Real-time digital audio signal processing with a STM32F746 Discovery board (v1.1)

S. Reynal

SIA - Fall 2021

1 Introduction : déroulement et évaluation

Ce mini-projet de 16h a pour objet l'étude de quelques aspects de la chaîne d'acquisition, de traitement et de restitution du signal audio en temps réel sur carte STM32F746-Discovery. L'objectif final est d'implémenter un effet audio temps réel (réverbération, compresseur dynamique, etc).

Répartition conseillée des 16h:

- 2h : présentation de la carte (CODEC, LCD, SDRAM) et du code fourni par l'encadrant
- 2h : prise en main du code STM32F746 fourni (compréhension de l'organisation du code, compilation et exécution, tests des fonctionnalités de base, modification de quelques paramètres audio et d'affichage) ; réalisation d'un écho simple.
- 4h : implémentation d'une FFT et affichage du spectrogramme du son capté par les microphones de la carte.
- 8h : implémentation d'un effet audio au choix avec édition des paramètres via l'écran tactile.

L'évaluation repose sur deux éléments :

- une démonstration d'une durée de 5 minutes, qui a lieu à la fin des 16h ;
- un rapport de mini-projet de 10000 signes (espaces compris) à remettre sur Moodle au plus tard le 24 janvier à midi.

2 Documentation utile

- La documentation de la carte STM32F746 Discovery : `https://www.dropbox.com/s/olvokchkjx7kv86/F746GDISCOVERY%20user%20manual.pdf?dl=0`

- STM32 DSP coding guidelines and CMSIS library, description des divers formats numériques (Q15, Q31 and floats): `https://www.st.com/resource/en/application_note/dm00273990-digital-signal-processing-for-stm32-microcont` `pdf` ainsi que le code de démonstration correspondant `https://www.dropbox.com/sh/kjdqlcsofcidh66/AAC2OoCopLxJzLsWAZ3CIsg3a?dl=0`
- Vous ferez une utilisation extensive des fonctions de la bibliothèque CMSIS-DSP :`https://www.keil.com/pack/doc/CMSIS/DSP/html/index.html`
- RTOS : `https://www.st.com/resource/en/user_manual/dm00105262-developing-applic` `pdf` et `https://www.keil.com/pack/doc/CMSIS/RTOS/html/index.html`

3 Quelques pistes pour démarrer (première séance)

Cette section propose quelques pistes à explorer et questions à investiguer lors de la première séance.

3.1 Quelques questions à se poser... pour savoir estimer rapidement les performances d'un système audio temps-réel

- Que dit le théorème de Shannon ?
- Pour du traitement de la parole, quelle fréquence d'échantillonage préconiseriez-vous ? Idem pour du signal de musique ?
- Qu'est-ce que le "double-buffering" ? Qu'est-ce que le DMA transfer ? Comment est implémenté le double-buffering dans le code fourni (rôle des IRQ Handler notamment, cf. audio.c) ?
- A 48kHz de fréquence d'échantillonnage, de combien de temps dispose-t-on pour traiter **un** échantillon audio ? A titre de comparaison, quelle est la durée moyenne d'une instruction (par exemple une addition de deux registres) sur un processeur STM32F7 cadencé à 200MHz ?
- La **latence audio** est définie comme le temps séparant l'arrivée d'un échantillon sur l'entrée audio et sa restitution sur la sortie casque (pour fixer les idées, imaginons un effet audio dont la fonction de transfert est simplement $T(z) = 1$) : donnez-en une estimation en fonction de `AUDIO_BUF_SIZE` et de la fréquence d'échantillonnage.
- Quelle est la dynamique (à estimer en dB) offerte par une quantification sur 16 bits (`int16_t`) ? quelle dynamique offre a contrario un encodage sous forme de `float` (32 bits au format IEEE) ?
- Que représentent les formats `q15_t` et `q31_t` ? Ils sont notamment utilisés dans la bibliothèque DSP-CMSIS (FFT, filtres, etc)

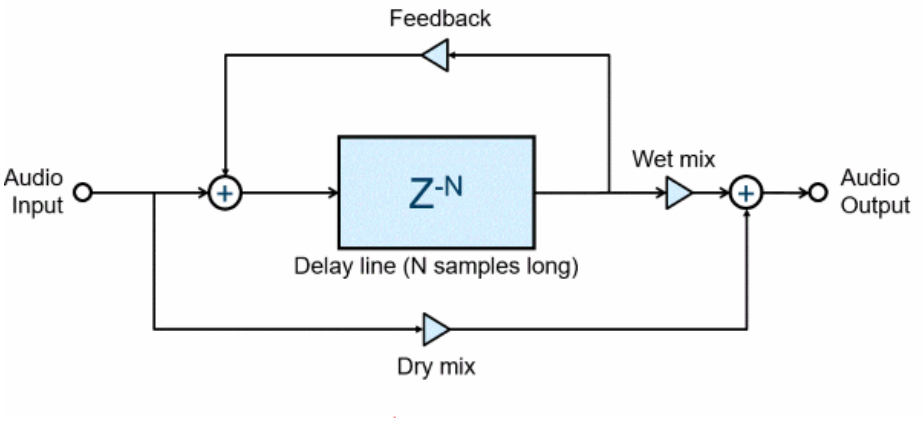

Figure 1: Schéma-bloc du "simple delay" constitué d'un bloc de retard de N échantillons. Faute de place en RAM (l'essentiel de la RAM étant occupé par le frame buffer de l'écran LCD), il sera nécessaire d'implémenter ce retard dans la SDRAM externe dès lors que N dépassera quelques milliers d'échantillons (soit au-delà de 20ms).

3.2 Tester l'influence de quelques paramètres audio

- En modifiant le paramètre `AUDIO_BUF_SIZE` (cf. préambule du fichier audio.c), estimez l'influence de la taille du buffer DMA sur la latence **perçue**.
- Quel est l'influence de la fréquence d'échantillonage sur la qualité perçue ? (on se reportera au fichier source main.c, ligne 858, où l'on peut modifier le paramètre `hsai_BlockA2.Init.AudioFrequency`)
- Quel est l'effet de la quantification sur la qualité perçue ? (comme le format est fixé à 16 bits par échantillon, on cherchera une méthode permettant de réduire artificiellement le nombre de bits de quantification)

3.3 Un premier algorithme d'effet audio : "the simple delay"

Il s'agit d'un effet qui réalise une fonction d'écho dont la période ("d"), le feedback ("fb", taux de ré-injection) et le taux de mélange dry/wet sont paramétrables. Son implémentation est illustrée par le schéma-bloc figure 1

Quelques pistes pour démarrer :

- En notant $x[n]$ l'échantillon d'entrée à l'instant n et $y[n]$ l'échantillon de sortie au même instant, donner la ou les équation(s) décrivant cet algorithme.
- Tracer la réponse impulsionnelle de cet effet pour comprendre comment il fonctionne.

Pour l'implémentation, il vous faudra utiliser :

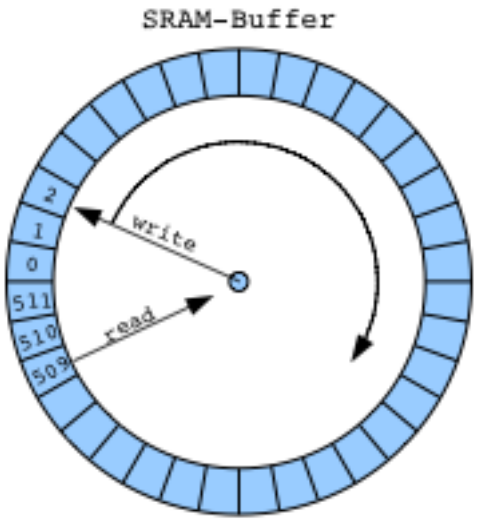

Figure 2: Un ring buffer (ici un buffer de taille 512) permet d'implémenter aisément un retard z^{-d} jusqu'à concurrence de la taille du buffer, grâce au "rebouclage" du buffer sur lui-même. Dans l'illustration, on écrit par exemple un échantillon audio à la position 2 et on lit à la position 509 qui se trouve d=5 échantillons en arrière chronologiquement. Le "rebouclage" peut être facilement implémenté par l'opérateur modulo.

- un buffer audio de grande taille qui permet de stocker des échantillons sur une longue durée (au moins "d") ; c'est le rôle que jouent le "scratch" et les fonctions readFromAudioScratch() et writeToAudioScratch() définies dans audio.c et qui permettent d'écrire dans la SDRAM externe (d'une capacité 64Mbits), la RAM interne ne disposant pas de suffisamment d'espace ;
- un "ring buffer"[1], c'est-à-dire un mécanisme permettant d'accéder à un buffer audio de taille finie sans jamais sortir des bornes, en rebouclant en l'occurence tout dépassement d'index à l'intérieur du buffer comme l'illustre la figure 2 ;
- un index implémenté sous forme de variable globale et indiquant à chaque instant à quelle position sera écrit le prochain échantillon d'entrée "x" ; cet index est incrémenté à chaque nouvel échantillon.

[1] On dit aussi "circular buffer"

3.4 Analyse expérimentale des performances

- à l'aide de la pin D13 du header ARDUINO (qui se trouve connectée électriquement à la LED verte de la carte), il est possible de mesurer à l'oscilloscope le temps consommé par un bloc de code, par exemple en faisant passer D13 au niveau haut puis au niveau bas respectivement à l'entrée et à la sortie du bloc et en affichant le signal disponible sur D13 sur un canal de l'oscilloscope (pensez à régler la synchronisation sur ce canal). Les fonctions `LED_on()` et `LED_off()` sont disponibles à cet effet (un exemple est déjà proposé dans l'implémentation par défaut de `processAudio()` dans le fichier audio.c).
- Tester l'influence de l'option "Optimized for speed" du compilateur (Properties →C/C++ Build →Settings →Tool Settings →MCU GCC Compiler →Optimization)
- Implémenter votre algorithme d'effet audio en format "float" puis en format "q15_t" et comparer.

4 Séance 2 : afficher un spectrogramme du son ambiant en temps réel

4.1 Utilisation des fonctionnalités de CMSIS-RTOS

RTOS[2] est un "mini" système d'exploitation dédié aux microcontrôleurs et orienté "temps réel" (en particulier il est préemptif : même si le processeur est en train d'exécuter des instructions d'un IRQ Handler, il peut être "préempté" par l'OS, c'est-à-dire interrompu par une tâche encore plus prioritaire, ce qui autorise une granularité fine des tâches, et une réactivité élevée du système).

Pour ce qui nous concerne ici, RTOS sera utilisé pour gérer des threads de manière totalement automatique, c'est-à-dire sans avoir à réaliser manuellement dans le code la répartition de la charge du processeur et l'ordonnancement des différentes tâches (ce qui constitue le rôle du *scheduler* de tâches de RTOS). Le système RTOS gère cette répartition notamment en fonction de la priorité affectée à chaque tâche.

Pourquoi utiliser RTOS ? Pour comprendre pourquoi il peut devenir indispensable d'utiliser un OS pour réaliser le *scheduling* des tâches, il suffit de réaliser une opération simple qui va pousser le système dans ses retranchements.

Dans audio.c, ligne 115 (environ) se trouve la fonction audioLoop() qui contient la boucle audio infinie `while(1)` : à l'intérieur de cette boucle a lieu, une fois sur vingt, un appel d'une fonction graphique, en l'occurence `uiDisplayInputLevel()`. Vous allez tout simplement ajouter des instructions gourmandes en temps dans cette dernière fonction, par exemple, une boucle `for` contenant des calculs trigonométriques (ou toute opération gourmande en temps, la fonction réalisée important peu ; il est ainsi tout-à-fait envisageable d'appeler une vraie fonction graphique de `disco_lcd.c`). Cela permettra de simuler un appel graphique très consommateur de temps de calcul.

[2] Il s'agit ici du portage CMSIS-RTOS de FreeRTOS vers les processeurs STM32.

Maintenant, augmentez le nombre d'itération de la boucle `for` jusqu'à ce que des artefacts audio commencent à apparaître (des clics) : il s'agit du seuil au-delà duquel le temps cumulé de toute les opérations à l'intérieur de la boucle `while` dépasse le temps permis pour traiter une trame audio DMA. Dans ce cas, une fois sur vingt, le processeur "saute" une trame audio DMA (il passera directement à la suivante, la trame courant ayant été écrasée avant qu'il n'ait pu commencer à la traiter), et ces sauts auront un effet clairement audible à cause des discontinuités de signal (le vérifier). Remarque : il peut être utile de désactiver le cache pour cette expérience (main.c:236), le cache ayant la fâcheuse tendance à ... optimiser les boucles `for` !

Comment résoudre le problème précédent ? On pourrait être tenté de découper la fonction `uiDisplayInputLevel` en plusieurs fonctions plus "courtes", par exemple `uiDisplayInputLevel1`, `uiDisplayInputLevel2`, etc, qu'on appellerait à tour de rôle à chaque itération de la boucle `while`. C'est une solution, mais c'est fastidieux à coder, et c'est là que RTOS entre en jeu : il va réaliser ce travail de découpage automatiquement, en interrompant tout simplement l'exécution de `uiDisplayInputLevel` lorsqu'une tâche plus prioritaire demande à être exécutée, ce qui sera le cas de la boucle audio.

En pratique, nous aurons deux tâches (ou "threads") de priorité différente, `DefaultTask` et `uiTask` (le support de RTOS est déjà intégré dans le code fourni en exemple et ces deux tâches ont déjà été crées dans le configurateur de Cube) :

- la première a une priorité élevée, elle sera dédiée à l'audio,
- la seconde, moins prioritaire, gérera l'affichage graphique.

La communication entre les tâches et les IRQ handler (notamment le DMA IRQ Handler de la SAI) repose sur l'utilisation de signaux, cf. `https://www.keil.com/pack/doc/CMSIS/RTOS/html/group__CMSIS__RTOS__SignalMgmt.html`. Les deux fonctions utiles seront : `osSignalWait (signal, ...)`, qui permet à une tâche d'attendre (en "bloquant") un signal provenant d'une autre tâche (ou d'un IRQ Handler) ; et `osSignalSet(task, signal)`, qui permet d'envoyer un signal à une tâche donnée, et de la "débloquer" en quelque sorte.

En pratique :

- La tâche `DefaultTask` dédiée à l'audio attendra un signal provenant de l'IRQ Handler du DMA (`HAL_SAI_RxCpltCallback`) et l'informant qu'une nouvelle trame DMA est prête, pour appeler `processAudio()` ;
- La tâche `uiTask` pourra, au choix, mettre à jour l'affichage de manière régulière (avec un timer), ou bien en réponse à un signal l'informant que l'affichage nécessite un rafraîchissement (ce signal pouvant être généré depuis n'importe quelle fonction).

Ainsi, la tâche dédiée à l'audio étant prioritaire, tout signal la "débloquant" la fera immédiatement passer au premier plan de l'exécution, même si la tâche graphique est encore en train de travail à rafrachir l'affichage.

A faire :

- Commenter l'appel `audioLoop()` dans `main.c:241` : ceci à pour effet d'autoriser l'exécution des lignes suivantes qui activent RTOS ;
- Dans `startDefaultTask()`, appeler une version modifiée de `audioLoop()` dont on aura éliminé les appels graphiques ;
- déplacer ces appels graphiques dans `startUITask()` ;
- insérer à une place judicieuse dans chacune de ces deux fonctions une attente de signal via un appel de `osSignalWait()` ;
- insérer les générateurs de signaux via `osSignalSet()` dans les fonctions appropriées : par exemple, dans `HAL_SAI_RxCplt/HalfCallback` pour l'audio...

Vous pouvez désormais reprendre votre test "push the system to the limit", et constater que désormais, l'audio n'est plus perturbé et que les clics ont disparu ! (néanmoins, comme tout a un prix, le rafraîchissement graphique peut sembler plus saccadé...)

4.2 Calcul en temps réel de la TF d'une trame audio et affichage

L'objectif de cette partie est d'afficher le spectrogramme du son ambiant capté par les microphones embarqués sur la carte. Il s'agit donc, pour chaque trame audio, d'en calculer la TF et d'en afficher le module de façon glissante (c'est-à-dire d'afficher le module de chaque coefficient sous forme d'une ligne verticale dont l'abscisse augmente d'un pixel pour chaque trame, à la façon d'un spectrogramme sous matlab, cf. figure 3).

Deux pistes à explorer et dont on pourra par exemple comparer les performances :

- Implémenter manuellement l'algorithme FFT (dit "papillon") ; les implémentations ne manquent pas, mais on tentera de trouver une version dédiée aux signaux réels, qui offre de meilleure performance pour l'audio ;
- Utiliser la bibliothèque CMSIS-DSP, cf ci-dessous.

Comment utiliser CMSIS-DSP ? CMSIS-DSP est une bibliothèque pour Cortex ARM dédiée au traitement du signal (cf. liens vers la documentation afférente en début de ce document). Elle contient une centaine de fonctions mathématiques ou pour le traitement du signal (filtres, FFT, convolutions, etc) implémentées à l'aide d'instructions spécifiques, comme MAC (Multiply and Accumulate) permettant de réaliser l'opération

$$z = z + x.y$$

Pour calculer une transformée de Fourier rapide sur un signal réel (ce qui est en général le cas d'un signal audio), il est optimal d'utiliser la fonction `arm_rfft_fast_f32()`. La documentation de CMSIS-DSP indique les étapes à suivre, notamment l'initialisation d'une structure `arm_rfft_fast_instance_f32`, etc. Le résultat de la transformée étant un tableau de complexes, il faudra encore le convertir en tableau de modules en utilisant par exemple `arm_cmplx_mag_f32()`.

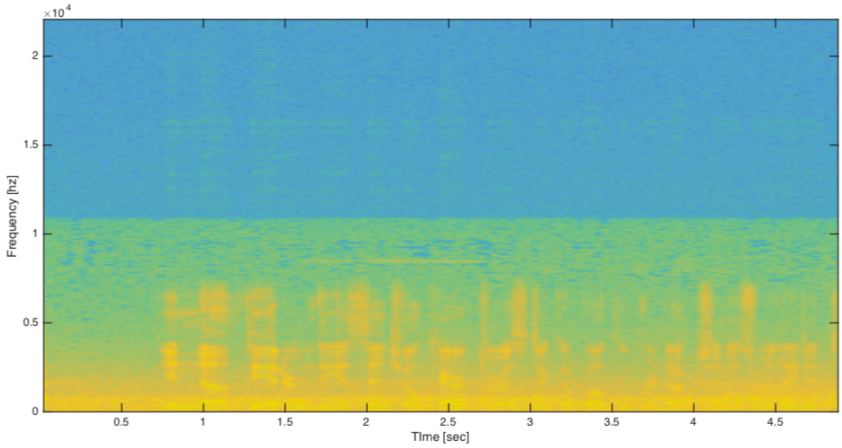

Figure 3: Spectrogramme typique d'un signal audio de parole.

Affichage sur l'écran LCD Vous pourrez utiliser l'ensemble des fonctions contenues dans le fichier `disco_lcd.c`, en particulier `LCD_DrawPixel()`. Attention, si vous utilisez RTOS (ce qui est vivement conseillé), toutes les fonctions graphiques devraient être placées dans le corps de la tâche `startUITask()`, cette dernière étant moins prioritaire que la tâche `startDefaultTask()` chargée du traitement audio. On garantit ainsi que c'est bien le traitement des trames audio qui est affecté en priorité au processeur, le processus d'affichage occupant le temps restant.

5 Séances 3 et 4 : implémenter un effet temps réel

Les séances restantes sont consacrées à l'implémentation **au choix** d'un effet audio **parmi** :

- Compresseur dynamique
- Réverbération
- Vocoder de phase (algorithme permettant de réaliser du "pitch shifting", c'est-à-dire d'altérer la hauteur tonale d'un son, d'une voix, etc sans en changer la durée)

Ces trois algorithmes sont indiqués par ordre de difficulté croissante.

Je fournis ci-après quelques pistes d'exploration qu'il s'agira d'approfondir de votre côté.

5.1 Compresseur dynamique

Si vous choisissez de travailler sur le traitement de la dynamique du signal, de nombreux algorithmes sont à votre disposition, qui ont été mis au point au cours de la longue histoire (plus d'un demi-siècle) du *mastering*, activité d'ingénierie sonore consistant à adapter la dynamique du signal audio musical au support sur lequel il sera enregistré (des disques 33 tours microsillons au streaming actuel, la dynamique a subit une forte augmentation, d'un petit cinquantaine de dB à près de 100dB). Parmi ces effets, le "noise-gate" et le compresseur-expandeur, qui vous sont présentés ci-après.

5.1.1 Réalisation d'un noise-gate

Il s'agit habituellement de la première étape de traitement de la dynamique d'un signal audio enregistré en condition réelles, c'est-à-dire avec du bruit ambiant. Le noise-gate est un effet qui coupe (=réduit au silence) les parties du signal se trouvant en-dessous d'un certain niveau, dans le but de supprimer le bruit. L'idée clé est qu'en présence de signal utile (parole ou musique), le bruit est généralement masqué (cf. cours psychoacoustique sur le masquage fréquentiel), et l'auditeur ne le perçoit pas ; en revanche il est clairement audible lors des plages de silence musical ou de pause entre deux phrases. Lorsque le signal sera mélangé à d'autres signaux lors de l'étape de mixage, il est souvent primordial (sauf raisons artistiques particulières) que le moins de bruit ambiant subsiste.

On dit que la porte (gate) est ouverte lorsque le signal passe tel quel au travers du traitement ; qu'elle est fermée si le signal est réduit (ou atténué).

Réalisation naïve Il s'agit simplement d'atténuer d'une quantité fixée (paramètre `attenuation` en dB) les échantillons se trouvant sous un seuil donné (paramètre `threshold`). Tester l'algorithme pour différents seuils et différentes atténuations et critiquer.

Réalisation avancée Une noise gate améliorée dispose de paramètres dynamiques supplémentaires (cf. figure 4) :

- attack time : c'est le temps que met la porte à s'ouvrir lorsque le signal dépasse le seuil
- holding time : c'est le temps minimal pendant lequel la porte reste ouverte (même si le signal passe sous le seuil) ; ceci permet notamment d'éviter les effets de hachage, la porte ne pouvant pas se fermer tant que ce temps n'est pas écoulé.
- release time : c'est le temps (supérieur au précédent, donc) que met la porte à se fermer lorsque le signal passe sous le seuil (à condition que le holding time ait été atteint).

Pour faciliter l'écriture de l'algorithme, je vous suggère d'utiliser un programmation par état en utilisant une variable `state` valant le cas échéant :

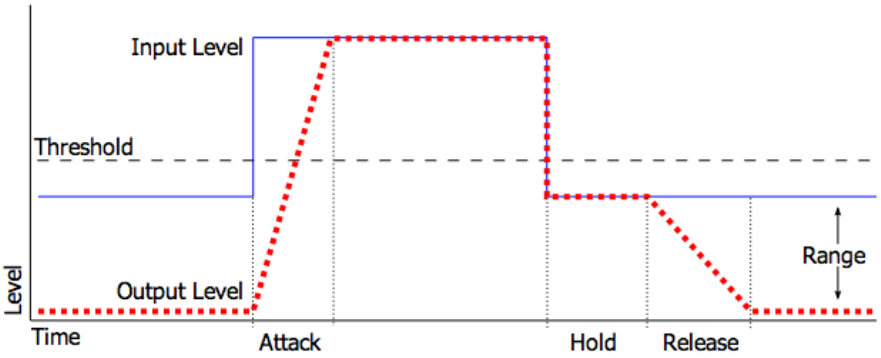

Figure 4: Paramètres de la noise-gate (le paramètre ”range” représente l'atténuation).

- CLOSED : porte fermée
- OPENING : le signal vient de passer le seuil, la porte est en train de s'ouvrir (prise en compte de l'attaque)
- OPEN : la porte est ouvre
- CLOSING : le signal vient de passer sous le seuil, la porte est (si le hold time est respecté) en train de se fermer (prise en compte du release time).

Vous choisirez des pentes linéaires (cf. figure 4) pour l'ouverture et la fermeture (des pentes exponentielles sont généralement plus esthétiques, mais aussi plus compliquées à programmer). Dans ce cas, un simple compteur suffira pour l'ouverture et la fermeture.

5.1.2 Réalisation d'un compresseur dynamique

Afin d'optimiser le nombre de bits utilisés pour numériser le signal, il est souvent intéressant de rehausser les faibles niveaux, ou, alternativement, de réduire les forts niveaux. Cette opération est appelée *compression dynamique*, et consiste à appliquer la courbe de gain de la figure 5 au signal d'entrée.

Un autre intérêt du compresseur est esthétique : il permet, en réduisant la dynamique globale du signal, de faire ressortir certains passages ou instruments qui seraient à peine audible autrement. Il donne également une impression de ”lissage” qui, dans certains styles musicaux, est appréciée.

Les paramètres d'un compresseur dynamique sont :

- compression ratio : la réduction de gain au-dessus du seuil
- threshold : le seuil (en dB sur le signal d'entrée) au-dessus duquel le signal est compressé.

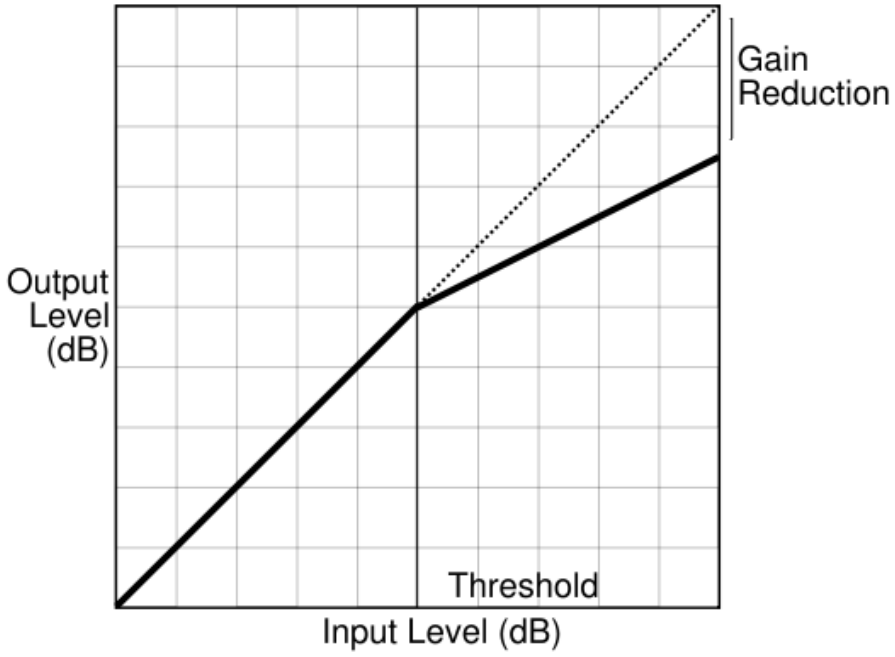

Figure 5: Le compresseur dynamique atténue tous les signaux dont l'amplitude est supérieure à un certain seuil (threshold paramètre). Ici le rapport de compression (compression ratio) vaut 2:1.

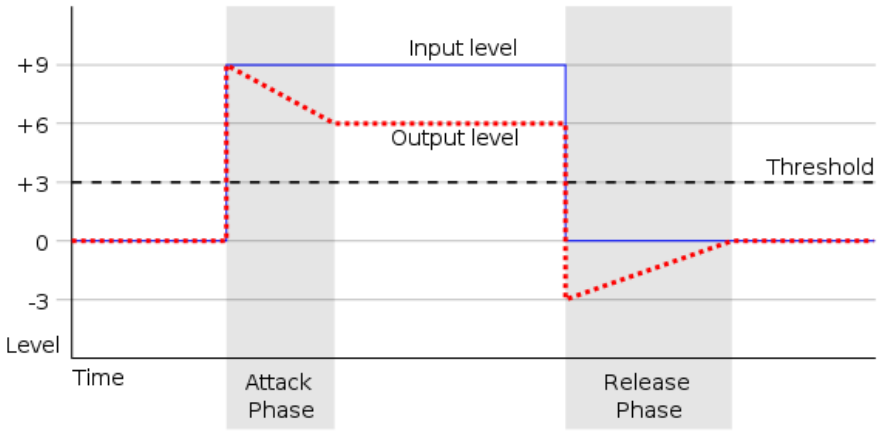

Figure 6: Le changement de gain n'est pas immédiat, mais suit au contraire une enveloppe similaire à la noise-gate (attack/release).

- attack : le temps nécessaire au compresseur pour se déclencher
- release : le temps nécessaire, lorsque le signal repasse sous le seuil, pour que le gain reprenne sa valeur unitaire.

Ces quatre paramètres standards sont parfois complétés d'un paramètre "hold" (comme pour la noise-gate), d'un paramètre "knee" adoucissant la courbe de compression afin que le compresseur rentre plus ou moins progressivement en action, et enfin de la possibilité de faire agir un filtre passe-bande sur le signal de compression (cf. de-esser).

Remarque : un compresseur avec un taux de compression infini, est appelé un **limiteur**.

Le fonctionnement standard d'un compresseur est le suivant :

- le niveau RMS du signal d'entrée est mesuré régulièrement
- lorsque ce niveau dépasse le seuil, le gain est réduit à sa valeur donnée par le taux de compression
- cette réduction de gain devient effective au bout d'un temps donné par le paramètre d'attack
- lorsque le niveau RMS du signal passe sous le seuil, le gain revient à sa valeur initiale (avec une courbe temporelle dépendant du paramètre "release")

Le paramètre "make up gain" est un paramètre additionnel, permettant après compression de redonner au signal une amplitude telle qu'il est perçu avec la même intensité que le signal non compressé. Il s'agit simplement d'une amplification additionnelle.

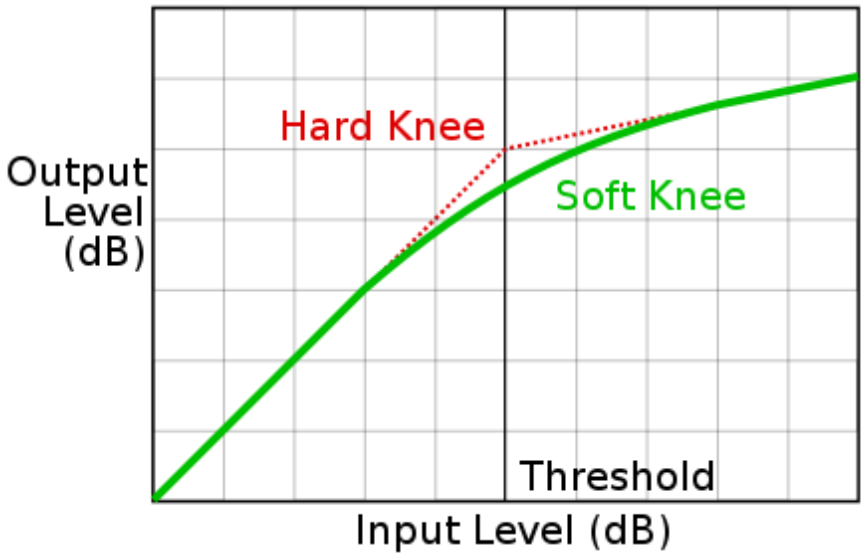

Figure 7: Un compresseur à Soft Knee voit le compresseur entrer progressivement en action. La compression est plus douce.

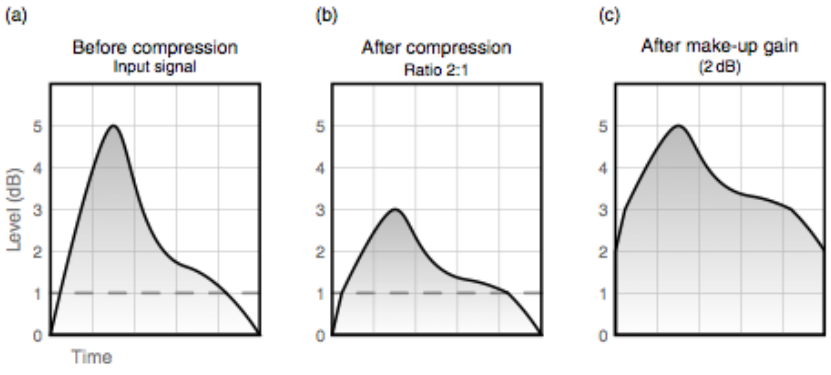

Figure 8: Le make-up gain permeant de redonner un niveau acceptable au signal.

5.2 Réverbération

Il s'agit implémenter un algorithme de spatialisation à partir de délais et de réverbération. L'objectif est de donner une impression subjective de spatialisation, comme si la voix avait été enregistrée dans une pièce à l'acoustique réverbérante.

Reprendre l'implémentation du délai de la séance 1 en testant cette fois la superposition de plusieurs (jusqu'à 5) délais dont les paramètres sont très différents entre eux (quel est l'effet de la congruence des durées ?).

L'implémentation de la réverbération repose elle sur l'algorithme de convolution par transformée de Fourier. Des réponses impulsionnelles se salles typiques sont fournies sur mon site (fichiers SDIR). La sortie de la réverbération est obtenues en réalisation la convolution du signal d'entrée $x(n)$ avec la réponse impulsionnelle de la salle $h(n)$: $y = x * h$. Pour accélérer le calcul (qui est proportionnel au carré de la longueur N de la réponse impulsionnelle), on passe dans le domaine fréquentiel : $Y = X \cdot H$. Il est utile de calculer la transformée de Fourier de h une fois pour toute, et celle de x, par FFT (algorithme en temps $N \ln N$). On pourra alors se référer à l'implémentation de **convolution hybride (ou partielle)** proposée dans l'article historique suivant : `https://cse.hkust.edu.hk/mjg_lib/bibs/DPSu/DPSu.Files/Ga95.PDF`

Remarque : attention au choix du fenêtrage (hamming, blackman, hanning,...) lors de l'application de la FFT. Attention également au fait que le signal réverbéré est plus long que le signal original (de N exactement).

5.3 Vocoder de phase

Il s'agit d'un algorithme permettant de réaliser deux opérations non-triviales sur un signal audio :

- Modifier la hauteur tonale d'un signal (voix, instrument...) sans en modifier la durée
- Inversement, modifier la durée (le rythme, le tempo) d'un signal sans en modifier la hauteur tonale

Expliquer pourquoi ce n'est pas trivial ! Que se passe-t-il si on se contente de "relire" un signal avec une fréquence d'échantillonage différente de celle à laquelle il a été enregistré ?

5.3.1 Re-synthèse d'un signal par TFCT inverse

Le vocoder de phase s'appuie sur l'analyse par TFCT et la re-synthèse audio par TFCT inverse. Il faut commencer par implémenter une fonction calculant le signal audio résultant de la TFCT inverse du tableau contenant les spectres complexes d'une trame audio (opération inverse de celle demandée à la séance 2). Il est conseillé d'utiliser la FFT inverse rapide de la bibliothèque CMSIS-DSP.

Il s'agit ensuite de réaliser l'analyse et la re-synthèse sur des trames "glissantes", c'est-à-dire qui se recouvrent partiellement :

- chaque trame (précédemment multipliée par une fenêtre *hanning*) possède idéalement une longueur `L=60 ms`
- le pas d'avancement `step` est égal à 1/3 de la longueur de la trame soit 20 ms environ

5.3.2 Dilatation temporelle sans changement de hauteur tonale par vocodeur de phase

Pour modifier la durée du signal d'une valeur `alpha=0.6` (ce qui correspond à un étirement), on utilisera l'algorithme du vocoder de phase tel que présenté en cours (`http://www-reynal.ensea.fr/docs/audio-sia/cours/diapos_cours_traitement_signal_audio.pdf`, à partir de la diapo 116).

On reconstruira la phase selon trois approches différentes (de la moins à la plus fidèle) :

- Solution 1 : la phase est gardée constante
- Solution 2 : la phase est incrémentée selon un modèle théorique
- Solution 3 : la phase est incrémentée selon la fréquence instantanée