



::{ Rapport n°3 }::

---

Julien-Pierre AVÉROUS  
Guillaume CALAS  
Patrice DE SAINT STEBAN  
Fabien DEBUIRE

---



# Table des matières

<b>Introduction</b>	<b>1</b>
<b>1 Le projet</b>	<b>2</b>
1.1 Le groupe de projet . . . . .	2
1.2 Avancement du projet . . . . .	2
<b>2 Plugs-In &amp; SDK-Bridge [Julien-Pierre]</b>	<b>4</b>
2.1 Introduction . . . . .	4
2.2 Nouveautés . . . . .	4
2.3 Conclusion sur mon travail . . . . .	6
<b>3 L'égaliseur (suite) [Guillaume]</b>	<b>7</b>
3.1 Avant-propos . . . . .	7
3.2 Égaliseur . . . . .	8
3.2.1 Plug-in clfft . . . . .	9
3.2.2 Plug-in clequa . . . . .	9
3.2.3 Problème rencontré . . . . .	10
<b>4 Interface graphique [Fabien]</b>	<b>11</b>
<b>5 Les filtres [Patrice &amp; Fabien]</b>	<b>12</b>
5.1 Introduction [Patrice] . . . . .	12
5.2 <i>Trémolo</i> [Patrice] . . . . .	13
5.3 Écho [Patrice] . . . . .	14
5.4 Flanger [Patrice] . . . . .	15
5.5 Chorus [Patrice] . . . . .	16
5.6 Codevoice [Patrice] . . . . .	17
5.7 Noise et Distortion [Patrice] . . . . .	18
5.8 Filtage de la voix [Fabien] . . . . .	19

<b>6</b>	<b>Bibliothèque</b> [Patrice]	<b>20</b>
6.1	Introduction . . . . .	20
6.2	cltag . . . . .	21
<b>7</b>	<b>Conclusion générale</b>	<b>23</b>
7.1	Ce qu'il reste à faire . . . . .	23
7.2	Conclusion . . . . .	23

# Introduction

Cette année, **CUNIXLINGUS Team** s'est lancé dans la mise au point d'un logiciel multi-plateformes (LINUX et MAC en priorité) de mixage audio (une platine numérique en quelque sorte).

Ce choix a été motivé par deux choses ; d'abord nous sommes tous des passionnés de musique électronique (Fabien est *Disk Jockey*), et ensuite nous sommes également très intéressés par la mise au point numérique de procédés mathématiquement très avancés.

De ce fait, notre projet, baptisé *Cl;MaX : Advanced Audio Shaker*, se veut résolument techniquement poussé sur certains points particuliers. Si nous en avons le temps, nous tenterions de tout recréer de A à Z de façon à interagir à tous les niveaux du traitement audio numérique, mais malheureusement nous devons poursuivre nos études en parallèle et notre niveau actuel en algorithmique avancée n'est pas encore suffisant de même que nos connaissances informatiques. Ainsi, nous avons et seront amenés à réutiliser du code en Open Source pour certains traitements très proches du hardware.

Quoi qu'il en soit, notre but final est de proposer aux utilisateurs un logiciel de mixage minimaliste et résolument évolutif grâce au développement de notre application sous formes de plugs-in. Un SDK (*Standard Development Kit*) sera d'ailleurs mis à disposition dans la version finale pour les développeurs extérieurs à notre équipe. Il est d'ailleurs actuellement utilisé par notre équipe pour la mise aux points des plugs-in de base.

Enfin, il va s'en dire que ce projet risque de nous apporter énormément de connaissances à tous dans de nombreux domaines tels que le développement d'applications sous formes de plugs-in et le traitement avancé d'un signal quel qu'il soit.

# Chapitre 1

## Le projet

### 1.1 Le groupe de projet

Notre groupe, constitué de vieux briscards de la SupB2, fonctionne, à allure réduite certes, mais dans la joie et la bonne humeur. Nous avons un peu de mal à nous mettre au travail, mais quand on arrive à se mettre d'accord sur le travail il n'y a généralement pas de problèmes et tout le monde est à son poste.

### 1.2 Avancement du projet

Lors de la précédente tranche de travail, nous avons appris à nous servir de notre SDK ce qui fut relativement aisé compte tenu de la bonne ergonomie de ce dernier. Notre travail pour cette soutenance consista donc principalement à réaliser le maximum de plugs-in pour notre logiciel, *ClMaX*. Nous n'avons pas d'objectif précis à réaliser, si ce n'est de faire de notre mieux. Nous sommes dans les temps, il ne nous reste plus qu'à paufiner le visuel de nos plugs-in qui reste, pour l'heure, assez rudimentaire.

Pour des raisons purement scolaires, nous n'avons pas souhaité travailler tous les week-end comme nous en avons pris l'habitude précédemment. Nous avons uniquement conservé notre petite semaine de travail en commun, ce qui, au final, s'est révélé suffisant.

Pour cette avant-dernière soutenance, notre bilan est le suivant :

- 8 filtres audio
- Vus-metter
- balance (gestion d'une double *playlist*)
- égaliseur (conversion sous forme de plug-in)

Comme précisé précédemment, la bonne ergonomie de notre SDK nous à permis d'avancer à un rythme soutenu et avec une grande simplicité. Par exemple, une fois les algorithmes trouvés, il n'a fallut qu'une seule journée de travail à Patrice pour réaliser les 8 filtres audio !

Mais notre projet n'est pas encore achevé, il reste encore pas mal de travail notamment au niveau du visuel. Pour résumer, il nous reste à :

- paufiner le visuel
- automatiser de la procédure d'installation
- écrire le manuel d'utilisation de notre logiciel
- écrire le manuel d'utilisation de notre SDK
- réaliser d'autres plugs-in (bonus)

# Chapitre 2

## Plugs-In & SDK-Bridge [Julien-Pierre]

### 2.1 Introduction

Pour cette troisième soutenance, maintenant que le SDK est finit à 99.9% (il manquera toujours un petit truc), le but a été d'avancer l'application elle même. Faire une esquisse de l'interface, améliorer les plugs-in existants (stabilité, qualité audio), en faire des nouveaux : `clpiste` pour la gestion des pistes audio, `cldeco` pour la gestion de la décoration globale. Je vais lister les éléments qui ont été retouchés ou créés, et renseigner les modifications apportées .

### 2.2 Nouveautés

- *clspectre* – Il y avait quelques problèmes avec la prise des échantillons dans le buffer audio. On affichait des fois la voie gauche, des fois la voie droite. C'est maintenant la moyenne entre les deux voies qui est utilisée pour calculer l'affichage. De plus, l'agorithme utilisé est maintenant beaucoup plus optimisé à l'aide d'un *buffering* du tracé (L'affichage est plus rapide que le rafraîchissement du buffer audio, or à chaque affichage je recalculais la courbe). Finalement, l'affichage a été agrandis pour mieux prendre en compte le buffer audio qui a changé de taille.
- *cloau* – Simple changement au niveau de la taille du buffer de sortie audio pour avoir une meilleur réactivité de l'application (le buffer a été réduit, ce qui implique une fréquence de rafraîchissement plus élevée, et donc une interface plus proche de la sortie son sur les haut-parleurs).
- *cliau* – Adaptation du code à l'évolution du format `clflow`, représentant des données audio en mémoire.

- *clplayer* – De même que *cliau*, il a du être adapté au nouveau *clflow* (ce nouveau *clflow* apporte, entre autres, la possibilité de définir un coefficient d’amplitude, ce que *clplayer* prend en compte lors du mix final du flux audio). Finalement, le mixage entre les pistes audio globales a été amélioré : avant, à chaque ajout de mix, il y avait une diminution du volume sonore, ce qui n’est plus le cas. Et des tests de saturations sont effectués pour éviter d’avoir des grésillements.
- *clvolume* – Il a une nouvelle interface créée à partir d’une texture (elle-même créé sous Photoshop) et peut dorénavant être modifiée par un drag&drop de souris ou par la molette de la souris.
- *clvum* – De même que *clvolume*, *clvum* bénéficie d’une peau neuve, tirée cette fois du VuMetter de GarageBang d’Apple. Mais la texture a été pas mal remaniée sous Photoshop pour l’agrandir et l’adapter à nos besoins. Le plus gros problème rencontré avec cette nouvelle apparence vient de la difficulté à trouver des renseignements spécifiques sur OpenGL. En effet j’ai choisi la solution optimisée d’avoir une texture de *vu metter* au maximum et une texture de ce même *vu metter* au minimum, puis avec OpenGL d’afficher le *vu metter* minimum et le *vu metter* maximum découpé en fonction du volume audio par dessus le *vu metter* minimum. *Clvum* bénéficie aussi maintenant d’un meilleur calcul du volume audio de sortie. Je me basais avant sur l’amplitude du signal, ce qui n’est pas correct. En effet, le son est produit par le haut-parleur par différence d’amplitude entre 2 points du temps  $t$  et  $t + 1$ . Plus la différence d’amplitude est grande entre  $t$  et  $t + 1$ , plus le son produit par les haut-parleurs est fort, c’est-à-dire que plus le coefficient de la droite d’amplitude entre  $t$  et  $t + 1$  est important, plus le son est fort. Je calcule donc maintenant la dérivé de l’amplitude audio pour en déduire le volume audio de sortie.
- *claddsong* – Même si *claddsong* n’est pas destiné à rester, il a quand même rapidement évolué pour prendre en compte le plug-in *clpiste* qui permet de gérer des pistes audio sur deux zones (voir le paragraphe sur *clpiste*).
- *clpiste* – Grande nouveauté de cette nouvelle version de *Cl<sub>i</sub>MaX*, *clpiste* permet d’avoir une interface agréable, intuitive et ergonomique pour gérer une application multipiste répartie en deux zones audio. Effectivement, les DJ utilisant du matériel audio, ont souvent un système symétrique : deux platines vinyle, deux lecteurs CD, ... ceci dans un but de préparer l’enchaînement pendant qu’une musique ou un morceau est joué. Grâce à

la programmation orientée objet, `clpiste` permet de mettre à disposition ces deux zones parfaitement symétriques et identiques. Ces deux zones regroupent des pistes audio. Chaque zone peut en contenir théoriquement une infinité, mais l'interface ne permet pour le moment d'en afficher que 5 pour chaque zone. Chaque piste est représentée par 3 petits boutons d'action : la suppression (pour supprimer la piste en question), la pause (pour mettre en pause la piste) et le silence (pour continuer à lire la piste, mais sans jouer le son qu'elle contient) et par une large bande contenant l'intégralité de la courbe d'amplitude audio du son qu'elle contient. Est dessiné sur cette bande un curseur de positionnement qui permet de savoir où en est la lecture de cette piste et qui permet également au DJ de repositionner lui-même, à l'aide de la souris, et en direct, la position de lecture. Finalement, `clpiste` affiche un petit curseur permettant de passer le son de sorti d'une zone à une autre avec un *fading*.

- *cldeco* – Permet de mettre à disposition des autres plugs-in les éléments graphiques d'interface dont ils ont besoin, et ceci de manière très simple (`cldeco` gère lui-même sa mémoire pour éviter que chaque plug-in n'ait à le faire, et il suffit de passer l'id du composant que l'on souhaite obtenir pour l'avoir immédiatement). Enfin, `cldeco` dessine la frame globale du logiciel avec le logo, en se chargeant du redimensionnement dynamique de cette frame en cas de redimensionnement de la fenêtre)

## 2.3 Conclusion sur mon travail

*CljMaX* suit son chemin, il avance bien, et le résultat devient possible à exploiter. L'interface arrive, les fonctionnalités sympas aussi. Il ne restera plus grand-chose à faire pour la soutenance finale : finir l'interface, coller les filtres sur les pistes selon les désirs du DJ, permettre peut être une gestion de flux audio via des réseaux.

# Chapitre 3

## L'égaliseur (suite) [Guillaume]

### 3.1 Avant-propos

L'égaliseur est un élément essentiel de tout logiciel ayant pour vocation de jouer un quelconque fichier audio. Il permet de modifier le rendu audio en agissant directement sur les composantes fréquentielles de la piste audio. Cela permet de modifier, par exemple, l'ambiance de lecture, ou d'amplifier certaines caractéristiques d'un morceau de musique comme les basses ou les aigus.

Pour rappel, la bande d'audition humaine se situe entre 20Hz et 20kHz, c'est sur cette particularité que se base l'audio numérique qui ne conserve que les composantes qui se trouvent dans ce domaine (pour des raisons évidentes d'espace mémoire). De même, tous les formats de compression audio numériques suppriment automatiquement les composantes qu'ils jugent inaudibles<sup>1</sup>. Pour ce qui est de l'égaliseur, on va découper ce domaine audible en sous-domaines qui correspondront, pour la plupart du temps, aux octaves<sup>2</sup>. Une fois ce découpage effectué, il ne reste plus qu'à appliquer un certain coefficient, représentant le gain, à chacune des bandes de fréquence selon les préférences de l'utilisateur. Mais avant de pouvoir appliquer ces coefficients, il faut d'abord décomposer le signal audio.

Ce qu'il faut bien comprendre c'est que chaque son est composé d'une multitude de signaux, dits "de base", de différentes amplitudes et fréquences. Le problème c'est que, dans la vraie vie, l'on dispose rarement de la fonction du signal. Heureusement, le mathématicien français Joseph FOURIER a mis au point un opérateur qui permet de décomposer n'importe quel signal : la transformée de Fourier. Le problème avec cet opérateur c'est que c'est un opérateur continu, ce qui

---

<sup>1</sup>C'est le cas du MP3 et de l'Ogg Vorbis.

<sup>2</sup>Un octave signifie que pour chaque "note", il faut doubler sa fréquence avant de retrouver cette "note" avec une fréquence plus élevé.

le rend inutilisable en l'état par des ordinateurs, il existe bien un équivalent discret, mais il est très couteux en termes de calculs ce qui le rend inutilisable même par les ordinateurs les plus puissants. La solution est venue de deux chercheurs, COOLEY et TURKEY, qui ont mis au point un algorithme<sup>3</sup> qui, par le biais de quelques astuces, permet de simplifier considérablement les calculs du moment le nombre d'échantillons à traiter est une puissance de 2.

Le principe d'un égaliseur est donc, en théorie, assez simple :

1. récupérer un *sample* dont la longueur est une puissance de 2
2. décomposer le signal du domaine temporel au domaine fréquentiel (*FFT*)
3. traitement du signal en fréquence, en appliquant les coefficients de réglage
4. recombinaison du signal (*FFT inverse*)

## 3.2 Égaliseur

Lors des précédentes soutenances, j'ai fait de nombreuses recherches sur l'implémentation de la transformée rapide de Fourier. Il existe un grand nombre de variations selon l'utilisation que l'on fait de cet opérateur. En effet, les FFTs sont utilisées dans de nombreux domaines scientifiques et pas seulement dans le traitement des signaux, on les utilise également dans les processeurs afin d'accélérer les gros calculs à virgule flottante.

Au départ, j'avais pour but d'implémenter la FFT personnellement, mais cet exercice s'est vite révélé au dessus de mes propres moyens, mes bagages scientifiques ne me permettent pas encore de venir à bout d'un tel *challenge*. J'ai donc dû m'inspirer très fortement d'algorithmes trouvés sur Internet, en particulier de l'implémentation de Don CROSS.

J'ai donc utilisé une adaptation de cet algorithme pour réaliser une première mouture de notre égaliseur qui fonctionnait indépendamment du reste de notre logiciel, et en mode texte. Ce n'était évidemment qu'un premier essai qui n'était pas très facile à manipuler. J'ai donc entrepris de porter mon égaliseur sous forme d'un plug-in afin de pouvoir l'utiliser directement dans notre application. C'est ce que j'ai fait pour cette troisième soutenance : porter l'égaliseur sous forme de plug-in et en réaliser l'interface graphique.

Pour des raisons pratiques, j'ai découpé le fonctionnement de l'égaliseur en deux plugs-in : `clequa`, qui est le plug-in égaliseur à proprement parler, et `clfft`, qui est un plug-in outil qui permet l'utilisation de la FFT dans différents plugs-in.

<sup>3</sup>Fast Fourier Transform ou Transformée Rapide de Fourier.

### 3.2.1 Plug-in clfft

Ce plug-in est un plug-in outil (kClUtilPlug) qui permet aux autres plugs-in d'utiliser la Transformée de Fourier Rapide (FFT) pour diverses utilisations.

Avant toute utilisation, il est nécessaire d'initialiser certaines variables propres à la procédure. Cette initialisation se fait grâce à la fonction suivante :

```
void initFFT(short *sample, unsigned nbr_ech, double *RealIn,
             double *ImagIn)
```

Nota bene : il est nécessaire d'allouer convenablement RealIn et ImagIn avant l'appel de cette fonction.

On peut ensuite appliquer la FFT à son signal afin de le décomposer. Le prototype de la fonction est le suivant :

```
void fft(int inv, unsigned int nbr_ech, double *RealIn, double
        *ImagIn, double *RealOut, double *ImagOut)
```

Le paramètre *inv* permet de déterminer s'il s'agit d'une FFT normale (*inv*=0) ou une FFT inverse (*inv*=1). Il faut allouer RealOut et ImagOut de la même manière que pour RealIn et ImagIn.

**Attention** : il ne faudra pas oublier de libérer la mémoire en fin d'utilisation.

### 3.2.2 Plug-in clequa

Ce plug-in correspond à l'interface et au traitement de l'égaliseur. Il s'agit d'un plug-in "filtre" (kClFiltrePlug). Le fonctionnement est très simple. On récupère le *sample* à traiter et on le décompose grâce au plug-in outil clfft. Une fois décomposé on traite les données obtenues grâce à la fonction

```
void trFFT(short nbr_ech, double *RealIn, double *ImagIn , bFr
          *mod)
```

Où *mod* est un pointeur sur une structure de donnée qui contient les gains à appliquer aux bandes de fréquences.

L'interface graphique est tout ce qu'il y a de plus standard. Il y a un bouton pour **activer/désactiver** l'égaliseur, un bouton pour **charger** un modèle de pré-réglage, un bouton pour **sauvegarder** ses propres réglages, un pour **ouvrir** un fichier de réglages personnels et bien sûr, on peut **régl**er directement le gain de chaque bande de fréquence (de -12 à +12dB) ainsi que la pré-amplification du signal (de -20 à +20dB). En voici un aperçu :

### 3.2.3 Problème rencontré

A l'heure actuelle, l'égaliseur souffre d'un problème assez gênant : le bruit. En effet, lorsque l'on applique un gain trop important à un *sample*, un bruit assez gênant apparaît, une sorte de grésillement dont l'origine m'est encore inconnue.

J'ai d'abord pensé que cela venait des bits extrêmes de chaque bloc qui sont, à cause de la FFT, souvent accentués, ce qui pourrait générer ce "clic", mais après de nombreux tests et essais sur ce point, je ne suis jamais arrivé à diminuer cette gêne. J'ai également essayé de modifier ma méthode d'application du gain par bandes de fréquence en utilisant un schéma gaussien, mais ça n'a fait qu'amplifier le bruit. Le mystère reste donc entier, mais je de bons espoirs de venir à bout de ce désagrément.

## Chapitre 4

### Interface graphique [Fabien]

Les choix fait précédement pour l'interface graphique ce sont avérés très mauvais et j'en suis le seul responsable, a savoir : pour afficher un template, des boutons et tous ce genre de graphiques le choix du format jpeg était très mauvais. Cependant la transition au format bmp s'avère très simple à mettre en oeuvre. De plus, l'interface, très inspiré d'un autre logiciel de mixage car très complète (voir trop) est très lourde à charger mais c'est un bon compromis dans le sens où l'on gagne en performances.

# Chapitre 5

## Les filtres [Patrice & Fabien]

### 5.1 Introduction [Patrice]

Un filtre modifie (ou filtre) certaines parties d'un signal d'entrée dans le domaine temps et dans le domaine fréquence. D'après le théorème de Fourier, tout signal réel peut être considéré comme composé d'une somme de signaux sinusoïdaux (en nombre infini si nécessaire) à des fréquences différentes ; le rôle du filtre est de modifier la phase et l'amplitude de ces composantes.

Il existe 5 principes de modification du signal et qui sont utilisés par tous les effets.

- *Variations de volume* (*volume = niveau sonore*) : varier le volume consiste à multiplier le signal par un coefficient qui varie. Avec une variation sinusoïdale, on crée l'effet *trémolo* qui fait paraître une vibration.
- *Filtration des harmoniques* (*graves, médiums, aigus, etc.*) : le théorème de FOURIER nous dit : "Une fonction périodique non sinusoïdale peut être considérée comme la somme algébrique d'un terme constant et de fonctions sinusoïdales dont les fréquences sont des multiples entiers de la fréquence de la première fonction (série de Fourier)", On apprend donc qu'un son est composé de plusieurs sinusoïdes de fréquences et d'amplitudes différentes. Grâce à la FFT<sup>1</sup> on peut ainsi décomposer toutes les harmoniques du son. On applique alors un coefficient différent sur chaque harmonique. On travaille souvent sur des bandes de fréquences : grave, médium, aiguë. Il existe 4 filtres de fréquences : passe bas, qui coupe les hautes fréquences, le passe haut, qui coupe les basses fréquences, le passe-bande qui ne laisse passer qu'une bande de fréquences, et enfin le coupe bande, qui coupe une bande de fréquence médium.
- *Variations de fréquence* (*fréquence = hauteur du son*) : a ne pas confondre

---

<sup>1</sup>Fast Fourier Transformer

avec la filtration des harmoniques ! La fréquence d'un son correspond à ce que l'on appelle "note" en musique. En bouchant un ou plusieurs trous d'une flûte, en pinçant une corde de guitare à tel ou tel endroit, on change la fréquence (vitesse) à laquelle l'air ou la corde oscille. La fréquence d'un son s'exprime en hertz (Hz). On peut ainsi faire varier la fréquence d'un son autour d'une valeur fixe pour faire un effet de *vibrato* ou alors accélérer ou ralentir une musique.

- *Effets retard (Écho, réverbération ou delay en anglais)*. L'écho est un mélange simple du signal d'origine avec le même signal décalé (retardé) et atténué dans le temps. La réverbération est une addition d'échos successifs, avec des retards différents, chaque écho faisant l'effet d'une filtration plus ou moins grave ou aiguë. L'effet de réverbération tente de simuler l'ambiance d'une salle dans laquelle le son va rebondir sur plusieurs obstacles (murs, colonnes, etc.). Chaque obstacle étant à une distance différente de l'auditeur, les retards seront multiples.
- *Saturation* : si vous posez un micro à 10 cm d'un réacteur de Boeing, vous allez expérimenter un exemple typique de saturation (*overdrive*) : le son est tellement fort que votre micro ne sera pas en mesure de l'enregistrer d'une façon fidèle. Les systèmes d'enregistrement, de traitement et de restitution du son ont tous un volume maximum au-delà duquel le signal est perdu ou déformé.

Dans la plupart des cas, les filtres ne se contentent pas d'opérer une modification constante du son. Diviser le volume d'un son par deux présente peu d'intérêt. Faire varier ce volume d'une façon régulière est plus intéressant (comme si vous tourniez le bouton de volume de votre ampli dans un sens puis dans l'autre sans arrêt).

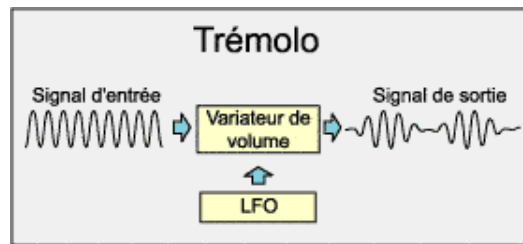
La plupart des effets sont des combinaisons de ces 5 filtres principaux. Souvent en découpant le signal en plusieurs pour appliquer des filtres différents sur chacun. Voici maintenant la description des quelques filtres audio que j'ai créé pour *Cl<sub>i</sub>M<sub>a</sub>X*. Chaque filtre est un plug-in qui implémente la fonction `clfilter`. Cette fonction prend en paramètre un `clflow` qui contient le signal audio représenté par des entiers intercalés pour la voie droite et la voie gauche.

## 5.2 Trémolo [Patrice]

Ce filtre est un filtre qui modifie le volume du son grâce à une fonction sinusoidale. Appliquée à la parole, il donne une voix chevrotante, utilisée par les acteurs de théâtre pour exprimer la tristesse la plus profonde. Pour mettre du *trémolo* dans votre voix, faites varier la quantité d'air (le souffle) expiré par vos poumons

pendant que vous parlez.

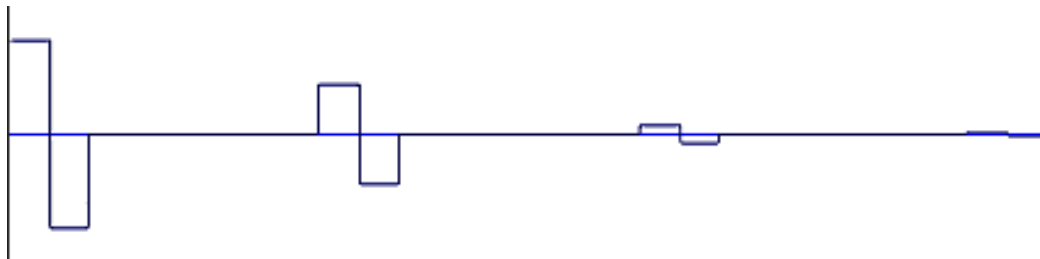
Pour réaliser ce filtre, je fais simplement une multiplication de tous les point du signal par un coefficient qui varie grâce à une fonction sinusoïdale, ce coefficient varie très peu.



### 5.3 Écho [Patrice]

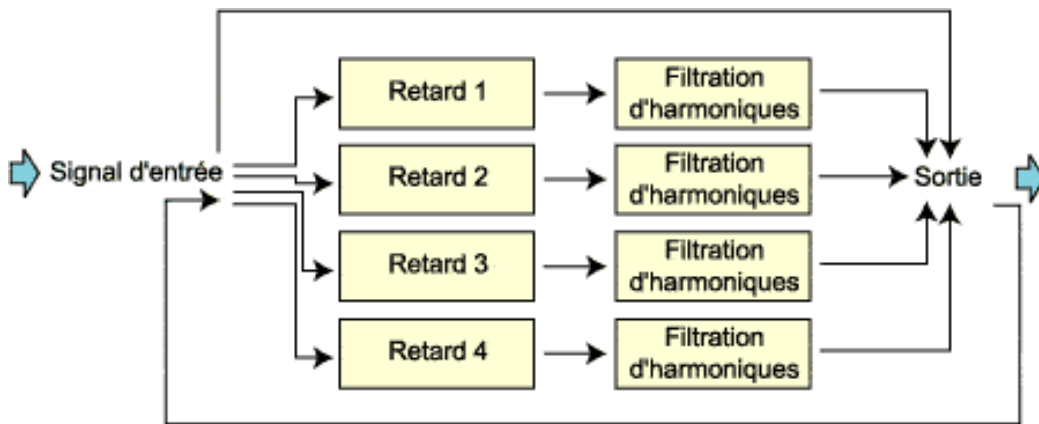
L'écho (*delay* en anglais) est un effet qui répète le signal qu'on lui soumet à intervalles réguliers et en l'atténuant. Le *delay* est un effet qui reproduit un phénomène acoustique naturel que tout le monde a déjà expérimenté : l'écho. D'où vient cette répétition ? Des multiples rebonds de l'onde sonore que vous produisez et qui vous reviennent aux oreilles, à chaque fois un peu plus atténués et filtrés (lorsqu'elle rentre en contact avec de la matière, une onde est en partie absorbée par cette dernière et perd en intensité...). Les premiers échos utilisaient une technique assez particulière, grâce à une bande magnétique, où ils enregistraient le signal et qu'ils relisaient un peu plus loin sur la bande, ils arrivaient ainsi à reproduire l'écho.

Pour reproduire cet effet, j'utilise un tableau de la taille du retard. Et pour chaque point j'ajoute le point du retard que je divise par l'atténuation et je met à la place mon nouveau point. Ce tableau reste en mémoire pour chaque échantillon.

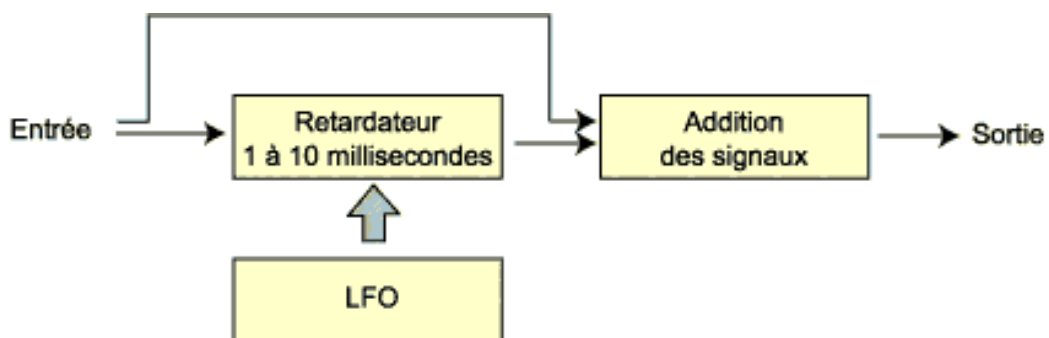


La réverbération (*reverb* en anglais) est une addition d'échos successifs, avec des retards différents, chaque écho faisant l'effet d'une filtration plus ou moins grave ou aiguë. L'effet de réverbération tente de simuler l'ambiance d'une salle dans laquelle le son va rebondir sur plusieurs obstacles (murs, colonnes, etc.). Chaque obstacle étant à une distance différente de l'auditeur, les retards seront multiples. Chaque obstacle étant constitué d'une matière différente (pierre, tissus, bois), le son sera plus ou moins atténué dans les graves ou les aigus avant de parvenir à l'auditeur.

J'ai eu quelques problèmes avec cet effet car il n'est pas très perceptible.

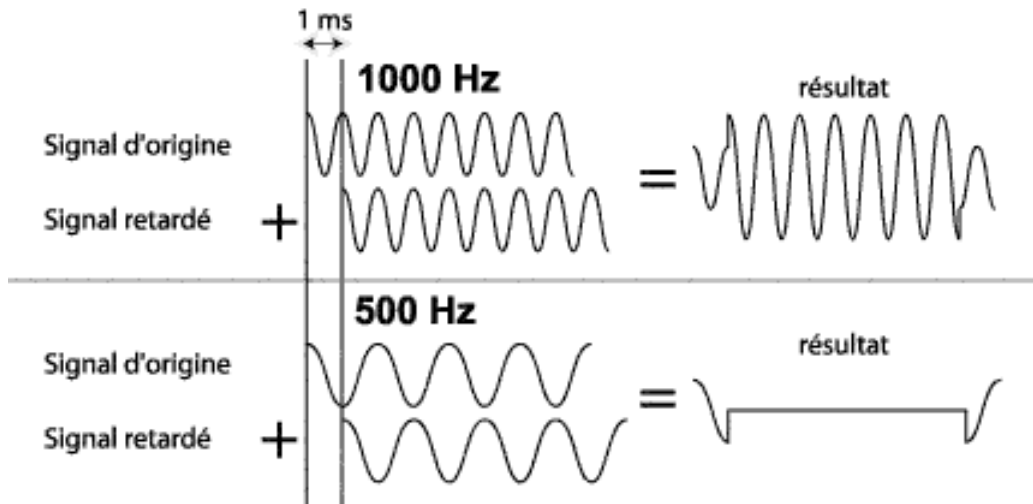


## 5.4 Flanger [Patrice]



A l'écoute, l'effet *flanger* évoque le bruit d'un réacteur d'avion. Il est souvent utilisé sur la guitare électrique ou la voix. Son principe n'est pas très compliqué,

même si le son produit est bigrement sophistiqué. Il s'agit d'un effet retard variable de courte durée (de 1 à 10 millisecondes) qui est simplement mélangé au signal d'entrée. Le graphique ci-dessous illustre la conséquence d'un mélange de ce type :

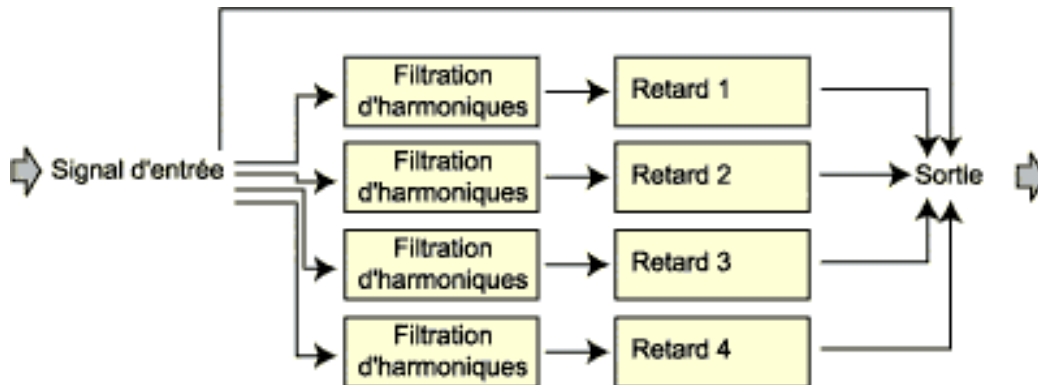


On constate que la fréquence de 1000 Hz, multiple du retard de 1ms choisi pour l'illustration, est amplifiée par ce montage. Les périodes du signal s'additionnent en effet les unes aux autres. La fréquence de 500 Hz est par contre totalement occultée. Bien que cela ne soit pas représenté sur notre graphique, toutes les harmoniques multiples ou sous multiples de 1000 Hz seront affectées (125 Hz, 250 Hz, 500Hz, 1000 Hz, 1500 Hz, 2000 Hz, etc.).

Appliqué à un signal complexe, cet effet va donc modifier profondément la répartition des harmoniques en amplifiant certaines fréquences et en occultant d'autres. Le retard de 1ms étant dans la réalité variable ce qui produit un effet "tournoyant" tout à fait caractéristique et perceptible.

## 5.5 Chorus [Patrice]

Le schéma de principe de l'effet chorus (également appelé effet "ensemble") est tout à fait similaire à celui de l'effet *flanger*. Le rendu en est pourtant assez différent du fait que les retards appliqués en sortie de filtration sont fixes et sont nettement plus importants que pour le *flanger* (environ 30 millisecondes contre 2 à 15 millisecondes).



L'idée est de donner l'impression que plusieurs instruments d'un orchestre jouent simultanément le même morceau, alors que le signal d'entrée ne correspond qu'à un seul instrument.

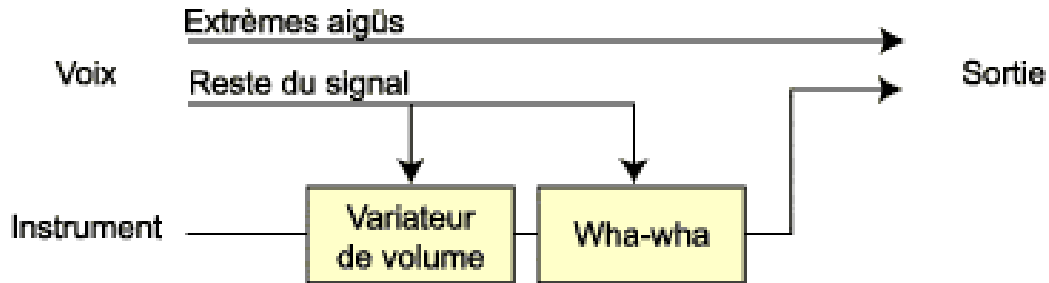
Dans un véritable orchestre, bien que les musiciens tentent de jouer en parfaite synchronisation, ils ne peuvent y parvenir tout à fait. Les légers retards (20 à 60 millisecondes, environ) générés par les modules figurant sur le graphique donnent à l'auditeur l'illusion que plusieurs instruments jouent en "quasi" synchronicité. Les modules de filtration ont pour objectif de différencier légèrement le timbre du signal d'origine, donnant à l'auditeur l'illusion que tous les instruments de l'orchestre ne sont pas exactement identiques. Certains filtres chorus font également varier légèrement la fréquence de chaque voie (effet "pitch") pour rendre compte du fait que les différents instruments ne peuvent pas être accordés exactement de la même manière. On peut enfin modifier légèrement le volume de chaque voie pour tenir compte du fait que tous les instruments ne sont pas à la même distance du micro.

Mon filtre de chorus n'est pas complètement au point, il produit plus un bourdonnement qu'un effet de chorus.

## 5.6 Codevoice [Patrice]

Dans les années 70, le guitariste Peter FRAMPTON a l'idée géniale de loger un minuscule haut-parleur dans sa bouche. Il envoie dans ce haut-parleur le son de sa guitare et articule des paroles mémorables : "do you feeeeeeeeeeeel like we doooooo". Le son de la guitare est modulé par sa bouche et donne l'impression que l'instrument parle. Le résultat ressemble à un super wha-wha.

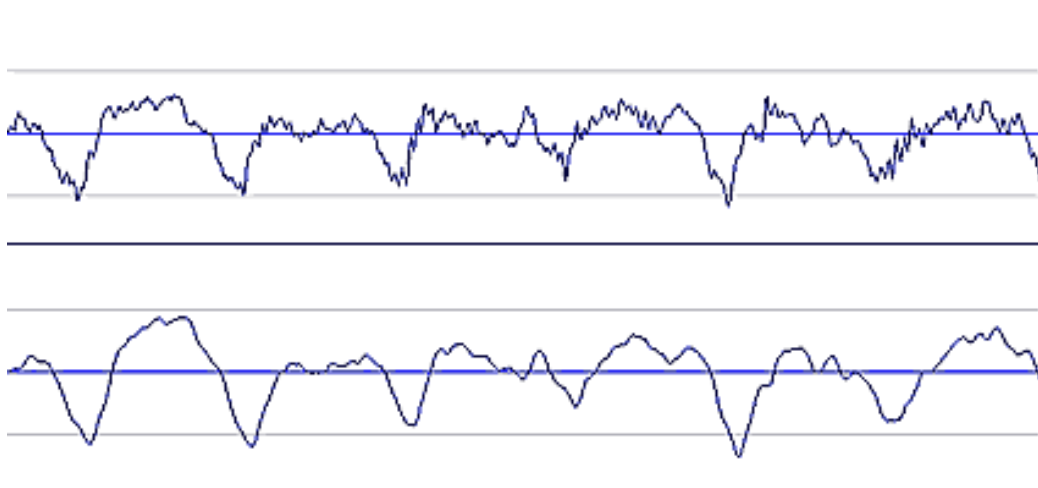
C'est un effet de ce type que permet de produire le *vocoder*. Le signal de la



voix est filtré pour n'en conserver que les extrêmes aigus. Les aigus "portent" en effet la plupart des consonnes et notamment les sifflantes (s, f) et les occlusives (p, t, m, etc.) qui ne peuvent être produites par un instrument. Le reste du signal de la voix sert à moduler le volume et les harmoniques du son d'un instrument.

## 5.7 Noise et Distortion [Patrice]

Ce filtre Noise permet de réduire les bruits de fond, à l'inverse, le filtre Distortion permet d'en rejouer. Leurs fonctionnements sont assez simples. :



*Noise* : Chaque point du signal est la moyenne des 20 points précédents, ce qui a pour effet d'atténuer les grandes variations. Comme on peut le voir sur le schéma.

*Distortion* : On utilise un tableau de distortion qui est composé de nombres aléatoires. Chaque point du signal est alors multiplié par un de ces coefficients,

ce qui produit des petits grésillements et ajoute un effet d'enregistrement avec un micro de mauvaise qualité.

## 5.8 Filtage de la voix [Fabien]

Sinon à part l'interface graphique qui se trouve plus complexe que prévu à mettre en œuvre j'ai commencé un travail sur le filtrage de voix. Il s'agit d'un filtre qui peut supprimer la voix d'une musique (si celle-ci n'est pas trop excentrique !) Il s'agit en fait de filtrer certaines fréquences. Par conséquent on peut aussi aboutir à un autre filtre qui lui supprime la musique. Cependant je tiens à faire remarquer que les algorithmes de détection de la voix sont très complexes et que moi je les ai traités de façon somme toute très basique à savoir que je supprime juste que le spectre de la voix. Le rendu n'en n'est que moyennement appréciable suivant les musiques sur lesquelles il est testé.

# Chapitre 6

## Bibliothèque [Patrice]

### 6.1 Introduction

J'ai commencé à réaliser le plug-in gérant la base de donnée lors de la dernière soutenance. Il permettait d'afficher toutes les musiques de la bibliothèque ainsi que d'y ajouter des musiques. Mon but pour cette soutenance était de finir ce plug-In. J'ai donc ajouté la possibilité de modifier les musiques, ainsi que les supprimer. Pour cela j'utilise le nom du fichier qui est unique pour modifier les fichiers dans la base de donnée. On ne peut donc pas modifier le nom du fichier ce qui est logique car ce nom est unique et ne change pas. Si on renomme le fichier, on peut supprimer l'entrée puis la rajouter. Mais une grande nouveauté est la recherche dans la base de donnée. Par défaut, toute le base de donnée est affichée mais quand on a un grand nombre de musiques ce n'est pas facile de s'y retrouver. J'ai donc ajouté un champ **rechercher** qui permet de rechercher une musique selon son titre, son artiste ou son album. La liste est alors rafraîchie avec seulement les musiques recherchées.

Enfin pour ajouter rapidement des musiques, une nouvelle fonction est apparue : "Scanner". En cliquant sur ce bouton, on peut sélectionner le répertoire où est contenue notre musique. Puis ce répertoire est scanné, c'est à dire que tous les fichiers de musiques sont récupérés puis ajoutés dans la base de donnée. Mais pour chaque fichier, on récupère les tags ID3<sup>1</sup> qui contiennent les informations de la musique comme le titre, l'artiste ou l'album et plein d'autres informations. Seuls ces derniers sont nécessaire pour la base de donnée. J'ai donc écrit un petit plug-in qui s'occupe seulement de récupérer ces informations.

---

<sup>1</sup><http://www.id3.org>

## 6.2 ctag



Ce petit plug-In s'occupe de récupérer les tags ID3 dans les fichiers audio. ID3 est le nom des métadonnées pouvant être insérées dans un fichier audio comme par exemple MP3. Ces métadonnées permettent d'avoir des informations sur le contenu du fichier comme le titre, le nom de l'interprète, ou encore la date de sortie. Il existe plusieurs versions de l'ID3 tag. Tout d'abord la version 1 est composé de 128 octets placés à la fin du fichier.

Elle contient les champs suivants :

- Identifiant "TAG" (3 octets)
- Titre de la chanson (30 octets)
- Nom de l'interprète (30 octets)
- Nom de l'album (30 octets)
- Année de parution (4 octets)
- Commentaire sur la chanson (30 octets)
- Genre musical (1 octet)

Une version v1.1 a été créée pour ajouter le champ numero de piste. Pour cela on a pris 2 octets à la fin du champs commentaire, le premier est mis à NUL par compatibilité avec les anciens lecteurs et le deuxième contient le numero de piste de la chanson.

Enfin, mettre juste les tags Titres, Artistes, Album, Année et Commentaire, cela n'était pas suffisant. Alors une version 2 est apparue, qui contient un nombre impressionnant de champs différents de longueurs différentes. On peut ainsi rentrer tous les artistes d'une chanson, mais aussi les paroles de la chanson ou l'image de la pochette. A la différence de la première version, ces tags sont contenus au début du fichier.

Pour créer le plug-In, j'ai utilisé la librairie Taglib<sup>2</sup>. Cette librairie qui a été créée par le projet KDE est rapide, simple et puissante. Et surtout permet de lire les tags ID3 dans toutes les versions, et permet même de lire les tags des fichiers Ogg Vorbis. Voici quelques exemples d'utilisation :

<sup>2</sup><http://developer.kde.org/wheeler/taglib.html>

```
TagLib::FileRef f("Latex Solar Beef.mp3");  
TagLib::String artist = f.tag()->artist(); // artist == "Frank Zappa"
```

Enfin, à la compilation avec Gcc, il faut ajouter la commande : ``taglib-config -cflags -libs`` qui va simplement ajouter les fichiers à inclure ainsi que les emplacements de la bibliothèque pour *linkage*. C'est cette commande qui m'a posé quelques soucis, car je n'ai pas réussi à la mettre dans le fichier `.pro` du projet. Pour compiler ce projet, on est donc obligé de passer par une commande simple.

Le plug-in *cltag* contient donc trois fonctions, en plus des fonctions obligatoire d'un Plug-in, pour récupérer chacun des 3 champs : le titre, l'artiste, et l'album.

# Chapitre 7

## Conclusion générale

### 7.1 Ce qu'il reste à faire

Nous poursuivons sur notre lancer et sommes dans les temps. Notre projet a bien avancé, cela commence enfin à ressembler à quelque chose.

Bien sûr, notre travail n'est pas encore terminé, il nous reste à paufiner le visuel de notre logiciel, à réaliser le "scratch" ainsi que les documentations et l'installateur.

### 7.2 Conclusion

Notre projet est sur de bons rails, nous sommes dans les temps. Il s'agit là d'une enrichissante expérience au sein de l'EPITA pour ce deuxième projet. Nous apprenons les uns des autres, mais également sur nous même. Ce projet nous permet de mieux appréhender nos propres limites ce qui sera bénéfique pour notre futur.

La prochaine soutenance sera la finale, et nous serons fins prêts pour présenter notre projet duement terminé et fonctionnel.