TP 6 : Traitement des signaux audio

Notions : Buts : Allocation dynamique de tableau, pointeurs

Ce TP nécessite de se munir d'un casque ou d'écouteurs audio.
Les fonctions des sections 3.1 à 3.3 au moins doivent être tapées sur machine avant la séance, dans le fichier `son.c`, au moyen d'un éditeur de texte

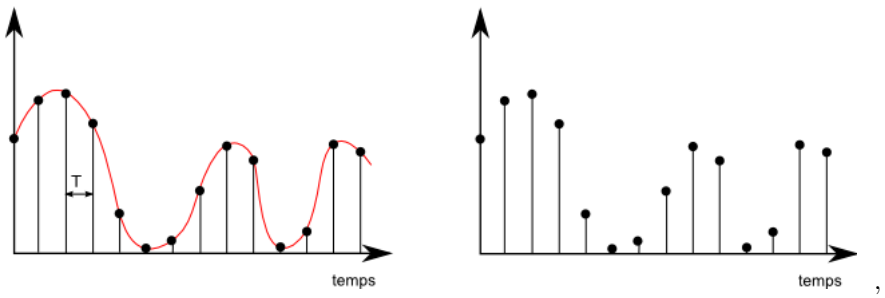
1 Le signal acoustique

Un son est produit par la vibration d'un corps provoquant des ondes qui se propagent dans l'air. Un exemple est la vibration d'une membrane de tambour lorsqu'on la percute. Si la vibration aérienne est composée de fréquences dans la gamme de 20Hz à 20kHz, et qu'elle est d'intensité (amplitude) suffisante, alors elle peut être captée par l'oreille humaine : c'est un son.

Un signal acoustique (ou signal sonore) analogique est un signal continu qui représente un son. Lorsqu'on amplifie un tel signal pour exciter un haut parleur, la membrane du haut parleur oscille en suivant ce signal. Cette vibration est transmise à l'air et produit un son qui se propage jusqu'à l'oreille : on l'entend.

1.1 Signal acoustique échantillonné

Pour enregistrer un signal sonore sur support numérique (monde discret), ce signal est **échantillonné**, c'est à dire qu'une valeur est enregistrée toutes les T secondes (figure 1).



(a) son en rouge, période T (b) signal précédent échantillonné

FIGURE 1 – Principe de l'échantillonnage

Pour un signal sonore numérique, si la fréquence d'échantillonnage $f_s = 1/T$ est suffisamment grande [1], le son est restitué sans distorsion perceptible.

Comme l'oreille est sensible jusqu'à 20 KHz environ, l'industrie a adopté une fréquence d'échantillonnage standard supérieure à 40 KHz. La plus couramment utilisée, par exemple sur les CD audio, est 44.1 KHz.

C'est la fréquence d'échantillonnage qui sera utilisée dans ce TP (jusqu'à la partie 3.5 optionnelle).

1. D'après le théorème de Shannon : si la fréquence d'échantillonnage est supérieure au double de la fréquence la plus haute présente dans le signal analogique, alors le signal échantillonné décrit sans perte le signal analogique.

1.2 Représentation des échantillons d'un signal acoustique échantillonné

Dans un signal sonore échantillonné, chaque échantillon est stocké avec une certaine précision. Sur un CD, par exemple, les échantillons sont stockés au format entier sur 16 bits (`short`, 65536 valeurs possibles pour chaque échantillon).

Pour simplifier ce TP, un son sera représenté sur 8 bits par un tableau d'octets (`char`). Chaque élément du tableau représente l'intensité d'un échantillon du signal sonore avec une valeur comprise entre -128 et +127 ($-2^7..2^7-1$).

Le tableau correspondant au signal échantillonné de la figure 1 est illustré table 1 :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
t[i]	12	25	27	17	-18	-28	-25	-10	10	2	-30	-24	10	8

TABLE 1 – Représentation du signal présenté figure 1

Dans ce TP, un seul canal sera considéré (signal monophonique uniquement). Les fichiers audio utilisés sont des fichiers .wav (WAVeform audio format).

2 Code fourni

Pour lire un fichier audio, écouter un signal sonore, visualiser un signal sonore, nous utiliserons la bibliothèque SDL à laquelle nous avons ajouté quelques fonctions pour simplifier ces manipulations.

Les fonctions suivantes sont dans la bibliothèque `SDL_phelma` :

- `int SDL_PH_PlaySoundWav (char *filename)` : envoie le fichier audio au format wav de nom `filename` sur la carte son du PC. Permet d'écouter directement un fichier sonore sur votre casque.
- `char * SDL_PH_GetCreateWavS8 (char *filename, int *p_freq, int *p_nb_samples)` : charge le signal contenu dans le fichier audio nommé `filename` dans un tableau de `char`. La fréquence d'échantillonnage du signal est stockée dans `*p_freq` et sa taille dans `*p_nb_samples` (notez les passages par adresse). Le tableau contenant les échantillons lus dans le fichier est alloué dynamiquement par la fonction au moyen de la fonction C `calloc()`. La fonction retourne l'adresse de ce tableau.
- `int SDL_PH_PlaySoundS8 (char *son, int nb_samples, int freq)` : envoie le signal sonore contenu dans le tableau d'échantillons `son`, de taille `nb_samples`, sur la carte son du PC, à la fréquence d'échantillonnage `freq`.
- `int SDL_PH_PlayDrawSoundS8 (char *son, int nb_samples, int freq, SDL_PHWindow *f, int coul)` : identique à la fonction précédente avec, en plus, l'affichage d'un graphique traçant le signal sonore dans la fenêtre `f` avec la couleur `coul`.
- `void SDL_PH_PrintSoundS8 (char *son, int nb_samples)` : affiche, dans le terminal, tous les échantillons du tableau d'échantillons `son`, de taille `nb_samples`. Bien utile pour débugger...

Sur le site Web des tp, nous vous fournissons :

- les fichiers `test1.c`, `test2.c` et `test3.c` (plus `test4.c` et `test5.c` pour la partie optionnelle) contenant chacun un programme principal `int main()`, qui permet de tester vos fonctions au fur et à mesure des questions,
- un fichier Makefile pour compiler ces programmes,
- le fichier d'en tête `son.h` (complet) et le fichier source `son.c` (à compléter dans ce TP).

Pour compiler un des programmes de test fournis, il faut exécuter la commande `make <nom du programme a compiler>` (par exemple `make test1`). Pour exécuter ensuite le programme, il faut exécuter la commande `./<nom du programme a executer>` (par exemple `./test1`).

Pour en savoir plus sur la commande `make` et les Makefile, voir le support de cours sur le site Web, section Cours.

3 Travail à réaliser

3.1 Allocation dynamique d'un tableau son et lecture des sons

Dans le fichier `son.c`, complétez le code des fonctions suivantes :

— `char* allouer_son(int nb_samples)` : alloue dynamiquement l'espace mémoire pour un signal sonore numérique de `nb_samples` échantillons, où chaque échantillon est de type `char`. Toutes les valeurs du tableau alloué seront initialisées à 0 (pas de son : la membrane du haut parleur ne bouge pas du tout lorsqu'on lit le signal) ;

— `void liberer_son(char* son)` : libère l'espace mémoire occupé par le tableau `son`.

Testez ces fonctions avec le programme principal (`main`) du fichier `test1.c` fourni, qui utilise :

1. l'écoute directe d'un son depuis un fichier .wav (`SDL_PH_PlaySoundWav`),
2. le chargement en mémoire d'un fichier son (`SDL_PH_GetCreateWavS8`),
3. l'écoute/visualisation d'un son sous la forme d'un tableau de char (`SDL_PH_PlayDrawSoundS8`),
4. la création et l'écoute d'un nouveau son (dirac, signal continu).

La compilation se fera à l'aide du fichier Makefile donné sur le site internet. Pour compiler, tapez la commande `make test1` dans le terminal.

Voici le contenu du fichier `test1.c`:

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL2/SDL_phelma.h>
#include "son.h"

int main(int argc, char * argv[]) {
    /* Les tableaux contenant les donnees */
    char* son=NULL;
    char* son2=NULL;
    /* Le nom du fichier contenant le son */
    char wavfile[] = "sons/johnny.wav";
    int nb_samples, fd, amplitude=100, fs;

    int i=0;
    /* La variable manipulant la fenetre graphique */
    SDL_PHWindow* f=NULL;

    puts("Ecouter le son initial: tapez une touche");  getchar();
    /* Jouer le son contenu dans le fichier*/
    SDL_PH_PlaySoundWav(wavfile);

    /* Charger un fichier son dans un tableau */
    if ( (son = SDL_PH_GetCreateWavS8(wavfile,&fs,&nb_samples))==NULL) {
      fprintf(stderr, "Erreur ouverture %s\n", wavfile);
      exit(EXIT_FAILURE);
    }
    /* afficher ses caracteristiques et l'ecouter*/
    printf("le fichier %s a %d echantillons et une frequence d'echantillonnage de %d\n", wavfile,
   nb_samples, fs);
    puts("Ecouter le son recupere: tapez une touche"); getchar();

    /* Creer une fenetre pour l'affichage graphique */
    f=SDL_PH_CreateWindow(800,400);
    /* Ecouter et afficher le trace si la fenetre existe */
    if (f) SDL_PH_PlayDrawSoundS8(son, nb_samples, fs, f, SDL_PH_RED);
    else SDL_PH_PlaySoundS8(son, nb_samples, fs);

    /* Creer un train de dirac a  441Hz, frequence a 44100 Hz*/
    fs = 44100;
    fd = fs/100;
    son2 = allouer_son(nb_samples);
    for(i=0; i<nb_samples; i++)
```

```
    son2[i] =  ((i%fd)==0)*amplitude;
printf("Signal en peigne de Dirac a %d Hz: tapez une touche\n",fd);
getchar();
/* Ecouter ce train de dirac et l'affciher si possible */
if (f) { SDL_PH_ClearWindow(f);
    SDL_PH_PlayDrawSoundS8(son2, nb_samples, fs, f, SDL_PH_RED);
}
else  SDL_PH_PlaySoundS8(son2, nb_samples, fs);
/* liberer la memoire */
liberer_son(son2);

/*Ecouter un signal continu */
son2 = allouer_son(nb_samples);
for(i=0; i<nb_samples; i++)
    son2[i] = amplitude;
puts("Signal continu: tapez une touche");   getchar();
/* Ecouter et visualiser le signal continu */
if (f) {  SDL_PH_ClearWindow(f);
    SDL_PH_PlayDrawSoundS8(son2, nb_samples, fs, f, SDL_PH_RED);
}
else  SDL_PH_PlaySoundS8(son2, nb_samples, fs);

/* Liberer la memoire */
liberer_son(son2);
liberer_son(son);
if (f) SDL_PH_DestroyWindow(f);
return EXIT_SUCCESS;
}
```

3.2 Un son sinusoïdal pour s'entraîner

Pour écrire une fonction qui réalise un signal sinusoïdal simple, il faut :

1. créer un tableau contenant les échantillons,
2. remplir chaque échantillon avec une valeur d'amplitude représentant une sinusoïde, comme présenté figure 2. Sur cette figure, se trouve en abscisse les indices des échantillons dans le tableau (et leur conversion en valeur temps) et en ordonnée les valeurs des échantillons à stocker dans le tableau. Sur cet exemple, la fréquence d'échantillonnage est `fs=10 Hz` (10 échantillons par seconde), l'amplitude du sinus `amp=120`, la fréquence du sinus `freqsin` vaut un peu moins de $0,5Hz$ (environ 1 oscillation toutes les deux secondes). Bien sûr, ce signal ne serait pas audible, car sa fréquence est trop basse. L'échantillon d'indice i, correspondant au temps t/fs, sera donc rempli selon la formule suivante :

$$s(i) = amp * \sin(2 * M\_PI * freqsin * i/fs)$$

Cette fonction sera écrite dans le fichier `son.c` et **testée en complétant le fichier** `test1.c`.

— `char* creer_sin(unsigned char amp, int taille, int freqsin, int fs)` : crée un signal sinusoïdal d'amplitude `amp`, de fréquence `freqsin` et durant `taille` échantillons, avec la fréquence d'échantillonnage `fs`. La figure 2 décrit l'allure d'un tel signal sinusoïdal échantillonné.

La fonction `sin` et la constante `M_PI` sont définies dans le fichier d'entête `math.h`.

Attention au type entier ou flottant dans les calculs : rappelez-vous qu'en langage C, `5/2` vaut `2` (entier) et non pas `2.5` (flottant).

Pour tester votre fonction, complétez le fichier `test1.c` avec par exemple `amp=120`, `taille=22050` (une demi seconde), `freqsin=400`, `fs=44100`.

Quelle est la valeur maximale acceptable pour le paramètre `amp` ?

Quelle est la valeur maximale acceptable pour le paramètre `freqsin`, en fonction de `fs` ?

3.3 Un premier traitement simple : écouter un signal à l'envers

Cette fonction renverse le signal temporel. Vous devrez écrire la fonction suivante dans le fichier `son.c` :

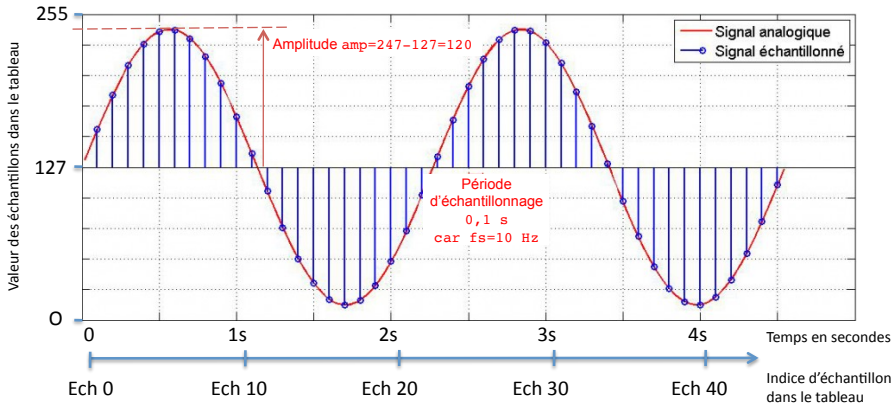


FIGURE 2 – Signal sinusoïdal échantillonné.

1. `char* inverse(char* son, int taille)` qui crée et retourne un son dont les éléments seront inversés dans le temps par rapport au son initial `son` (figure 3). Le son inversé sera stocké dans un tableau alloué dynamiquement et retourné par la fonction `inverse`.

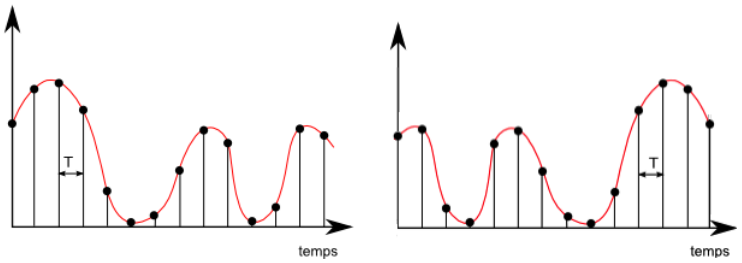


FIGURE 3 – inversion temporelle

Cette fonction est testée dans le programme principal du fichier `test2.c`. Pour compiler, tapez la commande `make test2` dans le terminal.

3.4 Vers des traitements plus complexes

Nous vous proposons ici d'altérer le son pour créer quelques effets sonores très simples. Le premier consiste à moduler l'amplitude du signal sonore par une autre fonction, puis nous proposons des filtres en peigne ou passe-tout, qui sont les briques de base pour réaliser une réverbération numérique. La dernière fonctionnalité proposée est la réalisation d'un écho, superposant le signal antérieur avec atténuation au signal courant.

Modulation : la modulation d'un son est simplement la multiplication de tous les échantillons d'un son par une fonction sinusoïde $\sin(2 * \pi * f * t)$ de fréquence f . Comme le signal est échantillonné, les échantillons de la sinusoïde de modulation sont obtenus par $modul[i] = \sin(2 * \pi * f * i/f_s)$, où $f_s = 1/T$ désigne la fréquence d'échantillonnage.

Si la fréquence de modulation est comprise entre 10 et 40Hz, on obtient un effet de tremolo.

Déphaseur ou passe-tout : le filtre déphaseur change la phase du son en fonction de la fréquence. Ce filtre dépend de 2 paramètres : un gain réel g avec $-1 < g < 1$ et un retard entier m positif. Vous pouvez tester avec des valeurs de l'ordre de $g = \pm 0.95$ et $m = 1000$. Sa fonction de transfert en z

reliant le signal d'entrée e et le signal de sortie s est :

$$H(z) = \frac{S(z)}{E(z)} = \frac{-g + z^{-m}}{1 - g.z^{-m}}$$
$$\text{soit } s[n] = -g.e[n] + e[n-m] + g.s[n-m]$$

Filtre en peigne : le filtre en peigne permet de supprimer ou accentuer certaines fréquences du son et de réaliser du *flanging*. Le filtre de base dépend de 2 paramètres : un gain réel g avec $0 < g < 1$ et un retard entier m positif. Les valeurs de g intéressantes se situent à proximité de 1, avec $g = 1 - 10^{-\frac{3.m}{1000}}$, pour des valeurs de m comprises entre 100 et 300. La fonction de transfert en z reliant le signal d'entrée e et le signal de sortie s est :

$$H(z) = \frac{S(z)}{E(z)} = 1 + g.z^{-m}$$
$$\text{soit } s[n] = e[n] + g.e[n-m]$$

Réverbération : pour réaliser une réverbération numérique, on peut enchaîner plusieurs traitements. On peut par exemple faire la moyenne de plusieurs filtres en peigne de retards différents, et passer le résultat dans une série de filtres passe-tout. Une autre méthode consiste à enregistrer un son (bruit de type impulsionnel comme un ballon qui éclate, un tir de pistolet, ...) dans une salle et à le convoluer ensuite avec le son devant subir la réverbération.

La convolution du son *son* avec la réverbération *reverb* de taille $2n+1$ est simplement l'opération suivante sur chaque échantillon du son :

$$sortie[i] = \sum_{j=-n}^{n} son[i+j].reverb[j]$$

Flanging : lorsque l'on joue le même disque simultanément sur deux platines en mélangeant les deux sorties, tout en ralentissant tour à tour les deux platines, le son produit est parfois synchronisé, parfois légèrement déphasé et le son a une coloration spéciale qui varie au cours du temps. Pour réaliser du *flanging*, il faut utiliser un retard variable, compris entre 0 et $retard\_max$. Cela peut être un retard aléatoire, ou une fonction continue (un sinus : $retard[i] = fabs(retard\_max * \sin(2 * \pi * i * 10./f_s))$). Dans les 2 cas, le retard risque de ne plus être à valeur entière. Il faut alors réaliser une interpolation (inventer le signal entre 2 échantillons). Si p est la partie fractionnaire de $retard[i]$ et $m = \lfloor retard[i] \rfloor$ sa partie entière, les équations deviennent :

$$H(z) = \frac{S(z)}{E(z)} = 1 + g.\left((1-p).z^{-m} + p.z^{-m-1}\right)$$
$$\text{soit } s[n] = e[n] + g.\left((1-p).e[n-m] + p.e[n-m-1]\right)$$

Phasing : comme l'effet de flanging, le phasing est basé sur l'utilisation d'un filtre en peigne variant au cours du temps. C'est le gain g qui varie périodiquement.

Les fonctions suivantes seront écrites dans le fichier `son.c` puis testées à l'aide du programme principal `test3.c`. Commentez (ou décommentez) les appels aux différentes fonctions, selon celles que vous voulez tester. Vous pouvez aussi les appliquer successivement pour obtenir des effets différents.

1. `char* modulation(double fs, double fmod, char* son, int taille)`, qui module le son `son`, avec une sinusoïde de fréquence `fmod`. La fréquence d'échantillonnage est `fs` et le nombre d'échantillons est donné par `taille`. Le son retourné est alloué dans la fonction.
2. `char* passetout(double gain, int retard, char* son, int taille)` , qui retourne un son déphasé par un filtre passe-tout de gain `gain` et de retard `retard`. On ne traitera pas les valeurs pour lesquelles le son n'existe pas, c'est à dire que le traitement commencera à l'échantillon `retard` pour ne pas chercher la valeur d'éléments hors du tableau.

   Cette fonction sera écrite avec **2 pointeurs** et sans indice entier pour parcourir les tableaux `son` et celui retourné par la fonction. On rappelle que si le pointeur `p` pointe l'élément `i` du tableau `son`, ce pointeur `p` a la valeur `son+i` tandis que `p-m` pointe l'élément `son[i-m]` du tableau `son`.

3. `char* peigne(double gain, double retard, char* son, int taille)`, qui retourne un son filtré par un filtre en peigne de gain `gain` et de retard `retard`. Cette fonction sera écrite avec **2 pointeurs** et sans indice entier pour parcourir les tableaux `son` et `sortie`.

Remarques :

1. toutes les fonctions ci dessus calculent le son de sortie $s[n]$ en fonction d'un ou plusieurs échantillons précédents $e[n-1], e[n-2], ..., e[n-k]$. Pour cette raison, on ne calculera le son de sortie qu'à partir de l'échantillon k et on s'arrêtera à l'échantillon `taille` $-k$. Les k premières et k dernières valeurs du son seront : soit recopiées depuis le son original, soit mises à 0.
2. Pour compiler, tapez dans le terminal : `make test3`
   Pour exécuter, tapez dans le terminal : `./test3`

3.5 Facultatif : écho

Le phénomène d'écho consiste à ajouter au signal courant une version atténuée (plus faible) du signal antérieur (retard de la durée de l'écho). La solution la plus simple pour réaliser cet écho est la suivante :

1. allouer le nouveau son de la taille voulue,
2. l'écho voulu a pour longueur un nombre d'échantillons égal à `tailleEcho = dureeEchoSec * fs`,
3. recopier l'ancien son dans le nouveau son,
4. à chaque nouvel échantillon i, ajouter une copie de l'ancien échantillon $i-$`tailleEcho`, atténuée du facteur constant `attenuation`.

```
pour tous les echantillons i de nouveauson a partir du numero tailleEcho
  echo         := ( nouveauson [i - tailleEcho] ) * attenuation
  nouveauson[i] := nouveauson[i] +  echo
fin pour
```

Décommenter l'appel à la fonction `echo` dans le fichier `test3.c` et écrire la fonction :

`int echo(char* son, int taille, int fs, float dureeEchoSec, float attenuation, int tailleSonFinal, char** ptabnouveauson)` , qui crée un nouveau son dans `*ptabnouveauson`, dans lequel un effet d'écho a été appliqué au son `son` passé en paramètre. La durée de l'écho est `dureeEchoSec` secondes. Chaque écho est atténué du facteur `attenuation` (un réel entre 0. et 1.). La taille du nouveau son est fixée à `tailleSonFinal`. Cette fonction retournera 0 si tout s'est bien passé, un nombre différent de 0 en cas de problème. Puisque cette fonction retourne un "code d'erreur" entier, on utilise un passage par adresse pour "retourner" le nouveau son alloué dynamiquement : `ptabnouveauson` est l'adresse du pointeur contenant l'adresse du tableau résultat.

Remarques

— Si l'allocation du tableau nouveau son échoue, c'est une erreur.

— Si le facteur d'atténuation n'est pas compris dans l'intervalle]0.; 1.], c'est une erreur.

— Si la valeur maximale acceptée par le format des échantillons lorsque l'on ajoute l'effet d'écho à chaque échantillon n'est pas comprise entre -128 et 127, c'est une erreur. Pour s'en assurer, une solution est de calculer la valeur de chaque nouvel échantillon dans une variable locale de type `int`, et vous assurer que cette variable est bien comprise entre -128 et 127, avant de la stocker dans le tableau du nouveau son.

— En appliquant plusieurs échos très courts les uns après les autres, de durée et facteur d'atténuation différents, il est possible d'approcher un effet de "réverb" (guère meilleure que celle de votre salle de bain, certes ...)