# Le langage JSFX et la librairie CookDSP

Olivier Bélanger

1er février 2019

# Table des matières

# Chapitre 1

# Introduction au langage JSFX

## 1.1 Qu'est-ce que le langage JSFX

JSFX est un langage de programmation audio propre au logiciel sonore *Reaper*. Il permet la création rapide de *plugins* audio, et/ou MIDI (*Musical Instrument Digital Interface*), grâce à sa syntaxe simple et à une intégration directe et efficace à l'architecture du logiciel. Un *plugin* JSFX consiste simplement en un fichier texte, compilé à la volée lorsque chargé dans l'environnement, contenant les directives nécessaires pour l'exécution du processus sonore désiré. Les effets JSFX distribués avec le logiciel, sous format texte, peuvent être consultés, modifiés ou utilisés comme point de départ pour la création de traitements originaux. Nous introduirons dans ce chapitre les fonctionnalités de base du langage, la structure d'un fichier JSFX ainsi que les règles de syntaxe essentielles à l'écriture d'effets audio. Les éléments de langage seront illustré par des exemples de *plugins* qui implémentent des opérations simples sur les échantillons audio.

Reaper effectue, sur demande, un balayage automatique des *plugins* contenus dans le dossier prévu à cet effet (les *plugins* contenus dans les sous-dossiers sont aussi détectés) :

- Windows XP :

  ```
  C:\Documents and Settings\<username>\Application Data\REAPER\Effects
  ```

- Windows Vista/7/8/10 :

  ```
  C:\Users\<username>\AppData\Roaming\REAPER\Effects
  ```

- MacOS :

  ```
  /Users/<username>/Library/Application Support/REAPER/Effects
  ```

- Linux (Reaper natif) :

  ```
  /home/<username>/.config/REAPER/Effects
  ```

- Linux (Reaper sous Wine) :

  ```
  /home/<username>/.wine/drive_c/users/<username>/Application Data/REAPER/Effects
  ```

Plusieurs banques de *plugins* créées par des tiers peuvent être téléchargées à des fins d'utilisation et/ou de consultation à l'adresse suivante :

http://stash.reaper.fm/tag/JS-Effects

Pour consulter la documentation complète du langage JSFX, veuillez vous référer au manuel en ligne, à l'adresse :

http://www.reaper.fm/sdk/js/js.php

## 1.2 Structure d'un fichier JSFX

Un fichier JSFX est un simple fichier texte, sans extension, contenant un certain nombre de lignes de description, suivi par une ou plusieurs sections de code. Les descriptions servent principalement à gérer l'allure de l'interface graphique tandis que les sections de code spécifient les actions à poser (modification des échantillons) en fonction des manipulations effectuées par l'utilisateur.

### 1.2.1 Les descriptions

Une description est une commande, sous la forme d'un mot-clé, suivi du symbole deux-points ( : ), puis des arguments spécifiques à la commande. Deux éléments de description seront détaillés dans cette section, *desc* et *sliderX*.

#### desc

La commande *desc* permet de spécifier le nom (ou une courte description) du *plugin* qui sera affiché dans l'interface graphique. Cette commande ne peut être utilisée plus d'une fois par *plugin* et vient généralement en tout début de script, ce qui facilite l'identification du processus lorsqu'ouvert en mode texte. La description d'un effet vide pourrait ressembler à ceci :

```
/*
Empty plugin, a plugin that does nothing!
*/

desc:Empty Plugin
```

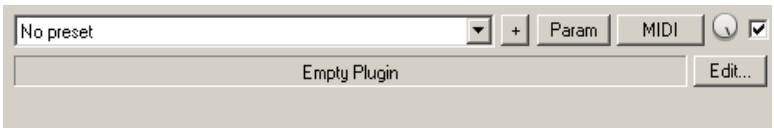



FIGURE 1.1: Affichage du titre d'un *plugin*.

Comme l'illustre la figure ci-dessus, le texte « Empty Plugin » sera affiché dans la région réservée à la description du *plugin*. Les trois premières lignes de cet exemple représentent un commentaire, c'est-à-dire un section de texte qui sera ignorée à la compilation du code. Les commentaires sont particulièrement utiles afin de donner des indications supplémentaires sur les

différentes parties du script. Plus de détails sur les commentaires dans la section sur les éléments de langage.

### slider1, slider2, ..., slider64

La commande *sliderX*, où *X* prend une valeur entre 1 et 64, permet de spécifier les attributs des potentiomètres qui seront affichés dans l'interface graphique. Les potentiomètres doivent obligatoirement être déclarés dans l'ordre, c'est-à-dire en commençant par *slider1*, puis *slider2*, jusqu'à *sliderN*, *N* étant le nombre total de potentiomètres utilisés. Le nom de la commande (*slider1, slider2, etc.*) sera le symbole par lequel la valeur courante du potentiomètre pourra être récupérée dans les autres sections du code. Les différents éléments de la déclaration d'un potentiomètre sont :

```
slider1:5<0,10,1>description
```

Dans l'ordre, on retrouve :

- **slider1**, le « nom » du potentiomètre, tel qu'il sera utilisé dans les différentes sections du code, suivi des deux-points.
- **5**, la valeur initiale du potentiomètre.
- **<0,10,1>**, entre les symboles « plus petit » et « plus grand », sont spécifiés la valeurs minimum, la valeur maximum ainsi que le « pas » (la distance entre deux valeurs successives) du potentiomètre.
- **description**, le texte qui suit le symbole « plus grand » sera affiché dans l'interface en tant qu'indication sur le rôle du potentiomètre.

La déclaration suivante donnera, dans l'ordre, un potentiomètre d'entier (pas de 1) entre 0 et 10, un potentiomètre de nombre décimaux (le pas est omis) entre 0 et 1 et un potentiomètre donnant une fréquence entre 20 et 20000 par saut de 200.

```
slider1:0<0,10,1>Compteur d'entiers
slider2:0.5<0,1>Panoramisation
slider3:1000<20,20000,200>Frequence en Hz
```


FIGURE 1.2: Affichage des potentiomètres.

#### Menu déroulant

Une définition particulière du potentiomètre permet de transformer celui-ci en un menu déroulant. Il suffit de donner une liste de symboles, entre accolades, tout juste après la valeur du « pas » (qui, dans ce contexte, doit obligatoirement être 1). La ligne suivante générerait un menu offrant les choix « bleu », « blanc » et « rouge », avec « bleu » sélectionné à l'initialisation (la valeur par défaut étant 0).

```
slider1:0<0,2,1{bleu,blanc,rouge}>Choisir une couleur...
```

La déclaration suivante donnera, dans l'ordre, un menu contenant deux items (*On* et *Off*), un menu offrant la sélection parmi les huit premières puissances de 2 et enfin, un menu permettant de choisir parmi différents types d'enveloppe.

```
slider1:0<0,1,1{Off, On}>Commutateur On/Off
slider2:4<0,8,1{1,2,4,8,16,32,64,128,256}>Puissances de 2
slider3:0<0,4,1{Rectangle,Triangle,Cloche,Trapeze,Parabole}>Enveloppe
```


FIGURE 1.3: Affichage des menus déroulants.

### 1.2.2 Les sections de code

Les sections de code représentent des actions à poser, c'est-à-dire des manipulations à effectuer sur les signaux, en différents moments du processus. Par exemple, certaines opérations doivent être effectuées à l'initialisation du *plugin*, tandis que d'autres doivent être appliquées à chacun des échantillons, ou encore seulement quand un potentiomètre change de valeurs, etc. Une section est déclarée à l'aide d'une ligne unique, contenant le nom de la section, précédée du symbole arobas ( @ ). Toutes les lignes suivantes appartiennent à cette section, jusqu'à la déclaration de la prochaine section (ou jusqu'à la fin du fichier). Seulement deux de ces sections seront exposées ici :

- **@sample**, où sont traités chacun des échantillons audio.
- **@slider**, exécutée lorsqu'un potentiomètre est manipulé.

#### @sample

Le code dans cette section est exécuté pour chacun des échantillons du signal sonore. Il permet de lire et/ou d'écrire dans les variables audio **spl0**, **spl1**, ..., **spl63** (il y a 64 canaux disponibles). Les variables **spl0** et **spl1** contiennent respectivement l'échantillon audio courant des canaux gauche et droite (en stéréo).

Un *plugin* qui applique un traitement quelconque sur le signal sonore doit donc nécessairement contenir une section *@sample*, puisque c'est dans cette section que les échantillons seront manipulés. Le *plugin* suivant est un amplificateur « linéaire », où les échantillons audio sont directement multipliés, à tour de rôle, par la valeur courante de l'unique potentiomètre de l'interface. On le dit « linéaire », en opposition à « logarithmique » (plus proche de la perception humaine du volume), parce que la distance entre chacune des valeurs d'amplitude successives est la même. De fait, les potentiomètres de Reaper sont toujours linéaires, ce qui représente une certaine limitation, notamment en ce qui concerne la manipulation des amplitudes et des fréquences. Des techniques de manipulation des données seront exposées pour palier à ce problème.

```
/*
Amplification lineaire d'un signal stereo.
*/

desc: Gain

// Gain est une valeur entre 0 et 2
slider1:1<0,2>Gain

@sample
// Echantillon de gauche * la valeur du potentiometre.
spl0 = spl0 * slider1;
// Echantillon de droite * la valeur du potentiometre.
spl1 = spl1 * slider1;
```

La section **@sample** de ce code sera exécutée à chaque période d'échantillonnage ($T_s$), où les variables **spl0** et **spl1** représentent respectivement la valeur courante du signal dans le canal de gauche et dans le canal de droite. La ligne :

```
spl0 = spl0 * slider1;
```

indique donc que l'échantillon du canal gauche vaut maintenant ($spl0 =$) sa valeur courante pondéré par la valeur du potentiomètre ($spl0 * slider1$). Le symbole point-virgule (;) sert à séparer les déclarations (par exemple, une opération mathématique dont le résultat est assigné à une variable). À des fins de clarté, nous nous limiterons autant que possible à une déclaration par ligne.

Pour transformer un amplificateur linéaire en un amplificateur logarithmique (où la valeur d'amplitude est spécifiée en décibels), il suffit d'appliquer une opération de conversion sur la valeur du potentiomètre.

### Petits rappels

Conversion des valeurs linéaires ($x$) en décibels :

$$db = 20 * \log_{10}(x)$$

Conversion des décibels en valeurs linéaires ($x$) :

$$x = 10^{db/20}$$

Cette opération pourrait très bien être effectuée à l'intérieur de la section **@sample**, mais cela représenterait un gaspillage considérable de CPU. Afin d'économiser des cycles de calcul, la conversion décibels vers linéaire (pour effectuer la multiplication avec les échantillons) sera exécutée seulement lorsque le potentiomètre changera de valeur. Une section de code est appelée chaque fois qu'un élément d'interface est manipulé, c'est la section **@slider**.

#### @slider

Le code contenu dans la section **@slider** est exécuté chaque fois qu'un potentiomètre change de valeur. Cela permet, par exemple, de mettre à jour certaines variables, pour les réutiliser telles quelles dans les autres sections, sans avoir à effectuer sans cesse des calculs redondants.

Le programme suivant implémente un amplificateur « logarithmique », où le potentiomètre est défini en décibels, de -60 à +18 dB. Cette valeur est convertit en amplitude linéaire dans la section **@slider**, et le résultat gardé en mémoire dans une variable *amp*. Nul besoin de déclarer les variables (symbole à gauche du signe =), le langage s'en charge. La multiplication des échantillons audio par la variable *amp* (amplitude linéaire) est effectuée, comme dans le programme précédent, dans la section **@sample**.

```
/* Amplification logarithmique d'un signal stereo. */

desc: DB Gain

slider1:0<-60,18>Gain en dB

@slider
amp = 10 ^ (slider1 / 20);

@sample
// Echantillon de gauche * amp (amplitude lineaire).
spl0 = spl0 * amp;
// Echantillon de droite * amp (amplitude lineaire).
spl1 = spl1 * amp;
```

Notez l'utilisation des parenthèses pour gérer la priorité des opérations. Comme la plupart des environnements mathématiques, l'ordre des priorités avec le langage JSFX peut être mémorisé à l'aide de l'expression mnémonique **PEMDAS** : **P**arenthèses, **E**xposant, **M**ultiplication et **D**ivision, et enfin **A**ddition et **S**oustraction. Sans les parenthèses, la formule se lirait comme suit : $10^{slider1}/20$, et non $10^{slider1/20}$.

## 1.3 Les instructions conditionnelles

Voici un *plugin* un peu plus élaboré, faisant usage des instructions conditionnelles afin de transformer une rampe en une onde triangulaire. L'objectif est d'appliquer une modulation, à l'aide d'un oscillateur à basse fréquence (LFO - *Low Frequency Oscillator*), sur l'amplitude d'un signal stéréo. Afin de générer une onde triangulaire, une rampe entre 0 et 1, à la fréquence désirée, est d'abord construite. Pour obtenir une rampe, une valeur d'incrément (calculée en prenant le rapport de la fréquence désirée sur la fréquence d'échantillonnage, $f_d/f_s$) est ajoutée à une variable (*ramp* dans le code) à tous les instants d'échantillonnage. Ceci génère une valeur

qui augmente sans cesse. Lorsqu'elle atteint le seuil de 1, elle est remise à 0 et recommence sa course. L'extrait suivant isole la génération de la rampe :

```
@slider
inc = slider1 / srate; // increment pour la rampe (f / fs).

@sample
ramp = ramp + inc; // On avance la rampe d'un increment.
ramp >= 1 ? ramp = 0; // On remet la rampe a 0 lorsqu'elle atteint 1.
```

À partir d'une rampe entre 0 et 1, pour obtenir une onde triangulaire, il suffit de comparer le signal avec un seuil de 0.5. Si le signal est sous le seuil, l'onde triangulaire est égale à la rampe, c'est la portion ascendante de l'onde. Si le signal est au-dessus du seuil, on soustraie cette portion de rampe ($0.5 \rightarrow 1.0$) à la valeur 1. On obtient ainsi la portion descendante de l'onde triangulaire, un signal qui descend de 0.5 ($1-0.5$) à 0 ($1-1$). Cette opération est réalisée à l'aide de la ligne suivante :

```
// Onde triangulaire a partir d'une rampe
ramp < 0.5 ? amp = ramp : amp = 1 - ramp;
```

Voici le code complet du *plugin* :

```
/*
Construction d'une onde triangulaire, a partir d'une
rampe, modulant l'amplitude d'un signal stereo.
*/

desc: LFO Modulated Gain

slider1:1<0.1,20>Speed (Hz)

@slider
// increment pour la rampe (f / fs).
inc = slider1 / srate;

@sample
// Tant que la rampe est < 0.5, amp vaut la rampe,
// c'est la trajectoire ascendante de l'onde triangulaire.
// Sinon (ramp est > 0.5), amp vaut 1.0 - la rampe,
// c'est la trajectoire descendante de l'onde triangulaire.
ramp < 0.5 ? amp = ramp : amp = 1 - ramp;
// Modulation du signal stereo par l'onde triangulaire.
spl0 = spl0 * amp;
spl1 = spl1 * amp;
// On avance la rampe d'un increment.
ramp = ramp + inc;
// On remet la rampe a 0 lorsqu'elle atteint 1.
ramp >= 1 ? ramp = 0;
```

## 1.4 Les éléments de langage

Tout langage de programmation, aussi simple soit-il, implique un certain nombre de règles de syntaxe. Voici un condensé des notions essentielles concernant le langage JSFX élaborées jusqu'ici. Des notions plus avancées seront introduites en temps et lieu.

### 1.4.1 Syntaxe

Éléments de syntaxe exposés jusqu'ici :

- Un **commentaire** est une ligne débutant par une double barre oblique ( // ) ou tout le texte placé entre les symboles /* et */. Les commentaires sont ignorés à la compilation mais s'avèrent extrêmement utiles pour indiquer à quoi servent les différentes parties du code.
- Une **description** est spécifiée à l'aide d'un mot-clé (*desc, sliderX*) suivi du symbole deux-points ( : ). Vient ensuite, selon le type de description, les arguments de la fonction.
- Le symbole **point-virgule** ( ; ) sert à séparer les déclarations (opérations mathématiques, assignation à une variable, etc.), en particulier à la fin d'une ligne.
- Les **section de code** sont déclarées à l'aide d'une ligne unique commençant par le symbole arobas ( @ ), suivi du nom de la section. Les lignes suivantes appartiennent à cette section jusqu'à ce qu'une nouvelle section soit déclarée.
- Les **parenthèses** servent à gérer l'ordre des priorités dans une expression mathématique.
- Les **variables** n'ont pas besoin d'être déclarées et elles sont « globale » par défaut (c'est-à-dire qu'elles sont accessibles dans toutes les sections de code).
- Les **expressions conditionnelles** utilisent la syntaxe :

  condition ? code si vrai : code si faux

  Une condition est d'abord énoncée, si la réponse est « vrai », le code suivant le point d'interrogation est exécuté tandis que si la réponse est « faux », c'est le code suivant les deux-points qui est exécuté.

### 1.4.2 Variables

Quelques noms de variables réservées :

- **slider1, slider2, ... slider64**, noms réservés des 64 potentiomètres possibles. On déclare un potentiomètre en faisant suivre le nom du symbole deux-points ( : ). Dans les sections de code, on récupère la valeur d'un potentiomètre via son nom de variable.
- **spl0, spl1, ..., spl63**, noms réservés des 64 canaux audio possibles. En stéréo, le signal de gauche est accessible via la variable **spl0** tandis que le signal de droite est accessible via la variable **spl1**.
- **srate**, variable donnant accès à la fréquence d'échantillonnage courante de Reaper. Cette variable peut être utilisée chaque fois qu'un calcul doit tenir en compte la fréquence d'échantillonnage.

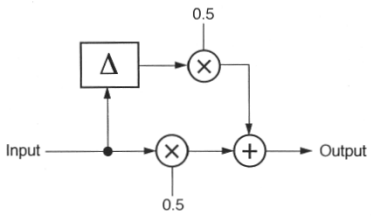
# Chapitre 2

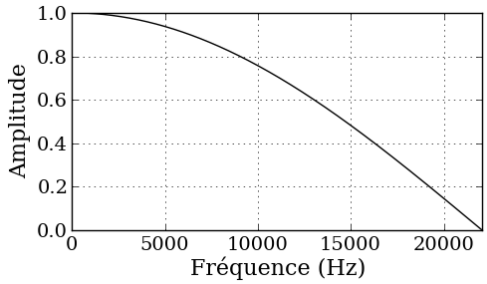
# Les filtres numériques

## 2.1 Exemples de filtres FIR

### 2.1.1 Un filtre passe-bas FIR de premier ordre

Pour réaliser un filtre passe-bas FIR simple (qui atténue les fréquences élevées du signal), il suffit d'effectuer la moyenne des valeurs entre l'échantillon présent et l'échantillon précédent, tel qu'illustré sur le schéma-bloc ci-dessous,




et tel qu'exprimé par l'équation :

$$y[n] = 0.5x[n] + 0.5x[n-1]$$

Intuitivement, on peut comprendre l'effet passe-bas de cette opération puisqu'en effectuant la moyenne, on atténue les variations brusques du signal, ce qui résulte en un **lissage** du signal d'entrée. Ce filtre est un exemple de filtre à moyenne mobile (*moving average filter*).

La réponse en fréquence de ce filtre est représentée ci-dessus (figure de droite). Le gain en fonction de la fréquence est donné par :

$$A(f) = \cos(\pi f / f_s)$$

La fréquence de coupure vaut $f_s/4$ (le quart de la fréquence d'échantillonnage). On peut se rendre compte que la simplicité du filtre (son ordre faible) se paie par une bande de transition importante. Le plugin suivant implémente un filtre passe-bas FIR de premier ordre.

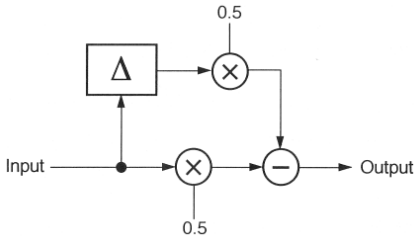
```
// Simple Mono FIR Lowpass Filter.
desc: Mono FIR Lowpass Filter

@init
sig0 = 0.0;
x0 = 0.0;

@sample
// Effectue la moyenne entre 2 echantillons successifs
sig0 = (spl0 + x0) * 0.5;
// Garde en memoire l'echantillon courant (pour le prochain tour)
x0 = spl0;
// Envoie le resultat du filtrage dans la variable de sortie
spl0 = sig0;
```

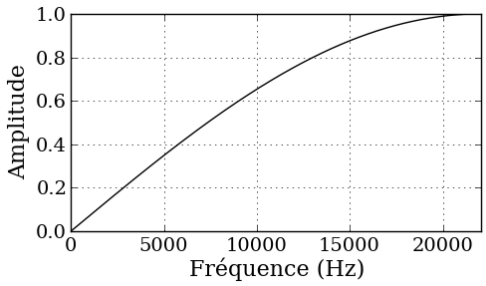
### 2.1.2 Un filtre passe-haut FIR de premier ordre

La structure d'un filtre passe-haut FIR élémentaire est très similaire à celle du filtre passe-bas FIR vu précédemment, mais au lieu d'additionner deux échantillons consécutifs, on les soustrait.

$$y[n] = 0.5x[n] - 0.5x[n-1]$$

En effectuant la différence entre deux échantillons successifs, on atténue le signal là où il varie lentement (basses fréquences) et on l'accentue là où il varie rapidement (hautes fréquences).




Le gain en fonction de la fréquence est donné par :

$$A(f) = \sin(\pi f / f_s)$$

```
// Simple Mono FIR Highpass Filter.
desc: Mono FIR Highpass Filter

@init
sig0 = 0.0;
x0 = 0.0;

@sample
// Soustraie l'echantillon precedent a l'echantillon courant
sig0 = (spl0 - x0) * 0.5;
// Garde en memoire l'echantillon courant (pour le prochain tour)
x0 = spl0;
// Envoie le resultat du filtrage dans la variable de sortie
spl0 = sig0;
```

### 2.1.3 Un filtre passe-bande FIR de second ordre

L'équation d'un filtre passe-bande simple non-récursif de second ordre est :

$$y[n] = 0.5x[n] - 0.5x[n-2]$$

Ce filtre est du second ordre puisqu'il utilise un délai de deux périodes d'échantillonnage au maximum (pour obtenir $x[n-2]$). Le gain en fonction de la fréquence est ici donné par :

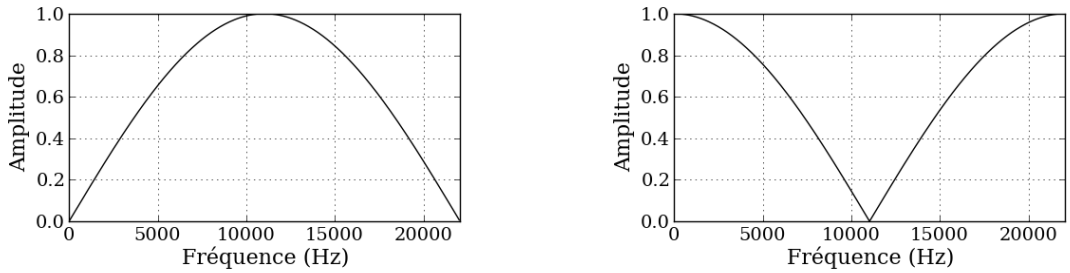
$$A(f) = \sin(2\pi f/f_s)$$

ce qui correspond à un filtre passe-bande de fréquence centrale $f_s/4$.

```
// Simple Mono FIR Bandpass Filter.
desc: Mono FIR Bandpass Filter

@init
sig0 = 0.0;
x0 = x1 = 0.0;

@sample
// Soustraie le delai de 2 echantillons a l'echantillon courant
sig0 = (spl0 - x1) * 0.5;
// Garde en memoire les 2 derniers echantillons d'entree
x1 = x0;
x0 = spl0;
// Envoie le resultat du filtrage dans la variable de sortie
spl0 = sig0;
```

Si, au lieu de les soustraire, on additionne les échantillons $x[n]$ et $x[n-2]$, alors on obtient un **réjecteur de bande** de fréquence centrale $f_s/4$ :

$$y[n] = 0.5x[n] + 0.5x[n-2]$$




La figure ci-dessus représente la réponse en fréquence d'un filtre passe-bande FIR d'ordre 2 (à gauche) et celle d'un filtre réjecteur de bande FIR d'ordre 2 (à droite).

## 2.2 Exemples de filtres IIR

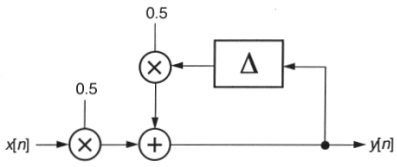
### 2.2.1 Un filtre passe-bas IIR de premier ordre

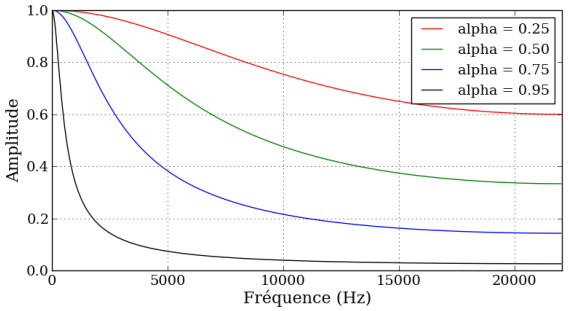
Un filtre passe-bas récursif de premier ordre effectue simplement une pondération entre l'échantillon d'entrée présent et l'échantillon de sortie précédent, comme illustré par le schéma-bloc ci-dessous et l'équation suivante :

$$y[n] = (1 - \alpha)x[n] + \alpha y[n - 1]$$

où $\alpha$ prend une valeur entre 0 et 1.

La réponse en fréquence (sur la figure de droite) a une forme différente que celle du filtre passe-bas non-récursif. Ce filtre, généralement rencontré sous l'appellation *Tone* dans les langages de programmation musicale, est dit ETA (*exponential time average*).




Une variation du coefficient de réinjection ($\alpha$) permettra de contrôler la fréquence de coupure du filtre. Lorsque $\alpha$ croit, le signal de sortie est réinjecté dans le système avec plus d'amplitude que l'échantillon d'entrée courant. Le lissage du signal est alors plus prononcé et la fréquence de coupure baisse.

Pour déterminer le coefficient de réinjection correspondant à une fréquence de coupure donnée en Hertz ($f$), on utilisera la formule suivante :

$$\alpha = e^{-2\pi f / f_s}$$

Le code suivant implémente un filtre passe-bas IIR stéréo :

```
// Tone, un filtre passe-bas IIR de premier ordre.
desc: One-Pole Lowpass Filter

slider1:800<100,10000,1>frequency (Hz)

@init
y0 = y1 = 0;

@slider
a = exp(-2 * $pi * slider1 / srate);

@sample
y0 = (1 - a) * spl0 + a * y0;
y1 = (1 - a) * spl1 + a * y1;
spl0 = y0;
spl1 = y1;
```

### 2.2.2 Un filtre passe-haut IIR de premier ordre

La façon la plus simple d'obtenir un filtre passe-haut IIR de premier ordre consiste à calculer le filtre passe-bas correspondant, puis à le soustraire au signal original, éliminant ainsi les fréquences graves du signal original. Le code suivant illustre les sections **@slider** et **@sample** du filtre passe-haut IIR de premier ordre :

```
@slider
a = exp(-2 * $pi * slider1 / srate);

@sample
y0 = (1 - a) * spl0 + a * y0;
y1 = (1 - a) * spl1 + a * y1;
spl0 = spl0 - y0;
spl1 = spl1 - y1;
```

### 2.2.3 Un filtre passe-bande IIR de second ordre

Un filtre populaire en traitement de signal est le filtre dit « résonateur » (communément appelée *Reson*). C'est un filtre passe-bande du second ordre permettant d'ajuster indépendamment la fréquence centrale et la largeur de bande (via le facteur de qualité $Q$). Le diagramme suivant illustre le fonctionnement de ce filtre :

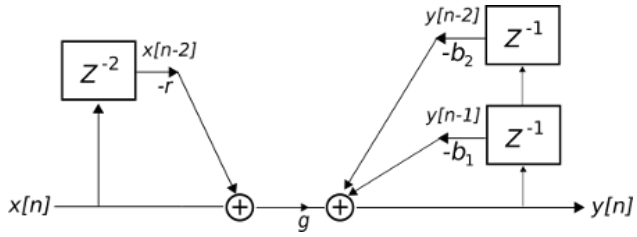



FIGURE 2.1: Schéma-bloc d'un résonnateur de second ordre.

Comme on le constate sur ce schéma, ce filtre utilise un mélange d'échantillons d'entrée ($x[n]$ et $x[n-2]$) ET d'échantillons de sortie ($y[n-1]$ et $y[n-2]$) pour permettre des courbes de réponse en fréquence plus complexes. Voici ce filtre en équation :

$$y[n] = a_1 x[n] - a_1 x[n-2] - b_1 y[n-1] - b_2 y[n-2]$$

On obtient les coefficients du filtre ($a_1$, $b_1$ et $b_2$), en fonction de la fréquence centrale et du facteur de qualité, à l'aide des équations suivantes :

$$\begin{aligned} bw &= f_c/Q \\ b2 &= exp(-2\pi/f_s \times bw) \\ b1 &= (-4.0 \times b2)/(1.0 + b2) \times cos(2\pi f_c/f_s) \\ a1 &= 1.0 - sqrt(b2) \end{aligned}$$

Le code suivant implémente une version mono (à des fins d'épuration du code) du filtre passe-bande illustré ci-dessus :

```
/*
Reson, un filtre passe-bande IIR du second ordre (version mono).
*/

desc: Reson: Second-Order Bandpass Filter

slider1:800<100,5000,1>Frequency (Hz)
slider2:1<0.7,50>Filter Q

@init
// Initialisation a zero des quatre espaces memoires servant
// a conserver les deux derniers echantillons d'entree
x1 = x2 = 0;
// et les deux derniers echantillons de sortie
y1 = y2 = 0;

@slider
// Calcul de la largeur de bande (frequence centrale / Q)
bw = slider1 / slider2;

// Calcul des coefficients (en suivant la recette)
b2 = exp(-2 * $pi / srate * bw);
b1 = (-4.0 * b2) / (1.0 + b2) * cos(2 * $pi * slider1 / srate);
a1 = 1.0 - sqrt(b2);

@sample
// Equation du filtre (spl0 = echantillon present, x2 = 2e delai
// du signal d'entree, y1 = 1er delai du signal de sortie, y2 = 2e
// delai du signal de sortie (x1 n'est pas utilise dans l'equation)
tmp0 = (a1 * spl0) - (a1 * x2) - (b1 * y1) - (b2 * y2);

// On prepare les memoires pour le prochain calcul qui aura lieu
// a la periode d'echantillonnage suivante.
// Le 1er delai du signal d'entree devient le 2e delai d'entree
x2 = x1;
// L'echantillon courant devient le 1er delai du signal d'entree
x1 = spl0;
// Le 1er delai du signal de sortie devient le 2e delai de sortie
y2 = y1;
// La sortie du filtre devient le 1er delai du signal de sortie ...
y1 = tmp0;
// ... et est envoyee a la sortie audio
spl0 = tmp0;
```

## 2.3 Éléments de langage JSFX

Deux nouveaux éléments de langage sont introduits dans ce chapitre, une nouvelle section de code (**@init**) et la librairie de fonctions mathématiques.

### 2.3.1 Section de code « @init »

Le code inscrit dans la section **@init** est exécuté au chargement du *plugin*, aux changements de fréquence d'échantillonnage ainsi qu'au départ de la lecture de la séquence (activation du bouton « Play »). On y place généralement du code servant à initialiser des variables qui seront utilisées avant même de recevoir une quelconque valeur (par exemple, les variables qui gardent en mémoire les échantillons délayés utilisés dans les équations de filtrage). Le code suivant reprend l'initialisation nécessaire au bon fonctionnement du *plugin* de filtre passe-bande résonant vu précédemment.

```
@init
// Assigne une valeur initiale aux variables utilisees par le filtre
x1 = x2 = 0;
y1 = y2 = 0;

@sample
// Equation du filtre (utilise les variables x1, x2, y1 et y2)
tmp0 = (a1 * spl0) - (a1 * x2) - (b1 * y1) - (b2 * y2);
// Et seulement ensuite, ces variables recoivent une valeur
x2 = x1;
x1 = spl0;
y2 = y1;
y1 = tmp0;
spl0 = tmp0;
```

### 2.3.2 Les fonctions mathématiques disponibles

Le langage JSFX donne accès aux fonctions mathématiques usuelles en programmation. Quelques-unes de ces fonctions on été introduites dans ce chapitre :

- **exp(x)**, retourne la constante $e$ élevée à la puissance $x$ ($e^x$). Plus efficace que la fonction **pow** ou le symbole « exposant ».
- **sqrt(x)**, retourne la racine carrée de $x$.
- **cos(x)**, calcule et retourne le cosinus de $x$. $x$ doit être spécifié en radians (entre 0 et $2\pi$).

Plusieurs autres fonctions sont disponibles. La liste complète peut être consultée en ligne à l'adresse suivante (cliquez sur le lien *Simple math functions*) :

http://www.reaper.fm/sdk/js/js.php

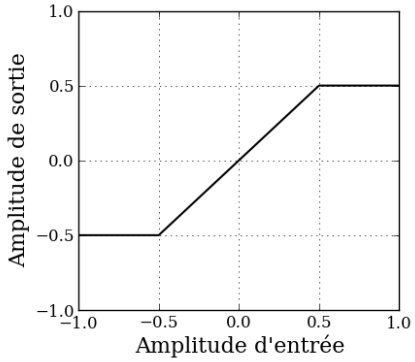
# Chapitre 3

# La distorsion

## 3.1 La distorsion en équation

### 3.1.1 Le *clipping*

La distorsion la plus simple que l'on peut imaginer consiste à « couper » le signal de façon radicale si celui-ci dépasse un seuil (à la façon d'un signal numérique dépassant les limites permises par le système). Toute portion de l'onde (positive et négative) qui dépasse le seuil se voit attribuée la valeur même du seuil. La fonction de transfert de cet algorithme, obtenue à l'aide de la formule suivante, est illustré sur la figure ci-dessous :



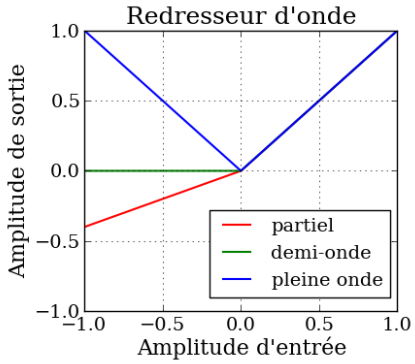

$$y[n] = max(min(x[n],\ seuil),\ -seuil)$$

Où $min(x[n],\ seuil)$ retourne systématiquement la valeur la plus petite entre l'échantillon courant et le seuil fixé, c'est-à-dire que le signal prend la valeur du seuil pour chaque échantillon plus élevé. La fonction $max$ effectue le même travail mais à l'inverse, c'est-à-dire qu'elle retourne le signal le plus élevé entre la sortie de la fonction $min$ et le seuil négatif.

```
// Distorsion Mono "Hard Clipping".

desc: Hard Clipping

slider1:0.9<0,1>Thresh

@sample
// min(x, y) -> retourne la valeur la plus petite entre x et y
// max(x, y) -> retourne la valeur la plus grande entre x et y
spl0 = max(min(spl0, slider1), -slider1);
```

### 3.1.2 Le « redresseur » d'onde

Le redressement de l'onde (ou la « rectification positive ») consiste simplement à inverser la polarité des échantillons négatifs dans le signal. Cela a pour effet de réduire de moitié la périodicité du signal, et par conséquent de doubler la fondamentale perçue (saut à l'octave). Pour obtenir différents plateaux de redressement, on effectue une pondération entre le signal original et le signal redressé « pleine onde », tel qu'indiqué par l'équation suivante :



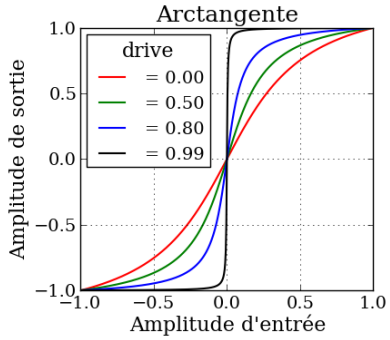

$$y[n] = (1 - \alpha) * x[n] + \alpha * abs(x[n])$$

Où $\alpha$ est la valeur de pondération entre 0 et 1. À 1, le signal est complètement redressé dans le positif.

```
// Redressement positif de l'onde.

desc: Positive Rectification

slider1:0<0,1>Amount of Rectification

@sample
// Interpolation entre la valeur reelle et la valeur absolue.
spl0 = (1 - slider1) * spl0 + slider1 * abs(spl0);
```

### 3.1.3 La distorsion *arctangente*

Une des fonctions de transfert très répandue dans les modèles de distorsion numérique est la fonction « arctangente ». Elle est très facile à manipuler et permet de créer des courbes allant d'une pente très « molle » à une pente très « dure ». La figure ci-dessous illustre quatre fonctions de transfert obtenues en modifiant le paramètre $\alpha$ de l'équation suivante :



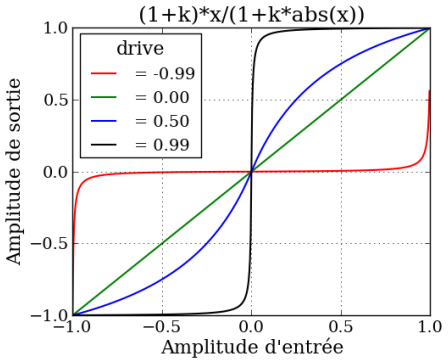

$$y[n] = atan2(x[n],\ (1 - \alpha) * 0.3999)$$

Où $\alpha$, une valeur entre 0 et 1, contrôle la raideur de la pente. À 1, le signal est distordu par une quasi-onde carrée.

```
// La fonction de transfert arctangente

desc: Arctangent Distortion

slider1:0.7<0,1>Drive

@slider
// Calcul du parametre "drive"
drive = (1.0 - slider1) * 0.3999;

@sample
// atan2(b,a) -> calcul l'arctangente de b/a
spl0 = atan2(spl0, drive) * 0.5;
```

### 3.1.4 Un *waveshaper* particulier

Un signal sonore peut être distordu de différentes façons à l'aide d'une panoplie d'équations mathématiques. Une formule particulièrement efficace et versatile (provenant des archives du site internet http://musicdsp.com/) s'exprime comme suit :




$$y[n] = (1 + k) \times x[n]/(1 + k \times abs(x[n]))$$

Où $k$, pour un paramètre *drive* entre -0.999 et 0.999, vaut :

$$(2.0 \times drive)/(1.0 - drive)$$

Les fonctions de transfert illustrées sur la figure ci-dessus montrent la vaste étendue de possibilités de l'équation. Pour un paramètre *drive* négatif, la fonction atténue fortement tous les échantillons sauf pour les extrêmes. À 0, le signal passe sans transformation tandis que pour un *drive* positif, la fonction ressemble à une *arctangente*.

```
// Waveshaper maison

desc: Waveshaper

slider1:0<0,1>Drive

@slider
// Restreint le drive sous la valeur 1
drive = slider1 * 0.999;
// Calcul la valeur de "k"
k = (2 * drive) / (1 - drive);

@sample
// Assure que le signal reste dans les limites -1 a 1.
spl0 = max(min(spl0, 1), -1);
// Applique la formule de waveshaping
spl0 = (1 + k) * spl0 / (1 + k * abs(spl0));
```

## 3.2 Pré-filtrage, post-filtrage et mixage

La fonction de transfert ne constitue qu'une portion d'un bon module de distorsion. Certaines opérations (filtrage, contrôle de gain, etc.) effectuées avant et/ou après la distorsion proprement dite peuvent améliorer grandement le résultat final du traitement.

Tout d'abord, il faut savoir que la distorsion peut ajouter énormément de composantes harmoniques, généralement concentrées dans les hautes fréquences, au signal original, provoquant ainsi une sonorité très dure. Il conviendra d'ajouter une filtre passe-bas après la distorsion afin d'atténuer la trop grande concentration d'énergie dans le registre élevé.

Une autre problématique de la distorsion vient de l'amplitude du signal d'entrée. Comme mentionné plus haut, un signal faible ne lira que la portion centrale de la fonction de transfert, sans atteindre les extrémités. Un contrôle de volume en entrée, pour permettre à des signaux d'amplitude différentes d'utiliser pleinement la fonction de transfert, ainsi qu'en sortie, afin de rétablir la dynamique du signal à un niveau raisonnable, permettront de régler les problèmes inhérents aux variations de dynamique.

Le processus de la distorsion a pour effet de générer de nouvelles composantes à des fréquences qui sont des multiples entiers de toutes les composantes présentes dans le signal original. Si le signal est riche en basses fréquences, le résultat sera une grande concentration d'énergie dans le registre médium-basse (autour de 500 Hz). Les différentes harmoniques générées entrent alors en conflit les unes avec les autres, créant une sonorité souvent déplaisante (dissonante). La solution à ce problème consiste à atténuer le contenu en basse fréquence du signal, avant d'appliquer la distorsion, à l'aide d'un filtre passe-haut (de fréquence entre 80 et 200 Hz). De même, la distorsion des composantes de fréquences élevées aura tendance a créer rapidement du repliement de spectre. Un filtre passe-bas appliqué au signal d'entrée, avant distorsion, pourra atténuer les problèmes de repliement.

Finalement, après avoir éliminé certaines composantes en entrée de la distorsion, on voudra probablement effectuer un mixage entre le signal original et le signal distordu. Voici un schéma-bloc qui résume la construction d'un module (parmi tant d'autres) de distorsion.

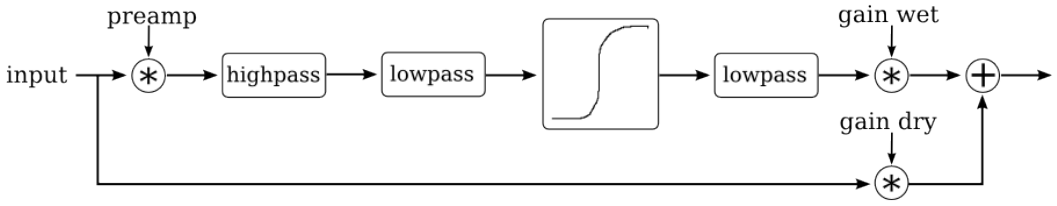



FIGURE 3.1: Schéma-bloc d'un modèle de distorsion numérique.

# Chapitre 4

# Les effets basés sur les délais

## 4.1 Éléments de langage JSFX

### 4.1.1 La gestion de l'espace mémoire

Comme nous l'avons vu précédemment, un effet basé sur les délais nécessite un espace mémoire où garder les échantillons qui seront potentiellement rejoués plus tard. Chaque *plugin* JSFX se voit automatiquement attribuer un espace mémoire équivalent à environ 8 millions d'échantillons (c'est-à-dire un peu plus de 3 minutes de signal audio monophonique pour une fréquence d'échantillonnage de 44,1 kHz). Cet espace mémoire peut être utilisé (on non) selon les besoins du processus audio développé dans le *plugin*.

Dans le langage JSFX, les crochets ([ ]) permettent d'indexer la mémoire allouée au *plugin*. On accède aux différents emplacements dans la mémoire en utilisant un décalage fixe :

```
1024[512];
```

ou à l'aide d'une variable :

```
bufferLeft = 1024;
bufferLeft[512];
```

La somme de la valeur à gauche des crochets et de la valeur à l'intérieur des crochets est utilisée pour indiquer l'emplacement (« index ») dans la mémoire du *plugin*. On peut tout aussi bien « écrire » une valeur en mémoire et la « lire » par la suite, par exemple :

```
/* Ecriture */
bufferLeft[127] = spl0;

/* Lecture */
spl0 = bufferLeft[127];
```

### Enregistrer des échantillons audio en mémoire

Pour enregistrer des échantillons audio en mémoire, on doit d'abord déclarer l'index de départ et le nombre d'emplacements nécessaires pour le bon fonctionnement du processus. Ces déclarations ont généralement lieu dans la section de code **@init**. Ensuite, on doit incrémenter un compteur à chaque instant d'échantillonnage et s'assurer que le compteur reste bien à l'intérieur des limites de la mémoire déclarée ($0 <= count < max$). La gestion du compte se fait dans la section **@sample**.

Voici une illustration de l'évolution du pointeur d'écriture dans une mémoire de 16 emplacements. Notez le retour à la position zéro lorsque le dix-septième échantillon ($n = 16$) entre dans la mémoire, il vient écraser l'échantillon préalablement présent (0) à cet endroit.

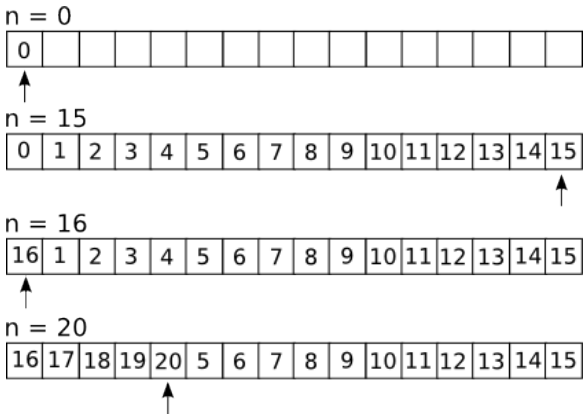



FIGURE 4.1: Évolution du pointeur d'écriture dans un espace-mémoire ($n$ est le temps en échantillons).

Le programme suivant conserve en mémoire les 1024 derniers échantillons en entrée du *plugin* :

```
/* Conserve les 1024 derniers echantillons en memoire. */

desc: Memoire Simple

@init
/* Utilise les premiers emplacements de la memoire du plugin. */
memory = 0;
/* Nombre d'emplacements maximum. */
maxlen = 1024;
/* Initialisation du compteur. */
count = 0;

@sample
/* Ecrit l'echantillon d'entree dans la memoire. */
memory[count] = spl0;
/* Incremente le compteur. */
count += 1;
/* Revient a zero lorsque la limite est atteinte. */
count == maxlen ? count = 0;
```

## 4.2 Les délais fixes en *plugins*

Voici quelques exemples concrets de *plugins* implémentant des effets basés sur l'utilisation des délais fixes.

### 4.2.1 Délai monophonique simple

Le premier *plugin* met en place un simple délai monophonique dont le temps de délai est contrôlé à l'aide d'une durée donnée en échantillons.

L'élément important de ce code est la relation entre la position du pointeur d'écriture dans la ligne de délai et la position du pointeur de lecture. Le pointeur d'écriture représente le temps courant, c'est-à-dire que les échantillons d'entrée sont enregistrés les uns à la suite des autres dans l'espace-mémoire, en avançant une position à la fois, jusqu'à ce que la limite maximum soit atteinte. À ce moment, le pointeur revient à zéro et les échantillons les plus anciens sont remplacés par les nouveaux. Le pointeur de lecture, quant à lui, est tout simplement $X$ échantillons derrière le pointeur d'écriture (position d'écriture - temps de délai). Si le pointeur de lecture est plus petit que zéro (ce qui arrivera forcément), on le bascule dans le positif en lui additionnant la longueur maximale de la mémoire. Dans l'image ci-dessous, la flèche en pointillé indique l'emplacement qui sera lu dans la mémoire pour un délai de 4 échantillons :

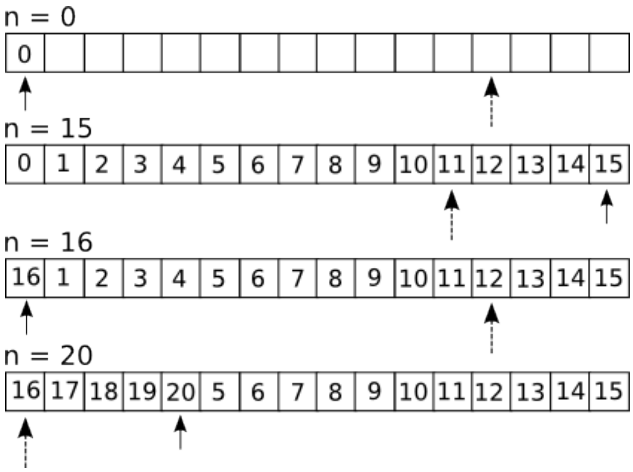



FIGURE 4.2: Évolution du pointeur de lecture (en pointillé) dans un espace-mémoire, pour un délai de 4 échantillons.

Le *plugin* à la page suivante délaye un signal monophonique d'une durée pouvant atteindre 4410 échantillons, soit 100 millisecondes pour une fréquence d'échantillonnage de 44100.

```
/*
Ligne de delai mono, avec controle
du temps de delai en echantillons.
*/

desc: Simple Delay Line

/* 4410/44100 = 0.1 seconde de delai maximum */
slider1:25<1,4409,1>delay time (samples)

@init
/* Delai maximum */
maxlen = 4410;
/* Position d'ecriture dans la memoire */
curPos = 0;
/* Indice de depart dans la memoire allouee au plugin. */
bufL = 0;

@sample
/* Calcul de la position de lecture
   (pos d'ecriture - temps de delai) */
pos = curPos - slider1;
/* Si la position est negative, on bascule dans le positif. */
pos < 0 ? pos += maxlen;

/* Lecture de l'echantillon a l'indice "pos" dans la memoire. */
readL = bufL[pos];
/* On ecrit l'echantillon d'entree a l'indice courant. */
bufL[curPos] = spl0;

/* Echantillon de sortie =
   echantillon d'entree + echantillon lu dans la memoire. */
spl0 = (spl0 + readL) * 0.5;

/* On incremente la position courante de 1. */
curPos += 1;
/* Si la position courante est egale au delai
   maximum, on rammene a zero. */
curpos == maxlen ? curpos = 0;
```

### 4.2.2 Délai stéréo simple

Pour retarder un signal stéréo, le *plugin* a besoin de deux espaces-mémoire, un pour conserver le signal du canal de gauche et un autre pour conserver le signal du canal de droite. On déclare donc une seconde variable ($bufR$ dans l'exemple ci-dessous) qui pointera sur un indice de départ au moins une longueur maximum de délai plus loin que l'indice de départ du canal de gauche ($bufL$).

```
@init
/* Delai maximum */
maxlen = 4410;
/* Indice de depart, pour le canal de gauche,
   dans la memoire allouee au plugin. */
bufL = 0;
/* Indice de depart, pour le canal de droite,
   dans la memoire allouee au plugin (a la suite
   de l'espace reserve pour le canal de gauche + 1). */
bufR = bufL + maxlen + 1;
```

Voici le *plugin* complet permettant de retarder un signal stéréo d'un temps donné en échantillons :

```
/*
Ligne de delai stereo simple, avec controle
du temps de delai en echantillons.
*/

desc: Simple Stereo Delay Line

/* 4410/44100 = 0.1 seconde de delai maximum */
slider1:25<1,4409,1>delay time (samples)

@init
/* Delai maximum */
maxlen = 4410;
/* Position d'ecriture dans la memoire */
curPos = 0;
/* Indice de depart, pour le canal de gauche,
   dans la memoire allouee au plugin. */
bufL = 0;
/* Indice de depart, pour le canal de droite,
   dans la memoire allouee au plugin (a la suite
   de l'espace reserve pour le canal de gauche + 1). */
bufR = bufL + maxlen + 1;

@sample
/* Calcul de la position de lecture
   (pos d'ecriture - temps de delai) */
pos = curPos - slider1;
/* Si la position est negative, on bascule dans le positif. */
pos < 0 ? pos += maxlen;

/* Lecture de l'echantillon a l'indice "pos" dans les memoires. */
readL = bufL[pos];
readR = bufR[pos];

/* On ecrit l'echantillon d'entree a l'indice
   courant dans chacun des espaces-memoire. */
bufL[curPos] = spl0;
bufR[curPos] = spl1;

/* Echantillon de sortie =
   echantillon d'entree + lecture dans la memoire (delai). */
spl0 = (spl0 + readL) * 0.5;
spl1 = (spl1 + readR) * 0.5;

/* On incremente la position courante de 1. */
curPos += 1;
/* Si egale au delai maximum, on rammene a zero. */
curpos == maxlen ? curpos = 0;
```

### 4.2.3 Délai avec interpolation linéaire à la lecture

Il est généralement souhaitable de spécifier le temps de délai en secondes plutôt qu'en échantillons (indépendance vis-à-vis de la fréquence d'échantillonnage, échelle temporelle naturelle, conversion à partir d'une fréquence en Hz, etc.). Lorsque le temps de délai est spécifié en secondes, la conversion vers des indices d'espace-mémoire, qui eux, sont nécessairement des positions en nombres entiers, pose problème. En effet, il est assez rare qu'une durée donnée

en secondes (ou en millisecondes) corresponde à un nombre exact (entier) d'échantillons. Pour effectuer la conversion des millisecondes vers des échantillons, on appliquera la formule suivante :

$$samples = ms/1000 * f_s$$

où $f_s$ est la fréquence d'échantillonnage.

Voyons ce qui arrive si on tente d'obtenir la correspondance en échantillons d'une durée de 75 ms ($f_s$ vaut 44100) :

$$75/1000 * 44100 = 3307.5$$

Une durée qui correspond à un demi-échantillon ! Il n'existe pas de demi-emplacement dans un espace-mémoire.

Alors quelle valeur devrait-on utiliser ? On tronque et on récupère la valeur à la position 3307 ? Ou bien on arrondit et on se retrouve probablement avec la valeur à la position 3308 ?

Aucune de ces deux solutions n'est réellement satisfaisante. Par exemple, pour faire résonner un délai à une fréquence $X$, où la durée du délai sera très courte, une différence de 0.5 échantillons dans le calcul de la période occasionnera des problèmes de justesse de la note. Il convient donc d'utiliser un algorithme d'**interpolation**, qui permettra une lecture plus précise du signal gardé en mémoire.

L'algorithme d'interpolation le plus simple est l'**interpolation linéaire**, où on effectue une pondération directe entre deux valeurs successives, en fonction de la fraction d'échantillon désirée. La figure ci-dessous illustre ce processus.

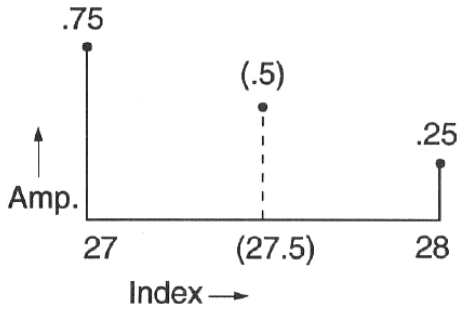



FIGURE 4.3: Interpolation linéaire entre deux points existants. La position 27.5 retourne 0.5 * la valeur à la position 27 + 0.5 * la valeur à la position 28.

En programmation, on effectue une interpolation linéaire à l'aide de la formule suivante :

$$y = x + (x1 - x) * frac$$

où $y$ est la valeur de sortie, $x$ est la valeur au temps de délai entier inférieur, $x1$ est la valeur au temps de délai entier supérieur et $frac$ est la partie fractionnelle du temps de délai en échantillons.

Voici un exemple de *plugin* implémentant un délai avec interpolation linéaire :

```
/*
Ligne de delai stereo avec interpolation lineaire,
et controle du temps de delai en millisecondes.
*/

desc: One Second Delay Line

slider1:250<0.1,1000>delay time (ms)

@init
/* "srate" echantillons = 1 seconde de delai maximum. */
maxlen = srate + 1;
curPos = bufL = 0;
bufR = bufL + maxlen + 1;

@slider
/* Conversion ms -> echantillons (ms / 1000 * srate). */
delay = slider1 * 0.001 * srate;

@sample
/* Calcul de la position de lecture. */
pos = curPos - delay;
pos < 0 ? pos += maxlen;

/* Interpolation lineaire: y = x + (x1 - x) * frac */
/* Partie entiere du temps de delai en echantillon. */
ipart = floor(pos);
/* Partie fractionnelle du temps de delai en echantillon. */
fpart = pos - ipart;

/* Lecture dans les espaces-memoire, avec interpolation. */
readL = bufL[ipart] + (bufL[ipart+1] - bufL[ipart]) * fpart;
readR = bufR[ipart] + (bufR[ipart+1] - bufR[ipart]) * fpart;

bufL[curPos] = spl0;
bufR[curPos] = spl1;

/* Afin d'eviter un click lorsque le pointeur de lecture arrive
a la toute fin de l'espace memoire, on copie le premier echantillon
au dernier emplacement de la memoire (seule l'interpolation utilise
cette valeur). */
bufL[maxlen] = bufL[0];
bufR[maxlen] = bufR[0];

spl0 = (spl0 + readL) * 0.5;
spl1 = (spl1 + readR) * 0.5;

curPos += 1;
curPos == maxlen ? curpos = 0;
```

### 4.2.4 Délai récursif

Un délai récursif est créé lorsque le signal de sortie du délai est réinjecté, en addition au signal original, en son entrée. Le *plugin* suivant offre un contrôle sur la quantité de signal réinjecté (*feedback*) ainsi qu'un contrôle de volume en sortie. Étudiez la section **@sample** pour comprendre la réinjection.

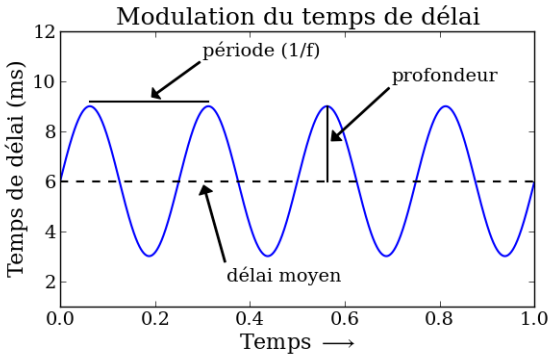
```
/*
Ligne de delai stereo avec interpolation lineaire,
feedback et controle du temps de delai en ms.
*/

desc: One Second Delay Line With Feedback

slider1:250<0,1000,1>delay time (ms)
slider2:0.5<0,1>feedback
slider3:-6<-60,12,1>gain (dB)

@init
maxlen = srate + 1;
curPos = bufL = 0;
bufR = bufL + maxlen + 1;

@slider
/* Conversion ms -> echantillons (ms / 1000 * srate). */
delay = slider1 * 0.001 * srate;
/* Feedback (amplitude du signal reinjecte). */
feed = slider2;
/* Conversion des dB en amplitude lineaire. */
gain = pow(10, slider3 * 0.05);

@sample
pos = curPos - delay;
pos < 0 ? pos += maxlen;
ipart = floor(pos); /* Partie entiere */
fpart = pos - ipart; /* Partie fractionnelle */
/* Lecture dans les espaces-memoire, avec interpolation. */
readL = bufL[ipart] + (bufL[ipart+1] - bufL[ipart]) * fpart;
readR = bufR[ipart] + (bufR[ipart+1] - bufR[ipart]) * fpart;
/* Ecrit l'echantillon d'entree + le signal delaye, multiplie
   par le feeback, a l'indice courant dans les memoires. */
bufL[curPos] = spl0 + readL * feed;
bufR[curPos] = spl1 + readR * feed;
/* Controle du volume. */
spl0 = readL * gain;
spl1 = readR * gain;
curPos += 1;
curPos == maxlen ? curpos = 0;
```

## 4.3 Les délais variables : effets de *flange*, *phasing* et *chorus*

On obtient les effets de délais variables par la **modulation** de l'emplacement du (des) pointeur(s) de lecture dans la mémoire-tampon, c'est à-dire en variant la durée du délai pendant sa lecture. Les effets les plus connus de cette technique sont le **flanging**, le **phasing**, le **chorus** et toute une gamme d'effets de **transposition**.

### 4.3.1 L'effet de *flange*

Le *flanging* numérique utilise une ligne de retard dont le délai peut varier dans le temps. Au lieu d'une pression manuelle sur le bord de la bobine, le temps de délai d'un *flanger* électronique est contrôlé par un oscillateur à basse fréquence (LFO) dont la forme d'onde est habituellement une **onde sinusoïdale** (comme l'illustre la figure ci-dessous) ou **triangulaire** et dont la fréquence se situe **entre 0,1 et 20 Hz**.




L'effet de *flange* pourrait être appelé « **effet de filtre en peigne balayant** », exprimant l'idée d'un filtrage en peigne dont les pics de résonance se déplacent le long de l'axe des fréquences.

Les paramètres de l'effet de *flange* sont :

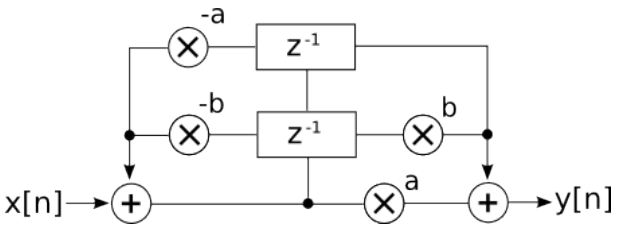
- le **temps de délai moyen**
- la **fréquence** du LFO qui contrôle la modulation du délai
- la **profondeur** ou **amplitude de la modulation** du délai

### 4.3.2 L'effet de *phasing*

Le *phasing* est un effet similaire à celui du *flanging* mais l'altération du timbre du son produite par le balayage du ou des filtres n'est habituellement pas aussi prononcée.

Dans le cas du *phasing*, un signal à spectre riche est envoyé au travers d'une série de filtres passe-tout (qui sont des filtres dont la réponse en fréquence est plate, c'est-à-dire des filtres qui n'atténuent aucune fréquence, mais qui affectent la **phase** du signal). Un oscillateur à basse fréquence peut être utilisé pour contrôler la quantité de déplacement de la phase (*phase shift*) introduit par chaque filtre passe-tout. Les sorties des filtres sont ensuite mélangées au signal original.

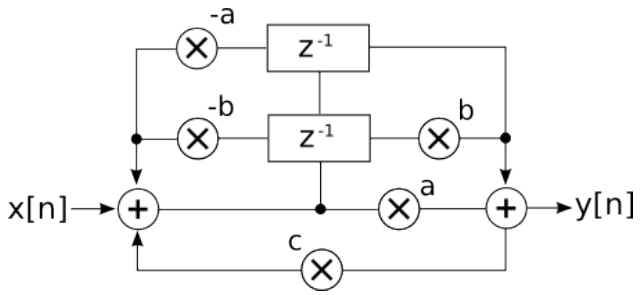
Le schéma-bloc ci-dessous illustre un filtre passe-tout IIR de second ordre. Ce filtre n'affecte pas le contenu en fréquence du signal d'entrée mais créé un déphasage de $180^o$ à la fréquence centrale. Les fréquences autour seront plus ou moins affectées en fonction de la largeur de bande du filtre. Si on additionne le signal de sortie du filtre au signal original, les composantes à ces fréquences s'annulent, créant ainsi un trou dans le spectre.



Pour une fréquence centrale ($f$) et une largeur de bande ($bw$) données en Hertz, on obtient les coefficients $a$ et $b$ de ce filtre à l'aide des formules suivantes :

$$r = e^{-\pi bw/f_s}$$
$$a = r^2$$
$$b = -2r \times cos(2\pi f/f_s)$$

Une variante de ce filtre, illustrée dans le schéma-bloc suivant, consiste à réinjecter la sortie du filtre en son entrée pour ajouter un pic de résonance (en plus du trou causé par le déphasage) dans le spectre de sortie. Ceci permet de créer des effets de *phasing* plus prononcés.




### 4.3.3 L'effet de *chorus*

Dans les systèmes numériques, l'effet chorus peut être réalisé en envoyant le son dans une ligne de retard *multi-tap*, où les temps de délai varient constamment sur une petite plage de valeurs. Ces variations causent des effets de désaccord et de doublage variant dans le temps. Ceci est équivalent à envoyer le signal au travers d'une banque de *flangers* en parallèle, bien que **les délais dans un *flanger* tendent à être plus courts que ceux utilisés pour un effet de *chorus***. En effet, les temps de délai pour l'effet de *chorus* se situent généralement entre 15 et 35 millisecondes. Les petites variations du délai sont contrôlées par un LFO autour de 3 Hz.

Ces techniques peuvent être enrichies en utilisant une rétroaction négative, comme pour le *flanging* (négative plutôt que positive, la rétroaction minimise les risques de résonance et de saturation du système).

### 4.3.4 Les effets de transposition

Lorsqu'on fait avancer ou reculer le pointeur de lecture d'une ligne de délais, on remarque que le son obtenu au point de lecture monte ou descend en hauteur selon qu'on raccourcit ou allonge le temps de délai. Cet effet est tributaire du fait que la lecture variable dans une mémoire-tampon est en fait un **re-échantillonnage** du signal d'entrée, au même titre qu'une transposition directe, puisque le taux d'échantillonnage se trouve accéléré ou ralenti selon la vitesse de variation du délai (illustré sur la figure ci-dessous). Il est à noter que **lorsque la vitesse de variation du délai est constante, la transposition obtenue est constante**.

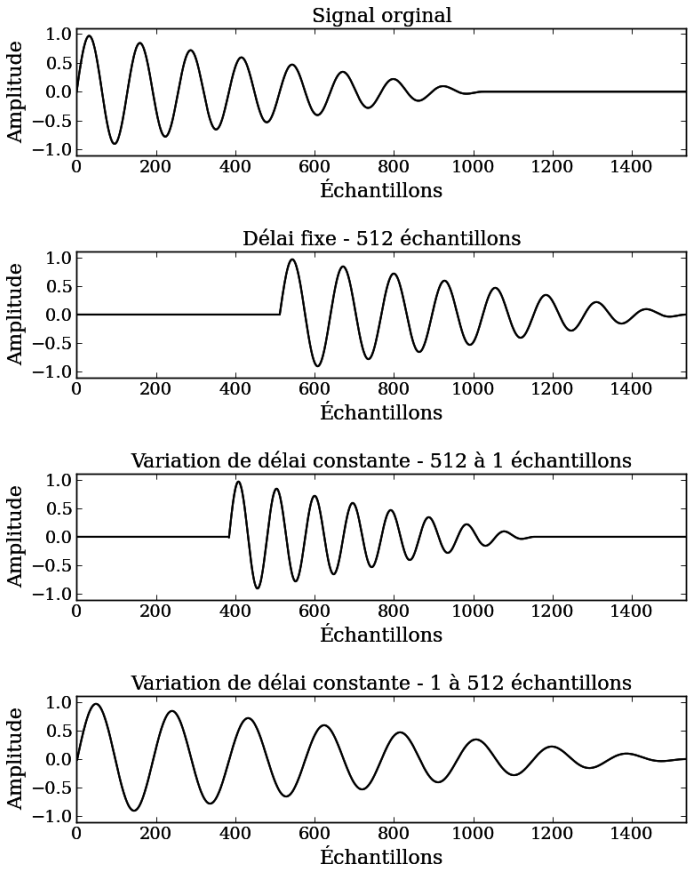



FIGURE 4.4: Effets de transposition causés par la variation du temps de délai. En haut, le signal original. Pour un délai fixe (en deuxième), la période, et par conséquent la fréquence, reste identique. En troisième, le temps de délai varie linéairement de 512 à 1 échantillons. Le signal ré-échantillonné présente une période plus courte que le signal original. L'inverse a lieu (en quatrième) lorsque le temps de délai augmente, la période du signal augmente aussi (sa fréquence diminue).

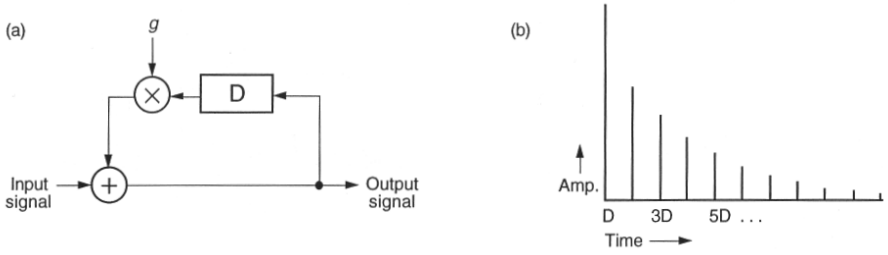
# Chapitre 5

# Les espaces artificiels et la spatialisation

### 5.0.1 La réverbération artificielle

Manfred Schroeder des Laboratoires Bell est le pionnier des réverbérateurs artificiels. Bien que certains réverbérateurs analogiques conçus selon les principes de Schroeder aient eu de belles carrières, ce n'est que depuis l'avènement de l'audionumérique que la réverbération artificielle offre des simulations très réalistes de tout type d'espaces.

Pour simuler les effets de la réverbération, les algorithmes de Schroeder proposent l'utilisation de **délais**, de **filtres passe-tout** et de **filtres en peigne**. Les délais sont utilisés pour reproduire les réflexions primaires de la salle et les filtres sont ensuite connectés en série et en parallèle pour créer un nuage très dense de réverbération.

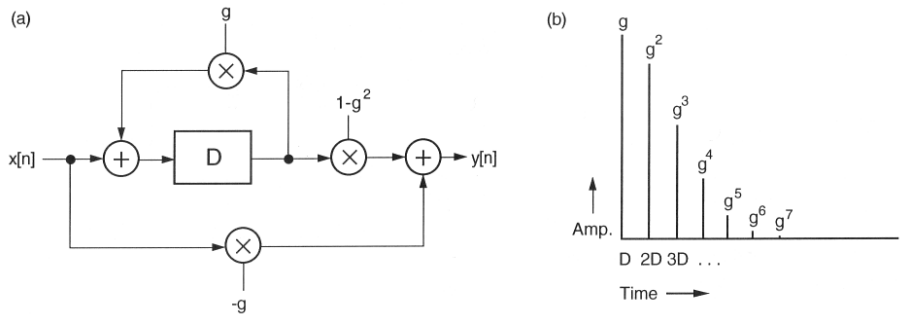



Les **filtres en peigne** de type IIR, tel que représenté ci-dessus, comprennent une boucle de réinjection et créent de multiples échos. La valeur du délai $D$ est généralement supérieure à 10 ms.

En contrôlant le niveau de réinjection du filtre en peigne, on peut ralentir le taux d'atténuation des échos et ainsi rallonger la durée de résonance du filtre. Le gain $g$ prend une valeur inférieure à 1. À chaque tour dans la boucle, une impulsion voit son amplitude multipliée par ce facteur de gain (d'atténuation, plus précisément). On obtient alors la série de puissances $g$, $g^2$, $g^3$, ...

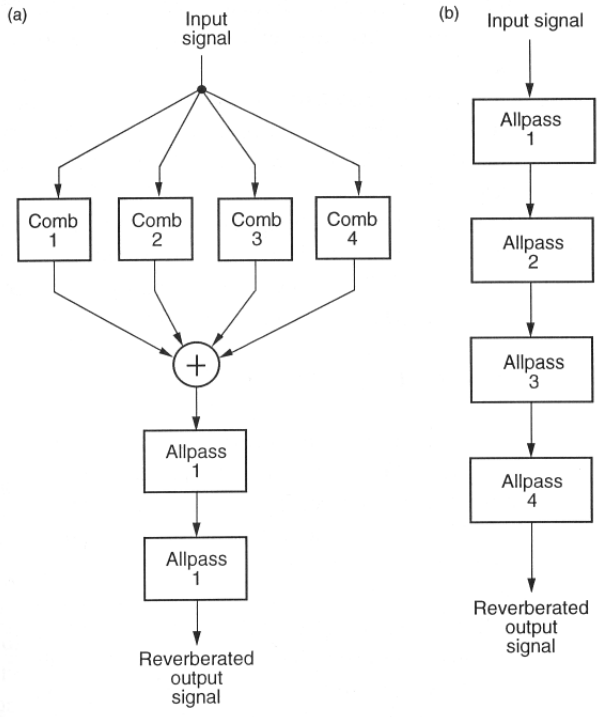
Les **filtres passe-tout**, tel que représenté ci-dessous, sont construits avec un filtre en peigne dont la sortie est mélangée à une copie du signal original inversé (en opposition de phase avec le

signal de sortie du filtre en peigne). Cette inversion de phase vient contrer les effets spectraux du filtre en peigne. Les filtres passe-tout transmettent donc toutes les fréquences des signaux stables mais ont un effet important sur la phase de ces signaux.




### 5.0.2 Les algorithmes de Schroeder

Schroeder propose deux modèles de réverbérateurs, illustrés sur les figures (a) et (b) ci-dessous, dont la sonorité est fonction des délais et des facteurs de réinjection. Il sera aussi essentiel de spécifier des temps de délais pour chacun des filtres, qui seront de préférence des nombres premiers, et ce afin d'éviter que les échos ne soient dédoublés et viennent donner un effet périodique à la réverbération.



Comme l'illustre la figure (a), une **unité de réverbération de base** est composée de :

— 4 filtres en peigne en parallèle, suivis de
— 2 unités de filtres passe-tout en série.

Les temps de boucle sont fixés pour une réponse en fréquence « naturelle ».

Pour construire une réverbération simple, La librairie *CookDSP* offre les deux objets suivants :

```
comb(delay, feed)
allpass(delay, feed)
```

Le signal d'entrée est réverbéré pendant un temps déterminé par le paramètre *feed* à l'aide d'un filtre dont la réponse en fréquence est :

— « colorée » avec le filtre en peigne (*comb*)
— « plate » avec le filtre passe-tout (*allpass*)

Ces filtres répètent l'entrée avec une densité d'écho déterminée par le temps de bouclage, en secondes. Le taux d'atténuation est indépendant et est déterminé par la quantité de signal ré-injecté dans le filtre. La sortie d'un filtre en peigne apparaît seulement après que le temps de délai soit écoulé. La sortie d'un filtre passe-tout apparaît immédiatement.

Le temps de bouclage détermine la densité d'échos de la réverbération, qui à son tour caractérise la couleur du filtre en peigne dont la réponse en fréquence contiendra $(delay \times sr/2)$ pics espacés régulièrement entre 0 et $sr/2$ (la fréquence de Nyquist). Le temps de bouclage peut être aussi grand que la mémoire disponible le permet.

Puisque la sortie de l'unité de réverbération de base commencera à apparaître après un délai correspondant à la durée du filtre en peigne le plus court, et souvent avec moins des 3/4 de la puissance originale, il est normal d'envoyer vers la sortie la source ET le signal réverbéré.

Un réverbérateur efficace pourra donc utiliser un ensemble de filtres en peigne et passe-tout afin de créer la densité d'échos nécessaire pour une simulation convaincante.

Réverbérateur de Schroeder (modèle *a* sur la figure) en plugin :

```
desc:Schroeder Reverb A

import cookdsp.jsfx-inc

slider1:3500<500,10000>Lowpass Cutoff
slider2:0.2<0,1>Balance Dry/Wet

@init
// Left ch. comb filters          Right ch. comb filters
c1L.comb(0.0297, 0.65);           c1R.comb(0.0277, 0.65);
c2L.comb(0.0371, 0.51);           c2R.comb(0.0393, 0.51);
c3L.comb(0.0411, 0.5);            c3R.comb(0.0409, 0.5);
c4L.comb(0.0137, 0.73);           c4R.comb(0.0155, 0.73);
// Left ch. allpass filters       Right ch. allpass filters
a1L.allpass(0.005, 0.75);         a1R.allpass(0.00507, 0.75);
a2L.allpass(0.0117, 0.61);        a2R.allpass(0.0123, 0.61);
// Left ch. lowpass filter        Right ch. lowpass filter
lpL.lop(3500);                    lpR.lop(3500);

@slider
lpL.lop_set_freq(slider1);
lpR.lop_set_freq(slider1);

@sample
// Left ch. comb filters          Right ch. comb filters
sig1L = c1L.comb_do(spl0);        sig1R = c1R.comb_do(spl1);
sig2L = c2L.comb_do(spl0);        sig2R = c2R.comb_do(spl1);
sig3L = c3L.comb_do(spl0);        sig3R = c3R.comb_do(spl1);
sig4L = c4L.comb_do(spl0);        sig4R = c4R.comb_do(spl1);
// Comb filters summation
sumL = spl0 + sig1L + sig2L + sig3L + sig4L;
sumR = spl1 + sig1R + sig2R + sig3R + sig4R;
// Left ch. allpass filters       Right ch. allpass filters
all1L = a1L.allpass_do(sumL);     all1R = a1R.allpass_do(sumR);
all2L = a2L.allpass_do(all1L);    all2R = a2R.allpass_do(all1R);
// Left ch. lowpass filter        Right ch. lowpass filter
sigL = lpL.lop_do(all2L) * .2;    sigR = lpR.lop_do(all2R) * .2;
// Balance and output
spl0 = spl0 + (sigL - spl0) * slider2;
spl1 = spl1 + (sigR - spl1) * slider2;
```

Réverbérateur de Schroeder (modèle $b$ sur la figure) en plugin :

```
desc:Schroeder Reverb B

import cookdsp.jsfx-inc

slider1:0.2<0,1>Balance Dry/Wet

@init
// Left ch. allpass filters      Right ch. allpass filters
a1L.allpass(0.0204, 0.35);       a1R.allpass(0.02011, 0.35);
a2L.allpass(0.06653, 0.41);      a2R.allpass(0.06641, 0.41);
a3L.allpass(0.035007, 0.5);      a3R.allpass(0.03504, 0.5);
a4L.allpass(0.023021, 0.65);     a4R.allpass(0.022987, 0.65);
// Left ch. lowpass filters      Right ch. lowpass filters
lp1L.lop(5000);                  lp1R.lop(5000);
lp2L.lop(3000);                  lp2R.lop(3000);
lp3L.lop(1500);                  lp3R.lop(1500);
lp4L.lop(500);                   lp4R.lop(500);

@sample
// Left channel                  Right channel
sig1L = a1L.allpass_do(spl0);    sig1R = a1R.allpass_do(spl1);
sig2L = a2L.allpass_do(sig1L);   sig2R = a2R.allpass_do(sig1R);
sig3L = a3L.allpass_do(sig2L);   sig3R = a3R.allpass_do(sig2R);
sig4L = a4L.allpass_do(sig3L);   sig4R = a4R.allpass_do(sig3R);
lop1L = lp1L.lop_do(sig1L);      lop1R = lp1R.lop_do(sig1R);
lop2L = lp2L.lop_do(sig2L);      lop2R = lp2R.lop_do(sig2R);
lop3L = lp3L.lop_do(sig3L);      lop3R = lp3R.lop_do(sig3R);
lop4L = lp4L.lop_do(sig4L);      lop4R = lp4R.lop_do(sig4R);
sigL = (lop1L + lop2L + lop3L + lop4L) * 0.5;
sigR = (lop1R + lop2R + lop3R + lop4R) * 0.5;
spl0 = spl0 + (sigL - spl0) * slider1;
spl1 =  spl1 + (sigR - spl1) * slider1;
```

## 5.1 La panoramisation

Afin de situer un son sur l'axe horizontal entre deux haut-parleurs, on fait varier l'amplitude du même signal dans deux haut-parleurs. Plus l'amplitude est forte d'un côté, plus le son semble venir de cette direction.

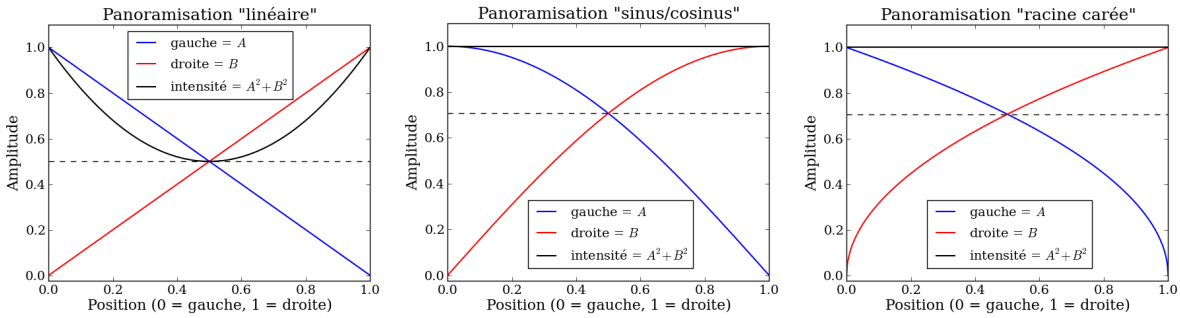



FIGURE 5.1: Comparaison de trois fonctions de panoramisation.

Le *plugin* suivant implémente les trois méthodes de panoramisation illustrées ci-dessus :

```
/* Trois algorithmes de panoramisation:
   linear: pan lineaire
   sincos: pan avec sinus et cosinus
   squareroot: pan avec racine carree */
desc: Panoramisation

slider1:.5<0,1>Pan
slider2:0<0,2,1{linear,sincos,squareroot}>Method

@slider
/* pan lineaire */
slider2 == 0 ? (
  panL = 1 - slider1;
  panR = slider1;
) :
/* pan sinus/cosinus */
slider2 == 1 ? (
  panL = cos(slider1 * $pi / 2);
  panR = sin(slider1 * $pi / 2);
) :
/* pan racine carree */
slider2 == 2 ? (
  panL = sqrt(1 - slider1);
  panR = sqrt(slider1);
);

@sample
/* mixage mono du signal stereo */
input = (spl0 + spl1) * 0.707;
/* amplitudes gauche et droite */
spl0 = input * panL;
spl1 = input * panR;
```

## 5.2 Un *plugin* d'effet Doppler

Le code suivant implémente un effet **Doppler « à bascule »** (il oscille constamment entre l'extrême gauche et l'extrême droite) à l'aide des éléments suivants :

1. Un LFO triangulaire (oscillant entre 0 et 1) servant à « basculer » le signal de l'extrême gauche à l'extrême droite en alternance.
2. Une panoramisation **sinus/cosinus** pour ajuster l'azimut du signal (0 = extrême gauche, 0.5 = centre, 1 = extrême droite).
3. Une ligne de délai servant à transposer le son. Le temps de délai est élevé aux extrémités et diminue en se rapprochant du centre (près de l'auditeur).
4. Un filtre passe-bas pour simuler une plus grande distance de la source aux extrémités (fréquence de coupure plus basse qu'au centre).

```
desc: Doppler Pendulum

import cookdsp.jsfx-inc

slider1:.05<0.01,2>Frequence d'oscillation (Hz)
slider2:0.2<0.001,0.499>Profondeur du delai (sec)

@init
phase = 0;            /* phase courante du LFO */
del.delay(srate/2); /* Initialisation du delai */
filt.lop(1000);       /* Initialisation du filtre */

@slider
/* Increment pour une rampe de 0 -> 2. */
inc = 2 * slider1 / srate;

@sample
input = (spl0 + spl1) * 0.707; /* Input stereo -> mono. */

/* Rampe de 0 a 2 ===> triangle entre 0 et 1. */
lfo = min(phase, 2-phase);
/* Incremente la phase courante */
phase += inc;
/* Restreint a l'interieur des limites 0 a 2. */
phase > 2 ? phase -= 2;

/*** Lecture dans la ligne de delai ***/
/* Triangle entre 0 et 1 ===> triangle entre 0.5 et 0. */
delay = (0.5 - min(lfo, 1-lfo));
/* On ajuste l'ambitus, 0.001 (centre) => 0.5 (extremites). */
delay = delay * 2 * slider2 + 0.001;
val = del.delay_sread2(delay);

/*** Ecriture dans la ligne delai ***/
del.delay_write(input);

/*** Filtrage passe-bas ***/
/* Triangle entre 0 et 1 ===> triangle entre 0 et 0.5. */
freq = min(lfo, 1-lfo);
/* On ajuste l'ambitus, 10000 (centre) ==> 250 (extremites). */
filt.lop_set_freq(freq * 20000 + 250);
val = filt.lop_do(val);

/*** Panoramisation ***/
pan = lfo * $pi * 0.5;
spl0 = val * cos(pan);
spl1 = val * sin(pan);
```

# Chapitre 6

# Le traitement de la dynamique

## 6.1 Le suivi de l'enveloppe d'amplitude

Le suivi d'amplitude est une technique d'analyse simple et efficace qui permet, par exemple, de sculpter l'enveloppe temporelle d'un son en fonction d'un second son, renforçant ainsi l'homogénéité des deux corps sonores.

Pour obtenir le suivi d'amplitude d'un signal, on a d'abord recours à un procédé de **rectification positive de l'onde**. La portion négative du signal est « rectifiée » vers le positif par inversion de la polarité (en prenant la valeur absolue de chacun des échantillons). Ce signal unipolaire est ensuite filtré à l'aide d'un filtre passe-bas à fréquence de coupure très basse (généralement autour de 10 Hertz) afin d'éliminer les petites variations de la forme d'onde et de ne garder que le contour général. Ce signal constitue un suivi relativement fidèle de l'enveloppe d'amplitude du signal d'origine. La fréquence de coupure, en Hertz, du filtre passe-bas permet d'ajuster le temps de réponse du suivi d'amplitude, en lissant plus ou moins les données d'analyse. Le *plugin* suivant illustre cette technique.

```
desc: Envelope Follower

slider1:10<1,100>Lowpass Cutoff (Hz)
slider2:-60<-60,6>Noise Gain (dB)

@init
y0 = y1 = 0; /* Initialisation des variables du filtre. */

@slider
/* Calcul du coefficient du filtre. */
c = exp(-2 * $pi * slider1 / srate);
/* Calcul du gain du bruit blanc. */
gain = pow(10, slider2 * 0.05);

@sample
/* Rectification positive du signal. */
aL = abs(spl0); aR = abs(spl1);
/* Filtre passe-bas. y0 et y1 contiennent le suivi d'amplitude. */
y0 = (1-c) * aL + c * y0;
y1 = (1-c) * aR + c * y1;
/* Signal original + bruit blanc * suivi * gain. */
spl0 += (rand(2) - 1) * y0 * gain;
spl1 += (rand(2) - 1) * y1 * gain;
```

## 6.2 La porte de bruit (*Noise Gate*)

Le *plugin* suivant donne une implantation d'une porte de bruit simple. On y effectue un test sur la valeur de la puissance RMS (« quasi-RMS » obtenue ici à l'aide d'un suivi d'amplitude simple). Si l'amplitude du signal est supérieur à au seuil fixé par l'utilisateur (*thresh*), alors *gateX* prend la valeur 1. Sinon, *gateX* prend la valeur 0. Un filtre passe-bas est ensuite appliqué à ce signal binaire de manière à lisser les transitions (sinon elles seront très audibles). Ce filtre passe-bas agit comme un « portamento ». La valeur visée (0 ou 1) sera atteinte dans une durée équivalent à la période du filtre, c'est-à-dire l'inverse de sa fréquence. Si le « portamento » est ajusté à 50 millisecondes, la fréquence de coupure du filtre sera :

$$f = 1/(ms/1000) = 1/0.05 = 20 \text{ Hz}$$

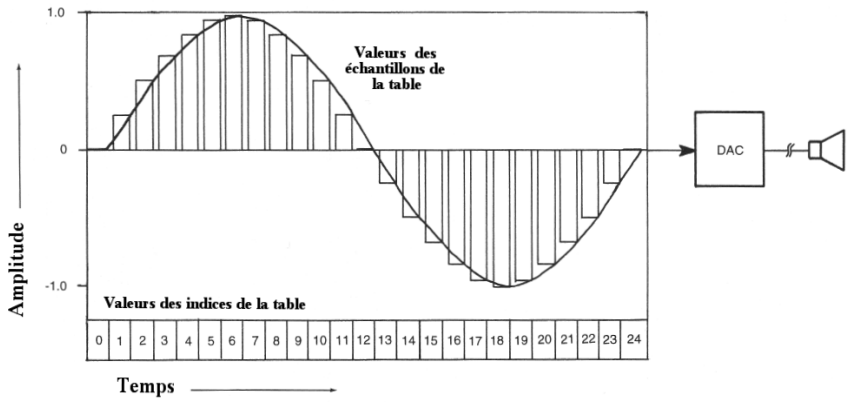
```
/* Une porte de bruit simple. */
desc: Noise Gate

slider1:-30<-70,0>Threshold (dB)
slider2:50<1,250>Portamento (ms)

@init
/* Initialisation des variables du filtre. */
y0 = y1 = 0;
/* Calcul du coefficient du filtre (fixe a 10 Hz). */
c = exp(-2 * $pi * 10 / srate);
/* Initialisation des variables d'amplitude. */
gate0 = gate1 = 0;
/* Initialisation des variables du portamento. */
port0 = port1 = 0;

@slider
/* Calcul du seuil lineaire. */
thresh = pow(10, slider1 * 0.05);
/* Calcul du coefficient du filtre passe-bas
   utilise pour filtrer les changements de gain
   (portamento). hz = 1 / (ms / 1000). */
freq = 1 / (slider2 * 0.001);
coeff = exp(-2 * $pi * freq / srate);

@sample
/* Rectification positive du signal. */
aL = abs(spl0); aR = abs(spl1);
/* Filtrage passe-bas. "y0" et "y1"
   contiennent le suivi d'amplitude. */
y0 = (1-c) * aL + c * y0;
y1 = (1-c) * aR + c * y1;
/* Ouverture de la porte si suivi > seuil. */
y0 > thresh ? gate0 = 1 : gate0 = 0;
y1 > thresh ? gate1 = 1 : gate1 = 0;
/* Filtrage passe-bas. "port0" et "port1"
   contiennent la courbe de gain de la porte. */
port0 = (1 - coeff) * gate0 + coeff * port0;
port1 = (1 - coeff) * gate1 + coeff * port1;
/* Signal de sortie : splX * courbe de gain. */
spl0 *= port0;
spl1 *= port1;
```

# Chapitre 7

# Le traitement par granulation

### 7.0.1 La lecture de tableaux par incrémentation d'un index

La méthode la plus simple pour générer un onde sonore consiste simplement en une lecture séquentielle d'un espace-mémoire contenant une forme d'onde. Cette lecture est contrôlée par un **index** (ou pointeur sur une adresse du tableau) qui est incrémenté entre chaque prélèvement, comme illustré sur la figure ci-dessous.




Ainsi, pour obtenir des hauteurs différentes, on lit le tableau en variant l'incrément du pointeur. Des échantillons sont passés ou répétés. Idéalement, le processus de lecture comprend une opération d'interpolation entre les échantillons pour plus de précision.

Si le tableau (mémoire) contient la définition d'une seule période d'une forme d'onde, on pourra générer un signal continu en « bouclant » le pointeur/index lorsqu'il arrive à la fin du tableau. Pour les lectures de tableau dans un environnement à échantillonnage à taux fixe (comme c'est le cas de *Reaper*), l'équation suivante résume l'opération à effectuer :

$$phase = (last\_phase + inc\ )\ \%\ L$$

où *phase* est la phase courante, *last_phase* est la phase précédente, *inc* est l'incrément appliqué au pointeur de lecture, $L$ est la longueur de la table en échantillons et % est l'opérateur *modulo*, qui restreint les valeurs entre zéro et la longueur du tableau $L$.

On calcul l'incrément en fonction de la fréquence désirée ($f_d$), de la longueur du tableau ($L$) ainsi que de la fréquence d'échantillonnage ($f_s$) :

$$inc = f_d \times L \ / \ f_s$$

Le *plugin* suivant illustre la lecture de tableau avec interpolation :

```
desc:Lecture de tableau

slider1:500<40,2000>Frequence fondamentale
slider2:-6<-60,0>Gain (dB)

@init
/* longueur du tableau */
len = 1024;
/* espace memoire */
buf = 0;
/* avec une boucle, on ecrit une periode
   de la forme d'onde dans le tableau */
i = 0;
loop(len,
  buf[i] = sin(2 * $pi * i / len);
  i += 1;
);
/* phase de lecture courante */
phase = 0;

@slider
/* calcul l'increment de lecture en
   fonction de la frequence fondamentale */
inc = slider1 * len / srate;
/* calcul du gain */
amp = pow(10, slider2 * 0.05);

@sample
/* genere le signal seulement en mode play */
play_state == 1 ? (
  /* lecture du tableau avec interpolation */
  ipos = floor(phase);
  frac = phase - ipos;
  sig = buf[ipos] + (buf[ipos+1] - buf[ipos]) * frac;
  /* incremente la phase */
  phase += inc;
  /* retour a 0 quand la phase atteint la fin du tableau */
  phase >= len ? phase -= len;
  /* gestion d'amplitude et sortie */
  spl0 = spl1 = sig * amp;
);
```

### 7.0.2 La mise en mémoire d'échantillons dans un tableau

Le fondement même de la granulation consiste à réorganiser la lecture des échantillons d'un son mis en mémoire dans un tableau (dans la mémoire RAM de l'ordinateur). La granulation n'est pas un processus en temps réel, c'est-à-dire que l'on ne peut pas granuler les échantillons successifs qui entrent et qui sortent du *plugin*. Pour mettre en place un processus de granulation, on doit posséder un bloc plus ou moins long d'échantillons en mémoire. La première étape d'un *plugin* de granulation consiste donc à enregistrer des échantillons dans un tableau.

L'exemple suivant utilise l'enregistrement d'échantillons en mémoire pour étendre les possibilités de la lecture de forme d'onde avec interpolation. Ici, la forme d'onde n'est pas préalablement dessinée dans le tableau mais enregistrée à partir du signal d'entrée du *plugin*. Une enveloppe d'amplitude est appliquée aux échantillons enregistrés afin d'assurer que le premier et le dernier échantillons du tableau ont bien une valeur de zéro.

```
desc:Remplir un tableau via l'input audio

slider1:60<20,250>Frequence fondamentale
slider2:-6<-60,0>Gain (dB)

import cookdsp.jsfx-inc

@init
len = 512;     /* longueur du tableau */
phase = 0;     /* phase de lecture courante */
record = 0;    /* signal d'activation de l'enregistrement */
recpos = len; /* phase d'ecriture */

/* espace memoire stereo */
bufLmem = bufL.buffer(len);
bufRmem = bufR.buffer(len);

/* enveloppe du grain */
window.buffer(len);
window.buffer_window(1);

@slider
/* calcul l'increment de lecture en
   fonction de la frequence fondamentale */
inc = slider1 * len / srate;
/* calcul du gain */
amp = pow(10, slider2 * 0.05);

@sample
/* Si l'enregistrement est actif */
record == 1 ? (
    /* enregistre "len" echantillons multiplies par l'enveloppe. */
    win = window.buffer_read(recpos);
    bufL.buffer_write(recpos, spl0 * win);
    bufR.buffer_write(recpos, spl1 * win);
    /* incremente la position d'ecriture */
    recpos += 1;
    /* arrete l'enregistrement quand les
    tableaux sont pleins. */
    recpos == len ? record = 0;
);

/* genere le signal seulement en mode play */
play_state == 1 ? (
    /* lecture du tableau stereo avec interpolation */
    sigL = bufL.buffer_read2(phase);
    sigR = bufR.buffer_read2(phase);
    /* incremente la phase */
    phase += inc;
    /* retour a 0 quand la phase atteint la fin du tableau */
    phase >= len ? phase -= len;
    /* gestion d'amplitude et sortie */
    spl0 = sigL * amp;
    spl1 = sigR * amp;
);
```