive	47
5.1	Les types de sources	47
5.1.1	Les sources bruitées	47
5.1.2	Les sources harmoniques	48
5.2	L'objet delay~ et les filtres FIR simples	48
5.3	Les filtres IIR de bases	51
5.4	Les résonateurs	52
5.5	Les filtres en peigne	53
5.6	Le contrôle des paramètres	55
5.6.1	Oscillateur à basse fréquence	55

5.6.2	Variations aléatoires	55
5.7	Objets à étudier	56
5.8	Exercices	57
5.8.1	Exercices de révision	57
5.8.2	Lectures	57
6	Analyse des données et objets maisons	59
6.1	Informations en provenance du clavier MIDI	59
6.2	Un transpositeur intelligent	60
6.3	Les expressions et les conditions	60
6.3.1	Les expressions	61
6.3.2	Les conditions	62
6.4	Création d'un compteur maison	62
6.4.1	Un compteur simple	62
6.4.2	Ajout d'un contrôle de la direction	63
6.5	Fabrication d'objets externes	64
6.6	Ajout d'arguments d'initialisation au compteur maison	65
6.7	Contrôle du compteur et élaboration d'un contenu musical	67
6.8	Objets à étudier	68
6.9	Exercices	68
6.9.1	Exercice n° 1	68
6.9.2	Exercice n° 2 (expressions)	68
6.9.3	Exercice n° 3 (conditions)	68
7	Algorithmie et automate	69
7.1	Algorithmie rythmique	69
7.1.1	Rythmes aléatoires	69
7.1.2	Rythmes cellulaires	70
7.2	Introduction à l'objet line	70
7.2.1	Concept de "grain"	71
7.2.2	Contrôles aléatoires	71
7.2.3	Plusieurs étages de génération de données	72
7.3	Exercices	72
7.3.1	Exercice n° 1	72
7.3.2	Exercice n° 2	72
8	La synthèse additive	73
8.1	Construction d'un synthétiseur MIDI par synthèse additive	73
8.1.1	Étape 1 - Oscillateur simple	73
8.1.2	Étape 2 - Variations aléatoires des amplitudes	74
8.1.3	Étape 3 - Ajout d'une enveloppe d'amplitude globale	75
8.1.4	Étape 4 - Contrôle des fréquences	75
8.1.5	Étape 5 - Conversion en synthétiseur MIDI et gestion de la polyphonie	77
8.1.6	Étape 6 - Génération et lecture d'une forme d'onde complexe	79
8.2	Synthèse par balayage d'un fichier son	81

9	Les synthèses par modulation	83
9.1	Synthèse par modulation d'amplitude	83
9.1.1	Différence entre signal bipolaire et signal unipolaire	83
9.1.2	Modulation d'amplitude et trémolo	84
9.1.3	Modulation en anneaux	85
9.2	Synthèse par modulation de fréquence	86
9.2.1	Implémentation	86
9.2.2	Contrôle dynamique des paramètres	87
9.2.3	La modulation de fréquence à plusieurs oscillateurs	88
9.2.4	Un oscillateur récursif	89
9.3	Exercices	91
9.3.1	Exercice n° 1	91
9.3.2	Exercice n° 2	91
9.3.3	Exercice n° 3	91
10	Stockage et manipulation de données	93
10.1	Séquenceur polyphonique en temps absolu	93
10.1.1	Mécanisme d'enregistrement	93
10.1.2	Lecture et manipulation des données enregistrées	95
10.1.3	Interface de contrôle	96
10.2	Améliorations possibles	97
11	La distorsion non-linéaire	99
11.1	Fonction de transfert	99
11.1.1	Génération d'une fonction de transfert	100
11.1.2	Lecture d'une table bipolaire (<i>table lookup</i>)	101
11.2	Les fonctions mathématiques continues	101
11.2.1	Le sur-échantillonnage	103
11.3	Le secret d'une bonne distorsion	103

Chapitre 1

Introduction à Pure Data

1.1 Quelques ressources internet

- Site web officiel de Pure Data : <http://puredata.info>
- Page de téléchargement de Pure Data : <http://puredata.info/downloads>
- Manuel français de Pure Data : <http://fr.flossmanuals.net/puredata/>
- Documentation, références, tutoriels et articles : <http://puredata.info/docs>
- The Theory and Technique of Electronic Music (Miller Puckette) :
<http://msp.ucsd.edu/techniques.htm>
- Designing Sound - Introduction à Pure Data (Andy Farnell) :
http://aspress.co.uk/ds/pdf/pd_intro.pdf

1.2 Pure Data - bref historique

- Extrait du manuel FLOSS :

Pure Data fait parti de la famille des langages de programmation par patchs comme JMax, VVVV, Ingen, etc. Ces langages constituée de boîtes et de ficelles tire son origine de la conception modulaire expérimentée dans les premiers programmes musicaux de Max Mathews au cours des années 1950, programmes qui ont par la suite inspiré les premiers synthétiseurs analogiques.

Miller Puckette est le créateur de Pure Data. En 1988, à l'IRCAM (<http://www.ircam.fr>), une institution française dédiée à la recherche et à la création musicale contemporaine, il développe l'éditeur Patcher (<http://msp.ucsd.edu/Publications/icmc88.pdf>). Ce logiciel fut revendu à la société Opcode pour créer bien plus tard Max/MSP. Pour sa part, Miller Puckette reprit la conception de Patcher pour en faire un logiciel libre à des fins musicales : Pure Data.

Les conditions de la vente du logiciel Patcher à une entreprise commerciale, alors que la phase de recherche initiale était intégralement financée par des fonds publics, font encore aujourd'hui débat. Pure Data est pour sa part publié sous la licence libre BSD. Tout en garantissant de toujours pouvoir utiliser, partager et modifier librement Pure Data, cette licence autorise la privatisation du code source de ce programme : c'est ainsi que les créateurs de Max/MSP ont pu légalement copier le code source de Pure Data pour créer la partie qui s'occupe du traitement de signal audio de leur logiciel propriétaire.

1.3 Configuration de Pure Data

Le menu *Media* permet d'adapter le logiciel à l'environnement système sur lequel il est installé. On peut entre autre y sélectionner les interfaces audio et MIDI désirées, un clavier MIDI ou une carte de son externe par exemple. La première étape afin de s'assurer que Pure Data est bien configuré est d'ouvrir la fenêtre « Test Audio and MIDI... » et de vérifier que le son se rend bien aux pilotes audio (et ultimement aux haut-parleurs). Sous la mention « TEST TONES », allumez le volume du signal test en appuyant sur 60 ou 80 (valeurs en dB). Si vous entendez une onde sinusoïdale à 440 Hertz, cela signifie que les sorties audio de Pure Data sont correctement configurées. Si une entrée micro est activée sur le système, la valeur *rms* (amplitude moyenne du signal d'entrée) sous la mention « AUDIO INPUT » devrait varier.

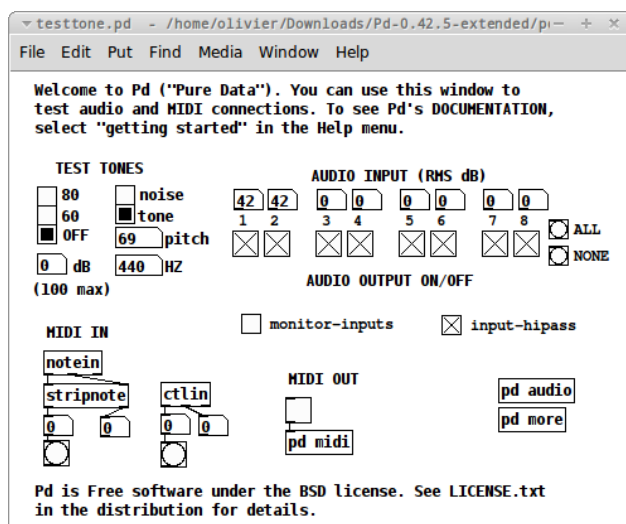


FIGURE 1.1 – Tester les entrées et sorties audio de Pure Data.

Quelques configurations supplémentaires peuvent être ajustées sous le menu *Media* → *Preferences*.

1.3.1 Path...

La fenêtre *Paths...* permet de spécifier à Pure Data le chemin vers les dossiers où sont sauvegardées les ressources externes ainsi que les objets maisons qui doivent être accessibles en tout temps par l'application. À l'ouverture de Pure Data, le contenu de ces dossiers sera connu et utilisable dans le développement de nouveaux projets. Il est conseillé de conserver les objets externes (téléchargés sur le web) ainsi que sa librairie d'objets personnels dans des dossiers hors de la distribution de Pure Data et de simplement spécifier où le logiciel doit chercher pour avoir accès à ces ressources. Cela facilite la migration vers une plate-forme différente ou le partage de projets incluant des ressources personnelles devant accompagner la « patch » principale.

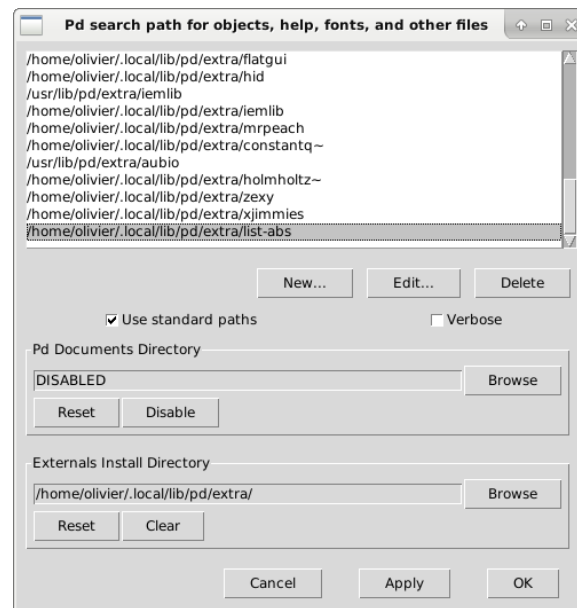


FIGURE 1.2 – Configuration des endroits où Pure Data doit chercher des ressources.

1.3.2 Startup...

La fenêtre *Startup...* permet de spécifier à Pure Data les bibliothèques qui doivent être activées à l'ouverture du logiciel. Une bibliothèque est une collection d'objets réunis dans un seul fichier binaire. Ce dernier doit être chargé dans l'environnement afin d'avoir accès aux objets individuellement. L'image ci-dessous présente trois bibliothèques contenant ce type de structure, *Gem*, qui permet de manipuler les flux vidéo en temps réel, *Aubio*, une collection d'outils pour l'analyse du signal audio et *zexy*, une bibliothèque contenant une multitude d'utilitaires.

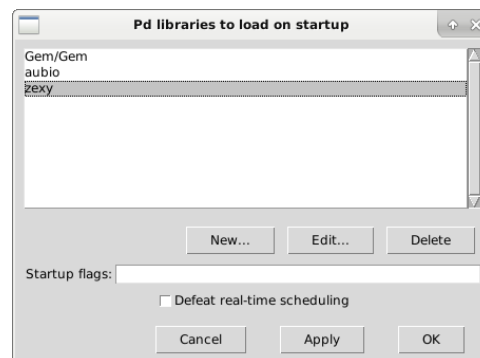


FIGURE 1.3 – Configuration des bibliothèques à charger au démarrage.

1.3.3 Audio Settings...

La fenêtre *Audio Settings...* prendra une allure différente selon l'interface choisie et permettra d'ajuster les paramètres audio de l'environnement tels que les entrées et sorties de la carte de son, la fréquence d'échantillonnage et la taille des blocs d'échantillons calculés par l'engin audio de Pure Data. Ces spécifications auront une incidence considérable sur la réponse temporelle du programme ainsi que sur la charge de processeur impliquée. Le compromis inévitable entre efficacité et rapidité d'exécution sera approfondi ultérieurement.

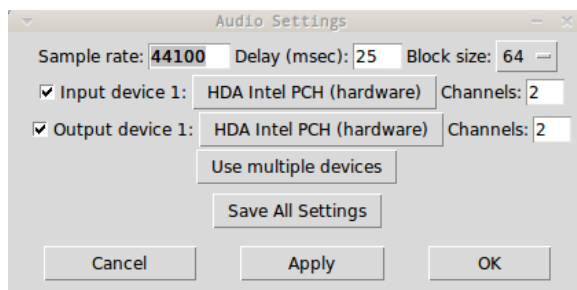


FIGURE 1.4 – Configuration audio de Pure Data.

1.3.4 Midi Settings...

La fenêtre *Midi Settings...* permet à l'utilisateur de sélectionner les ports d'entrées et de sorties de l'interface MIDI sélectionnée.

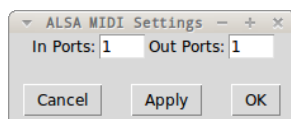


FIGURE 1.5 – Configuration midi de Pure Data.

1.4 L'environnement Pure Data

Pure Data est un environnement de programmation graphique utilisant le principe de flux de données. L'utilisateur met en place des processeurs (appelés « objets ») qu'il connecte les uns aux autres à l'aide de fils afin de créer une structure de contrôle ou une chaîne de traitement de signal audio. Les objets reçoivent les données en entrée, transforment l'information selon le processeur choisi et retournent de nouvelles valeurs. Le flux de data poursuit ensuite son chemin vers le prochain objet jusqu'à ce que la fin de la structure soit atteinte. Dans le cas d'une chaîne de traitement audio, le circuit est répété en boucle tant que l'audio est allumé.

1.4.1 La fenêtre Pd

La fenêtre Pd est celle qui apparaît à l'ouverture du logiciel. Elle ne peut être éditée mais donne de précieux renseignements sur la configuration du logiciel. Elle indique, par exemple, si un problème de pilote audio est survenu à l'ouverture du logiciel. À tout moment, on peut utiliser cette fenêtre pour afficher des données provenant de notre module afin de découvrir l'erreur de programmation qui corrompt le processus. Idéalement, cette fenêtre doit toujours être visible lors de l'élaboration d'un programme, car elle nous avertit, à l'aide de messages d'erreurs, si des connexions erronées sont effectuées. La boucle audio peut être démarrée via la fenêtre Pd en cochant la case « DSP ».

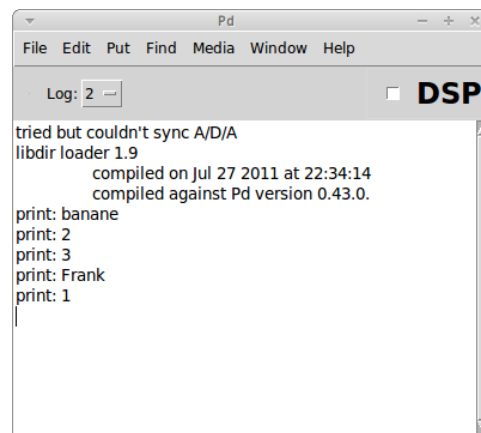


FIGURE 1.6 – La fenêtre Pd.

1.4.2 La fenêtre Patcher

La fenêtre patcher est la fenêtre principale d'un projet Pure Data, celle servant à éditer le programme. Elle consiste en une page blanche sur laquelle des objets seront connectés pour créer une chaîne de contrôle ou de traitement audio. Il y a deux modes opératoires dans cette fenêtre, un mode d'édition et un mode de jeu, que l'on peut alterner à l'aide du raccourci-clavier *Ctrl+E* (*Cmd+E* sous OSX). Toute modification du programme doit être effectuée dans le mode d'édition tandis que la manipulation des différents objets au cours d'une performance se fait dans le mode de jeu. Le mode d'édition est illustré par un curseur en forme de main pointant du doigt alors qu'en mode de jeu, le curseur reprend la forme d'un pointeur en forme de flèche.

1.4.3 La palette d'objets

Les différents objets possibles pour construire un programme sont accessible, soit à l'aide de raccourcis-clavier, soit via le menu *Put*. Il est fortement recommandé de développer l'habitude d'utiliser les raccourcis-clavier, cela augmente considérablement la vitesse de développement des projets.

Boîte d'objet

La boîte « objet » est l'outil principal de Pure Data. C'est avec cet objet que l'on met en place les différents processeurs nécessaires à la construction de l'algorithme désiré. Un processeur est un modificateur de données. Les opérateurs arithmétiques, les contrôleurs de direction, les oscillateurs et les filtres font tous partis de la gamme des processeurs. Ce sont généralement des structures primitives servant à construire des algorithmes plus complexes. La boîte d'objet est définie visuellement par un rectangle à l'intérieur duquel on inscrit le nom du processeur désiré.

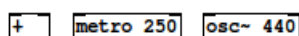


FIGURE 1.7 – La boîte d'objet en Pure Data.

Chaque objet, en fonction du processeur utilisé, présente un certain nombre d'entrées (onglets noirs du haut), pour connecter le data à modifier, et un certain nombre de sorties (onglets noirs du bas), d'où sort le data transformé. Les connexions, de sortie en entrée, déterminent l'ordre des opérations à effectuer sur les données de départ.

Boîte à message

La boîte à message, représentée par un rectangle au côté droit incurvé, permet de construire ou de conserver des messages de contrôle pouvant être activés à tout moment par un clic de souris (ou par l'envoi d'un *bang* en entrée). Un objet, selon la nature du processeur choisi, pourra accepter différents messages afin de modifier son comportement. Les messages seront donc fort utiles pour communiquer des informations importantes aux différents objets ainsi que pour la conservation de data destiné à une utilisation ultérieure. Le contenu d'une boîte à message peut être des chiffres, des symboles, une liste de valeurs ou un mélange de chiffres et de symboles.

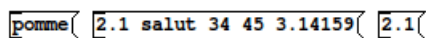


FIGURE 1.8 – La boîte à message en Pure Data.

Boîte à chiffre

La boîte à chiffre remplit deux rôles en Pure Data. D'abord, elle sert à afficher des valeurs numériques permettant une meilleure compréhension des différentes étapes de l'algorithme. Elle permet aussi de modifier des valeurs numériques en cours de performance. Par défaut, la boîte à chiffre affiche des entiers, mais comme tous les nombres sont représentés à l'interne en nombre à virgule flottante, elle peut aussi afficher des nombres décimaux (*floats*). On reconnaît visuellement la boîte à chiffre à son coin supérieur droit tronqué. On peut faire défiler les valeurs en gardant la souris enfoncée sur la boîte, puis en glissant vers le haut ou vers le bas. Cela créera un

défilement d'entiers. Pour une précision de contrôle plus fine, il suffit de tenir la touche SHIFT enfoncée durant défilant, l'incrément sera alors de 0.01. Il est aussi possible d'entrer des valeurs directement au clavier. Pour cela, cliquez sur l'objet, entrez la valeur au clavier puis appuyez sur la touche RETURN.

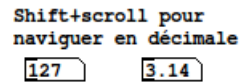


FIGURE 1.9 – La boîte à chiffre.

Boîte à symbole

La boîte à symbole remplit sensiblement le même rôle que la boîte à chiffre mais offre la possibilité d'entrer du texte. Elle a le même aspect visuel que la boîte à chiffre (coin supérieur droit tronqué) mais est un peu plus étendue. Pour générer un nouveau message, on clique sur l'objet, on entre le texte au clavier puis on appuie sur RETURN. Dans l'exemple ci-dessous, la boîte à symbole permet de spécifier le nom du fichier à lire.

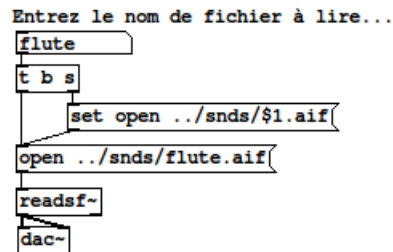


FIGURE 1.10 – La boîte à symbole.

Commentaire

Les commentaires ne jouent aucun rôle dans le fonctionnement du programme mais permettent de donner de précieuses informations sur l'utilité des différentes sections de la patch. Les commentaires facilitent la diffusion et la compréhension des programmes à des tiers n'ayant pas participé au développement.

Éléments graphiques

Pure Data offre quelques éléments d'interface graphique permettant d'interagir avec le processus ou d'en visualiser le comportement. Le bang, le bouton à deux états (*toggle*), les potentiomètres, les boutons radio et la table multi-points (*array*) sont autant d'éléments pouvant être

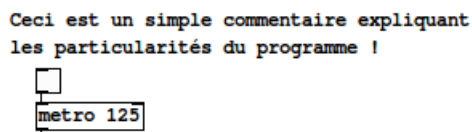


FIGURE 1.11 – Un commentaire.

contrôlés à la souris que par des valeurs en entrée. L'aspect visuel de ces objets peut être modifier via la fenêtre de propriétés de l'objet (clic droit sur l'objet → *Properties*). Seul le vu-mètre n'a qu'un rôle de témoin, sans possibilité d'interaction. Cet objet est configuré pour représenter des valeurs en décibels dans un ambitus de -99.0 à +12.0 dB. L'échelle des signaux oscillant entre -1.0 et +1.0 doit d'abord être ajustée avant d'arriver au vu-mètre.

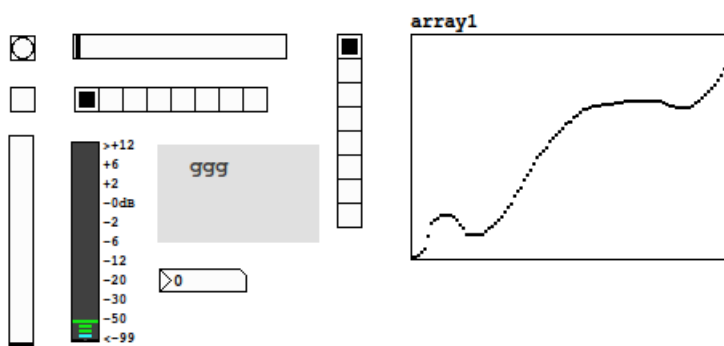
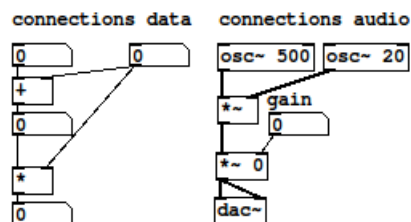


FIGURE 1.12 – Les différents objets d'interface graphique.

1.4.4 Les connexions

Les connexions (fils entre les boîtes) servent à acheminer l'information, nombres, messages ou signal audio, d'un objet à l'autre. Il y a deux types de connexions possibles en Pure Data.

Le premier type est la connexion de data, représenté par un fil mince. Ces fils véhiculent de l'information de contrôle tel que des messages, des nombres ou des *bangs*. Ils servent à créer la structure de contrôle. Le second type est la connexion audio, représenté par un fil gras. Ces connexions transportent le signal audio d'un objet à l'autre. Notez que le nom des processeurs audio se termine toujours par le caractère tilde (~).



1.4.5 Entrées chaudes versus entrées froides

L'ordre d'exécution des différentes opérations d'un algorithme étant généralement important, il est souvent nécessaire de préparer le terrain avant de lancer le processus en place. Le système mis en place dans Pure Data est de ne pas générer le calcul de l'objet (et par conséquent une sortie) aussitôt qu'une donnée arrive en entrée. Les différentes entrées d'un objet n'ont pas la même implication au niveau de l'exécution du processus. Seule l'entrée de gauche (entrée chaude) provoque le calcul et une valeur en sortie, les autres entrées (entrées froides, vers la droite) ne font que placer des données dans la mémoire de l'objet afin de le préparer pour le prochain calcul. Ainsi, l'entrée de droite d'un objet `+` ne fait que préparer la variable à droite de l'addition pour le calcul final, qui sera exécuté lorsqu'une donnée arrivera dans l'entrée de gauche. Si une sortie est souhaitée peut importe l'entrée qui reçoit du data, on utilisera un objet **trigger** pour acheminer un *bang* à l'entrée de gauche après avoir comblées les entrées froides.

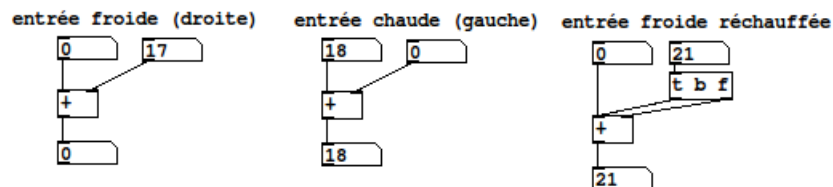



FIGURE 1.13 – L'entrée de gauche provoque le calcul.

1.5 Les types de messages

1.5.1 Bang

Le bang  est le message de base en Pure Data. C'est un simple déclencheur que l'on peut utiliser pour démarrer un processus quelconque, tel qu'un enregistrement audio dans la mémoire vive, le calcul d'une opération mathématique ou le départ d'un processus en boucle. La sortie d'un métronome, par exemple, envoie des bangs de façon cyclique dans le temps. Le bang est un message mais aussi un objet graphique, sur lequel on peut cliquer.

1.5.2 Nombres

Les nombres sont à la base de tout processus informatico-musical. Le signal audio lui-même est représenté par une suite de nombres à virgule flottante (*float*), naviguant entre -1.0 et +1.0, qui correspond à l'amplitude des échantillons successifs. Les traitements audio ne sont que des opérations mathématiques appliquées sur chacun des échantillons du signal. Il est à noter que dans l'environnement Pure Data, tous les nombres sont considérés comme des nombres à virgule flottante, même dans le cas où un entier est affiché à l'écran. Ceci signifie donc que toutes les opérations mathématiques sont effectuées en décimale. Pour obtenir une valeur entière, on aura parfois recours à un objet **int**, qui élimine la partie décimale et ne conserve que la partie entière d'un nombre.

1.5.3 Symbole

Un symbole est généralement un mot ou du texte constitué de lettres, de chiffres et de caractères de ponctuation tels que le point, le souligné ou le tiret. Un symbole peut représenter n'importe quoi et servira à envoyer des messages aux objets ou à identifier un fichier à lire sur le disque. À noter qu'un symbole ne peut contenir d'espace (ce que Pure Data considérerait comme une liste de symboles). Pour créer un symbole représentant un nom de fichier contenant des espaces, il faudra utiliser l'objet **makefilename** ou bien l'objet **openpanel**, un dialogue standard qui conserve aussi les caractères d'espacement.

1.5.4 Liste

Une liste est une succession ordonnée de nombres et/ou de symboles. Les éléments d'une liste sont séparés par des espaces et peuvent provenir d'une boîte à message, d'une lecture de fichier texte ou être assemblés à l'aide de divers objets tel que **pack**, un objet au sujet duquel nous élaborerons plus tard. Lorsqu'une liste de valeurs est envoyée à un objet quelconque (en cliquant sur une boîte à message par exemple), tous les éléments de la liste sont envoyés en même temps. Ils pourront donc être traités soit comme un ensemble ou bien un par un, en bouclant sur les éléments de la liste. La séquence $\{0\ 2\ 4\ 5\ 7\ 9\ 11\}$, dans une boîte à message, sera interprété par Pure Data comme une liste et pourrait servir à identifier le premier octave d'une gamme de Do majeur en note MIDI.

1.5.5 Table ou *array*

Une table, ou un *array*, est une structure de données à deux dimensions permettant de garder du data en mémoire. Ce type de structure sera utile notamment pour enregistrer des séquences d'échantillons destinées à une utilisation ultérieure, pour dessiner des enveloppes de contrôle ou pour afficher les variations dans un signal audio (à la manière d'un oscilloscope). On crée un objet *array* via le menu *Put* → *array* et un dialogue de propriétés apparaît permettant de spécifier le nom, la taille et le type désiré. La table de visualisation, créée en même temps que l'*array* permet non seulement de visualiser le contenu de la mémoire mais aussi d'interagir avec elle en dessinant des trajectoires à même l'interface. Le contenu de la mémoire peut aussi être lu et modifié de façon algorithmique avec les objets **tabread**, **tabwrite**, **tabread4~** et **tabwrite~**.

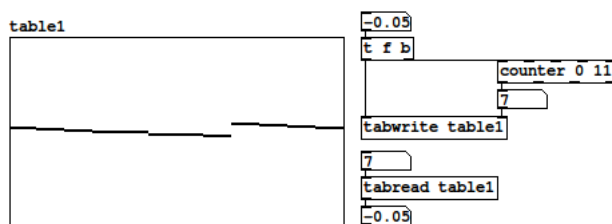


FIGURE 1.14 – Lecture et écriture dans un *array*.

1.6 La hiérarchie des messages

En Pure Data, lorsque la sortie d'un objet est connectée dans l'entrée de plusieurs autres objets, l'ordre d'envoi des messages est fonction de l'ordre de création des objets, du plus ancien au plus récent. Dans un algorithme où l'ordre des opérations est cruciale, cela peut vite devenir un problème puisqu'il est pratiquement impossible de se rappeler dans quel ordre les objets ont été mis en place. Par contraste, MaxMSP utilise la disposition des objets dans la page pour déterminer l'ordre d'arrivée des messages, de droite à gauche et de bas en haut. Avec Pure Data, il est nécessaire d'adopter une approche plus rigoureuse et de spécifier systématiquement, avec l'objet **trigger** (ou seulement **t**), l'ordre dans lequel on désire effectuer les opérations. Dans l'exemple suivant, les deux versions d'une même patch ne donnent pas tout à fait le même résultat. Seule la deuxième version permet de prédire exactement l'ordre d'arrivée des messages à l'objet **print**.

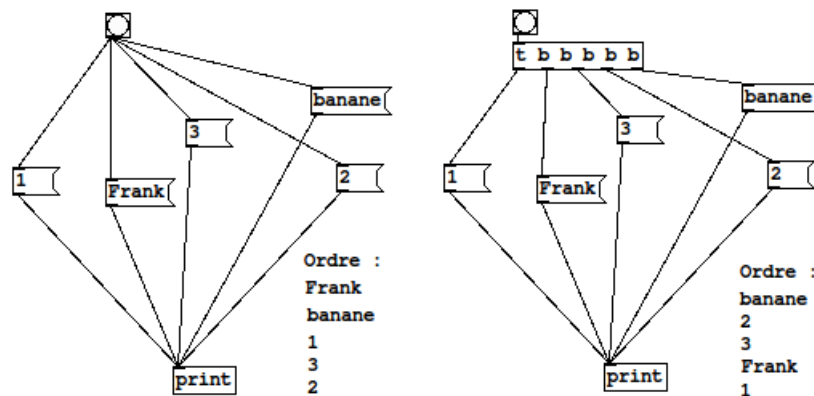


FIGURE 1.15 – Hiérarchie d'envoi des messages.

1.7 Documentation des objets

La documentation des objets, sous la forme de patchs Pure Data, est toujours accessible à même l'application. Un clic-droit sur l'objet ouvre un petit menu contextuel donnant accès à la fenêtre de propriétés (si applicable) ainsi qu'au *help* de l'objet. Comme le *help* est une patch Pure Data, il est tout à fait permis de copier-coller des morceaux de code et de s'en servir comme point de départ dans le développement de projets personnels.

Une fenêtre donnant accès à la page de documentation de tous les objets connus de l'application est disponible via le menu *Help* → *Browser...*

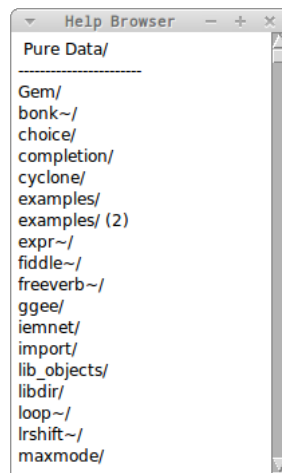


FIGURE 1.16 – Fenêtre de documentation.

1.8 Objets à étudier

- bang, toggle
- boîte à chiffre
- arithmétique (+, -, /, *, div)
- print, trigger
- metro, random
- makenote, noteout

1.9 Exercices

1.9.1 Génération d'une ligne mélodique aléatoire

Étape 1

Un métronome, activé à l'aide d'un *toggle*, doit générer de façon continue des valeurs entières pigées au hasard entre 48 et 72.

Étape 2

Ces valeurs correspondent à des notes MIDI qui doivent être envoyées à un synthétiseur MIDI virtuel à l'aide des objets **makenote** et **noteout**.

Étape 3

Ajoutez ensuite une voix d'harmonisation à l'octave inférieur.

Étape 4

Créez un mécanisme qui, à toutes les 5 secondes, change la vitesse du métronome de façon aléatoire.

Étape 5

Ajoutez un contrôle de la durée des notes indépendant de la vitesse de génération.

Étape 6

Modifier le mécanisme qui fait varier la vitesse de génération pour n'obtenir que des vitesses comprises dans le groupe {125, 250, 375, 500}.

1.9.2 Lectures

Parcourir les sections **Introduction**, **Installation** et **Configuration** du [manuel FLOSS](#) de Pure Data.

Chapitre 2

Boucles, décisions et automatismes

2.1 Lecture de segments en boucle

Erreur fréquente

Le message **stack overflow** apparaît lorsqu'une boucle perpétuelle et instantanée se glisse dans le système. Ce type de boucle est généralement dû à une erreur de programmation, par exemple, quand la sortie d'un objet est branchée dans l'entrée de ce même objet. La sortie provoque une entrée qui provoque une sortie qui provoque une entrée, etc. L'horloge de contrôle de Pure Data est alors arrêtée momentanément afin de permettre à l'utilisateur de réparer les connexions erronées.

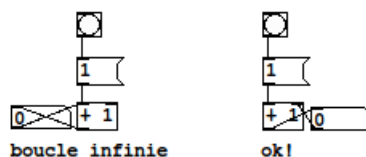


FIGURE 2.1 – Incrément du compte via l'entrée de droite.

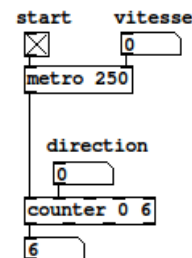
Joueur de gamme

Il y a trois éléments essentiels à la construction d'un module de lecture de notes en boucle : un incrémenteuse, un espace mémoire pour conserver des valeurs et un mécanisme de contrôle des distances. Il est à noter qu'il existe un nombre infini de façons d'arriver à un résultat intéressant, et que les techniques exposées dans ce cours n'ont pour but que de fournir des exemples et de stimuler l'imaginaire de chacun...

2.1.1 Incrémenteur

La première étape consiste donc à créer un mécanisme permettant d'incrémenter un compte d'entiers qui servira à lire des données de façon ordonnée.

Un incrémenteur peut être un simple compteur, dont on peut contrôler le minimum, le maximum et la direction. L'objet **counter** remplit cette fonction. Chaque fois que l'objet reçoit un *bang*, le compte augmente de un. Quand le maximum est atteint, le compteur est remis à zéro. On contrôle la direction à l'aide d'un entier dans la deuxième entrée et le compte maximum via la dernière entrée. Si on désire un minimum autre que zéro, il suffit de placer une addition à la sortie de l'objet. Un compteur plus sophistiqué permettrait également de déterminer la distance entre deux valeurs successives, qu'on appelle le *pas*. Nous construirons éventuellement un compteur plus évolué.



2.1.2 Espace mémoire

Un espace mémoire, comme l'objet **funbuff**, sert à enregistrer des données sous forme de paires de chiffres, le premier correspond à une adresse et le second au data à conserver. Il est très simple de mettre en mémoire les degrés d'une gamme dans un **funbuff** pour ensuite les rappeler par leur adresse respective. On lui envoie, par le biais d'une boîte à message, des paires de chiffres séparées par des virgules. Chaque paire correspond à un degré de la gamme et son adresse associée (adresse - degré). La virgule permet de séparer plusieurs messages à l'intérieur d'une seule boîte à message. Dans l'exemple ci-dessous, chaque message comporte deux entiers, le premier est l'adresse dans le **funbuff**, le second est le degré de la gamme qui lui est associé. L'objet **loadbang**, qui envoie un *bang* à l'ouverture de la patch, permet d'initialiser automatiquement certains objets, dans ce cas-ci, la mémoire du **funbuff**.

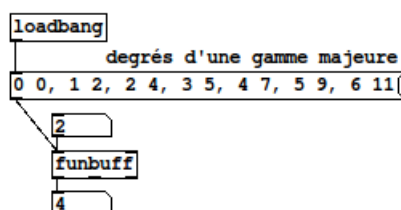


FIGURE 2.2 – Lecture de gamme avec l'objet **funbuff**.

On peut facilement intégrer un mécanisme de variation dynamique du matériau musical en alternant plusieurs listes contenant différentes gammes. Un objet **sel** (pour **select**) envoie un *bang* dans la sortie correspondant à un de ses arguments lorsque cette valeur est reçue en entrée. Dans l'exemple suivant, la valeur 0 active la gamme mineure tandis que la valeur 1 active la gamme majeure.

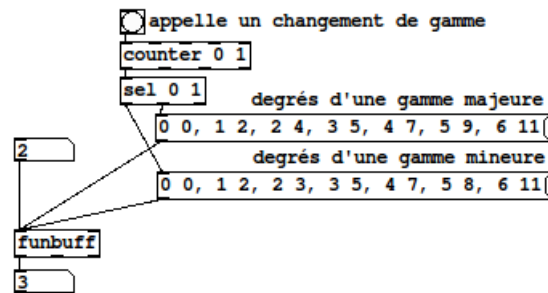
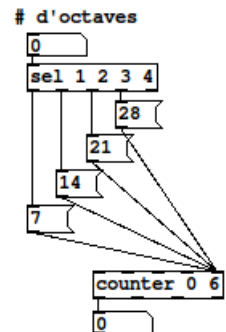


FIGURE 2.3 – Alternance des gammes.

2.1.3 Registre

Il faut maintenant trouver un moyen de contrôler les distances afin d'étendre l'ambitus de notre joueur de gammes.

La dernière entrée de l'objet **counter**, qui permet de fixer le compte maximum, sera utilisée pour déterminer le nombre d'octaves à parcourir. Pour un octave, on compte de 0 à 7, pour deux octaves de 0 à 14, pour trois octaves de 0 à 21, etc. Pour pouvoir utiliser directement cette méthode, il faudrait faire en sorte qu'il y ait toutes les adresses possibles dans la mémoire du **funbuff**, ce qui pourrait éventuellement occasionner des problèmes, comme entrer la mauvaise valeur pour un degré d'un des octaves ou pointer sur une adresse inexistante. Un moyen efficace pour éviter ce type d'erreur consiste à utiliser des opérations mathématiques afin de contraindre les adresses demandées au premier octave, puis d'additionner aux notes un facteur correspondant au registre désiré.



L'opérateur modulo (%) permet d'obtenir le restant d'une division entière de A par B. Si $B = 7$, n'importe quelle valeur en A sera ramenée entre 0 et 6, les degrés de notre gamme. Il ne restera qu'à déterminer à quel octave se trouve la note demandée. Encore là, une simple opération mathématique nous permettra de compter combien d'octaves on doit ajouter à la note de la gamme :

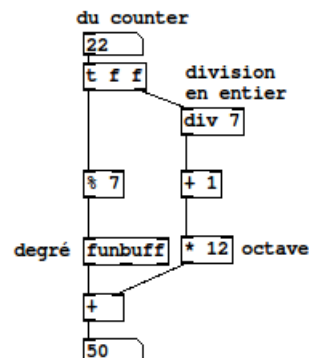
$$\text{nombre d'octaves en multiple de 12} = (\text{int}(A / 7) + 1) * 12$$

Pour une gamme de Do majeur, quelle sera la note jouée si le compteur est rendu à 22 ?

$$22 \% 7 = 22 / 7 = 3 \text{ reste } 1 = 1 \text{ (deuxième degré de la gamme)}$$

$$(\text{int}(22 / 7) + 1) * 12 = 48 \text{ (dans le quatrième octave)}$$

$$2 + 48 = 50 \text{ est la note entendue}$$

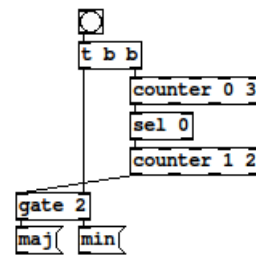


2.2 Décisions

Il est très important, dans un environnement comme Pure Data, de bien comprendre le chemin qu'emprunte le flux de données. Si on ne comprend plus ce qui se passe dans le programme, il est inutile de tenter de complexifier les choses encore plus, car c'est dans ces moments que l'on fait des erreurs de programmation. Avant d'ajouter un nouvel élément à un algorithme, on doit toujours être conscient des tours et détours qu'emprunte le programme.

Pour rendre un algorithme dynamique, il est indispensable d'y insérer des carrefours où les données peuvent changer de direction, afin de renouveler le matériau musical. Voici un petit exemple tout simple :

Lorsque le premier compteur passe par 0, le second compteur incrémente son compte et active la première sortie de la **gate**, forçant le système à utiliser une gamme majeure. Quand le compteur repasse par 0, la deuxième sortie de la **gate** est activée et l'algorithme passe en mineur. Le système alterne ainsi entre le majeur et le mineur tant que le programme est actif.



Un carrefour peut posséder plus de deux portes, selon les besoins en complexité de notre algorithme. La prise de décision pour le changement peut aussi être beaucoup plus élaborée. Voici un exemple de carrefour plus complexe, permettant d'ajuster le registre des valeurs en entrée.

Un premier objet **moses** identifie un seuil sous lequel les valeurs en entrée passent par la sortie de gauche. Les valeurs plus petites que 32 ouvriront donc la porte 1 de l'objet **gate**. Ces notes seront additionnées à 32 afin de les élever dans le registre souhaité, de 32 à 92. Les valeurs plus grandes ou égales à 32 sortiront à droite et seront à nouveau séparées en deux registres avec un nouvel objet **moses**. Cette fois-ci, les valeurs plus petites que 93 (mais plus grandes ou égales à 32) sortiront à gauche et ouvriront la porte 2. Ces notes ne seront pas affectées et seront donc entendues à leur hauteur originale. Finalement, les valeurs plus grandes ou égales à 93 activent la porte 3 de l'objet **gate** et se voient soustraire 35 afin de les ramener dans le registre désiré.

2.3.2 Encapsulation

Un programme peut vite devenir un gros fouillis de boîtes et de fils, communément appelé « spaghetti », où l'on ne distingue plus rien. Il devient donc nécessaire de regrouper les objets participant d'une même tâche et de les encapsuler dans un objet *sous-patcher*. On peut donner au *sous-patcher* autant d'entrées et de sorties que l'on désire à l'aide des objets **inlet** et **outlet**. Lorsque la fenêtre du *sous-patcher* est fermée, ce dernier occupe l'espace d'un seul objet à l'écran. On libère ainsi notre vision et on se donne une meilleure vue d'ensemble des commandes mises en place dans le programme. On crée un *sous-patcher* en inscrivant « pd » dans une boîte « objet » suivi d'un nom (un symbole) qui spécifie sa fonction. Pour visualiser ou modifier le contenu d'un *sous-patcher*, il suffit de cliquer sur l'objet en mode de jeu.

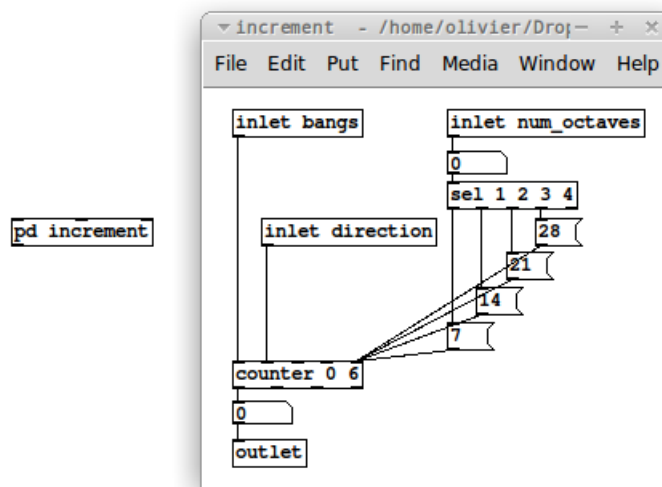


FIGURE 2.6 – Encapsulation d'objets participant d'une même fonctionnalité.

2.4 Objets à étudier

- opérateurs logiques (<, >, <=, >=, ==)
- div (division en entier)
- counter, funbuff
- gate, moses, select (sel)
- loadbang, delay, pipe
- pd, inlet, outlet

2.5 Exercices

2.5.1 Construction d'un métronome maison

Construire un métronome « maison » qui fonctionne comme l'objet **metro** de Puredata, c'est-à-dire qu'il doit répondre aux messages « bang » et « stop » ainsi qu'aux entiers 0 et 1. Encapsuler votre objet en *sous-patcher* (Comme les *sous-patchers* n'acceptent pas les arguments, la vitesse de votre métronome devra être spécifiée en entrée de droite). Vous aurez besoin des objets suivants :

- bang
- toggle
- boîte à message
- boîte à chiffre
- inlet, outlet
- delay
- sel (pour la gestion des contrôles numériques 0 et 1)
- route (pour la gestion des symboles « bang » et « stop »)

2.5.2 Lecture de mélodies en boucle

Étape 1

Construire quatre messages contenant chacun une mélodie de huit notes MIDI, dans un format prêt à être enregistré en mémoire dans un **funbuff**. La première mélodie doit être chargée à l'ouverture de la patch.

Étape 2

Créer un mécanisme qui lit, en boucle, le contenu du **funbuff**.

Étape 3

Assigner la mélodie générée à la sortie MIDI à l'aide des objets **makenote** et **noteout**.

Étape 4

Les quatre mélodies doivent alterner et chacune doit être entendue deux fois avant de passer à la suivante.

Étape 5

Marquer les temps forts en assignant une vitesse élevée à la première et la quatrième note de la boucle et une vitesse plus faible pour les autres notes.

2.5.3 Lectures

Parcourir les pages de la section **Prise en main rapide** du [manuel FLOSS](#) de Pure Data.

Chapitre 3

Introduction au traitement de signal audio

3.1 Lecture et écriture de fichiers sons sur le disque dur

Il existe deux méthodes pour accéder, en lecture ou en écriture, à un fichier son sur le disque dur. La première, plus simple mais aussi plus limitée, consiste à lire les échantillons directement sur le disque pendant le déroulement de la performance. Avec la seconde méthode, les échantillons seront tous lus une seule fois et enregistrés dans la mémoire vive de l'ordinateur, offrant ainsi un accès plus rapide au contenu du fichier son.

3.1.1 Lecture sur le disque dur

L'objet **readsf~** lit un son sur le disque dur et achemine le signal vers une ou plusieurs sorties de l'objet. Un argument permet de spécifier le nombre de canaux que l'on désire lire dans le fichier. L'objet présente toujours une sortie de plus que le nombre de canaux spécifiés. Cette dernière, complètement à droite, envoie un *bang* indiquant la fin de la lecture du son.

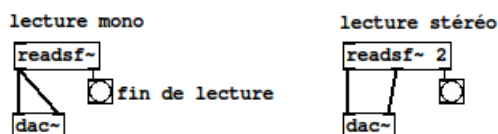


FIGURE 3.1 – Un argument à **readsf~** spécifie le nombre de canaux.

Pure Data accepte les fichiers *AIFF* et *WAVE* en format 16-bit ou 24-bit (entier) ainsi que 32-bit (virgule flottante, *float*). Un message « open », suivi du nom du fichier à lire servira à indiquer où l'objet doit chercher le son sur le disque. Un argument supplémentaire spécifiera un point de départ pour la lecture, en échantillons, autre que le début du fichier. Le chemin où lire le fichier son peut prendre deux formes : La première consiste à donner le chemin complet du fichier, depuis la racine de l'ordinateur, tandis que la seconde, si le chemin ne commence pas par la racine, cherche à partir du dossier courant de la patch.

```

Chemin complet à partir de la racine de l'ordinateur
open /home/olivier/Dropbox/private/snds/flute.aif{

Chemin relatif à partir l'emplacement de la patch
open ../snds/flute.aif{

Temps de départ dans le fichier son, en échantillons
open ../snds/flute.aif 44100{

```

FIGURE 3.2 – Exemples de messages d’ouverture d’un fichier.

La valeur 1 en entrée de l’objet démarre la lecture du son, qui arrête automatiquement lorsqu’il n’y a plus d’échantillons à lire. La valeur 0 force l’arrêt de la lecture. L’exemple suivant lit un son stéréo, applique un contrôle d’amplitude et envoie le signal aux haut-parleurs.

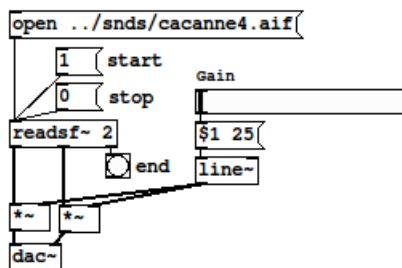


FIGURE 3.3 – Lecture d’un fichier son stéréo.

3.1.2 Écriture sur le disque dur

Pour enregistrer un signal audio dans un fichier sur le disque, on aura recours à l’objet **writesf~**. L’argument donné à l’objet spécifie le nombre de canaux audio dans le fichier et détermine le nombre d’entrées audio de l’objet.

```

fichier son 16-bit
open ../snds/test.wav{

fichier son 24-bit
open -bytes 3 ../snds/test.wav{

fichier son 32-bit float
open -bytes 4 ../snds/test.wav{

```

FIGURE 3.4 – Exemples de message d’ouverture d’un fichier pour l’écriture.

Tout comme pour l’objet **readsf~**, un message « open » servira à indiquer où enregistrer le fichier sur le disque, soit en donnant un chemin complet depuis la racine, soit en donnant un chemin relatif depuis le dossier où réside la patch. Certaines options peuvent être spécifiées au message d’ouverture, notamment l’option *-bytes* permettant de choisir un des trois formats sup-

portés par Pure Data, soit 16-bit (défaut), 24-bit ou 32-bit (*float*). Pour connaître les différentes options possibles, voir la patch d'aide de l'objet **writesf~**.

On démarre l'enregistrement du fichier ouvert avec un message « start ». Un enregistrement en cours continue tant que le message « stop » n'a pas été envoyé. L'objet **writesf~** est particulièrement approprié pour les enregistrements où la durée n'est pas connue à l'avance ou pour les enregistrements de longue durée, qui ne peuvent être effectués en RAM puisqu'ils demanderaient trop de mémoire vive.

Dans l'exemple suivant, lorsque le *bang* est activé, un message spécifiant où enregistrer le fichier « test.wav » est d'abord envoyé à l'objet **writesf~**, puis l'enregistrement de l'entrée microphone est lancé. Une minute plus tard (60000 millisecondes), le message « stop » est donné pour arrêter l'enregistrement.

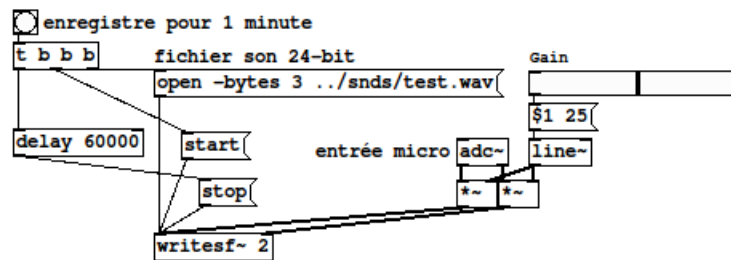


FIGURE 3.5 – Écriture d'un fichier stéréo sur le disque.

3.1.3 Construction dynamique des noms de fichier

Une lacune manifeste de l'exemple précédent vient du fait que chaque fois qu'un fichier « test.wav » est enregistré, il vient écraser l'enregistrement précédent, et ce, sans avertissement. Pour éviter de perdre le travail déjà accompli, il peut être parfois intéressant de mettre en place un système d'incrémentation du nom des fichiers enregistrés. L'objet **makefilename** permet de donner un nom de fichier générique dans lequel une variable attend une donnée pour construire le nom final envoyé à l'objet **writesf~**. La syntaxe utilisée est la même que dans la plupart des langages de programmation standards, c'est-à-dire que la variable ouverte sera déclarée à l'aide de l'opérateur %, suivi du type de donné désiré. Les deux types permis sont :

- **d** : permet d'insérer un entier.
- **s** : permet d'insérer un symbole.

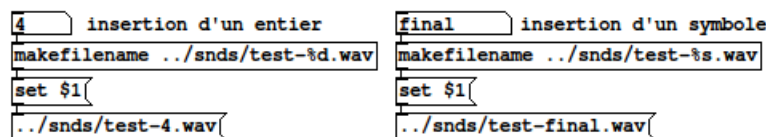


FIGURE 3.6 – Construction dynamique à l'aide d'un entier ou d'un symbole.

Pour assurer un ordre croissant sur plusieurs dizaines de fichiers, il est possible de spécifier le nombre de caractères utilisés pour la construction du message en glissant un 0 et le nombre de caractères désirés entre l'opérateur % et la lettre **d**. Ainsi, la valeur 5, donnée à une variable de syntaxe `%03d`, s'écrira 005.

```
5
makefilename ../snds/test-%03d.wav
set $1(
../snds/test-005.wav
```

FIGURE 3.7 – Gestion du nombre de caractères lors de la construction d'un entier.

La patch suivante permet d'enregistrer une centaine de fichiers d'une durée de 1 seconde chacun, simplement en appuyant sur un *bang* à chaque nouvel enregistrement.

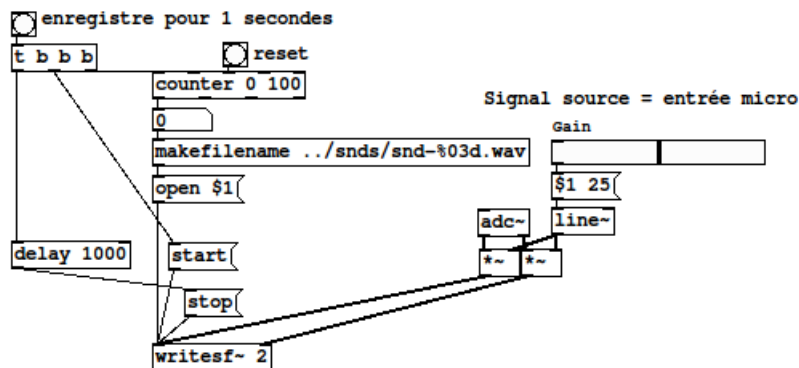


FIGURE 3.8 – Enregistrement ordonné de fichiers sonores.

Deux opérateurs % consécutifs éliminent l'insertion d'une variable et permettent de construire un message contenant encore une variable attendant une donnée. Ceci rend possible l'enchaînement d'objets **makefilename** et des constructions encore plus flexibles de noms de fichier.

```
crunchy nom générique du fichier
makefilename %s-%03d.wav
set $1( assigne crunchy-%03d.wav au second makefilename
5 incrémente le compte...
makefilename temp-symbol
set $1(
crunchy-005.wav
```

FIGURE 3.9 – Enchaînement d'objets **makefilename** pour une construction plus flexible.

Dans l'exemple précédent, le nom donné à l'objet **symbol** remplacera la variable `%s` dans le premier objet tandis que pour la variable « entier », le premier `%` indique que le caractère suivant doit être considéré comme un caractère standard, sans rôle particulier à jouer. Ce qui laisse donc `%03d` tel quel dans le symbole généré. Ce symbole est donné, à l'aide du message « set » au deuxième objet **makefilename** qui aura donc un nom générique choisi par l'utilisateur et une variable attendant un entier pour incrémenter le nom des fichiers enregistrés.

3.2 Lecture et écriture de fichiers sons en RAM

La seconde méthode pour charger des fichiers sons dans l'environnement Pure Data offre beaucoup plus de flexibilité. Il s'agit de charger le son dans un objet **array**, qui consiste en un espace alloué à même la mémoire vive de l'ordinateur. Les échantillons enregistrés dans un **array** sont accessibles beaucoup plus rapidement que ceux sur le disque dur. La manipulation d'un son à partir d'une mémoire permet plusieurs effets tel le bouclage, la lecture à vitesse variable, la granulation et bien d'autres...

3.2.1 Lecture en RAM

Pour déclarer un espace mémoire en RAM, il suffit de sélectionner un objet **array**, disponible via le menu *Put → array*. Une fenêtre de propriétés apparaît alors, permettant de donner un nom et une taille (en échantillons) à l'espace mémoire ainsi créé.

Après avoir ajustés les attributs de l'**array**, un nouvel élément graphique apparaît dans la patch. Le graphique, initialisé à zéro, peut servir à la fois d'élément visuel et d'objet de contrôle. Il est donc possible d'interagir avec le contenu de l'**array** à l'aide de la souris, sujet sur lequel nous reviendrons dans les chapitres suivants.

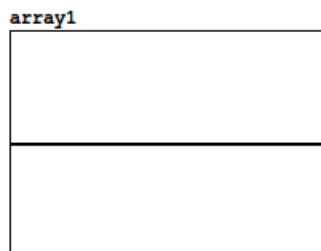
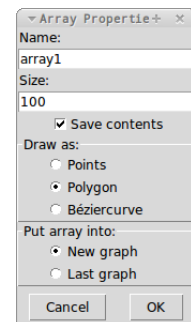


FIGURE 3.10 – Un **array** nouvellement créé.

Une fois l'espace mémoire créé, il est maintenant possible de le remplir avec les échantillons d'un fichier son. La communication avec la mémoire, en lecture comme en écriture, s'effectue à l'aide d'un objet **soundfiler**. Pour lire le contenu d'un fichier, il suffit d'envoyer un message « read » suivi du chemin du fichier à lire et d'un ou plusieurs noms d'**arrays** où charger les

échantillons. L'exemple suivant charge le contenu du fichier « flute.aif » dans une mémoire de une seconde nommée « array1 ».

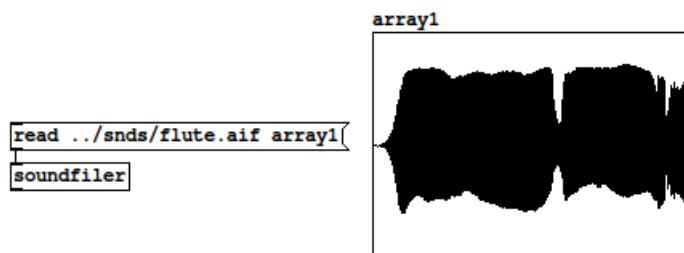


FIGURE 3.11 – Lire un son dans un **array**.

Comme on peut le constater à l'image du graphique, le fichier son a été tronqué, seule la première seconde de son a été mise en mémoire. On peut donner certaines options au message « read » afin de contrôler la méthode de chargement des échantillons. L'option qui nous intéresse ici est *-resize*, qui indiquera à Pure Data que la taille de la mémoire doit être réajustée en fonction de la longueur du fichier son choisi. Voici comment devrait être déclarée la lecture du son « flute.aif ». Notez la valeur de retour de l'objet **soundfiler**, qui indique le nombre d'échantillons chargés en mémoire. Cette valeur sera fort utile quand viendra le temps de déterminer à quelle vitesse la mémoire doit être lue pour entendre le son à la hauteur désirée.

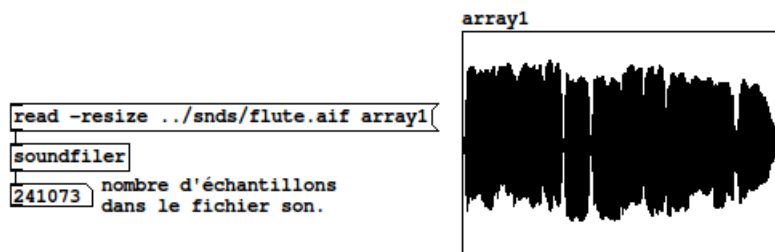


FIGURE 3.12 – Lire un son dans un **array** avec ajustement de la taille.

L'objet **openpanel** ouvre, sur réception d'un *bang*, un dialogue standard permettant à l'utilisateur de naviguer sur le disque dur afin de choisir un fichier à charger. L'objet retourne, sous la forme d'un symbole, le chemin complet du fichier choisi, en respectant les espaces et les caractères accentués. Ce symbole peut être utilisé pour construire un message « read » assigné à un objet **soundfiler**. Le programme suivant permet de charger un son stéréo dans deux **arrays** et utilise le nombre d'échantillons en sortie du **soundfiler** pour contrôler deux algorithmes de lecture différents. Ces algorithmes seront détaillés dans la prochaine section.

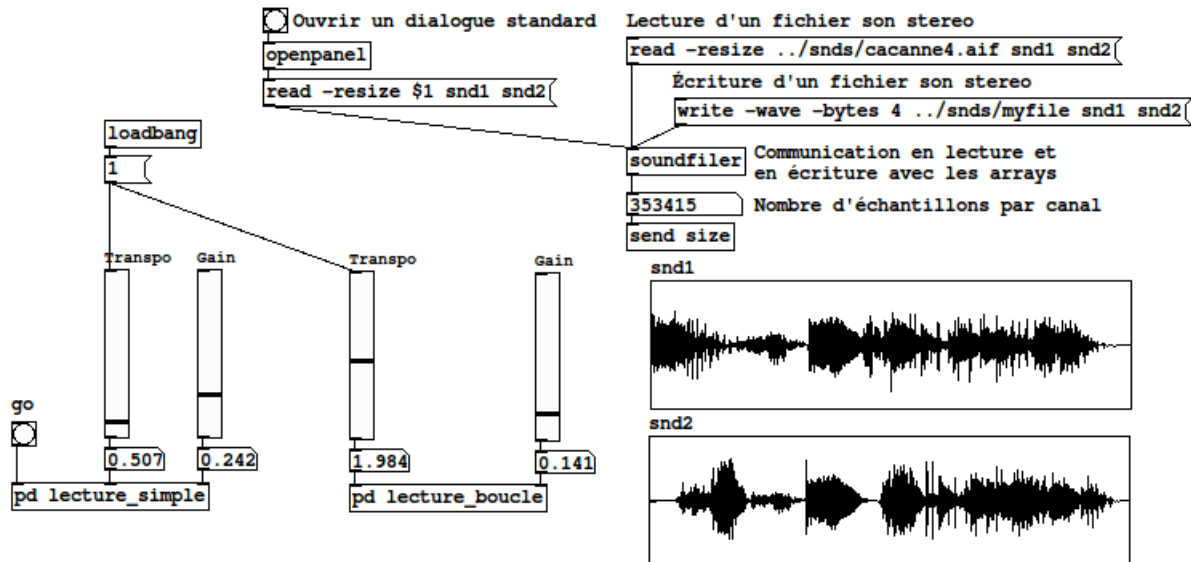


FIGURE 3.13 – Deux algorithmes de lecture d'un fichier son stéréo chargé en RAM.

3.2.2 Algorithmes de lecture en RAM

Pour lire le contenu audio d'un **array**, nous utiliserons l'objet **tabread4~**, qui offre une lecture avec interpolation cubique de bonne qualité. L'interpolation est nécessaire pour toute lecture où la vitesse ne correspond pas à la fréquence d'échantillonnage. Comme les échantillons ne sont pas lus un à la suite de l'autre, l'interpolation permettra d'extrapoler la valeur d'amplitude réelle du signal en pondérant les quatres échantillons autour du pointeur de lecture. Un argument à l'objet **tabread4~** indique de quel **array** il doit lire le contenu et on contrôle la position de lecture, en échantillons, à l'aide d'un signal audio dans sa première entrée.

Lecture simple

Le sous-patch « lecture_simple » met en place une lecture unique du son, à une hauteur modifiable, déclenchée à l'aide d'un *bang*. La position de lecture sera donnée par un objet **line~**, qui génère une rampe audio dont les balises et la durée sont contrôlées à l'aide d'une boîte à message. Une valeur unique, donnée à un **line~**, provoque un saut direct à la valeur tandis qu'une liste de deux valeurs correspond à une destination et une durée. Le message « 0, \$1 \$2 » indique donc de se rendre immédiatement à 0 puis d'effectuer une rampe vers la valeur fournie en remplacement de \$1 sur une durée fournie en remplacement de \$2. La destination et la durée seront calculées en fonction du nombre d'échantillons contenus dans la mémoire. Comme la durée d'un objet **line~** est donnée en millisecondes, il faut tout d'abord convertir la durée en échantillons en une durée en milliseconde. L'opération suivante effectue le travail :

$$ms = samples / sr * 1000$$

Cette valeur est ensuite divisée par le facteur de transposition, puis donnée en deuxième

entrée d'un objet **pack**, servant à construire une liste. La deuxième valeur de la liste est donc la durée de lecture de la mémoire. Ensuite, en première valeur de la liste est donnée la destination de la ligne de lecture, c'est-à-dire la durée du son en échantillons.

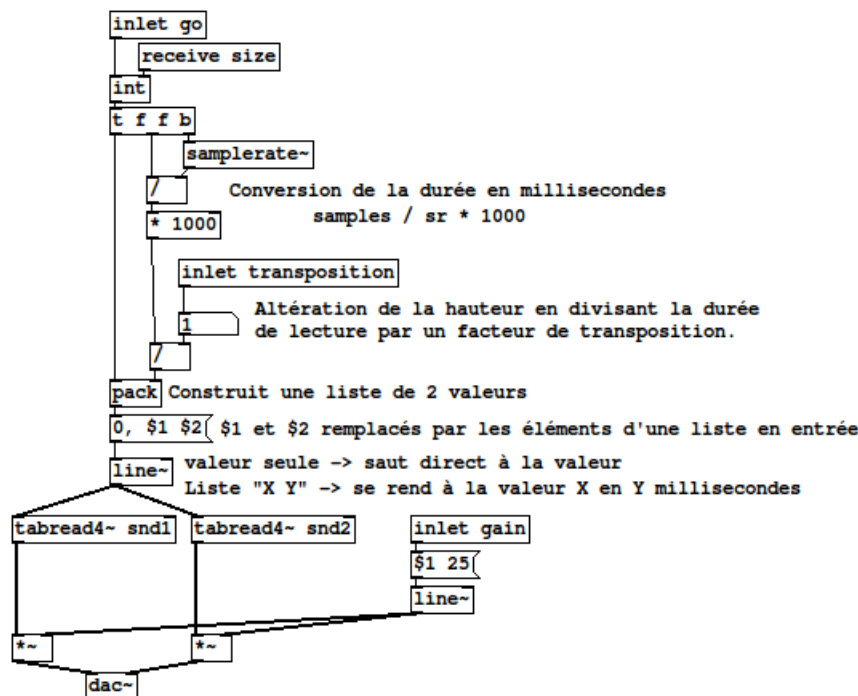


FIGURE 3.14 – Contenu du sous-patch « lecture_simple ».

Lecture bouclée

Le deuxième algorithme, que l'on retrouve dans le sous-patch « lecture_boucle », met en place une lecture bouclée du fichier son placé en mémoire. Nous aurons recours cette fois-ci à un objet **phasor~** dont le rôle est de générer une rampe de 0 à 1 à une fréquence spécifiée en Hertz. La première étape consiste donc à convertir la durée de la mémoire, donnée en échantillons, en une fréquence en Hertz. La formule suivante donne la fréquence à laquelle le son sera lu à sa hauteur originale :

$$hz = sr / samples$$

Cette valeur est ensuite multipliée par le facteur de transposition pour donner la fréquence réelle du **phasor~**. La rampe (de 0 à 1) sera ensuite multipliée par la longueur de la table, en échantillons, afin de respecter les balises attendues par l'objet **tabread4~**. Notez ici que pour profiter pleinement de l'interpolation cubique, il doit toujours y avoir un échantillon disponible au devant de la position de lecture courante ainsi que deux échantillons après. Cela explique le fait que la rampe est en réalité multipliée par la longueur de la mémoire - 3 et additionnée à 1.

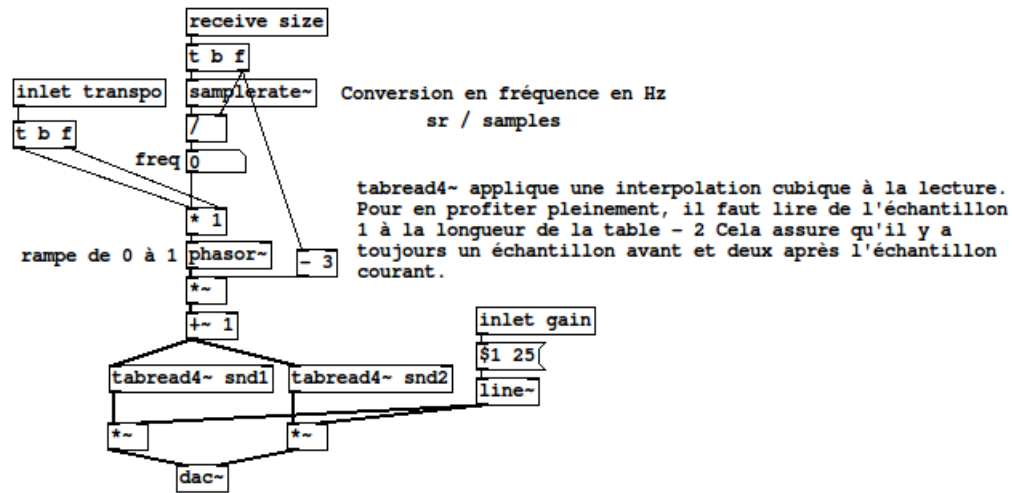
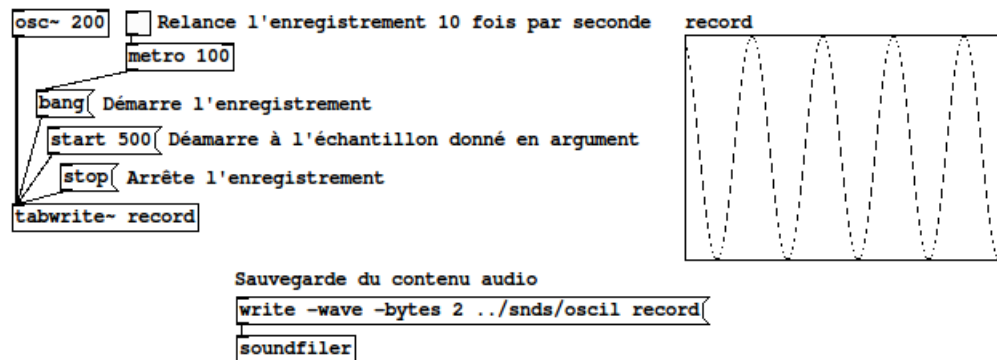


FIGURE 3.15 – Contenu du sous-patch « lecture_boucle ».

3.2.3 Écriture en RAM

L'objet **tabwrite~** permet d'enregistrer le signal en entrée dans un **array** dont le nom est donné en argument. Un *bang* ou un message « start » démarre l'enregistrement et un message « stop » permet de l'arrêter avant d'avoir atteint la fin de la mémoire.

FIGURE 3.16 – Enregistrement d'un signal audio dans un **array**.

Il est possible d'enregistrer le contenu d'un ou plusieurs **arrays** dans un fichier sur le disque dur en envoyant un message « write », suivi d'un nom de fichier puis du ou des **arrays** à sauvegarder, à un objet **soundfiler**. Les options « -wave » ou « -aiff » serviront à indiquer le type de fichier à enregistrer et l'option « -bytes » permet de spécifier le format désiré.

3.3 Objets à étudier

- pack
- dac~, adc~
- readsf~, writesf~
- makefilename, openpanel
- table (*array*), soundfiler
- tabwrite~, tabread4~
- samplerate~, phasor~, line~

3.4 Exercices

3.4.1 Lecture d'un fichier son par segmentation aléatoire

Étape 1

Charger un son (d'une durée de une seconde ou moins) dans un **array**. Le son doit être chargé automatiquement à l'ouverture de la patch. Récupérer la longueur du son en échantillons.

Étape 2

Créer un mécanisme de génération de valeurs aléatoires qui fournit une valeur, entre 0 et la longueur du son, à toutes les 250 millisecondes.

Étape 3

Utiliser la valeur aléatoire pour construire un message « X Y », destiné à un objet **line~**, où la valeur représente la destination et la durée est fixée à 250 ms.

Étape 4

Utiliser la ligne générée à l'étape précédente pour lire le contenu de la mémoire et acheminer le signal aux haut-parleurs.

Étape 5

Ajouter un contrôle de volume avant l'envoi aux haut-parleurs.

3.4.2 Lectures

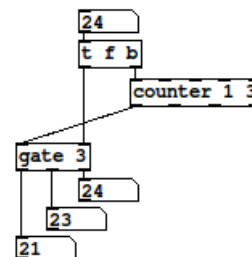
Parcourir les pages « Les bases de l'audio numérique », « L'audio dans Pd » et « L'échantillonnage » de la section **Audio** du [manuel FLOSS](#) de Pure Data.

Chapitre 4

Résolution de problèmes et aiguillage des données

4.1 Aiguillage des données

Un des principaux défis de la musique algorithmique consiste à acheminer l'information au bon endroit et au bon moment. Plusieurs stratégies peuvent être envisagées pour arriver au résultat souhaité. Un aiguillage très simple consiste à utiliser l'objet **gate** et à faire varier les ouvertures avec, par exemple, un compteur. Chacune des données fait d'abord avancer le compteur et passe ensuite dans la porte qui vient tout juste d'être ouverte.



Un deuxième exemple d'aiguillage simple est illustré avec l'objet **moses**. Si la valeur d'entrée est plus petite que l'argument donné à l'objet, elle sort à gauche, sinon, elle sort à droite. Dans l'exemple ci-dessous, l'ambitus de la vélocité, qui est de 0 à 127, est divisé en quatre groupes, du plus faible au plus fort. Les valeurs associées à chacun des groupes (1, 2, 3 et 4) pourraient être utilisées pour changer la tonalité, l'instrumentation, le registre ou un quelconque autre des paramètres du processus en cours.

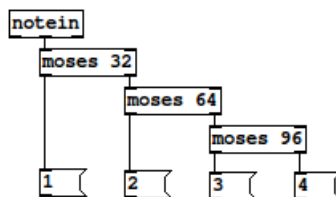


FIGURE 4.1 – Utilisation de la vélocité de la note à des fins extra musicales.

4.2 Aiguillage et contrôle

Pour créer un programme musical interactif, il est nécessaire de mettre en place des techniques de contrôle permettant de générer des événements ou de modifier les paramètres du jeu. Les interfaces MIDI représentent un moyen simple, accessible et efficace pour le contrôle interactif de processus sonores. Nous verrons plus tard que les caractéristiques d'un signal audio peuvent aussi servir de signaux de contrôle. Ici, nous agissons sur la musique avec les touches et les potentiomètres d'un clavier MIDI. Il faut garder en tête que nous ne sommes pas obligés d'utiliser les contrôleurs de façon conventionnelle, c'est-à-dire qu'une touche du clavier peut très bien déclencher autre chose qu'une note MIDI.

Observons un exemple dans lequel on fait varier la séquence rythmique d'un répéteur en fonction de la vélocité de la note. Nous avons un compteur activé par un métronome dont la vitesse reste constante. À chaque fois que le compteur donne un 1, la note est jouée. Avec l'aide de l'objet **moses**, la valeur maximale du compteur devient fonction de la vélocité de la note. Plus la note est forte, plus la valeur maximale est petite, le 1 revient donc plus souvent et le rythme est plus rapide. La hauteur est enregistrée dans une mémoire simple (un objet **int**) et y reste jusqu'à ce qu'une autre note vienne la remplacer.

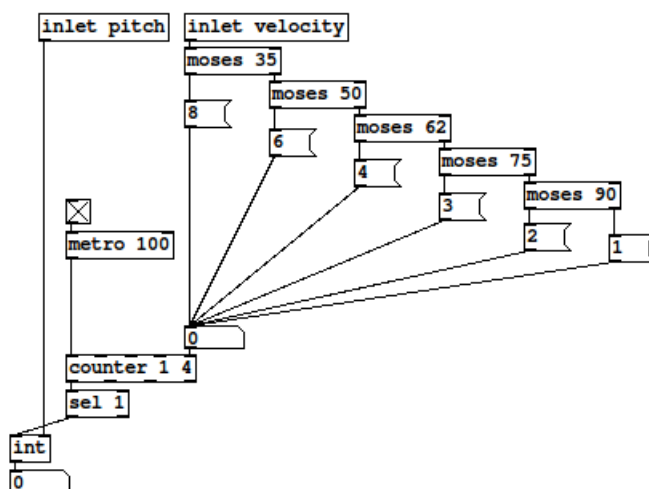


FIGURE 4.2 – La vélocité de la note contrôle la fréquence de répétition.

4.3 Gestion de la polyphonie

Lorsqu'un clavier MIDI est utilisé pour contrôler un programme de génération de notes, il est important de comprendre l'ordre et la nature des messages envoyés pour construire un module capable de gérer adéquatement la polyphonie jouée au clavier. Un message *note on* peut être capté via l'objet **notein**, qui enverra trois informations : le canal MIDI sur lequel la note est jouée, sa vélocité (positive) et sa hauteur, en valeur numérique (de 1 à 127). L'objet **notein** donnera les mêmes informations pour un message *note off*, toutefois, la vélocité sera

de 0. L'information qui importe pour mettre en place une répétition à vitesse variable est la hauteur de la note et sa vélocité, qui détermine si la note est enfoncée ou relâchée.

L'objet **poly** est conçu expressément pour gérer la polyphonie MIDI. Un premier argument indique le nombre de voix de polyphonie désirées. Si on fixe le nombre de voix à 5, par exemple, les cinq premières notes enfoncées se verront chacune assigner une voix, représentée par un entier dans la sortie de gauche. Si d'autres notes sont jouées sans relâcher les cinq premières, elles seront soit ignorées (comportement par défaut) ou bien elles viendront "voler" la voix de la plus ancienne note si un second argument, de valeur 1, est donné à l'objet. Un objet **poly** possède deux entrées, une pour la hauteur et une autre pour la vélocité d'une note MIDI. Pour savoir combien de notes sont enfoncées et quelle voix assigner à la prochaine note, l'objet garde en mémoire tous les *notes on* avec leur numéro de voix associé, et utilise la première voix disponible. Lorsqu'un *note off* arrive, il achemine l'information sur la voix qui a reçu un *note on* à cette hauteur. Voici l'ordre des sorties, de gauche à droite, de l'objet **poly** :

- Numéro de voix
- Hauteur MIDI
- Vélocité

On utilisera un objet **route** pour acheminer les listes de valeurs (hauteur, vélocité) aux différentes copies de notre module de répétition. Après avoir construit une liste avec les trois valeurs en sortie de l'objet **poly**, l'objet **route** dirigera le restant d'une liste vers la sortie correspondant à l'argument qui coïncide avec la première valeur de la dite liste. Le numéro de voix sera donc retiré de la liste et ne restera que la hauteur et la vélocité qui seront envoyées à la voix pour générer le processus.

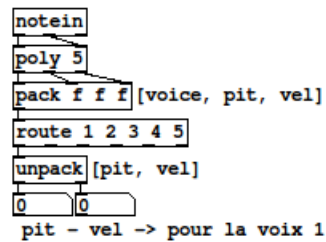


FIGURE 4.3 – Gestion de la polyphonie MIDI.

La combinaison du clavier MIDI (**notein**) et d'un système de gestion de la polyphonie (**poly**) nous permettra de mettre en place plusieurs répétitions de notes en parallèle, avec chacune leur hauteur et leur vélocité propre. En détournant la vélocité de son rôle habituel et en lui assignant le contrôle de la fréquence de répétition des notes, un contrepoint rythmique sera généré en variant la force avec laquelle chacune des notes est enfoncée au clavier.

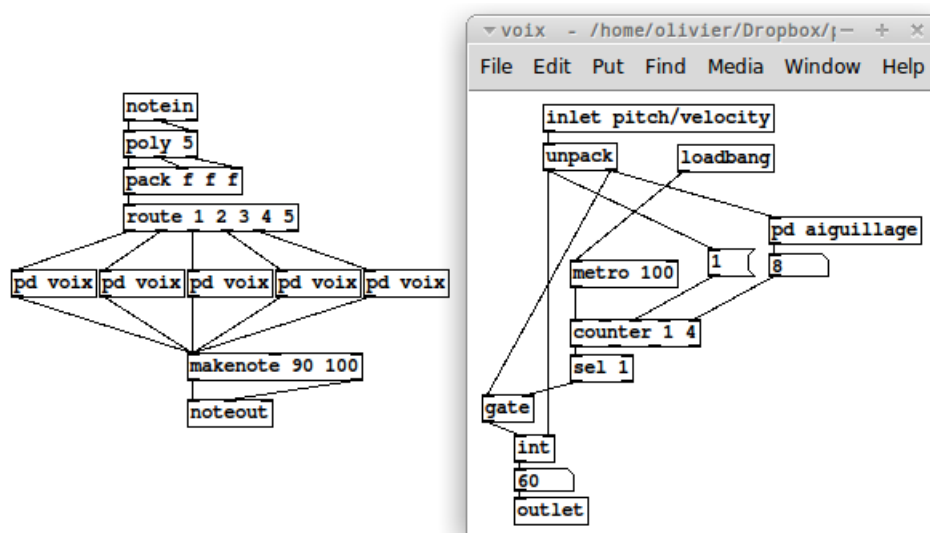


FIGURE 4.4 – Cinq étages de répétitions de notes en parallèle.

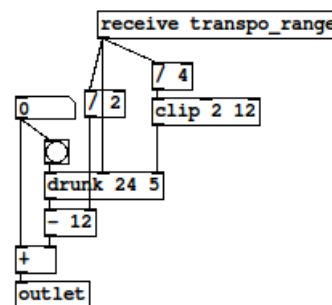
4.4 Éléments de hasard et contrôleurs MIDI

Afin de transformer notre répétiteur de notes en un "pianiste fou", nous allons maintenant ajouter quelques éléments de hasard dont les caractéristiques seront contrôlées à l'aide de potentiomètres MIDI.

4.4.1 Variations mélodiques

Le premier élément de hasard consiste à faire varier la note répétée en lui additionnant une valeur pigée au hasard à l'intérieur d'un ambitus contrôlé par un potentiomètre.

L'objet **drunk** constitue une alternative intéressante à l'objet **random**. Il génère, sur réception d'un bang, une valeur entre zéro et la valeur maximale spécifiée en premier argument, tout en respectant un écart maximum donné en deuxième argument. Chaque valeur générée reste ainsi dans un ambitus raisonnable par rapport à la valeur précédente, on appelle ce processus une "marche aléatoire". La valeur maximale est ici contrôlée par le potentiomètre tandis que l'écart maximum sera systématiquement le quart de cette dernière. En soustrayant la moitié de la valeur maximale à la sortie de l'objet, on obtient des valeurs positives et négatives donc, une mélodie qui tourne autour de la note jouée au clavier.



4.4.2 Variations rythmiques

Le second élément de hasard permettra de briser le caractère métronomique des générateurs en éliminant certaines des notes générées. Un potentiomètre contrôlera le pourcentage de notes entendues.

Pour mettre en place un tel système, il suffit de piger une valeur au hasard entre 0 et 100, puis de comparer cette valeur avec un seuil, donné ici par le potentiomètre MIDI. Si la valeur est plus petite que le seuil, la porte est ouverte et laisse passer le bang, sinon, la porte est fermée et on saute une note. Plus le seuil est élevé, plus il y a de chance que la valeur pigée ouvre la porte.

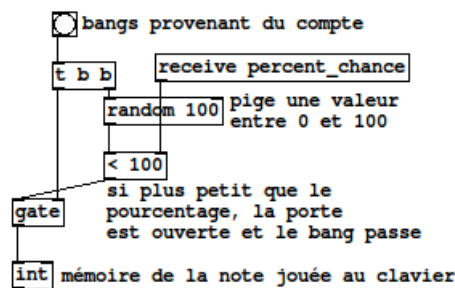


FIGURE 4.5 – Une porte laisse passer seulement un certain pourcentage des notes.

4.4.3 Contrôle des variations

Chacune des voix recevra les mêmes valeurs de contrôle qui seront générées au niveau principal du programme. L'objet **ctlin** permet de récupérer les valeurs d'un contrôleur MIDI en désignant le numéro du contrôleur en argument. Pour connaître le numéro du contrôleur désiré, il suffit de placer un objet **ctlin** sans argument dans la patch et de regarder la valeur en deuxième sortie de l'objet lorsque le contrôleur est manipulé. De simples opérations mathématiques permettent ensuite de restreindre les valeurs MIDI dans un ambitus correspondant au processus mis en place.

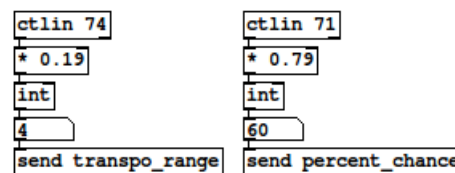


FIGURE 4.6 – Réception des contrôleurs MIDI.

Le programme suivant illustre l'ajout des variations aléatoires pour créer le pianiste fou, avec en prime la possibilité de varier le volume des notes via le contrôleur 7 et une gestion de la durée des notes via le contrôleur 1 (roue de modulation).

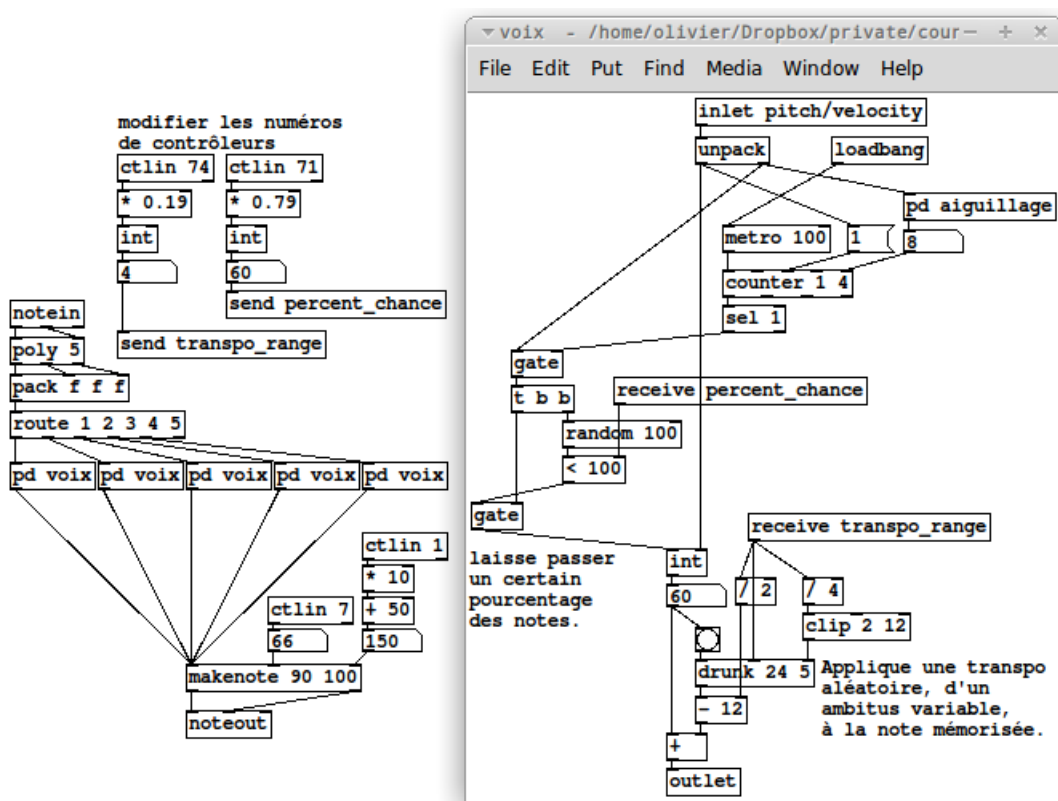


FIGURE 4.7 – Cinq étages mélodiques en parallèle.

4.5 Objets à étudier

- pack, unpack
- notein, ctlin
- poly, route
- drunk, urn
- send, receive

4.6 Exercices

4.6.1 Exercices de révision

Exercice n° 1

Écrire un programme qui joue dix notes, à intervalle de 200 ms, et qui arrête de façon automatique. Les notes seront choisies de façon aléatoire et devront être comprises dans l'intervalle allant du do2 (note MIDI) au do5.

Exercice n° 2

Écrire un programme qui compte à pas de 2 ou de 3 entre 0 et 12, inclusivement. Lorsque le programme est mis en marche, il doit toujours partir à 0. Le programme doit alterner automatiquement entre l'intervalle de 2 et celui de 3 à chaque nouveau cycle.

Exercice n° 3

Écrire un programme qui joue une gamme majeur sur un octave à intervalle de 200 ms. 100 ms après chaque note, on veut entendre cette même note transposée un octave plus haut.

Exercice n° 4

Écrire un programme qui joue des notes aléatoires sur une gamme précise, mémorisée dans un objet funbuff. On doit pouvoir changer la gamme d'un simple clic.

4.6.2 Lectures

Lire la page "Le MIDI" dans la section **Communication** du [manuel FLOSS](#) de Pure Data.

Chapitre 5

La synthèse soustractive

5.1 Les types de sources

Lorsque l'on travaille avec la synthèse soustractive, le choix de la source sonore à filtrer est aussi important que l'algorithme de filtrage lui-même. Afin d'obtenir un effet de filtrage convaincant, il est nécessaire que la source possède de l'énergie dans la région du spectre influencée par le filtre. Les sons environnementaux ont généralement un potentiel d'utilisation très riche avec la synthèse soustractive. Mais pour bien comprendre les manipulations effectuées avec les objets de filtrage, il est intéressant de commencer uniquement avec des sources provenant de synthèse. Et l'on verra qu'il est possible de donner vie, à l'aide de la synthèse soustractive, à un son qui, à la base, est plutôt statique.

On distingue principalement deux types de sources dont le spectre permet une synthèse soustractive efficace. Il s'agit des signaux bruités et des signaux harmoniques à large spectre.

5.1.1 Les sources bruitées

Les signaux bruités comprennent tous les sons apériodiques dont les plus populaires sont le "bruit blanc" et le "bruit rose". Ces signaux se distinguent par une répartition plus ou moins uniforme des composantes sur toute l'étendue du spectre sonore.

- Bruit blanc (**noise**~) : Énergie constante sur toute l'étendue du spectre.
- Bruit rose (**pink**~) : Énergie constante par bande d'octave (-3 dB par octave).

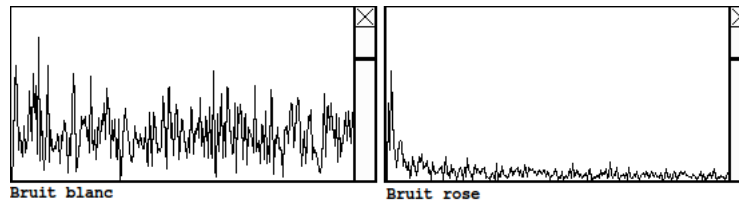


FIGURE 5.1 – Signaux de synthèse bruités.

Pour un aperçu des différentes couleurs de bruit, consulter la page wikipedia [les bruits colorés](#).

5.1.2 Les sources harmoniques

Les signaux harmoniques sont composés d'une grande quantité de composantes discrètes (appelées *harmoniques* ou *partiels* dans le cas de composantes inharmoniques). Ces sons présentent généralement une fréquence fondamentale reconnaissable, accompagnée d'une série de composantes harmoniques. L'image ci-dessous illustre le spectre d'une onde en dent de scie et d'un train d'impulsions.

- Dent de scie : Une onde en dent de scie est composée d'harmoniques dont l'amplitude est l'inverse de l'ordre, $A(n) = 1/n$. Un objet **phasor~** génère une dent de scie contenant un nombre virtuellement infini d'harmoniques.
- Train d'impulsions : Un train d'impulsions est un signal, souvent appelé "buzz", contenant un grand nombre d'harmoniques. Dans le cas d'un objet simple, les harmoniques auront tous la même amplitude. Un algorithme plus évolué permettra de définir une pente, c'est-à-dire un filtrage des hautes fréquences, comme c'est le cas pour l'objet **train~** de Pure Data.

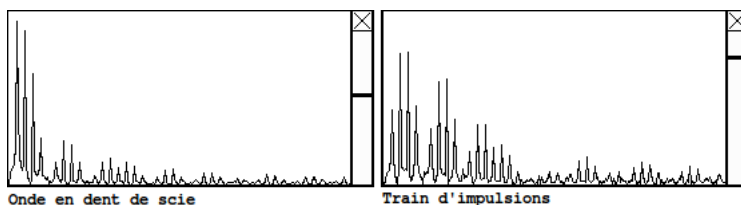


FIGURE 5.2 – Signaux de synthèse harmoniques à large spectre.

Les ondes carrées et triangulaire, que l'on peut construire à l'aide d'une somme de composantes sinusoïdales et/ou cosinusoïdales, sont souvent utilisées comme source d'une synthèse soustractive. Nous explorerons éventuellement la construction des différentes formes d'onde.

5.2 L'objet **delay~** et les filtres FIR simples

Le filtrage numérique de base s'effectue dans le domaine temporel. C'est par la pondération entre les échantillons successifs que sont mises en place les amplifications ou les annulations de fréquence dans le signal filtré. La notion de délai prend ici une importance capitale pour la compréhension de la nature d'un filtre. Plus on utilise d'échantillons passés, donc de délais, pour construire un filtre, plus celui-ci peut être complexe. On définit l'**ordre d'un filtre** par le plus grand temps de délai qu'il utilise.

Les filtres à réponse impulsionnelle finie (FIR) n'utilisent que des échantillons d'entrées, c'est-à-dire, les derniers échantillons qui sont précédemment entrés dans le système, pour effectuer son calcul. À l'inverse, les filtres à réponse impulsionnelle infinie (IIR) utilisent aussi des échantillons de sortie, c'est-à-dire des échantillons récupérés à même la sortie du filtre (*feedback*). Cela permet de programmer des filtres beaucoup plus complexes de façon économique.

Voici un exemple d'un filtre passe-bas FIR simple. En effectuant la moyenne entre un échantillon et celui qui le précède, on "lisse" le signal, donc on élimine les variations rapides, c'est-à-dire qu'on atténue les composantes aiguës du son.

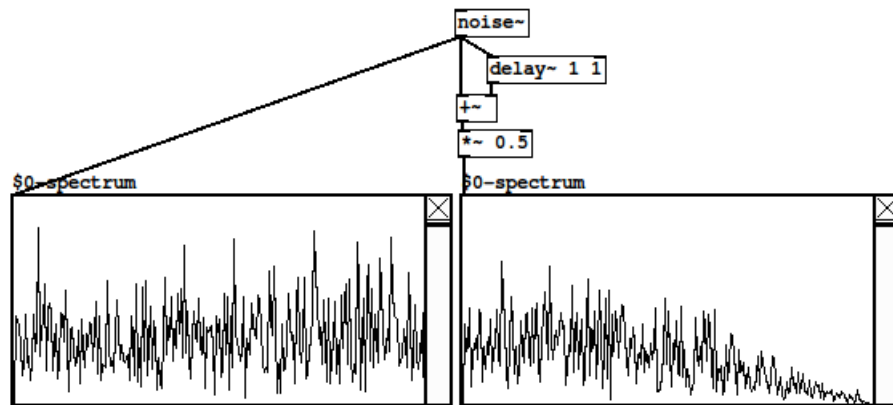


FIGURE 5.3 – Filtre passe-bas FIR de premier ordre.

Pour construire un filtre passe-haut, on voudra cette fois amplifier les variations rapides du signal (hautes fréquences), donc amplifier la différence entre les échantillons successifs. Il suffit de soustraire l'échantillon passé à l'échantillon actuel pour atténuer les composantes graves.

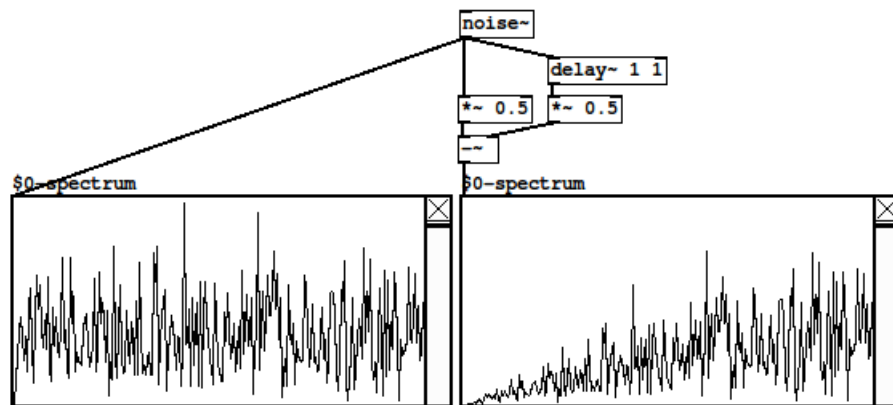


FIGURE 5.4 – Filtre passe-haut FIR de premier ordre.

En utilisant un délai de 2 échantillons, il est possible de créer un filtre FIR réjecteur de bande (additif) ou passe-bande (soustractif) de second ordre. Il suffit d'additionner ou de soustraire la valeur du délai de 2 échantillons à la valeur de l'échantillon actuel.

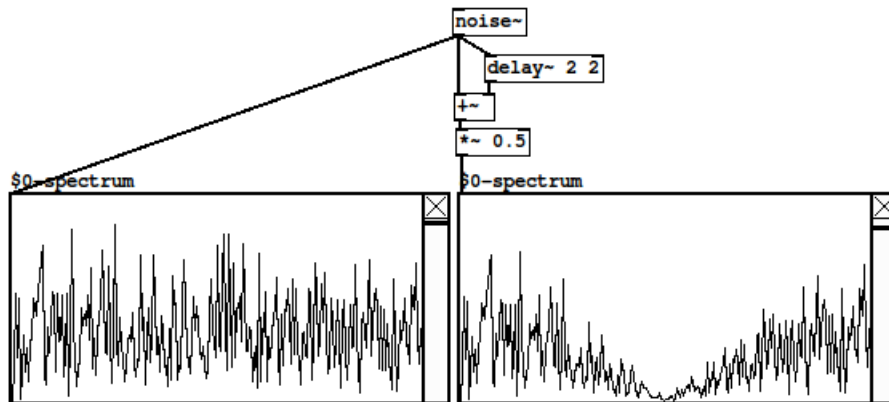


FIGURE 5.5 – Filtre réjecteur de bande FIR de second ordre.

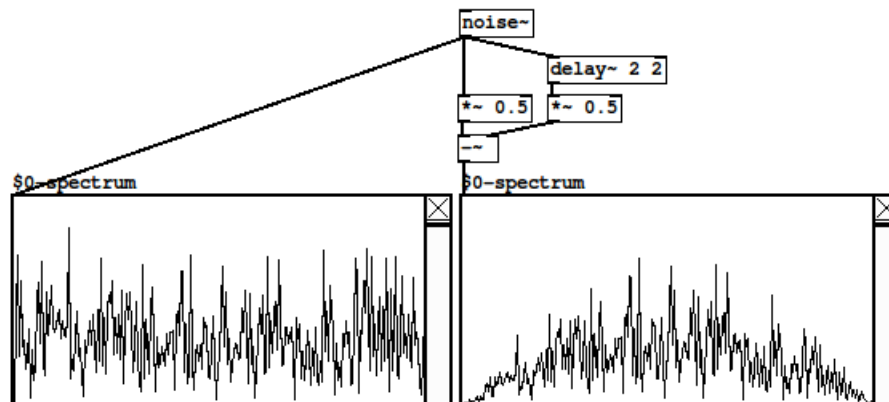


FIGURE 5.6 – Filtre passe-bande FIR de second ordre.

5.3 Les filtres IIR de bases

La librairie de base de Pure Data fournit quelques filtres récurrents de premier et de second ordre pour effectuer des opérations de filtrage simples. Ces objets offrent des entrées de contrôle pour modifier la fréquence de coupure du filtre ainsi que la largeur de bande des filtres passe-bande. Comme il n'est pas permis de contrôler ces objets aux taux audio, il faudra éviter les variations rapides des paramètres, afin de prévenir l'ajout d'artefacts dans le signal filtré. Ces objets sont économiques en temps de calcul et serviront principalement à faire des corrections fixes de spectre.

lop~

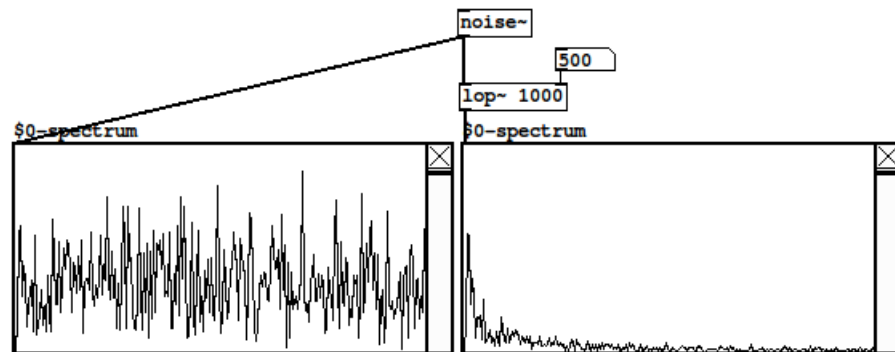


FIGURE 5.7 – Filtre passe-bas IIR de premier ordre.

hip~

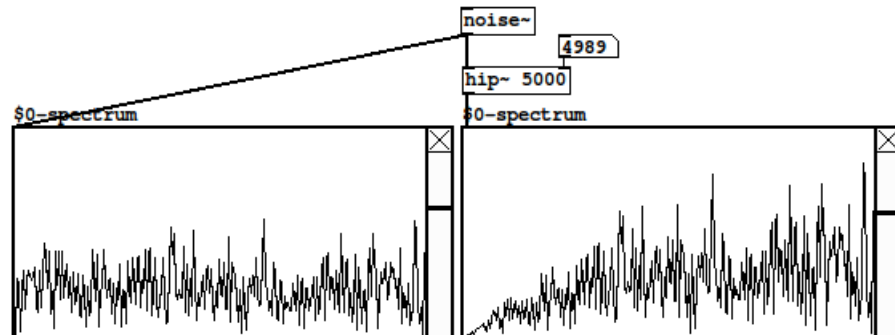


FIGURE 5.8 – Filtre passe-haut IIR de premier ordre.

bp~

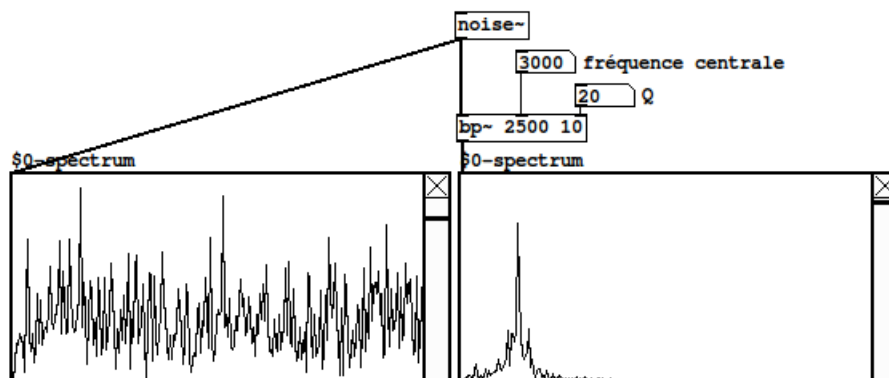


FIGURE 5.9 – Filtre passe-bande IIR de second ordre.

Une alternative avantageuse à l'objet **bp~** est l'objet **vcf~** (*voltage-controlled filter*). Cet objet attend un signal audio pour le contrôle de la fréquence centrale, permettant ainsi d'effectuer des transitions rapides sans créer d'artefacts dans le signal. À noter que le facteur de qualité (Q) ne prend toujours que des *floats*. La section suivante présente quelques objets de filtrage plus complexes mais aussi beaucoup plus versatiles. Ces objets proviennent principalement de la librairie *cyclone*, un clone des objets MaxMSP pour Pure Data.

5.4 Les résonateurs

Les filtres IIR permettent de créer des pentes de filtrage beaucoup plus serrées que les filtres FIR et ce, en utilisant un plus petit nombre de délais. Ainsi, un filtre passe-bande IIR de second ordre permet d'amincir la largeur de bande pour faire résonner le filtre à la fréquence centrale. L'objet **reson~** implémente l'équation suivante :

$$y[n] = gain * (x[n] - r * x[n - 2]) + c1 * y[n - 1] + c2 * y[n - 2]$$

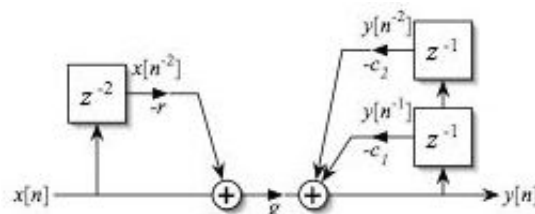


FIGURE 5.10 – Diagramme d'un filtre passe-bande IIR de second ordre.

Les paramètres d'amplitude r , $c1$ et $c2$ sont calculés à partir des données de fréquence centrale, d'amplitude et de "Q". Ces données peuvent être spécifiées en signaux ou en décimale. C'est un filtre qui est donc très simple à manipuler. Si vous désirez un filtre encore plus serré, donc plus résonnant, il suffit de brancher, en cascade, deux **reson~** recevant exactement les mêmes paramètres.

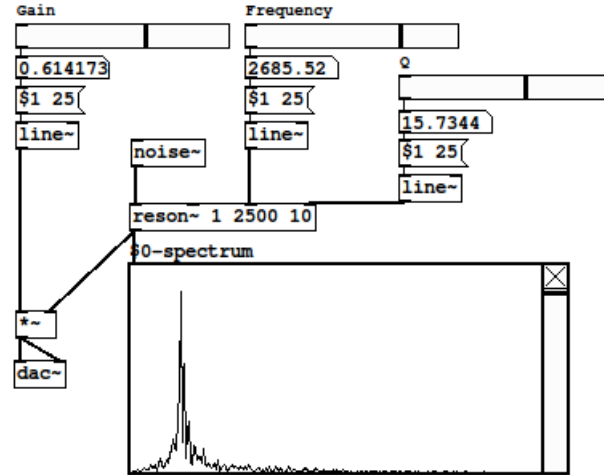


FIGURE 5.11 – Filtre passe-bande IIR de second ordre.

L'objet **bob~** implémente un filtre passe-bas résonant avec contrôle en audio sur la fréquence de coupure et la raideur de la pente (via le paramètre Q). Ce filtre constitue un bon choix pour déplacer de façon cyclique, à l'aide d'un LFO par exemple, une résonance dans le spectre d'un son.

5.5 Les filtres en peigne

Les filtres en peigne permettent de créer une série de pics et de vallées dans le spectre d'amplitude d'un son. L'implémentation de base est très simple, il suffit de faire la moyenne entre l'échantillon actuel et un échantillon passé dont le temps de délai se situera entre 0.1 ms et 10 ms, c'est-à-dire entre 4 et 440 échantillons pour une fréquence d'échantillonnage de 44100 Hz. Le temps de délai détermine l'emplacement des pics dans le spectre.

Les pics et vallées proviennent de l'annulation et du renforcement de phase des deux signaux (original et retardé). Le premier pic se situe à la fréquence :

$$f_o = f_s / D$$

où D est le délai en échantillons et f_s est la fréquence d'échantillonnage. Les pics successifs se retrouvent aux fréquences $2f_o$, $3f_o$, etc ...

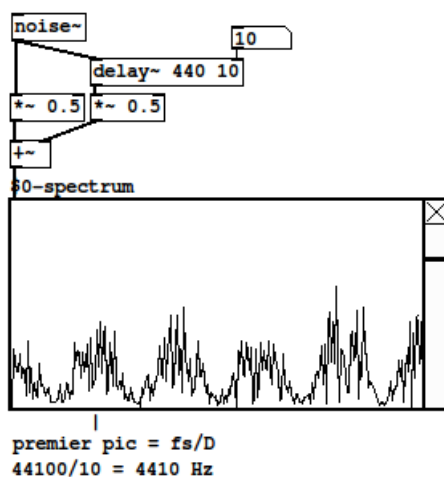


FIGURE 5.12 – Filtre en peigne FIR.

Sur l'image précédente est illustré un filtre en peigne FIR. On pourra, avec un filtre récursif (IIR), générer un filtrage en peigne où les pics seront beaucoup plus prononcés et la résonance de chacun plus grande. L'objet **comb~** est un filtre IIR dont les paramètres à contrôler sont le délai en ms, le gain du signal d'entrée, le gain de l'échantillon retardé et le feedback, c'est-à-dire le gain du signal de sortie retardé et réinséré dans le système. Voici l'implémentation de cet objet :

$$y[n] = a * x[n] + b * x[n - delay] + c * y[n - delay]$$

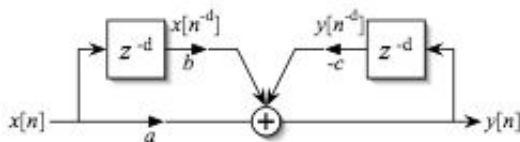


FIGURE 5.13 – Diagramme d'un filtre en peigne IIR.

Voici le spectre d'amplitude, de l'objet **comb~**, pour le même temps de délai que le filtre en peigne FIR ci-dessus. Le facteur de récursion du filtre IIR fait en sorte que les pics sont plus pointus et résonnent plus. En variant le temps de délai de façon dynamique, on peut créer des effets tel le *flange* et le *chorus*. Tous les paramètres de l'objet **comb~** acceptent les contrôles en signaux audio.

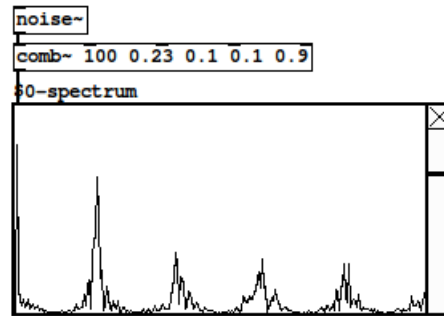


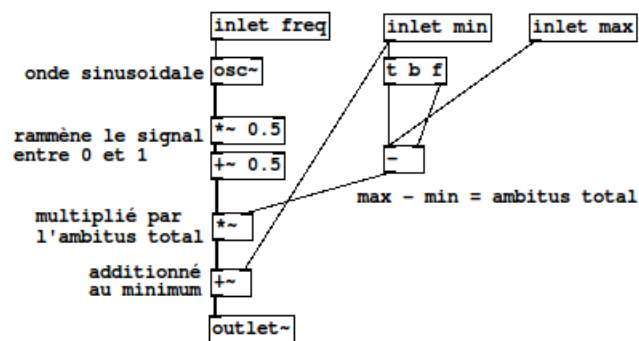
FIGURE 5.14 – Filtre en peigne IIR.

5.6 Le contrôle des paramètres

L'utilisation de filtres à des fins créatives nécessite l'élaboration de structures de contrôle efficaces et flexibles. Deux exemples sont présentés dans cette section : un oscillateur à basse fréquence (*LFO*) et une variation aléatoire à l'intérieur d'un ambitus donné.

5.6.1 Oscillateur à basse fréquence

Un *LFO* peut être implémenté simplement à l'aide d'un oscillateur sinusoïdal dont l'ambitus est adapté aux valeurs attendues par le paramètre à modifier. Dans l'exemple ci-dessous, la sortie de l'objet `osc~`, qui est normalisée entre -1 et 1, est d'abord ramenée entre 0 et 1. Ensuite, on lui applique le registre désiré en multipliant chacun des échantillons par l'ambitus total puis en lui additionnant la valeur minimum permise.

FIGURE 5.15 – Construction d'un *LFO* avec fréquence et ambitus variable.

5.6.2 Variations aléatoires

Le second exemple représente un générateur de rampes, au taux audio, dont les valeurs de destination sont comprises entre un minimum et un maximum contrôlable par l'utilisateur. Pour

ajouter de la flexibilité au processus, la vitesse de génération des valeurs est aussi modifiable via une entrée du *sous-patch*. Ce module utilise un métronome pour demander de nouvelles valeurs à un objet **random** et construit une liste (destination - durée) donnée à un objet **line~** pour obtenir les variations de valeurs au taux audio. Comme l'objet **random** ne connaît que des valeurs entières, le minimum et le maximum sont d'abord multipliés par 100, puis la valeur est générée et réajustée, en divisant par 100, en fonction du registre désiré.

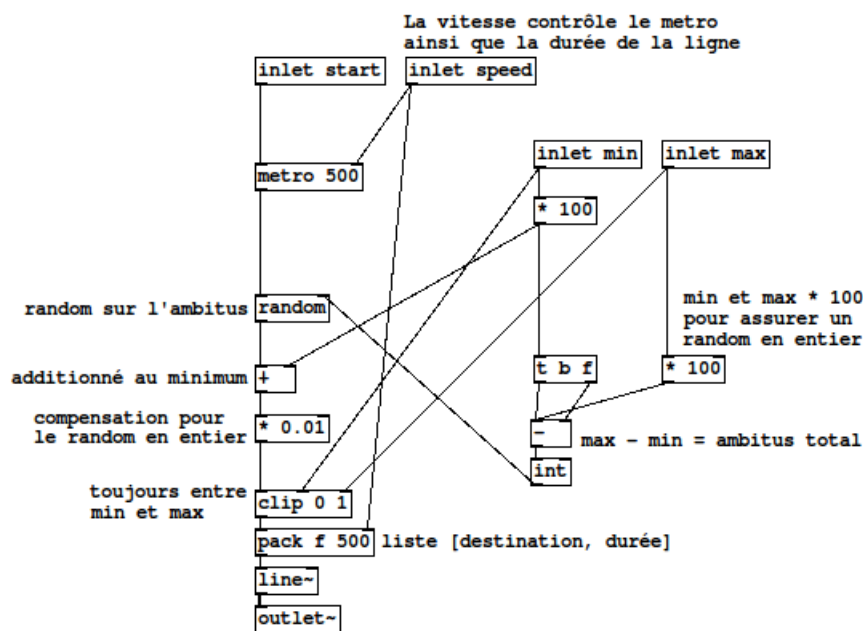


FIGURE 5.16 – Construction d'un générateur aléatoire avec vitesse et ambitus variable.

5.7 Objets à étudier

- `noise~`, `pink~`
- `phasor~`, `train~`, `osc~`
- `delay~`
- `lop~`, `hip~`
- `bp~`, `vcf~`
- `reson~`, `bob~`
- `comb~`
- `throw~`, `catch~`

5.8 Exercices

5.8.1 Exercices de révision

Sur la lecture en boucle d'un son chargé en RAM (dans un **array**), utilisez un filtre en peigne pour construire un effet de *flange* et un effet de *chorus "bon marché"*.

Le passage d'un effet à l'autre doit s'effectuer d'un simple clic sur une boîte à message contenant les paramètres de l'effet pour le LFO appliqué au temps de délai du filtre en peigne.

Flanger

- Vitesse du LFO : Plutôt lente, autour de 0.1 Hz.
- Ambitus du LFO : Large, presque 100% du délai moyen.
- Délai moyen : Très court, autour de 5 ms.

Chorus

- Vitesse du LFO : Un peu plus rapide que pour le *flanger*, autour de 1 Hz.
- Ambitus du LFO : Mince, autour 10% du délai moyen.
- Délai moyen : Un peu plus long que pour le *flanger*, autour de 12 ms.

5.8.2 Lectures

Lire la page "Filters", sur les filtres de base en Pure Data dans la section **Audio tutorials** du [manuel FLOSS](#) de Pure Data.

Chapitre 6

Analyse des données et objets maisons

6.1 Informations en provenance du clavier MIDI

On peut retirer beaucoup d'informations dans le jeu au clavier MIDI. Le programme ci-dessous compte le nombre de notes jouées dans un intervalle de 100 ms. et envoie un signal (1) si la présence d'un accord est détectée. Le principe est simple, on accumule toutes les notes jouées pendant une durée prédéterminée, ce que fait l'objet **thresh**, et ensuite, on sépare les éléments de la liste avec **iter** et on compte combien il y en a. Si plus de deux notes sont présentes, la comparaison est positive et le signal d'un accord détecté est envoyé.

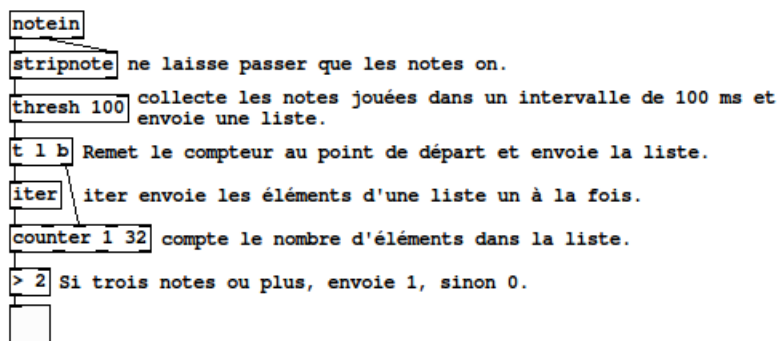


FIGURE 6.1 – Détection d'accords MIDI.

Le programme suivant calcule la durée d'une note en observant le temps écoulé entre le début et la fin de la note. Avec l'objet **timer**, il est très facile de savoir combien de temps s'est écoulé entre deux événements. Un **bang** dans l'entrée de gauche relance l'horloge à partir de zéro et un **bang** dans l'entrée de droite envoie le temps écoulé en sortie de l'objet.

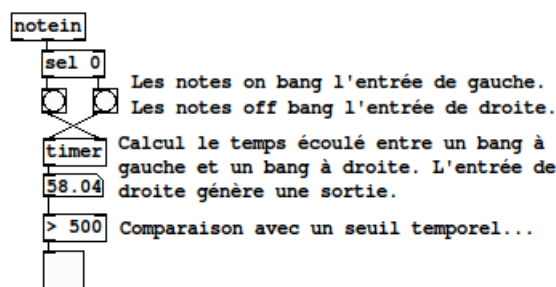


FIGURE 6.2 – Détection de la durée d’une note MIDI.

6.2 Un transpositeur intelligent

Voici un transpositeur qui évite d’avoir des notes collées lorsque la valeur de transposition change entre un *note on* et un *note off*. La hauteur réelle des *notes on* et la hauteur transposée sont enregistrées en tant que paire (x, y) dans le funbuff. Aussitôt après, la hauteur réelle appelle la transposition, prend la vélocité et joue la note. Lorsque la vélocité reçue est de zéro, aucune paire n’est enregistrée, mais la hauteur réelle appelle toujours la hauteur transposée pour reconstruire le *note off*. Même si la valeur de transposition est modifiée en cours de jeu, une note doit absolument recevoir son *note off* avant de pouvoir enregistrer une nouvelle transposition sur une même adresse, aucun *note on* ne peut donc être oublié.

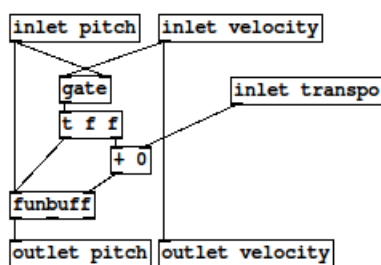


FIGURE 6.3 – Le transpositeur intelligent.

6.3 Les expressions et les conditions

Certaines conditions ou opérations mathématiques nécessitent parfois l’usage de plusieurs objets pour être complètes et opérationnelles. Dans la plupart des cas, on pourrait faire appel aux objets **expr**, pour "expression", et **expr if**, pour les conditions, et réduire toute la formule à un seul objet. Il suffit d’en maîtriser la syntaxe, qui est fortement inspirée du C, pour écrire des formules très puissantes.

6.3.1 Les expressions

Pour écrire une expression, vous pouvez utiliser un total de 9 variables appelées \$i#, \$f# ou \$s#, pour entier, décimale et symbole, où le symbole # sera remplacé par le numéro de l'entrée désirée pour la variable. Tous les opérateurs mathématiques sont permis, ainsi qu'un grand nombre de fonctions tels sinus, cosinus, racine, minimum, maximum, logarithme, random, etc. On peut même faire appel à des valeurs écrites dans une table avec le message *symbol* suivi du nom de la table. Voici quelques exemples d'expressions :

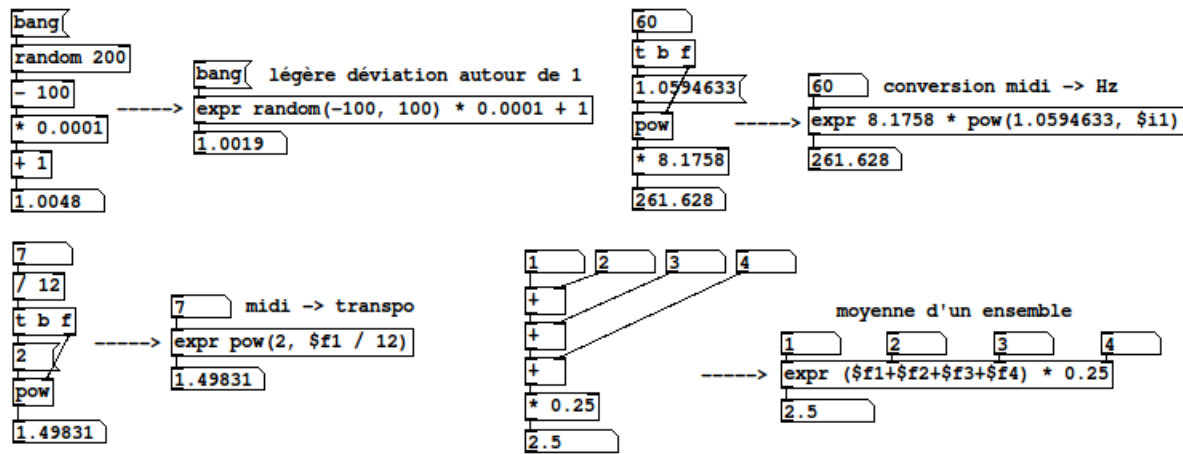


FIGURE 6.4 – Écriture d’algorithmes avec l’objet **expr**.

Une expression peut aussi être appliquée sur un vecteur audio grâce à l’objet **expr~**. L’expression sera exécutée pour chacun des échantillons en entrée. Une nouvelle variable est définie pour cet objet, \$v#, qui attend un signal audio dans son entrée correspondante. La première entrée d’un objet **expr~** doit absolument être un signal audio mais il est permis d’utiliser des données de contrôle pour les autres entrées. Voici un petit exemple de saturation du signal (*hard clipping*) implémentée à l’aide d’une expression audio :

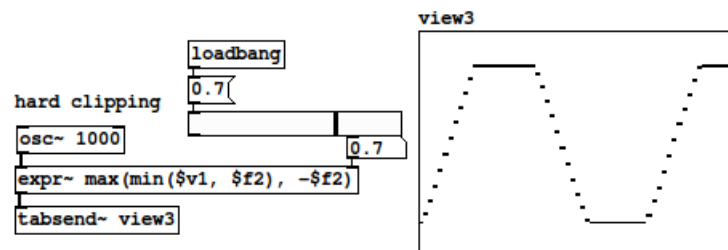


FIGURE 6.5 – Manipulation de vecteurs audio avec l’objet **expr~**.

Plusieurs expressions peuvent être données à l’intérieur d’un seul objet, il suffit de les séparer à l’aide du point-virgule (;). L’objet possèdera autant de sorties qu’il y a d’expressions, chaque

expression acheminant son résultat vers la sortie correspondante.

Il existe aussi un objet pour écrire des expressions encore plus complexes sur un signal audio, il s'agit de **fexpr~**. Cet objet offre un accès indépendant à chacun des échantillons passés, à l'intérieur d'un *block size*, en entrée comme en sortie. Cet objet, qui dépasse les limites de ce chapitre, est idéal pour implémenter des filtres récurrents (IIR).

6.3.2 Les conditions

Une condition peut être insérée dans une expression à l'aide du symbole *if*. La syntaxe des conditions est quelque peu particulière. Le symbole *if* est suivi d'une paire de parenthèse à l'intérieur desquelles sera donnée la condition ainsi que la valeur de retour pour un résultat vrai et la valeur pour un résultat faux, le tout séparé par des virgules.

if (condition, résultat si vrai, résultat si faux)

Ainsi, pour tester si une valeur est positive, et retourner 1, ou négative, pour un retour de 0, une condition serait écrite comme ceci :

expr if (\$f1 > 0, 1, 0)

Les conditions fonctionnent aussi en audio, avec les symboles pour donner accès aux *arrays* ou en conditions multiples avec le symbole "et" (&&) et le symbole "ou" (||). Par exemple, la condition suivante retourne 1 si la valeur en entrée est comprise dans l'intervalle de 0 à 1, sinon elle retourne 0 :

expr if (\$f1 >= 0 && \$f1 < 1, 1, 0)

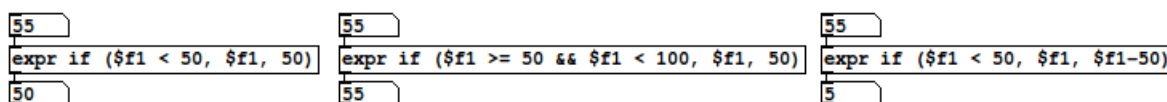


FIGURE 6.6 – Quelques exemples de conditions.

6.4 Création d'un compteur maison

6.4.1 Un compteur simple

Nous allons reprendre le concept de l'incrémenteur pour élaborer un compteur simple, capable d'avancer selon un pas variable. Le pas est, rappelons-le, la distance entre deux valeurs successives. Une boîte de décimale (**f**) sera notre espace mémoire, et avec une addition nous ferons avancer le compteur. L'objet **bondo** sera très utile pour contrôler les valeurs du minimum, du maximum et du pas. La particularité de cet objet est que n'importe laquelle de ses entrées provoque la sortie de tous les éléments en mémoire, contrairement à la plupart des autres objets

dont c'est uniquement l'entrée de gauche qui provoque une sortie. Ainsi, dès qu'une valeur est modifiée, tous les paramètres sont rafraîchis. Observons en détail comment fonctionne ce programme. On décide tout d'abord des paramètres qui régiront le compte. Lorsqu'un paramètre est modifié, l'objet **bondo** envoie le maximum dans l'entrée de droite de la comparaison, le pas dans l'entrée de droite de l'addition et, ensuite, le minimum vient se placer dans la mémoire. Le compteur est prêt. Chaque fois que l'on appuie sur le *bang*, la valeur en mémoire va s'additionner avec le pas et le résultat vient remplacer la valeur précédente dans la boîte de décimale. La sortie de l'addition se compare ensuite au maximum dans la boîte **plus grand que** et, si le maximum est dépassé, l'objet **sel** vient réactualiser les valeurs d'origine dans le **bondo**. Alors, le compte recommence au début, sinon, il continue. En dernier, la valeur qui était dans l'objet **f** au départ est affichée.

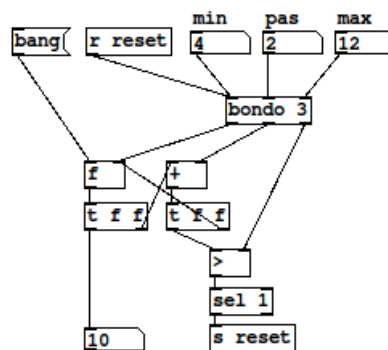
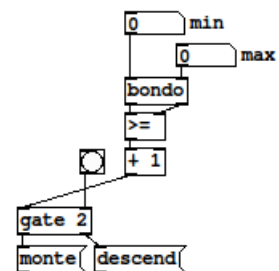


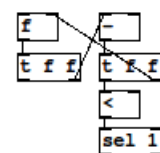
FIGURE 6.7 – Compteur simple.

6.4.2 Ajout d'un contrôle de la direction

Ajoutons maintenant une simple étape de comparaison du minimum et du maximum afin de contrôler la direction du compte. Cette opération doit s'effectuer avant tout le reste. Si le minimum est plus grand ou égal au maximum, le *bang* passera désormais par la porte de droite et le compte descendra, sinon, le compte montera comme avant. Les termes **minimum** et **maximum** peuvent maintenant être contradictoires, on devrait plutôt dire **départ** et **arrivée**!



Il ne reste qu'à doubler la section "incrémenteur et comparaison" et à modifier la copie pour que le compte descende. On remplace l'addition par une soustraction, le ">" par un "<" et le tour est joué, avec les mêmes contrôles, nous avons un compteur qui fonctionne dans les deux sens.



numéro de l'argument, ne peut être utilisée dans une boîte à message puisque cette dernière implémente une fonctionnalité différente pour ce symbole. \$1, dans une boîte à message, sera ignoré à l'initialisation et remplacé par une valeur donnée en entrée de l'objet. Les arguments doivent donc être récupérés dans une boîte d'objet standard, soit dans une boîte de décimale (f), une boîte de symbole (**symbol**) ou en remplacement d'un argument (ex. : *metro \$1*).

L'argument magique

Un cas particulier concerne l'argument \$0. Ce symbole, dans une boîte d'objet, n'attendra pas un argument à l'initialisation mais générera plutôt une valeur numérique unique pour chaque instance de l'objet (à partir de 1001). Cet argument sera fort utile pour indépendantiser les différentes instances d'un objet dans une même patch. Un cas classique consiste à faire précéder tous les noms des objets **send** et **receive** par "\$0-" afin d'éviter les conflits d'une instance à l'autre.

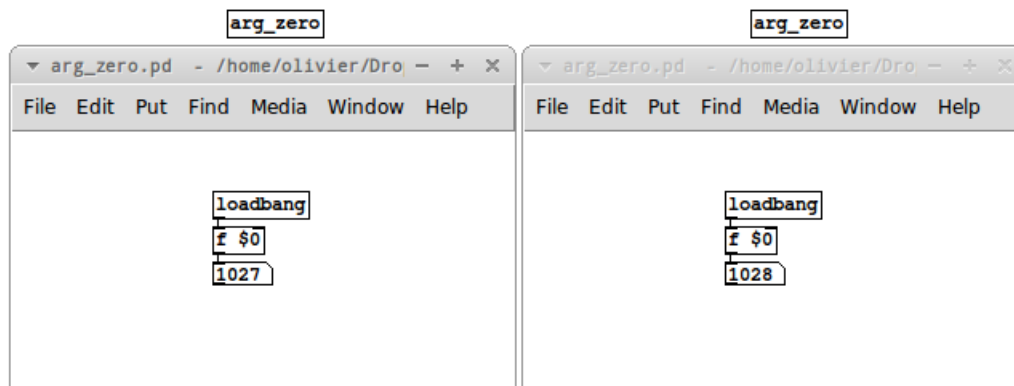


FIGURE 6.10 – L'argument 0 retourne une valeur unique par instance.

6.6 Ajout d'arguments d'initialisation au compteur maison

Nous allons maintenant modifier le compteur maison afin de pouvoir l'utiliser en tant qu'objet dans n'importe quel patch. Les entrées et sorties nécessaires au bon fonctionnement du compteur sont d'abord ajoutées puis la patch est sauvegardée sous le nom *fcounter.pd*. Il est donc maintenant possible d'utiliser plusieurs fois notre objet dans une même patch.

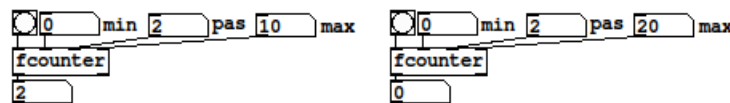


FIGURE 6.11 – Deux compteurs côte à côte.

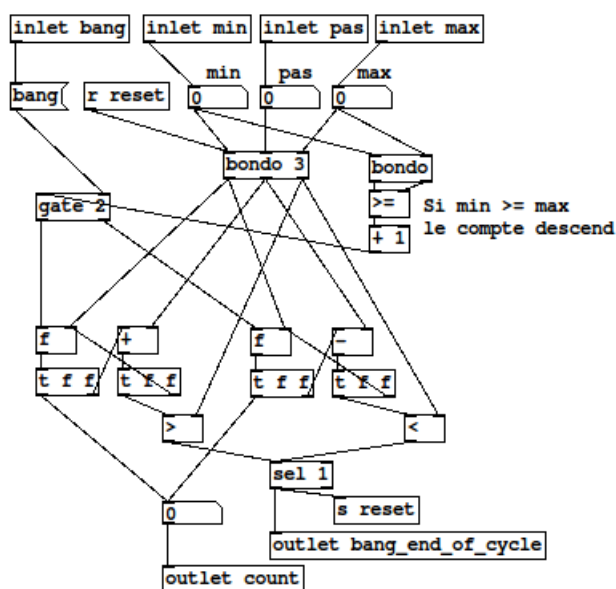


FIGURE 6.12 – Le compteur bi-directionnel avec entrées/sorties de contrôle.

L'utilisation de plusieurs compteurs dans un même patch met en évidence la problématique des **send/receive** multiples. Un objet **s reset**, tel que celui figurant dans le compteur, enverra un **bang** à tous les objets **r reset** présents dans l'environnement Pure Data, et ce, peu importe le patch dans lequel ils se trouvent. Ceci implique que lorsqu'un compteur est arrivé à la fin de sa course, il se replace au minimum et replace aussi tous les autres compteurs à leur minimum, peu importe l'état du compte de ces derniers. La solution à ce problème consiste à remplacer les objets **s reset** et **r reset** par **s \$0-reset** et **r \$0-reset** afin d'assurer que les symboles d'envoi et de réception sont différents pour chaque instance du compteur.

Les trois premiers arguments donnés au compteur seront ensuite acheminés respectivement vers le minimum, le pas et le maximum. Le *sous-patch* suivant, connecté aux boîtes à chiffre du compteur, offre en prime des conditions permettant d'attraper la valeur 0 (valeur par défaut si les arguments sont omis) et de la remplacer par une valeur qui fait du sens dans le contexte d'un compteur.

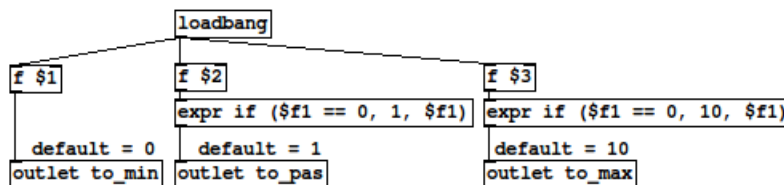


FIGURE 6.13 – Arguments et valeurs par défaut.

6.7 Contrôle du compteur et élaboration d'un contenu musical

Une application simple du compteur consiste à lui faire générer des valeurs de notes MIDI par segments. À l'aide de boîtes à message, on sauvegarde différents groupes de valeurs pour les paramètres minimum, pas et maximum. Ces groupes représenteront les segments mélodiques à jouer. Dans l'exemple ci-dessous, les groupes sont appelés de façon aléatoire à chaque fin de cycle grâce à la deuxième sortie du compteur, qui envoie un bang quand le compte a atteint le maximum. Pour permettre les répétitions du segment choisi, l'objet **random** génère des valeurs qui dépassent le nombre de groupes disponibles. Si la valeur pigée ne figure pas dans les arguments de l'objet **sel**, ils sortiront à droite et n'auront aucun effet.

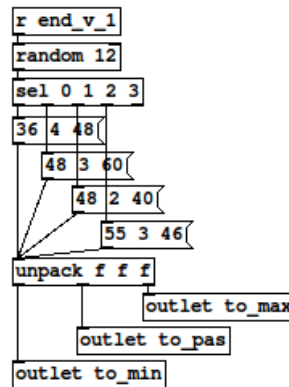


FIGURE 6.14 – Pige aléatoire parmi un groupe de valeurs prédéfinies (**pd values**).

Pour donner un peu de vie à la lecture mélodique, le métronome sera soumis à son propre compteur, dont le compte avancera aux fins de cycle, dans une proportion de 50% des *bangs* reçus.

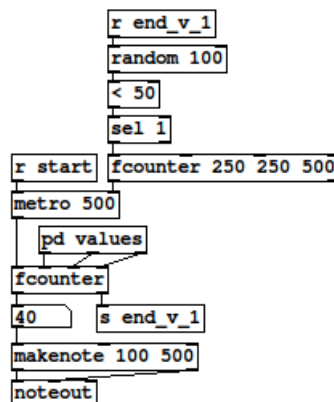


FIGURE 6.15 – Une voix mélodique avec variations de rythmes et de hauteurs des notes.

Pour créer un contrepoint quelque peu frénétique, il suffit simplement de dupliquer le module précédent, de changer les noms donnés en argument des objets **send**/**receive** puis de définir de nouveaux groupes de valeurs pour le rythme et les hauteurs des notes.

6.8 Objets à étudier

- thresh, iter
- timer
- expr, expr~
- opérateurs logiques (<, >, ==, !=, <=, >=)
- bondo
- stripnote

6.9 Exercices

6.9.1 Exercice n° 1

Construire une horloge maison qui affiche les heures, les minutes, les secondes et les millièmes de seconde. Tout le mécanisme doit être en sous-patch. Doit contenir une fonction départ/pause (toggle) et un bouton de réinitialisation. Simplicité, efficacité et transparence sont de mise.

6.9.2 Exercice n° 2 (expressions)

1. Écrire une expression qui convertit des valeurs en Hertz en temps de délai en millisecondes. Rappel : La fréquence est l'inverse de la période.
2. Écrire une expression qui génère une valeur aléatoire entre 0 et 1 avec 4 décimales de précision.
3. Écrire une expression qui fait dévier une valeur en entrée de +/- 1%.
4. Écrire une expression qui retourne la longueur de l'hypoténuse d'un triangle rectangle en fonction de la longueur des 2 autres côtés.

6.9.3 Exercice n° 3 (conditions)

1. Écrire une condition qui laisse passer les valeurs plus petites ou égales à 64 et remplace les autres valeurs par -1.
2. Écrire une condition qui laisse passer les valeurs pairs et remplace les valeurs impairs par la valeur pair tout juste au-dessus.
3. Écrire une condition qui envoie un 1 lorsque 3 valeurs sont identiques, un 0 si au moins l'une d'entre elles est différente.
4. Écrire une condition qui laisse passer les valeurs entre 36 et 84, inclusivement, seulement si une deuxième valeur vaut 1. Sinon, la condition retourne -1.

Chapitre 7

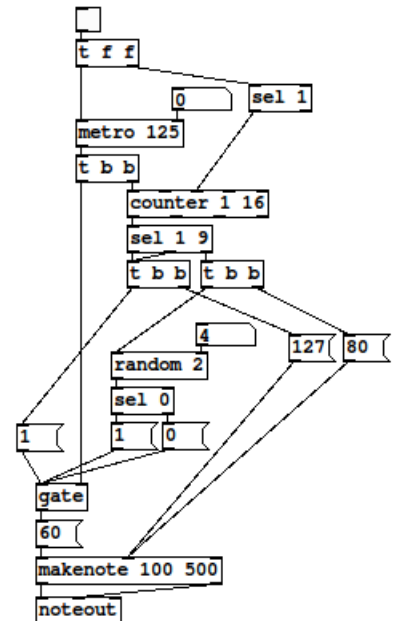
Algorithmie et automate

7.1 Algorithmie rythmique

Deux exemples d'algorithmes rythmiques vous sont présentés dans ce chapitre. Bien qu'ils soient très simples, ils n'en sont pas moins efficaces, et pourraient servir de point de départ pour l'élaboration d'algorithmes plus complexes, développant un contenu musical plus élaboré. Nous introduisons ici un univers qui n'a pour limites que votre imagination...

7.1.1 Rythmes aléatoires

Le premier exemple se base sur l'objet **random** pour varier les mesures successives de façon à ce que le matériel rythmique se renouvelle constamment. Un métronome bat la double-croche à une vitesse variable et fait avancer un compteur qui indique quel temps de la mesure va être joué. À partir du moment où l'on connaît la valeur numérique de chaque temps, il est très simple d'imposer des règles de génération rythmique. Ici, à l'aide de l'objet **select**, on isole les premier et neuvième "taps" et on assure qu'ils seront toujours joués avec une vitesse maximale. Le premier et le troisième temps seront donc des temps forts. Ensuite, pour les 14 autres valeurs, l'objet **random** détermine si la porte, qui laisse passer ou non les "bangs" du métronome, sera ouverte ou fermée. Plus le nombre de valeurs aléatoires possibles est grand, moins il y aura de notes entre les temps forts, car seul le zéro ouvre la porte.



7.1.2 Rythmes cellulaires

Dans cet exemple, nous créerons des cellules rythmiques afin de garder le contrôle sur les différents rythmes possibles à l'intérieur de la durée d'un temps. Plus on définit de cellules, plus la mesure pourra être complexe et intéressante. Par contre, nous allons laisser au hasard la succession des différents groupes rythmiques. Dans l'algorithme précédent, nous utilisons une seule porte dont l'ouverture était questionnée à chaque "tap", tandis que dans cet algorithme, il y aura plusieurs portes qui mèneront chacune vers une des différentes cellules possibles. Une seule porte peut être ouverte à la fois et elle reste ouverte pour toute la durée d'un temps. Lorsque le compteur arrive à la dernière valeur du groupe courant, on vient activer de nouveau l'objet **random** afin d'ouvrir une nouvelle porte. Les valeurs rythmiques sélectionnées sont envoyées, à l'aide d'un objet **send**, au métronome, qui s'ajuste pour suivre le rythme de la cellule.

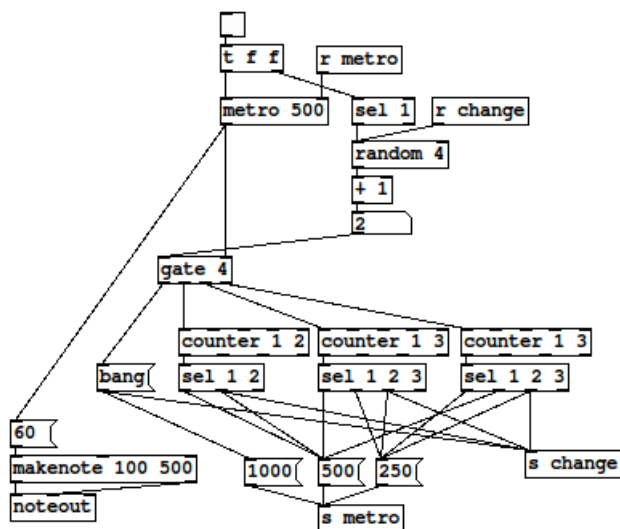


FIGURE 7.1 – Générateur de rythmes cellulaires.

7.2 Introduction à l'objet line

Nous allons maintenant voir qu'un contrôle judicieux des paramètres d'un objet peut parfois étendre ses possibilités bien au-delà de sa fonction principale. À la base, l'objet **line** sert à créer une ligne entre une valeur de départ et une valeur d'arrivée dans un temps donné. Si on lui envoie le message "34, 86 1000", l'objet **line** passera de 34 à 86 en 1 seconde. Cette fonction est très utile pour varier des valeurs en continu. Si le message ne contient qu'une seule liste de deux items, par exemple, "256 2000", la ligne partira de la dernière valeur en mémoire pour se rendre à 256 en 2 secondes.

7.2.1 Concept de "grain"

On peut préciser à l'objet **line**, indépendamment de la durée de la ligne, quel sera le grain, c'est à dire la durée écoulée entre deux valeurs successives. Ce temps est fixé à 20 ms par défaut. Donc, à toutes les 20 ms, l'objet rafraîchit la sortie de la ligne. Si la ligne avance rapidement, la valeur courante à chaque grain aura probablement changée, par contre, si elle avance lentement, il se peut que la même valeur soit envoyée plus d'une fois. Voici trois exemples de lignes avec les valeurs de sortie, tronquées à l'entier inférieur, pour un grain de 100 ms (la première valeur représente le saut direct) :

- ```

— 10, 20 1000
 sortie : 10 10 11 12 13 14 15 16 17 18 19 20
— 5, 10 1000
 sortie : 5 5 5 6 6 7 7 8 8 9 9 10
— 10, 110 1000
 sortie : 10 10 20 30 40 50 60 70 80 90 100 110

```

### 7.2.3 Plusieurs étages de génération de données

Nous possédons maintenant un module qui génère des lignes pseudo-aléatoires que nous pouvons utiliser à volonté. Après en avoir fait un objet externe, avec arguments pour l'initialisation, nous allons construire un programme qui jouera des séries de notes plus évoluées que des simples lignes. Un premier module servira de courant de notes principal auquel on viendra additionner la sortie d'un second module qui agira comme transpositeur. En fonction de la direction de chacune des lignes, le résultat final pourra contenir des notes répétées ou de grands sauts entre les notes. Un troisième module sera affecté à la vélocité afin de donner un caractère plus naturel au jeu.

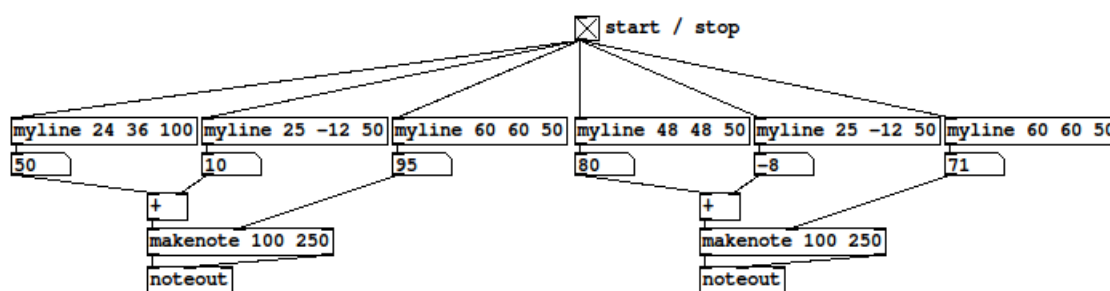


FIGURE 7.3 – Deux voix de génération pseudo-aléatoire.

## 7.3 Exercices

### 7.3.1 Exercice n° 1

Ajouter quelques extensions à l'algorithme de rythmes aléatoires.

1. Pouvoir varier la longueur des mesures.
2. Ajouter un mécanisme de variation des accents.
3. Ajouter un générateur aléatoire sur la vélocité des temps faibles.

### 7.3.2 Exercice n° 2

Ajouter quelques extensions à l'algorithme de rythmes cellulaires.

1. Assurer l'indépendance des **send** et **receive** afin de pouvoir utiliser plus d'un générateur à la fois.
2. Ajouter un mécanisme de contrôle global du tempo.
3. Inclure des accents sur la première note de chaque cellule.
4. Ajouter des cellules de rythme.

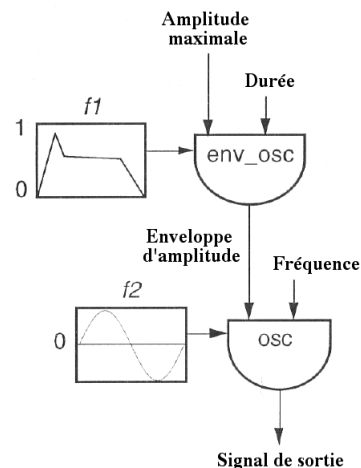


## Chapitre 8

# La synthèse additive

Bien qu'elle soit une des plus dispendieuses en temps de calcul, la synthèse additive est très puissante en ce qu'elle offre un contrôle précis sur chacun des paramètres des composantes de l'onde sonore. Pour arriver à créer une sonorité intéressante, il faut combiner une grande quantité d'oscillateurs, une centaine ou plus sont parfois nécessaires pour générer un timbre riche et puissant.

Le diagramme d'un oscillateur, dans sa forme la plus simple, est illustré à droite. Si on compte que chaque oscillateur possède une enveloppe d'amplitude d'au moins 5 points, une amplitude maximale, une durée et une fréquence de base, il faut donc spécifier autour de 1000 paramètres pour créer un son naturel. À cela, si nous ajoutons un algorithme de déviation sur la fréquence, la possibilité de dessiner la forme d'onde contenue dans la table ou un étirement de l'ordre des partiels, la quantité d'information à gérer devient énorme. Il est donc essentiel de se donner des moyens pour générer de grande quantité de données de façon automatique. C'est ici qu'un environnement comme Pure Data devient intéressant et fait revivre des méthodes de synthèse sonore telle que la synthèse additive.



Dans ce chapitre, nous allons créer, étape par étape, un synthétiseur MIDI dont le timbre sera généré à l'aide d'une synthèse additive.

### 8.1 Construction d'un synthétiseur MIDI par synthèse additive

#### 8.1.1 Étape 1 - Oscillateur simple

Nous devons d'abord créer un objet externe qui comprendra un oscillateur et un contrôle de son rang harmonique, que l'on pourra spécifier en argument. La fréquence fondamentale du son sera multipliée par le rang harmonique afin de déterminer la fréquence à laquelle chacun des partiels doit osciller. Il faut garder en mémoire que plusieurs instances de cet objet se combineront pour créer les différentes harmoniques d'un même son.

L'objet **osc~** implémente une lecture en boucle d'un cosinus à une fréquence spécifiée en Hertz. Si l'objet est créé sans argument, le contrôle de la fréquence se doit d'être en audio, tandis que si un argument numérique est donné à la création, la première entrée attendra une fréquence en data. Il est plus prudent de contrôler les variations de fréquence en audio afin d'éviter que des changements brusques produisent des discontinuités dans la forme d'onde, ce qui causera des clics en sortie. Un objet **line~** servira de portamento aux variations de fréquences de l'oscillateur, en effectuant une rampe de 10 ms entre la valeur courante et la nouvelle valeur demandée.

L'objet **osc~** permet d'accéder à une mémoire, contenant un cosinus par défaut. Nous élaborerons un peu plus loin sur la lecture de formes d'ondes complexes.

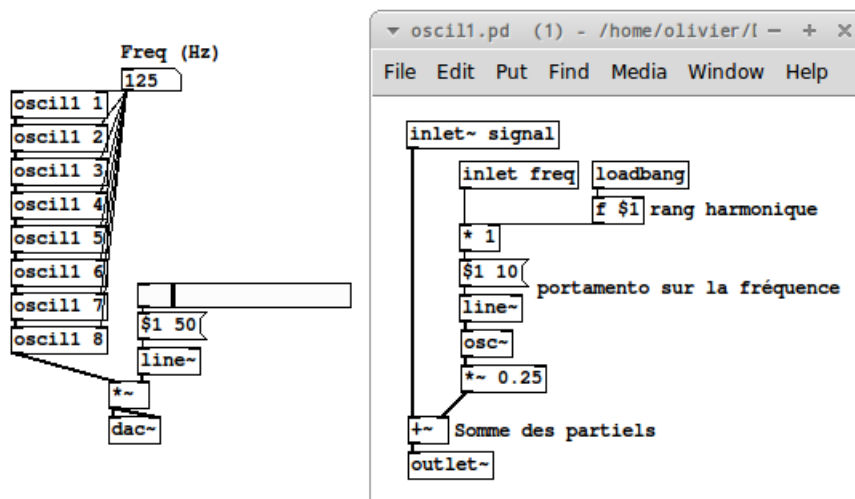


FIGURE 8.1 – Oscillateur simple avec le rang harmonique donné en argument.

### 8.1.2 Étape 2 - Variations aléatoires des amplitudes

L'étape numéro 2 consiste à ajouter une variation d'amplitude aléatoire à notre oscillateur afin de donner un peu de vie au son résultant de l'addition de plusieurs harmoniques. Chaque instance de notre objet possédera son propre générateur aléatoire qui fera varier l'amplitude de l'harmonique indépendamment de l'amplitude des autres harmoniques. La pondération des différents signaux sera donc en constante évolution. Un potentiomètre sera assigné à la vitesse de génération des valeurs d'amplitude tandis qu'un second potentiomètre contrôlera la profondeur des variations.

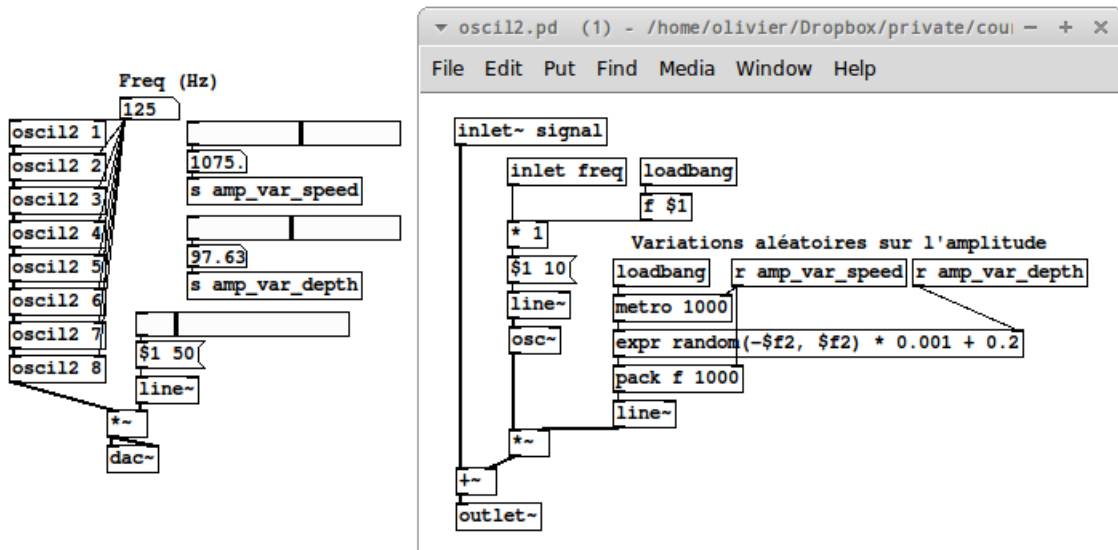
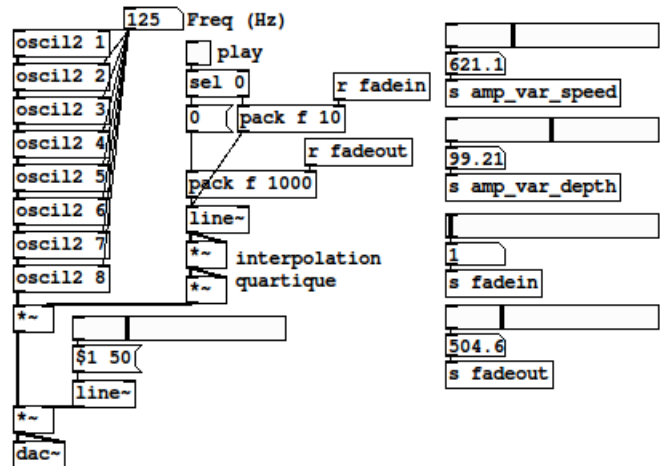


FIGURE 8.2 – Ajout des variations aléatoires sur l'amplitude des oscillateurs.

### 8.1.3 Étape 3 - Ajout d'une enveloppe d'amplitude globale

Nous allons maintenant ajouter une enveloppe d'amplitude sur la somme de tous les oscillateurs.

À cette étape-ci du programme, l'enveloppe sera activée à l'aide d'un commutateur, dont les valeurs possibles sont 0 ou 1. Cette enveloppe sera éventuellement soumise à la vélocité de la note MIDI. L'attaque de la note sera contrôlée par un temps de rampe assigné pour toute valeur positive (*fadein*) tandis que la relâche, pour une vélocité de 0, possèdera sa propre durée (*fadeout*). La rampe générée par un objet `line~` sera ensuite soumise à une interpolation quartique afin d'adoucir le contrôle linéaire de l'amplitude.



### 8.1.4 Étape 4 - Contrôle des fréquences

#### Étirement et compression d'une série de fréquences

Une méthode simple et efficace pour générer une grande gamme de spectres de fréquences à l'aide d'un seul paramètre consiste à écrire un algorithme d'étirement/compression des données.

La fonction d'étirement des partiels proposée ici est très simple, il s'agit d'élever le rang harmonique à une puissance donnée (paramètre *spreader* dans le module ci-dessous) avant la multiplication avec la fréquence fondamentale. Si la puissance est de 1, alors les harmoniques sont justes. Plus on se rapproche de 0, plus les partiels tendent à s'accorder sur la fréquence fondamentale tandis que pour une puissance plus élevée que 1, les partiels s'éloignent vers les aigus.

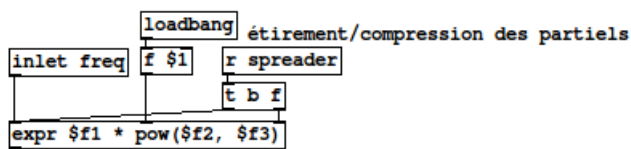


FIGURE 8.3 – Fonction d'étirement/compression des fréquences.

### Variations aléatoires sur la fréquence des partiels

La déviation sur la fréquence est implémentée à l'aide d'un **random** et d'un **line~**, activés par un métronome, pour déterminer les valeurs successives qui viendront multiplier la fréquence du partiel. La déviation sera donc différente pour chaque partiel, à l'intérieur d'une limite commune. La vitesse du changement est aussi ajustable.

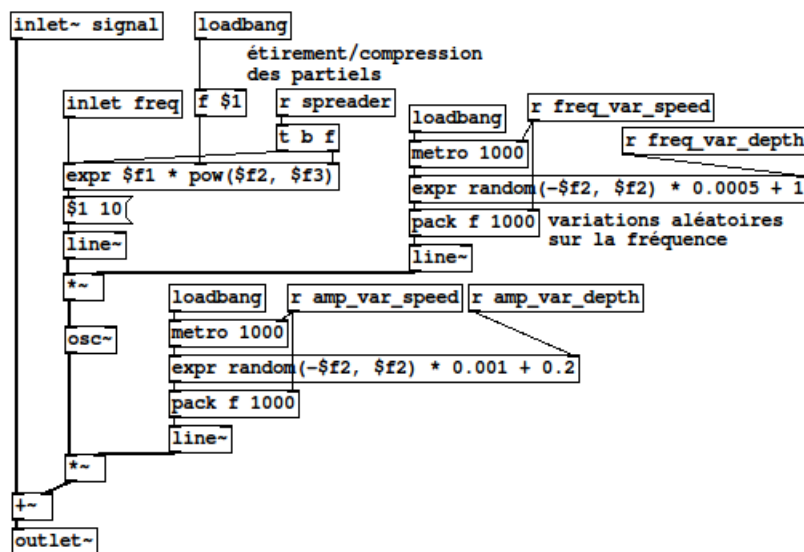


FIGURE 8.4 – Contenu de l'objet **oscil3.pd**.

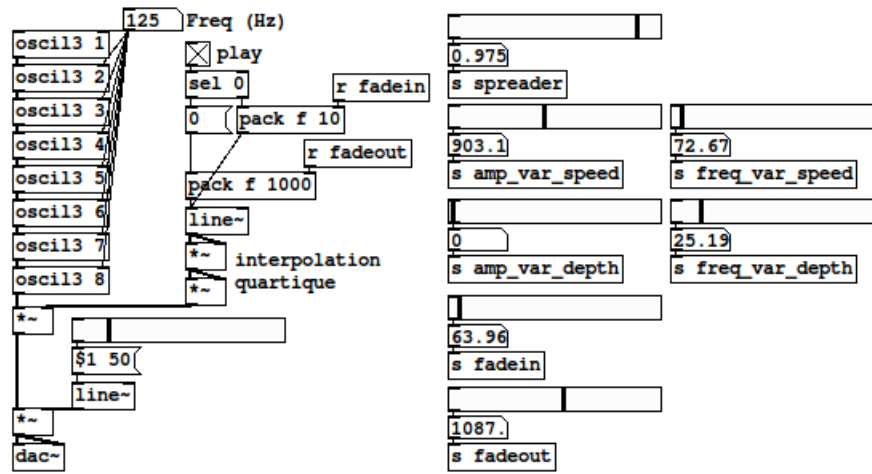


FIGURE 8.5 – Ajout des contrôles sur les fréquences.

### 8.1.5 Étape 5 - Conversion en synthétiseur MIDI et gestion de la polyphonie

La gestion de la polyphonie est un des éléments les plus difficiles à maîtriser. Il est important, avant de commencer à effectuer des modifications au programme, de bien évaluer lesquels des paramètres sont communs à tous les oscillateurs et lesquels sont indépendants. Dans ce programme, tous les paramètres contrôlés par des potentiomètres concernent le timbre général du synthétiseur et s'appliquent donc à tous les oscillateurs. Afin de prévenir l'interférence entre plusieurs synthétiseurs ouverts en même temps, nous allons donner un symbole unique à chacun des objets **send** associés aux potentiomètres. Cette opération est effectuée en ajoutant *\$0-* au devant des symboles déjà en place. Le numéro unique du programme (automatiquement attribué par Pure Data) sera donc intégré aux symboles de communication du synthétiseur. Afin que les objets **receives** correspondants répondent bien aux mêmes symboles, *\$0* sera donné en argument de l'objet **synth\_voice** (un objet **synth\_voice** par voix de polyphonie). À l'aide de l'objet **poly**, les valeurs de hauteur et de vélocité MIDI seront dirigées vers l'une ou l'autre des voix de synthèse.

L'objet **synth\_voice**, qui contient maintenant les huit oscillateurs de la synthèse additive, se charge de la conversion des hauteurs MIDI en valeur de fréquence en Hertz, à l'aide de l'objet **mtof**, et contrôle l'enveloppe d'amplitude de la voix en fonction de la vélocité MIDI reçue. Le numéro unique du programme, reçu en argument est redirigé en deuxième argument des oscillateurs afin que les objets **receives** à l'intérieur de ceux-ci reçoivent bien les valeurs modifiées dynamiquement via les potentiomètres.

Finalement, tous les objets **receives** des oscillateurs sont modifiés pour inclure *\$2-* devant le symbole identifiant le canal de communication. *\$2-* correspond au numéro unique du programme.

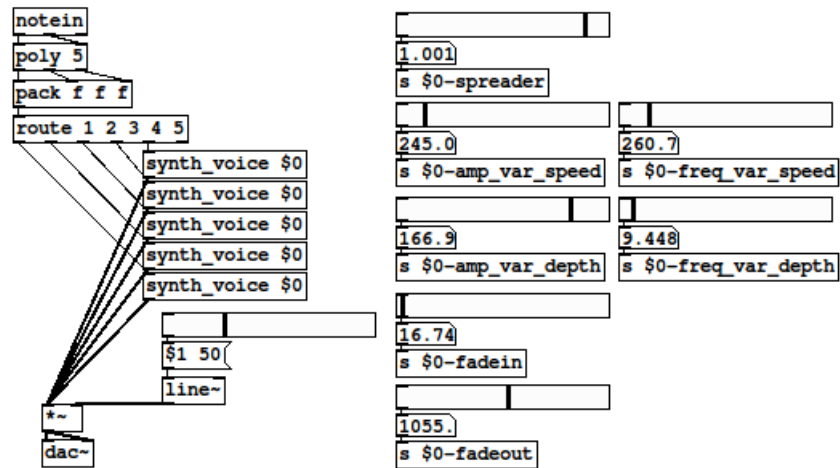
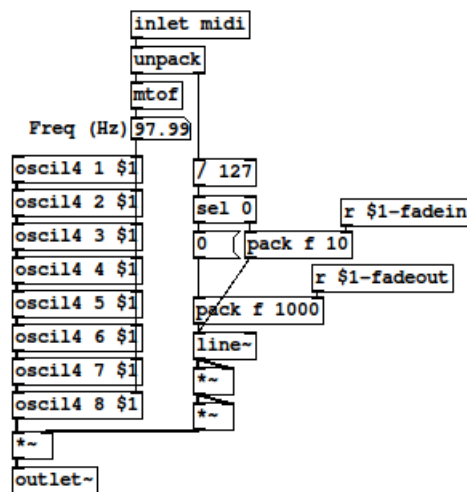
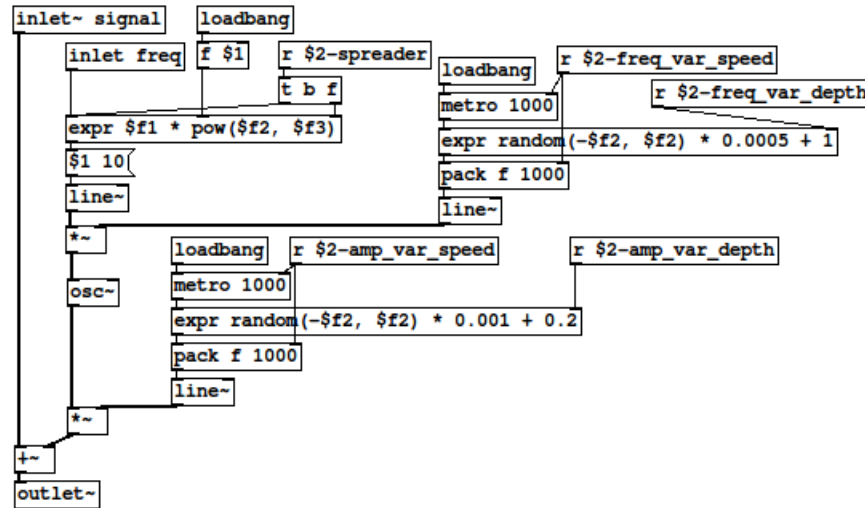


FIGURE 8.6 – Gestion de la polyphonie.

FIGURE 8.7 – Une voix de polyphonie (objet `synth_voice`).

FIGURE 8.8 – Un oscillateur (objet **oscil4**).

### 8.1.6 Étape 6 - Génération et lecture d'une forme d'onde complexe

Une technique économique pour découpler le nombre de composantes d'une synthèse additive consiste à lire une table contenant une forme d'onde complexe. Une forme d'onde complexe est généralement construite par l'addition de plusieurs harmoniques, d'amplitudes différentes, pour une période de la fondamentale. On appelle ce processus "synthèse additive fixe". Ainsi, sans augmenter le nombre d'oscillateurs présents dans le programme, on enrichit le spectre puisque chacun des oscillateurs génère plusieurs composantes. L'objet **array** offre la création rapide de forme d'onde complexe à l'aide du message *sinesum*. On spécifie, en argument du message, la longueur de la table en échantillons suivi de l'amplitude relative des harmoniques successives de la forme d'onde. En se rappelant qu'une forme d'onde carrée n'est constituée que d'harmoniques impairs dont les amplitudes sont l'inverse l'ordre, on pourrait construire une onde carrée avec le message suivant :

```
sinesum 2051 1 0 .33 0 .2 0 .143 0 .111 0 .091
```

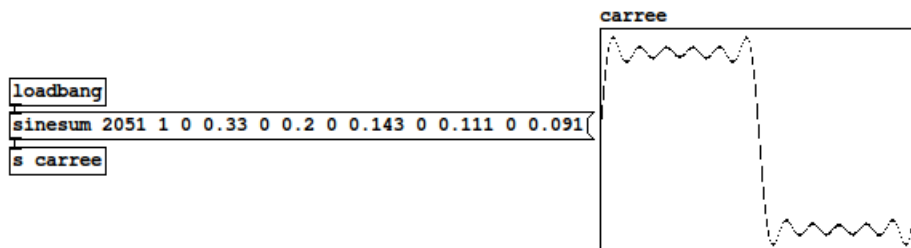


FIGURE 8.9 – Génération d'une forme d'onde carrée contenant 6 harmoniques.

La longueur de l'**array** sera automatiquement ajustée en fonction de la taille demandée. Si l'on calcul chacun des échantillons de la forme d'onde à écrire dans un **array**, il est important de prévoir les 3 échantillons nécessaires à la fonction d'interpolation cubique appliquée aux objets de lecture de table en Pure Data. Le message *sinesum* se charge automatiquement de copier le dernier échantillon au début de la table ainsi que les deux premiers à la toute fin.

Certains types de synthèse additive peuvent générer des pics d'amplitude qui dépassent largement le registre de -1 à 1. Le message *normalize* permet de rajuster la table en fonction de l'amplitude maximale donnée en argument.

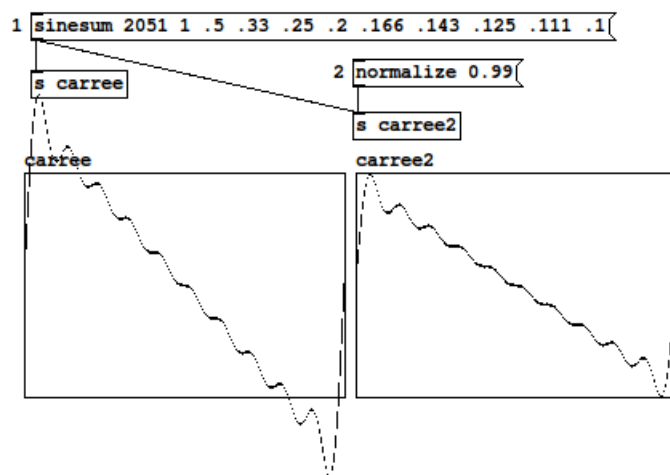


FIGURE 8.10 – Normalisation du contenu d'un **array**.

L'objet **osc~** peut être remplacé par un objet **tabosc4~** afin de lire en boucle le contenu d'un **array** (plutôt qu'un simple cosinus). **tabosc4~** prend aussi une fréquence de lecture en Hertz et applique une interpolation cubique à la lecture des échantillons contenus dans l'**array** spécifié en argument. Dans l'objet oscillateur créé précédemment, il suffit de remplacer l'objet **osc~** par **tabosc4~ \$2-wave** pour lire le contenu d'un **array** *\$0-wave* créé au niveau principal du programme.

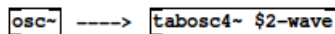


FIGURE 8.11 – Lecture d'une forme d'onde complexe.

Le module suivant utilise une banque de potentiomètre pour régler l'amplitude des harmoniques successives d'une synthèse additive fixe. En utilisant l'objet **bondo**, chacune des valeurs en entrée est toujours donnée en sortie peu importe l'entrée qui a reçu du data. L'objet **pack** construit une liste d'amplitudes, à laquelle on fait précéder le message "sinesum 515", que l'on envoie à un **array** via l'objet **send**. La forme d'onde est systématiquement normalisée afin d'éviter de trop grands écarts d'amplitude.



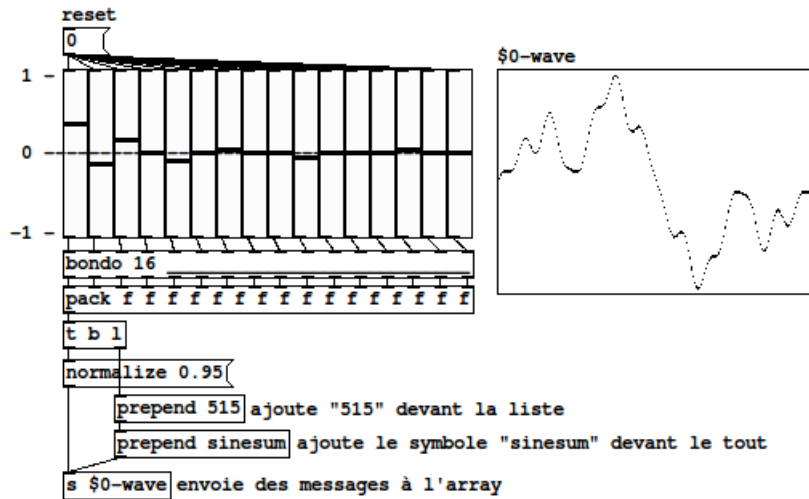
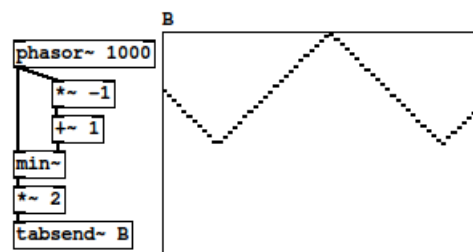


FIGURE 8.12 – Génération de formes d’ondes par synthèse additive fixe.

## 8.2 Synthèse par balayage d’un fichier son

Le module suivant utilise un fichier son, placé en mémoire dans un **array**, pour fournir des formes d’ondes complexes. Une onde triangulaire est utilisée pour balayer une région du son, dont la position de départ et la durée seront variables. La forme triangulaire permet d’éviter les discontinuités entre l’échantillon au début et celui à la fin de la région sélectionnée en effectuant un mouvement d’aller-retour. Un triangle est généré en conservant la valeur la plus petite entre une dent de scie (ascendante) et son inverse (descendante). On multiplie ensuite le signal par deux pour rétablir l’amplitude entre 0 et 1.

FIGURE 8.13 – Création d’une onde triangulaire à partir d’un **phasor~**.

La fréquence du **phasor~** devient la fréquence fondamentale du signal tandis que la forme d’onde est dépendante de la longueur de la région, une valeur en échantillons qui vient multiplier l’onde triangulaire, et de la position de départ dans le son, qui est ajoutée juste avant la lecture avec l’objet **tabread4~**. Un oscillateur à basse fréquence (*LFO*) ajouté à la position de départ fait en sorte que la forme d’onde est en constante évolution, créant des transformations continues du timbre du signal généré. Les filtres passe-bas en sortie de la lecture permettent d’atténuer les

hautes fréquences causées par la complexité d'une forme d'onde tirée d'un fichier son. Des filtres passe-bas sont aussi appliqués à tous les paramètres modifiables afin d'éviter les changements brusques de position à la lecture, ce qui aurait pour effet d'introduire des clics dans le signal audio.

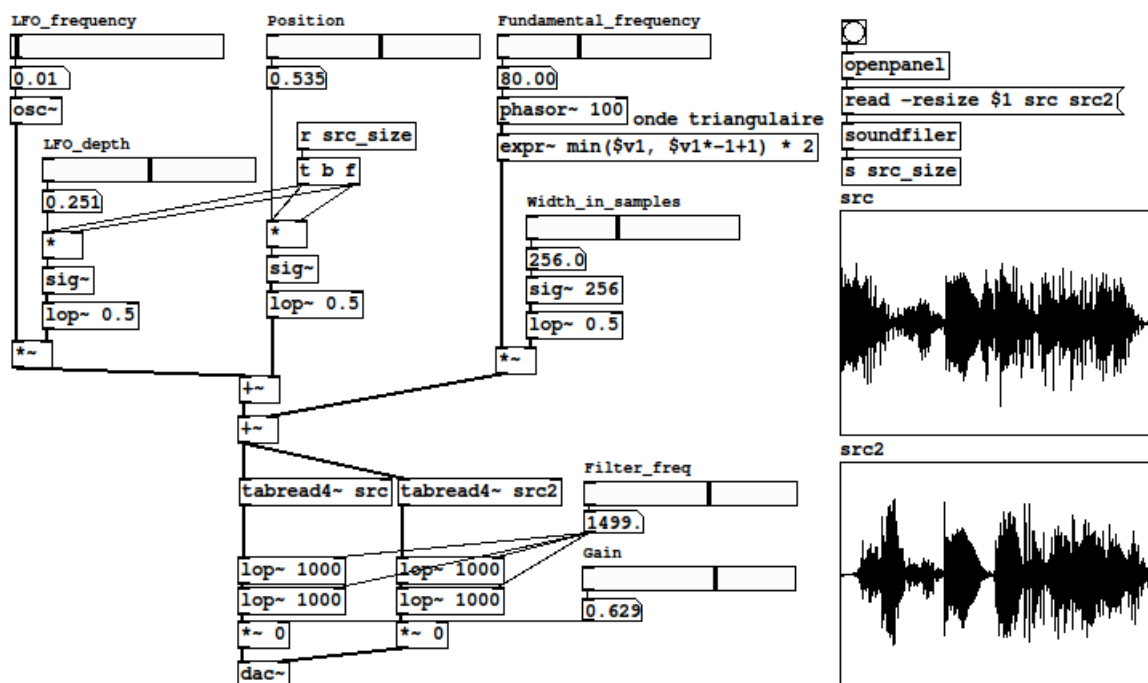


FIGURE 8.14 – Balayage d'un fichier son à l'aide d'une onde triangulaire.

## Chapitre 9

# Les synthèses par modulation

### 9.1 Synthèse par modulation d'amplitude

#### 9.1.1 Différence entre signal bipolaire et signal unipolaire

La différence entre un signal bipolaire et un signal unipolaire est très simple : le signal bipolaire oscille autour du zéro, prenant des valeurs positives et négatives tandis que le signal unipolaire est un signal bipolaire dont on a repositionnées les limites inférieures et supérieures pour qu'il oscille dans une seule zone, positive ou négative. La figure ci-dessous illustre deux ondes triangulaires, la première est unipolaire et la seconde bipolaire.

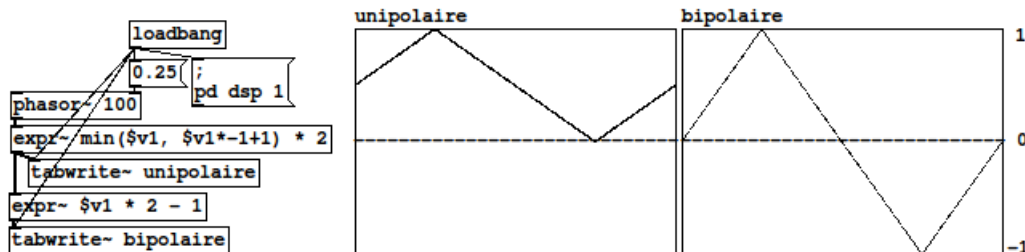


FIGURE 9.1 – Signal unipolaire versus signal bipolaire.

L'utilisation de l'une ou l'autre des formes d'ondes aura une incidence importante sur le résultat sonore. Une forme d'onde bipolaire est généralement utilisée comme oscillateur, c'est-à-dire comme générateur de son. Une utilisation typique de la forme d'onde unipolaire consiste à générer une enveloppe d'amplitude.

Dans le cas d'une modulation entre deux signaux, le signal modulé étant généralement bipolaire, le choix de la polarité du signal modulant sera très important, car de cette dernière dépend le contenu spectral du son résultant.

### 9.1.2 Modulation d'amplitude et trémolo

Voici une implémentation simple d'une modulation d'amplitude, c'est-à-dire une modulation entre un signal bipolaire et un signal unipolaire. Un sinus est multiplié par une fonction *hanning*, en forme de cloche, dessinée dans un **array** et lue à l'aide de l'objet **tabosc4~**.

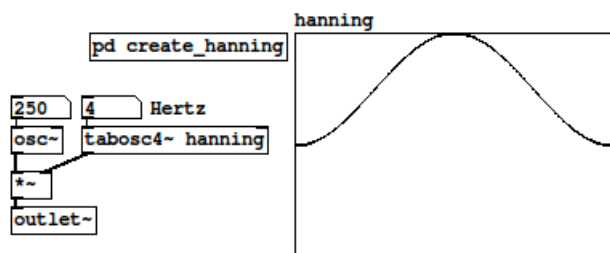


FIGURE 9.2 – Modulation d'amplitude simple.

Quels sont les résultats sonores possibles :

- Si la fonction *hanning* est lue à une vitesse très lente, on obtient une enveloppe d'amplitude, donc le volume du sinus augmente et diminue sur une période de temps allongée.
- Entre 4 et 15 Hertz, on obtient un trémolo, une variation rapide et cyclique de l'amplitude.
- Plus haut que 20 Hertz, on arrive dans le domaine audio, donc les harmoniques de chacun des deux signaux se multiplieront pour créer un nouveau spectre sonore.

Dans le cas d'une modulation entre un sinus bipolaire et un sinus unipolaire, le spectre résultant sera composé de la fréquence porteuse (sinus bipolaire) à plein volume, accompagnée de deux autres composantes, l'une à la fréquence  $P - M$  (porteuse - modulante) et l'autre à la fréquence  $P + M$  (porteuse + modulante). Ces deux nouvelles composantes ont une amplitude deux fois plus faible que la porteuse.

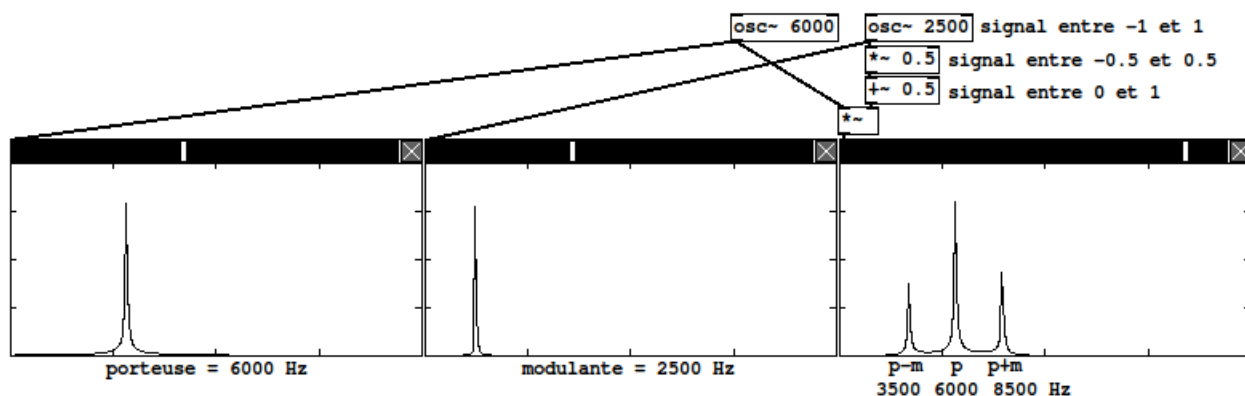


FIGURE 9.3 – Spectre résultant d'une modulation d'amplitude.

### 9.1.3 Modulation en anneaux

La différence fondamentale entre la modulation d'amplitude (modulante unipolaire) et la modulation en anneaux (modulante bipolaire), c'est que la fréquence porteuse disparaît totalement du spectre résultant. Une modulation en anneaux entre deux sinus de 6000 et 2500 Hertz donnera un spectre contenant deux composantes, l'une à 3500 Hz ( $P - M$ ) et l'autre à 8500 Hz ( $P + M$ ).

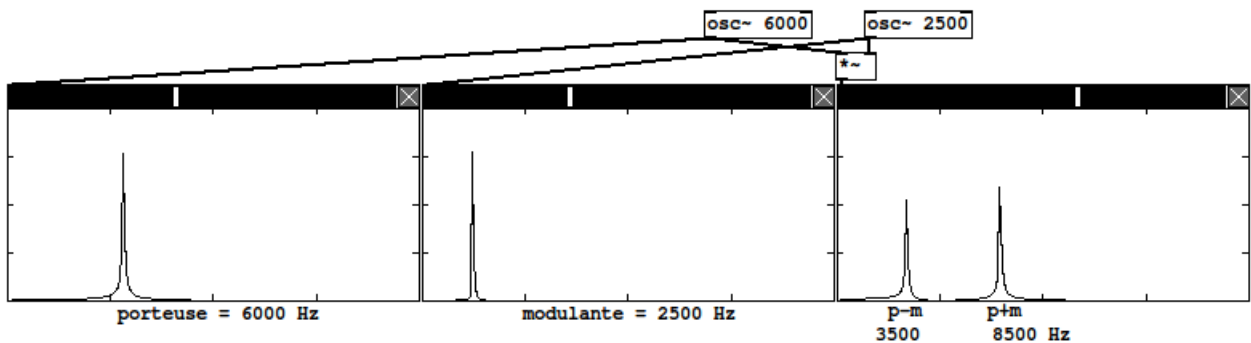


FIGURE 9.4 – Spectre résultant d'une modulation en anneaux.

La modulation en anneaux devient très puissante, quoique plutôt typée, lorsqu'appliquée sur un signal complexe. Chacune des composantes du signal porteur génèrera deux composantes à  $\pm$  la fréquence modulante. Ainsi, deux signaux complexes produiront un spectre où chaque harmonique de la porteuse produira deux composantes pour chacune des harmoniques de la modulante. Voici le spectre d'une modulation en anneaux entre deux signaux ayant chacun deux composantes :

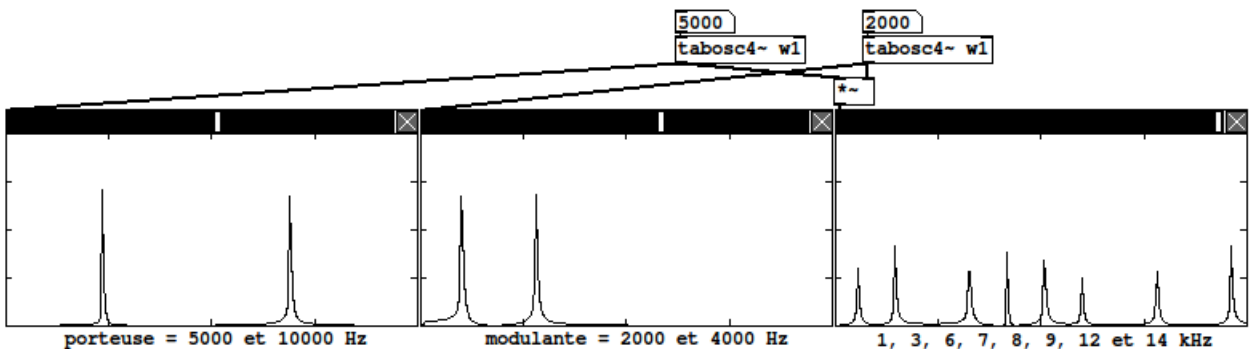


FIGURE 9.5 – Spectre résultant d'une modulation en anneaux complexe.

## 9.2 Synthèse par modulation de fréquence

### 9.2.1 Implémentation

Lors d'une synthèse par modulation de fréquence (FM), l'oscillateur modulant n'agit plus sur l'amplitude, mais sur la fréquence de l'oscillateur porteuse. Cette opération aura pour effet de générer non pas une, mais bien plusieurs bandes latérales de chaque côté de la fréquence porteuse. Nous utiliserons l'algorithme de synthèse FM développé par John Chowning, en 1973 au CCRMA de Stanford, et implémenté selon le graphique ci-dessous.

Nous retiendrons ici les trois paramètres importants dans le contrôle de cette synthèse :

- La **fréquence porteuse** : qui a pour effet de déplacer tout le spectre vers les graves ou vers les aigus.
- Le **ratio porteuse/modulante** : rapport entre la fréquence porteuse et la fréquence modulante. Si le ratio est une valeur en entier, le spectre sera harmonique, et il sera inharmonique pour tout ratio fractionnel.
- L'**index de modulation** : permet de contrôler la largeur de la déviation, donc la largeur du spectre résultant. Il a un effet direct sur le nombre de bandes latérales que contient le spectre sonore, environ  $I + 1$  de chaque côté de la fréquence porteuse.

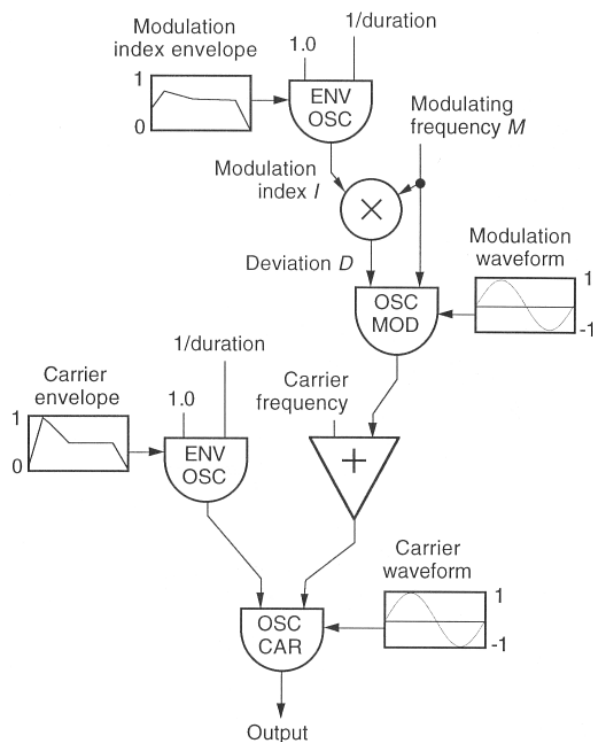


FIGURE 9.6 – Diagramme de l'algorithme de synthèse FM développé par John Chowning.

Le programme suivant synthétise la modulation de fréquence illustrée ci-dessus, avec en entrée les trois paramètres importants, c'est-à-dire la **fréquence porteuse**, le **ratio porteuse/modulante** et l'**index de modulation**.

Tout d'abord, la fréquence porteuse se multiplie au ratio pour donner la fréquence modulante. Cette fréquence détermine la distance en Hertz entre chaque bande latérale. Ensuite, la fréquence modulante se multiplie à l'index de modulation et devient l'amplitude de l'oscillateur modulant. L'amplitude de la modulante donne la déviation maximale, c'est-à-dire la distance, en Hertz, jusqu'où il y aura des bandes latérales significatives. Donc, la largeur de bande est représentée par la relation :

$$(\text{fréquence modulante} * \text{index de modulation}) * 2$$

La sortie de l'onde modulante vient osciller autour de la fréquence porteuse en s'additionnant à cette dernière, ce qui justifie que la fréquence porteuse déplace tout le spectre sur l'axe des fréquences. Le résultat est utilisé pour contrôler la fréquence de l'oscillateur porteur.

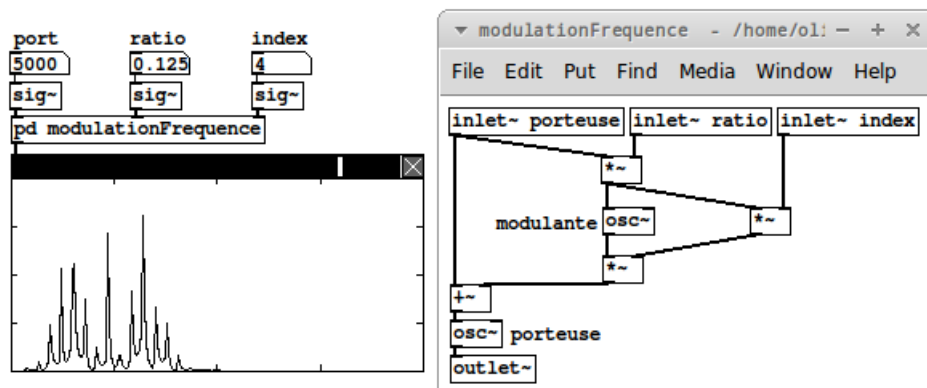


FIGURE 9.7 – Implémentation simple d'une synthèse par modulation de fréquence.

### 9.2.2 Contrôle dynamique des paramètres

La synthèse FM est un outil simple et efficace pour la génération de timbres dynamiques. Chacun des paramètres énumérés précédemment ayant un effet bien particulier sur le spectre de sortie, il est possible de créer des éléments de contrôle dont l'impact sur le signal sera facile à prévoir. Voici un petit programme qui met en place des automatisations sur les paramètres de la FM. L'index de modulation n'a pas d'effet sur la structure harmonique du signal, seulement sur la richesse du spectre en nombre de composantes. Ce paramètre est utilisé pour créer une chute plus rapide des harmoniques aiguës, en lui affectant la même enveloppe qu'au contrôle de l'amplitude de sortie, afin de simuler le comportement naturel des sons. Le ratio porteuse/modulante, qui affecte l'harmonicité du spectre, est, quant à lui, soumis à un oscillateur à basse fréquence qui, en oscillant lentement autour d'un ratio central, crée un effet de battement lorsque les harmoniques ne sont pas tout à fait justes. La fréquence porteuse est tout simplement pigée au hasard pour déplacer le spectre à chaque nouvelle note.

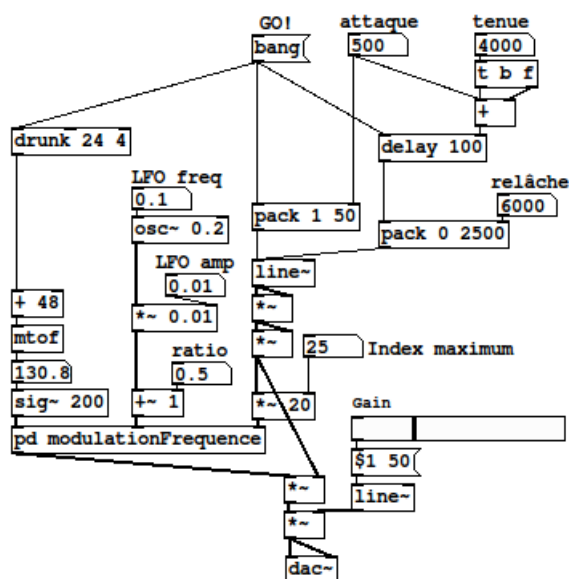


FIGURE 9.8 – Création de timbres dynamiques avec la modulation de fréquence.

### 9.2.3 La modulation de fréquence à plusieurs oscillateurs

Le programme suivant illustre la combinaison de trois oscillateurs pour former une synthèse FM complexe, un principe implémenté dans le désormais célèbre synthétiseur DX7 de Yamaha, où six oscillateurs étaient combinés selon 32 schémas possibles, pour créer les différents algorithmes sonores (<http://www.chipple.net/dx7/>). En voici un exemple :

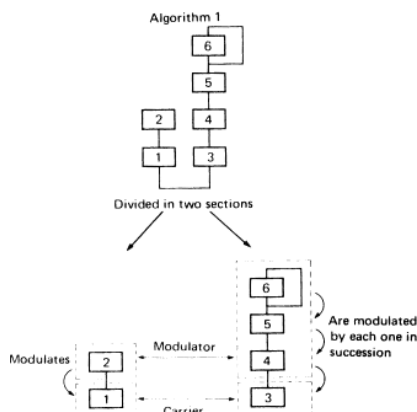


FIGURE 9.9 – Algorithme numéro 1 du DX7 de Yamaha. Quatre modulantes pour deux porteuses.

Dans notre programme, un premier oscillateur modulant est utilisé pour moduler la fréquence d'un second oscillateur modulant, qui lui, module la fréquence de l'oscillateur porteur. Chacun



des oscillateurs modulants possède son index de modulation propre, ce qui permet de contrôler indépendamment la largeur de bande des deux synthèses FM en présence. À la vue du spectre d'amplitude, on constate qu'en plus des bandes latérales créées autour de la fréquence porteuse (4000 Hz) par la modulante numéro 2, chacune de ces composantes possède ses propres bandes latérales générées par la modulation de fréquence (modulante numéro 1) sur l'oscillateur modulant. Cette technique permet de créer des spectres contenant plusieurs "pics" d'amplitude, à la manière des formants vocaux. Différentes configurations (algorithmes), telles que plusieurs modulantes pour une porteuse ou une modulante pour plusieurs porteuses, en parallèle ou en série, donneront différents types d'enveloppes spectrales. À ce stade-ci, l'exploration est essentielle !

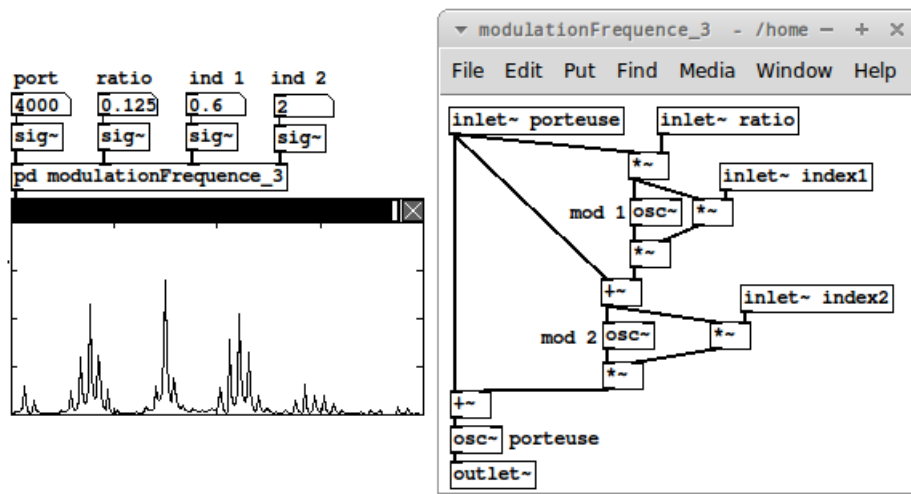
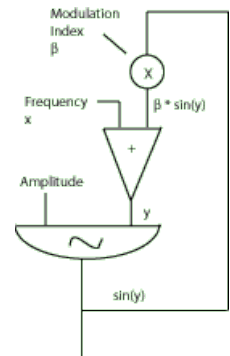


FIGURE 9.10 – Deux oscillateurs, connectés en série, modulant une porteuse.

#### 9.2.4 Un oscillateur récursif

En observant bien l'algorithme numéro 1 du synthétiseur DX7, illustré précédemment, on constate que l'oscillateur numéro 6 est défini avec une boucle sur lui-même. C'est un cas particulier de modulation de fréquence qui n'utilise qu'un seul oscillateur.

On appelle cet oscillateur un **oscillateur récursif** (*one-oscillator feedback*). Tel qu'illustré ci-contre, le signal de sortie de l'oscillateur est utilisée pour moduler la position du pointeur de lecture de la forme d'onde à l'aide d'une multiplication et d'une addition. La multiplication agit comme facteur de récursion et permet de contrôler la quantité de signal de sortie réinjecté à la lecture. Plus la récursion est élevée, plus le nombre d'harmoniques dans le signal augmente. Le rôle de l'addition est d'ajouter le signal au pointeur de lecture, qui lui, lit la forme d'onde à la fréquence demandée. Cet oscillateur, qui contient une quantité variable de composantes harmoniques, peut être utilisé pour enrichir le spectre d'une modulation de fréquence (ou d'amplitude).



Dans un environnement tel que Pure Data, il peut être ardu d'implémenter ce type de processus puisque le signal de sortie audio doit être aussitôt réinjecté dans l'algorithme de génération de ce même signal. Or, comme les signaux audio sont traités par bloc d'échantillons (64 échantillons par bloc étant la valeur par défaut de Pure Data), le temps de délai à la réinjection doit être plus grand ou égal à la taille d'un bloc. Ainsi, une récursion directe n'est pas permise.

Il existe un objet de la famille des expressions, **fexpr~**, qui non seulement accepte les signaux audio en entrée, mais donne accès au bloc d'échantillons courant, en entrée comme en sortie. Cet objet est extrêmement puissant pour écrire des algorithmes de traitement de signal tels que des filtres récurrents ou des distorsions. Deux nouvelles variables sont introduites dans cet objet d'expression, il s'agit de  $\$x$  et  $\$y$ .

- $\$x\#[n]$  permet d'accéder au bloc d'échantillons de l'entrée  $\#$  indexé à la position  $n$ , où  $n$  doit satisfaire  $0 \leq n \leq \text{-block size}$ . Si on omet de spécifier la position, l'échantillon courant est utilisé,  $\$x1 \rightarrow \$x1[0]$ .
- $\$y[n]$  permet d'accéder au bloc d'échantillons de sortie indexé à la position  $n$ , où  $n$  doit satisfaire  $0 \leq n \leq \text{-block size}$ . Si on omet de spécifier la position, le dernier échantillon de sortie est utilisé,  $\$y1 \rightarrow \$y1[-1]$ .

Cet objet nous permet donc de construire un oscillateur récursif en donnant le signal d'un objet **phasor~**, qui génère une rampe de 0 à 1 à une fréquence donnée en Hertz, à l'expression suivante :

$$\sin((\$x1 + \$f2 * \$y1) * 6.283185308)$$

Cette expression additionne tout d'abord à  $\$x1$ , c'est-à-dire le signal du **phasor~**, le dernier échantillon de sortie,  $\$y1$ , multiplié par un coefficient de récursion,  $\$f2$ . Le résultat est ensuite multiplié par  $2\pi$  et est donné à une fonction sinus pour générer le signal de sortie.

Le programme suivant, sauvegardé sous le nom **lsin~**, peut être utilisé pour générer un oscillateur récursif.

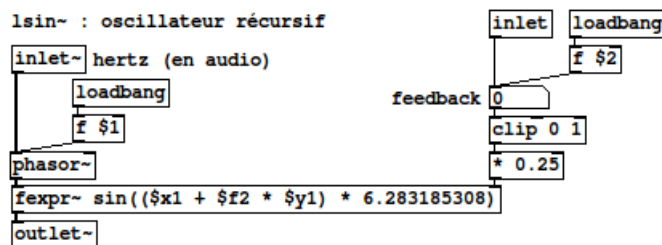


FIGURE 9.11 – Oscillateur récursif.

## 9.3 Exercices

### 9.3.1 Exercice n° 1

Remplacez un ou des oscillateurs d'une modulation de fréquence par un oscillateur récursif et expérimentez avec le facteur de *feedback*.

### 9.3.2 Exercice n° 2

Créez différents instruments de synthèse en mélangeant les modulations d'amplitude, en anneaux et de fréquence. Utilisez plus de deux oscillateurs par instrument.

### 9.3.3 Exercice n° 3

Créez un système de *presets* à l'aide de messages, contenant des listes de valeurs, appelés à l'aide de valeurs numériques et d'un objet **select**. Préparez différentes sonorités pour un instrument donné et sauvegardez-les en tant que *presets*.



## Chapitre 10

# Stockage et manipulation de données

Pour construire un séquenceur MIDI simple et efficace, il y a trois composantes essentielles auxquelles on doit s'attarder. Un mécanisme d'enregistrement, un mécanisme de lecture et un panneau de contrôle. Dans ce chapitre, nous allons conceptualiser, étape par étape, un séquenceur polyphonique en temps absolu. C'est-à-dire un séquenceur pouvant enregistrer plusieurs notes MIDI en même temps, en associant les notes à un index temporel provenant d'une horloge activée au début de l'enregistrement. Nous n'allons donc pas enregistrer la durée réelle de chaque note, mais le plutôt le moment, dans le temps, où sont arrivés le *note on* et le *note off* de chaque événement. La lecture se fera donc aussi avec une horloge, dont nous pourrions modifier la vitesse afin de varier le tempo.

### 10.1 Séquenceur polyphonique en temps absolu

#### 10.1.1 Mécanisme d'enregistrement

Le point central du mécanisme d'enregistrement de ce séquenceur réside dans le mode d'attribution de la référence temporelle. Au lieu de compter le temps écoulé entre deux notes (qui est la façon de fonctionner de certains séquenceurs), la fonction d'enregistrement démarre une horloge, qui servira de référence temporelle, en donnant le temps absolu, depuis le départ de l'enregistrement. Ces valeurs de temps serviront d'index dans une collection (objet **coll**) où sera conservées les données de hauteur et de vélocité des notes MIDI. Cette horloge doit compter suffisamment rapidement pour ne pas briser la rythmique naturelle du jeu, mais assez lentement pour fonctionner de façon efficace. La vitesse du compte a été fixée à 4 millisecondes pour deux raisons : elle respecte les deux conditions précédentes et permettra aussi de varier aisément la vitesse du jeu à la lecture.

La première étape du mécanisme d'enregistrement consiste donc à construire une horloge qui donnera les index temporels nécessaires à la mémorisation des notes jouées. Le programme suivant utilise un **metro** et un **counter** afin de créer une horloge flexible et efficace. La fréquence d'enregistrement est gérée avec le métronome et le compte des index de temps avec le compteur. Dans cet exemple la sortie du compteur est multipliée par 4 pour respecter la temporalité dictée par le métronome, qui envoie un *bang* à toutes les 4 millisecondes. Chaque sortie de l'horloge correspondra donc exactement au temps écoulé depuis le début de l'enregistrement. Pour être

certain que le compteur ne rebrousse jamais chemin, on laisse libre d'arguments l'objet **counter**, de cette façon il n'aura pas de compte maximum et il continuera toujours d'avancer vers le haut.

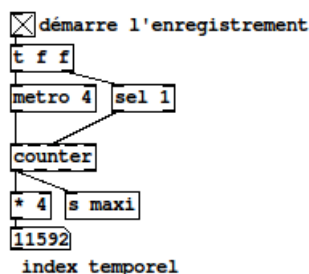


FIGURE 10.1 – Création d'une horloge donnant le temps absolu depuis le début de l'enregistrement.

L'étape suivante consiste à garder en mémoire tous les *note on* et les *note off* qui arrivent à l'intérieur de l'intervalle de 4 millisecondes entre deux tics de métronome. Ces données seront associées à l'index temporel suivant en sortie de l'horloge.

L'objet **thresh** accumule toutes les valeurs entrantes, données sous la forme de listes (hauteur - vitesse), pendant une durée définie en argument, et construit une nouvelle liste avec toutes les données reçues. Lorsque la durée en argument est écoulée, **thresh** envoie la liste cumulative en sortie. Cette liste vient ensuite s'ajouter à la suite d'un message, *store X*, dont la valeur *X* est incrémenté par le défilement du temps. Lorsqu'une ou des notes s'ajoutent au message, ce dernier est enregistré dans la collection avec la première valeur de la liste (le temps) comme index. L'objet **coll** fonctionne comme un objet **funbuff**, à la différence que ce dernier n'enregistre qu'un entier par index tandis que la collection peut enregistrer des données plus complexes, comme une liste de valeurs. En cliquant sur un objet **coll**, on peut visualiser, ou modifier, son contenu. Dans l'interface texte de la collection, l'index est séparé de sa valeur par une virgule et un point-virgule sépare deux éléments (un élément étant une paire index - valeur).

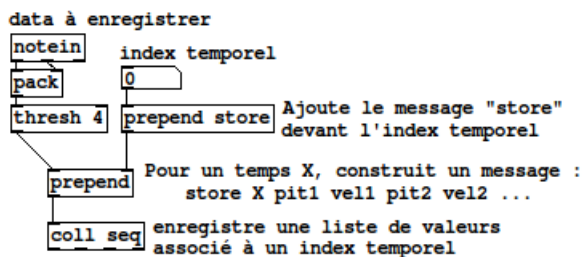


FIGURE 10.2 – Mémorisation des données dans une collection.

À noter que l'objet **coll** répond à plusieurs messages différents, permettant d'automatiser certaines tâches (voir le fichier d'aide de l'objet pour le détail de toutes les commandes possibles). Dans le module ci-dessus, le message "store" est utilisé pour enregistrer des données à un index

quelconque. Si l'index donné existe déjà dans la collection, les données associées seront automatiquement remplacées par les nouvelles valeurs en argument. On pourrait remplacer le message "store" par le message "merge" pour assurer une persistance du data et faire en sorte que les nouvelles valeurs viennent s'ajouter aux anciennes, sans les effacer. Un autre message essentiel est "clear", qui efface tout le contenu de la collection. Nous utiliserons ce message au départ d'un enregistrement afin d'éviter les éléments indésirables en provenance des enregistrements précédents. Voici le module d'enregistrement en entier.

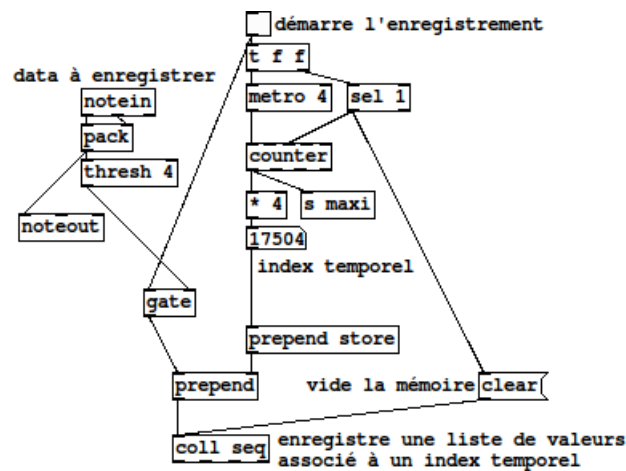


FIGURE 10.3 – Mécanisme d'enregistrement avec initialisation de l'horloge et de la collection.

### 10.1.2 Lecture et manipulation des données enregistrées

La lecture des données se fera sensiblement de la même façon qu'elles ont été enregistrées, c'est-à-dire avec un métronome qui incrémente un compteur. Les valeurs en sortie du compteur, après une multiplication par 4, serviront d'index de lecture pour la collection. Notez l'argument "seq" de l'objet **coll** qui est identique à l'argument de la collection qui a servi à l'enregistrement ; ainsi, elle fera référence au contenu enregistré précédemment dans la **coll** "seq". La suite de notes, restituée de façon intégrale, peut être directement envoyée dans un objet **noteout** ou passée préalablement dans un transpositeur.

Pour varier la vitesse de lecture, il suffit de changer la vitesse du métronome. S'il évolue à une vitesse de 2 millisecondes, la lecture se fera deux fois plus rapidement tandis que pour une vitesse de 8 millisecondes, elle sera deux fois plus lente. L'objet **metro** accepte les nombres réels comme contrôle de la vitesse, ce qui permettra de faire des variations de tempo sur un ambitus très large.

Comme la liste en provenance de la collection peut contenir plusieurs paires de valeurs (hauteur - vitesse), il faudra, avant de l'envoyer vers la sortie MIDI, la séparer en autant de listes de 2 valeurs qu'il sera nécessaire. L'objet **zl iter** fonctionne exactement comme l'objet **iter**, c'est à dire qu'il divise une liste en une succession pratiquement instantanée d'événements simples, avec comme différence qu'il peut diviser la grande liste en plusieurs petites listes, dont

la longueur est dépendante de l'argument donné à l'objet. Ainsi, **zl iter 2** diviserait une liste de 8 valeurs en 4 listes de 2 valeurs chacune, séparant 4 notes (hauteur - vitesse) qui auraient été jouées en même temps. Un transpositeur, ajouté juste avant l'envoi des notes au **noteout**, permet de relire la séquence dans une tonalité différente de l'enregistrement.

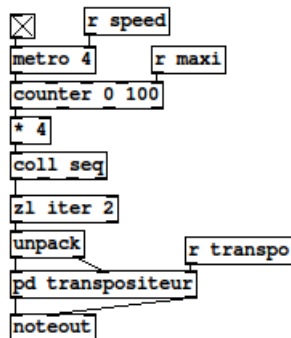
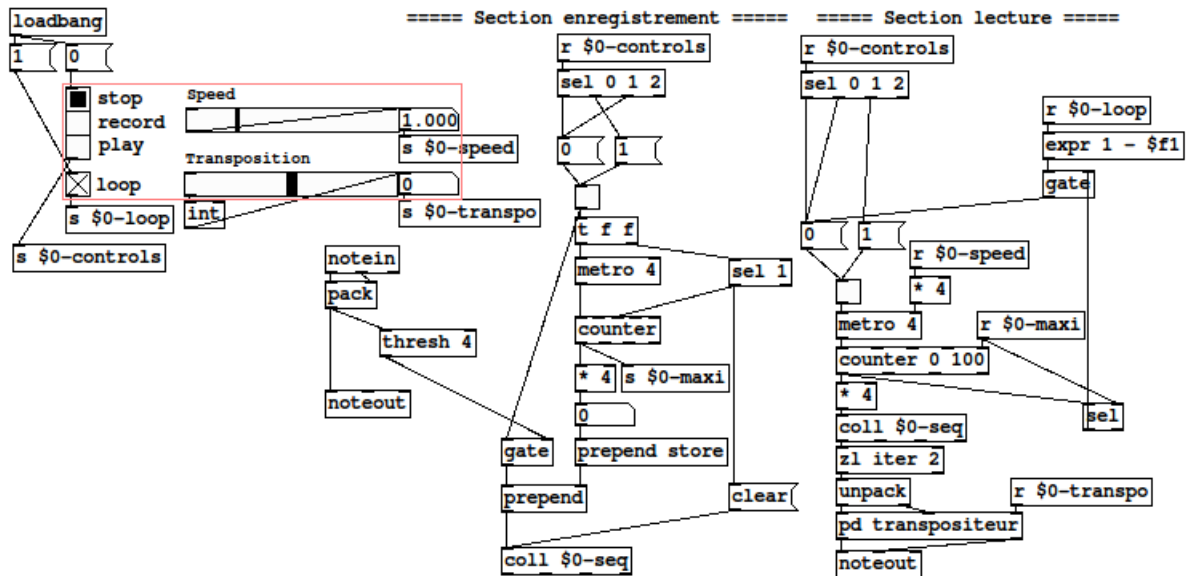


FIGURE 10.4 – Lecture en boucle de la séquence avec contrôle sur la vitesse de lecture ainsi que la transposition.

### 10.1.3 Interface de contrôle

Nous allons élaborer une petite interface de contrôle afin de simplifier la manipulation de notre séquenceur MIDI. Tout d'abord, dans le but d'assurer une totale indépendance de notre objet, le symbole "\$0-" sera ajouté au-devant du nom donné à la collection ainsi qu'au-devant de tous les symboles des objets **send** et **receive**. Ceci permettra d'éviter les conflits lorsque de multiples instances de notre séquenceur seront créés dans un même programme. Ensuite, les contrôles seront automatisés à l'aide d'objets graphiques. Un **radiobutton** servira à déterminer le mode d'opération, c'est-à-dire l'une des commandes *stop*, *record* ou *play*. Cet objet est idéal pour ce type de commandes puisqu'il ne permet qu'une seule valeur à la fois. Les sections d'enregistrement et de lecture reçoivent toutes deux la commande choisie et réagissent en conséquence. Un **toggle** permettra d'activer la lecture en boucle de la séquence, en fermant la porte pour l'arrêt du métronome à la fin de la séquence si la boucle est activée, et deux potentiomètres seront assignés respectivement à la vitesse de lecture et à la transposition. Le programme suivant illustre l'ajout des contrôles graphiques. Notez le rectangle rouge autour des objets de manipulation. Ce rectangle spécifie une section du programme qui sera visible, et accessible via la souris, lorsqu'un objet "sequenceur" sera créé dans un autre programme Pure Data. Pour afficher une région d'un programme lorsque ce dernier est créé en tant qu'objet, il suffit de cocher l'option *Graph-On-Parent*, qui se trouve dans les propriétés du programme, et de spécifier la position et la taille de la région. La fenêtre de propriétés est accessible via un clic-droit dans une section vide (libre d'objets) de la fenêtre *patcher*. Avec ce mode d'affichage, seuls les objets graphiques sont visibles, les connections et les boîtes d'objets restent cachées.



FIGURE 10.5 – Object **sequenceur** avec interface de contrôle.

## 10.2 Améliorations possibles

Ce programme donne les bases nécessaires à la conception d'un séquenceur MIDI polyphonique très performant. Plusieurs améliorations, à tenter en guise d'exercices, peuvent y être apportées :

- Sortir le **notein** de l'objet et donner une entrée afin de pouvoir enregistrer du data en provenance du programme principal. Le système pourrait être en mesure d'enregistrer du data différent d'une note MIDI, c'est-à-dire que le module de lecture devrait pouvoir séparer les données autrement qu'en groupe de deux valeurs (hauteur - vitesse).
- Les contrôles graphiques pourraient être manipulés depuis le programme principal, via des entrées, afin d'automatiser les opérations.
- Le séquenceur pourrait accepter une horloge maîtresse, par exemple un compte généré dans le programme principal, afin de synchroniser plusieurs instances de l'objet sur un tempo global. Les départs et arrêts de l'enregistrement, ainsi que de la lecture, seraient soumis à cette horloge, en spécifiant, par exemple, que ces commandes doivent être activées sur le premier temps de la mesure suivante.
- L'ajout d'un contrôle pour basculer le mode d'enregistrement de "unique" (message *store*) à "multiple" (message *merge*). La remise à zéro de la collection, avant le départ de l'enregistrement, devra être gérée en conséquence !



# Chapitre 11

## La distorsion non-linéaire

Un effet de distorsion est obtenu lorsqu'une forme d'onde est déformée, soit à l'aide d'une fonction de transfert, soit par des opérations mathématiques, de telle sorte que des nouvelles composantes apparaissent dans le spectre du signal. On dit que la distorsion est un effet non-linéaire puisque le gain du signal en entrée influence le spectre du signal de sortie. Ceci permet de simuler des comportements naturels des instruments acoustiques, c'est-à-dire que plus on joue fort, plus le signal émis est riche. Un signal plus fort utilisera une plus grande zone de la fonction de transfert et la déformation du signal d'entrée sera par conséquent plus drastique.

Il y a deux méthodes couramment utilisées pour appliquer une distorsion sur un signal, chacune avec ses forces et ses faiblesses. La première consiste à dessiner une fonction de transfert dans une table et d'effectuer une lecture de table bipolaire avec le signal à transformer comme tête de lecture. La seconde simule la fonction de transfert en appliquant, de manière continue, des opérations mathématiques sur les échantillons successifs du signal à modifier.

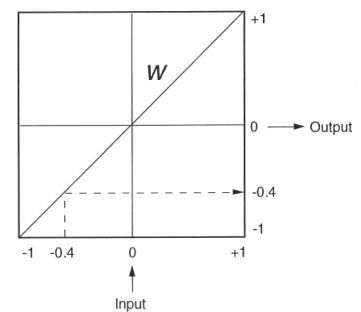
### 11.1 Fonction de transfert

On peut comprendre l'effet de la distorsion en considérant la **fonction de transfert** comme un **miroir déformant**. Ce miroir (la fonction de transfert) serait incliné à un angle donné, le sujet reflété (le signal d'entrée) se trouverait en-dessous du miroir et l'observateur (le signal de sortie) se trouverait sur le côté.

La figure ci-contre montre une fonction de transfert linéaire. Une telle fonction reflète précisément à sa sortie le signal qui est envoyée à son entrée.

L'angle du miroir détermine si l'image est reflétée de façon agrandie ou de façon diminuée, à la manière d'un amplificateur, dont le bouton de volume en contrôlerait l'inclinaison.

Dans le domaine numérique, la fonction de transfert  $W$  réside dans une table. Une valeur  $x$  du signal d'entrée est "localisée" dans la table et la valeur de sortie devient  $W(x)$ , le contenu de la table  $W$  à la position  $x$ .



### 11.1.1 Génération d'une fonction de transfert

En Pure Data, on peut générer une fonction de transfert en enregistrant dans un *array* les valeurs de sortie, entre -1 et 1, pour un signal d'entrée dont la valeur zéro correspondra au centre de la table. L'exemple suivant illustre la création d'une fonction de transfert dont la forme est contrôlée par le deuxième paramètre d'une fonction mathématique **arc-tangente**. L'utilisation de  $\text{atan2}(x, y)$ , plutôt que  $\text{atan}(x/y)$ , permet de spécifier indépendamment le numérateur et le dénominateur de la division  $x/y$  et ainsi d'éviter les erreurs de division par zéro.

Un compteur donne à tour de rôle chacune des positions possibles de l'*array*, mises en mémoire dans l'entrée de droite de l'objet **tabwrite**, et ces valeurs, ramenées entre -1 et 1, sont utilisées pour "balayer" une fonction **arc-tangente**. Le résultat de l'opération, donné en entrée de gauche de l'objet **tabwrite**, sera la valeur enregistrée à la position reçue à droite. Afin d'éviter tout débordement d'amplitude en sortie, la table est ensuite normalisée pour restreindre les valeurs entre -1 et 1.

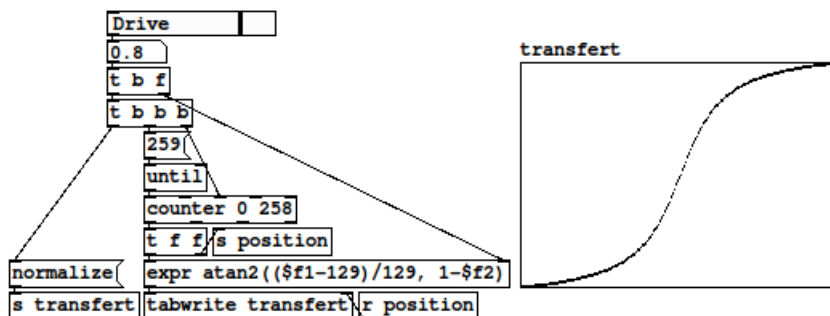


FIGURE 11.1 – Génération d'une fonction de transfert sauvegardée dans un *array*.

La clé de la fonction de transfert réside dans l'objet **expr** où sont calculées les valeurs à mettre en mémoire. Plusieurs variantes peuvent être apportées à cette formule, soit par l'utilisation de fonctions mathématiques différentes, soit par un contrôle plus détaillé des paramètres. L'exemple suivant ajoute au contrôle de la pente la possibilité de modifier la symétrie de la fonction.

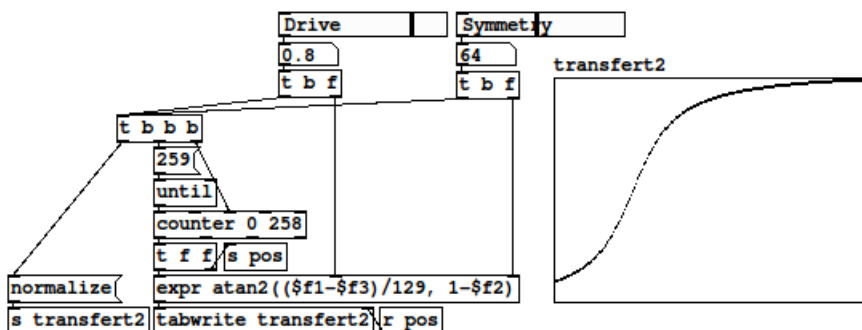


FIGURE 11.2 – Contrôle sur la pente ainsi que sur la symétrie de la fonction de transfert.

### 11.1.2 Lecture d'une table bipolaire (*table lookup*)

La lecture d'une table bipolaire (*table lookup*) consiste à utiliser une source sonore comme pointeur de lecture dans un *array*. La valeur d'amplitude de la source donne la position dans la table où piger la valeur de sortie. Le module suivant présente une onde sinusoïdale comme source d'une fonction de transfert appliquée avec l'objet **tabread4~**. Comme la position de lecture doit être spécifiée en échantillons, le signal est d'abord multiplié par 128 pour produire une onde bipolaire oscillant entre -128 et 128, puis est additionné à 129 afin de fournir une position de lecture entre 1 et 257.

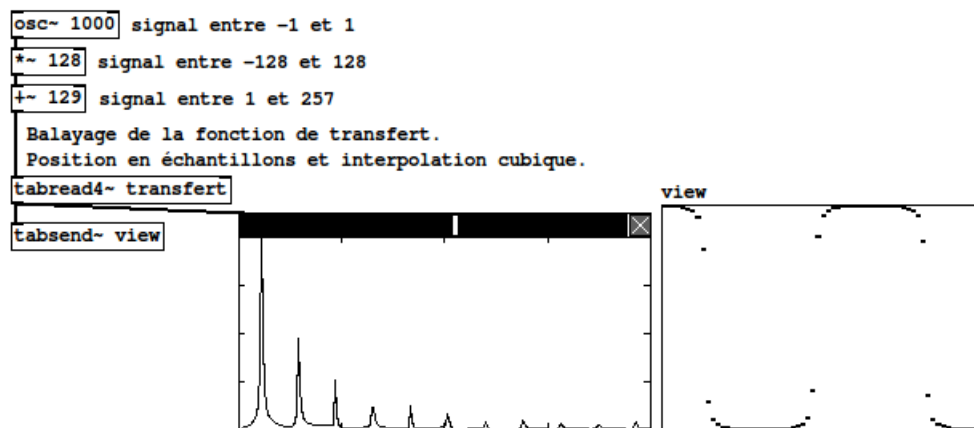


FIGURE 11.3 – Lecture de la fonction de transfert.

L'avantage de cette technique est sa faible consommation en temps de calcul. Il suffit d'une simple lecture de table pour appliquer une distorsion sur un signal. Le principal inconvénient est son manque de flexibilité. Si l'on veut modifier la fonction de transfert de façon dynamique, il faut réécrire tout le contenu de l'*array*. Les changements ainsi apportés à la fonction de transfert peuvent causer des discontinuités dans le signal de sortie, se traduisant par des clics dans le signal audio. Un moyen simple et efficace, quoique légèrement plus coûteux en temps de calcul, pour palier au manque de flexibilité de la fonction de transfert, consiste à appliquer, en continu, une fonction mathématique sur le signal à modifier.

## 11.2 Les fonctions mathématiques continues

Afin de permettre les variations dynamiques de la fonction de transfert, le système doit permettre la modification au taux audio des paramètres de la formule utilisée. L'application systématique d'une formule mathématique sur tous les échantillons d'un signal est plus demandant pour le processeur puisque l'opération est répétée sans cesse sur toute la durée de la performance (plutôt de d'effectuer le calcul une seule fois au moment de générer la fonction de transfert). Par contre, cela permet de modifier les paramètres de la fonction (*atan2* par exemple) de manière continue, sans causer de discontinuités dans le signal de sortie. Voici comment s'écrirait le processus précédent avec une expression au taux audio (objet **expr~**).

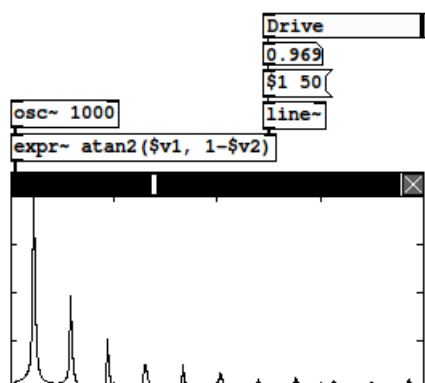


FIGURE 11.4 – Distorsion arc-tangente simple.

Différentes formules mathématiques peuvent être utilisées pour déformer un signal. La formule suivante retourne sensiblement le même résultat que la fonction **arc-tangente**, la différence se trouvant dans la vitesse à laquelle la distorsion est appliquée, en rapport avec la course du paramètre "drive" :

$x$  = signal d'entrée  $[-1 \leq x \leq 1]$

$d$  = drive  $[0 \leq d < 1]$

$k = 2 * d / (1 - d)$

$y = (1 + k) * x / (1 + k * \text{abs}(x))$

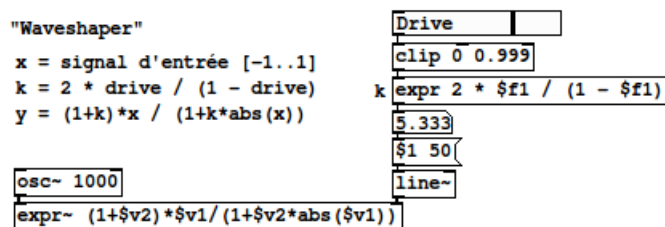


FIGURE 11.5 – Opérations mathématiques simples sur le signal audio.

Le principe de base de la distorsion étant d'ajouter des composantes à des fréquences multiples des fréquences déjà présentes dans le signal, il est très courant que des composantes soient créées au-delà de la fréquence de *Nyquist*, causant ainsi du repliement. Les filtres sont impuissants pour éliminer le repliement dans un signal puisqu'au moment du filtrage, les fréquences trop élevées sont déjà rabattues sous la fréquence de Nyquist et font parties intégrantes du nouveau son. Elles ne peuvent être distinguées des autres composantes présentes dans le signal d'origine. La seule façon de prévenir le repliement consiste à repousser la fréquence de Nyquist le plus loin possible, en sur-échantillonnant le signal, avant d'appliquer la distorsion.

### 11.2.1 Le sur-échantillonnage

Pour une fréquence d'échantillonnage de 192 kHz, la fréquence de Nyquist est repoussée à 96 kHz, c'est-à-dire qu'il est possible de créer de nouvelles composantes jusqu'à concurrence de 96 kHz avant de générer du repliement. Le principe d'une distorsion numérique consiste donc à préalablement sur-échantillonner le signal, ensuite lui appliquer la transformation (sans crainte du repliement), puis de filtrer le résultat avec un filtre passe-bas ajusté à la fréquence de Nyquist "réelle" (en fonction de la fréquence d'échantillonnage de l'environnement) pour éliminer les composantes qui pourraient se replier lors du sous-échantillonnage qui ramène le signal à la bonne fréquence d'échantillonnage.

Pure Data offre la possibilité de fonctionner avec plusieurs fréquences d'échantillonnage dans une même programme, en assignant un facteur de ré-échantillonnage à différents sous-patches (objet **pd**) via l'objet **block~**. Cet objet prend trois arguments permettant d'ajuster respectivement le *block size*, la quantité de chevauchement (*overlaps*) et le facteur de ré-échantillonnage du sous-patch dans lequel l'objet se trouve. Ces valeurs n'affectent que le sous-patch en question, laissant le reste du programme intact.

Voici un exemple qui reprend la distorsion **arc-tangente**, cette fois-ci à l'intérieur d'un sous-patch avec sur-échantillonnage par un facteur de 8. La distorsion est donc appliquée à une fréquence d'échantillonnage de 352,8 kHz (fréquence de Nyquist = 176,4 kHz), puis le signal est filtré pour ne conserver que les fréquences d'intérêts. À noter que le filtrage doit avoir lieu à l'intérieur du sous-patch, à la fréquence d'échantillonnage élevée, afin d'éviter le repliement que pourrait causer le sous-échantillonnage effectué par l'objet **outlet~**.

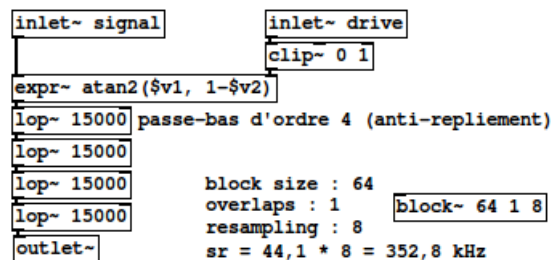


FIGURE 11.6 – Sous-patch avec sur-échantillonnage par un facteur de 8.

## 11.3 Le secret d'une bonne distorsion

Dans un module de distorsion, la fonction de transfert établit la nature de la série harmonique créée pour chacune des fréquences présentes dans le spectre de la source. En général, ce processus génère trop de composantes et parfois, il génère aussi des composantes indésirables. Le développement d'une distorsion originale implique un travail de réflexion et d'exploration en ce qui concerne le traitement de la source avant distorsion, le filtrage du signal de sortie ainsi que le mixage entre la source originale et le signal traité.

Le module suivant présente un module de distorsion contenant tous les éléments ci-haut mentionnés. Dans l'ordre, un filtre passe-haut pour retirer de la distorsion les fréquences très

graves, un filtre passe-bande pour isoler la région du spectre à traiter, la distorsion avec sur-échantillonnage, un filtre passe-bas pour atténuer les hautes fréquences et enfin, un contrôle de mixage entre la source et le traitement.

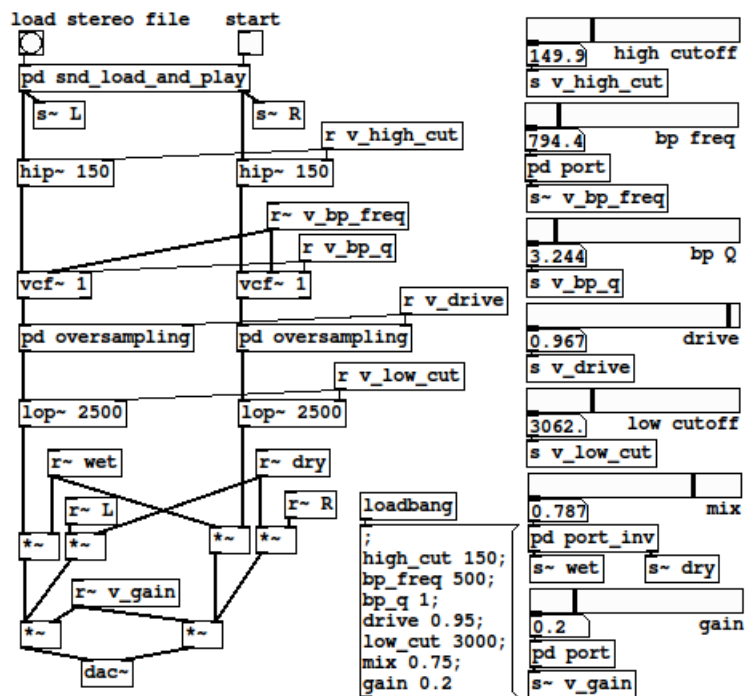


FIGURE 11.7 – Distorsion avec pré/post filtrages, sur-échantillonnage et mixage.