TP-2 Réseaux : Numérisation

1 Objectifs pédagogiques

L'objectif de ce TP est d'étudier un cas simple de numérisation d'un signal analogique : la Modulation par Impulsion et Codage ou MIC (en anglais *Pulse Code Modulation*, *PCM*). C'est la méthode de base qui est employée pour générer des fichiers ou des flux de données numériques d'audio. On utilisera le langage de programmation `C` dans ce TP.

2 Consignes

General : Lisez TOUT l'énoncé avant de taper une seule ligne de code.

Ressources : Vous pouvez utiliser les matériels du cours (slides de CM, TPs passés, ...) ainsi que toute ressource disponible sur Internet.

3 Numérisation des signaux analogiques

Dans les ordinateurs, le format dominant est le format numérique. En effet, un caractère d'un texte est codé come une séquence binaire de bits 0 et 1 (p. ex., la lettre A majuscule en ASCII correspond à la suite de 7 bits "1000001"). Ensuite, cette séquence binaire est, soit enregistré dans un fichier de texte dans le disc dur, soit transmis vers un autre ordinateur en utilisant des techniques de transmission numérique vues en cours (en bande base ou par modulation d'une onde porteuse). Ce schema est très approprié quand la donnée à exprimer en format numérique (suite de bits 0 et 1) fait partie d'un ensemble discret et fini, comme l'alphabet latin où il n'y a que 26 lettres. Un codage come l'ASCII (7 bits par caractère) permet de representer jusqu'à 128 caractères, plus que suffisant pour coder tout l'alphabet latin, majuscules et minuscules inclues.

Cependant, dans le monde physique il y a des données qui ne font pas partie d'un ensemble fini dénombrable, p. ex. les signaux analogiques. Un signal analogique n'est que l'evolution temporel d'une certaine propriété physique. Un exemple quotidien sont les signaux acoustiques, la voix humaine, la musique, qui ne sont que des changements de la pression atmosphérique crées par les cordes vocales ou par des instruments musicales. Donc, comment est-ce que on peut enregistrer et transmettre du son de manière numérique comme on fait avec un texte codée ASCIII ? Il faut passer par le processus de *numérisation* : transformer le signal analogique (du son dans notre exemple) en numérique (une sequence binaire). Pour cela, la Modulation par Impulsion et Codage ou MIC (en anglais *Pulse Code Modulation*, *PCM*) c'est la technique standard employée pour représenter de la voix et de la musique de manière numérique. Dans ce TP, on va étudier comme le PCM est employé pour numériser du sons.

4 Generation des sons purs

Dans le code source vous avez une première version du `main.c` qui vous demande comme argumente d'entrée le nom d'un fichier `<NOM>`. Ce `main.c` fait trois choses :

(1) fonction `generate_sinusoid` : remplir un buffer d'entiers courts sur 16 bits avec les échantillons d'une sinusoïde de 8 secondes à 440 Hz. Les échantillons sont pris avec une certaine fréquence d'échantillonnage `SAMPLE_RATE` Hz (échantillons / seconde).

(2) fonction `write_CSV_file` : enregistrer les échantillons dans un fichier appelé `<NOM>_samples.csv`, ou `<NOM>` c'est l'argument d'entré.

(3) fonctions dans le code source `wave.c` : enregistrer les échantillons dans un fichier de audio numérique `.wav` appelé `<NOM>.wav`. On reparlera de ce type de fichier dans les suivantes sections.

Compilez ce `main.c` avec les fonctionnes auxiliaires contenus dans `wav.c` et générez le fichiers `.csv` et `.wav`.

Avec l'aide du script `read_and_plot.py`, plotter les échantillons en tapant la commande sur le terminal `python read_and_plot.py <NOM>_samples.csv`. Ce script permet d'afficher deux courbes. Le courbe à gauche est l'ensemble de tous les échantillons dans le fichier. La courbe à droite est un zoom sur les 200 premiers échantillons. Vous pouvez régler ce zoom en tapant la valeur initiale et finale de zoom come 2è et 3è argument, respectivement. Ce que vous verrez c'est simplement une sinusoïde à 440 Hz composé 10*`SAMPLE_RATE` échantillons. Si cette sinusoïde est employé par un dispositif pour générer des vibrations dans l'air, vous gênerez un son pure qui correspond à la note musicale LA dans la troisième octave : LA_3, qui est la fréquence de référence donnée par un diapason. (Vous pouvez retrouvez plus de renseignements sur les rapports entres notes musicales et fréquences sonores sur : http://fr.wikipedia.org/wiki/Note_de_musique).

Le `main.c` ne fait que enregistrer les échantillons de la sinusoïde dans un `.csv`. En plus, il enregistre la sinusoïde dans un fichier `.wav`, ce qui nous permettra de reproduire et écouter ce note LA_3. Le fichier `.wav` (*Waveform Audio File Format*) est un format conteneur pour le stockage de l'audio numérique. Il peut recevoir des formats aussi variés que le MP3, le WMA, l'ATRAC3, l'ADPCM, et le **PCM**. C'est ce dernier qui est cependant le plus courant, et c'est pour cela que l'extension `.wav` est souvent, et donc à tort, considérée comme correspondant à des fichiers "sans pertes" (i.e. lossless), face aux formats de compression avec perte comme le **MP3.** Le format WAV est standardisé sous Windows ; son pendant sous la plate-forme Macintosh est l'AIFF/AIFC. Le conteneur WAV est désormais ancien, et peu pratique :

http://fr.wikipedia.org/wiki/WAVEform_audio_format

. . . mais il reste incontournable sous Windows et beaucoup de baladeurs.

4.1 Structure des fichiers WAV

Le format RIFF, sur lequel repose le format WAV, définit une structure de fichier qui repose sur une succession de blocs de données (*chunks*).

Chaque bloc est identifié par 4 octets (4 symboles ASCII) suivi de la taille du bloc codé sur 4 octets. Si un lecteur rencontre un bloc qu'il ne connaît pas, il passe au suivant. Un fichier `.wav` doit au minimum contenir trois blocs :

(1) un bloc appelé `<RIFF>`, puisque le format WAVE est construit conformément au Resource Interchange File Format.

(2) un bloc appelé `<fmt>` > (format), qui contient les métadonnées techniques, c'est-à-dire les informations relatives au codage du flux audio, informations indispensables pour interpréter les données.

(3) un bloc appelé `<data>`, qui contient la charge (payload), c'est-à-dire les données audio utiles.

Le bloc `<fmt >` doit être positionné en amont du bloc `<data>`. Donc, l'en-tête d'un fichier WAV commence dès le premier octet (offset 0). Il a une taille de 44 octets, et est constitué des champs suivants (listés dans l'ordre) :

```
[Bloc de déclaration d'un fichier au format WAVE]
   FileTypeBlocID  (4 octets) : Constante << RIFF >> (0x52,0x49,0x46,0x46)
   FileSize        (4 octets) : Taille du fichier moins 8 octets
   FileFormatID    (4 octets) : Format = << WAVE >> (0x57,0x41,0x56,0x45)

[Bloc décrivant le format audio]
   FormatBlocID    (4 octets) : Identifiant << fmt >> (0x66,0x6D, 0x74,0x20)
   BlocSize        (4 octets) : Nombre d'octets du bloc - 16  (0x10)

   AudioFormat     (2 octets) : Format du stockage dans le fichier (1: PCM, ...)
   NbrCanaux       (2 octets) : Nombre de canaux (de 1 à 6, cf. ci-dessous)
   Frequence       (4 octets) : Fréquence d'échantillonnage (en [Valeurs
```

```
                                standardisées  : 11.025, 22.050, 44.100 et
                                éventuellement 48.000 et 96.000]
    BytePerSec      (4 octets) : Nombre d'octets à lire par seconde (c.-à-d.,
                                Frequence * BytePerBloc).
    BytePerBloc     (2 octets) : Nombre d'octets par bloc d'échantillonnage
                                (c.-à-d., tous canaux confondus :
                                NbrCanaux * BitsPerSample/8).
    BitsPerSample   (2 octets) : Nombre de bits utilisés pour le codage de
                                chaque échantillon (8, 16, 24)

[Bloc des données]
    DataBlocID      (4 octets) : Constante << data >>  (0x64,0x61,0x74,0x61)
    DataSize        (4 octets) : Nombre d'octets des données (c.-à-d. "Data[]",
                                c.-à-d. taille_du_fichier - taille_de_l'entête
                                (qui fait 44 octets normalement).
   DATAS[] : [Octets du Sample 1 du Canal 1] [Octets du Sample 1 du Canal 2]
             [Octets du Sample 2 du Canal 1] [Octets du Sample 2 du Canal 2]

    * Les Canaux :
       1 pour mono,
       2 pour stéréo
       3 pour gauche, droit et centre
       4 pour face gauche, face droit, arrière gauche, arrière droit
       5 pour gauche, centre, droit, surround (ambiant)
       6 pour centre gauche, gauche, centre, centre droit, droit, surround (ambiant)
```

NOTES IMPORTANTES : Les octets des mots sont stockés sous la forme (c.-à-d., en "little endian") [87654321][16..9][24..17] [8..1][16..9][24..17] [...

Heureusement pour vous, les fichier `wav.c` et `wav.h` implémentent déjà les fonctionnes nécessaires pour écrire un fichier conforme au format RIFF/WAV ci-dessus. Regardez les signatures des fonctions contenues dans ces fichiers pour apprendre à les utiliser. Par défaut, on cree un fichier de qualité AUDIO CD : son numérisé par *PCM*, fréquence d'échantillonnage de 44.100 Hz et des échantillons de16-bits (des entiers courts). Par rapport à la vraie qualité AUDIO CD, on utilise des canaux mono à la place de stéréo afin de simplifier l'utilisation des échantillons. La fréquence d'échantillonnage est à 44.100 Hz parce que la borne supérieur de champ auditif humain est à 20.000 Hz (http://www.cochlea.org/entendre/champ-auditif-humain). Donc, la fréquence d'échantillonnage est deux fois cette valeur-là.

Dans la version actuelle du `main.c`, les fonctionnes du fichier `wav.c` sont utilisées pour écrire un fichier `.wav` contenant 8 seconds d'un son pure qui correspond à la note musicale LA dans la troisième octave : LA_3. Vous allez appeler ce fichier `LA.wav`. Ecoute-le afin de vérifier qu'il s'agit bel et bien d'un LA.

5 Generation des mélodies

Maintenant, vous allez générer une mélodie contenant un "DO, RE, MI, FA, SOL, LA, SI" dans la troisième octave, un "DO, RE, MI, FA, SOL, LA, SI" dans la quatrième octave et un "DO" dans la cinquième octave pour finir, c.-à.-d. une suite de 15 sons pures (sinusoïdes) avec les suivantes fréquences : [264, 297, 330, 352, 396, 440, 495, 528, 594, 660, 704, 792, 880, 990, 1056], respectivement. Pour simplifier, on garde la durée totale de la mélodie à 8 seconds, ce qui fait une durée de chaque note d'un demi-second (vous pouvez metre un silence d'un demi-second à la fin pour arriver à 8 seconds).

5.1 Votre travail

Il s'agit là de démontrer que êtes capables de concaténer simplement des signaux. Pour faire cela, vous créez dans le `main.c` une nouvelle fonction `generate_melody` en vous inspirant de `generate_sinusoid`

et vous écrivez la sortie de cette fonction dans un fichier `.wav` en utilisant les fonctionnes du fichier `wav.c`. Vous allez appeler ce fichier `mel.wav`. Ecoute-le afin de vérifier qu'il s'agit bel et bien d'un un "DO, RE, MI, FA, SOL, LA, SI" dans la troisième octave suivi d'un "DO, RE, MI, FA, SOL, LA, SI" dans la quatrième octave.

6 L'échantillonnage

Comme on a vu en cours, la toute premiere étape de la numérisation par PCM est l'échantillonnage. Le paramètre de contrôle de cette étape est la fréquence d'échantillonnage, qui doit être au moins deux fois la fréquence maximale présente dans la signal è échantillonner. Par défaut, on utilise une fréquence d'échantillonnage de 44.100 Hz, qui est largement suffisante pour échantillonner notre mélodie dont la fréquence supérieure est 1.056 Hz.

6.1 Votre travail

Maintenant, mettez la fréquence d'échantillonnage (`SAMPLE_RATE`) à la valeur de1.000 Hz et enregistrez le fichier comme `mel_fe1000Hz.wav`. Ecoute-le.

Question 1 : Est-ce que vous écoutez le "DO, RE, MI, FA, SOL, LA, SI" dans le deux octaves ? Pour quoi ?

7 La quantification

Dans tout le TP, on a travaillé avec des entiers courts signés (16 bits). Donc, on avait droit à $2^{16} = 65536$ niveaux pour representer les valuers des sinusoïdes. Plus précisément, les valuer entiers de nos sinusoïdes évoluaient dans l'interval continu ($analogique$) [$-$ `SHRT_MAX` , `SHRT_MAX`], où `SHRT_MAX` = $2^{15} - 1 = 32767$, interval couvert par les entiers courts signés (***SHORT*** ou ***int16_t***). Cela veut dire que chaque valuer décimale (***DOUBLE***) peut être arrondi vers un entier court signé différant en faisant directement une conversion de type de données (*casting*) ***int16_t***. Autrement dit, il n'y aura pas de deux valuers décimales de la sinusoïde représentés avec le même entier court signé.

7.1 Votre travail

Maintenant, on va étudier ce qui se passe si on n'a le droit qu'à utiliser quatre bits, c.-à.-d. $2^4 = 16$ niveaux. Dans ce cas, il faudra représenter l'interval [$-$ 32767 , 32767] uniquement avec 16 niveaux. Cela implique distribuer les 65536 valuers possibles sur des intervalles de quantification de longueur $65536/16 = 4096$. P. ex. les valeurs entre -32768 et -28672 devraient être représentes par la première valeur sur 4 bits, -8. Créez une fonction `quantification4bits` qui permet transformer la sortie de `generate_melody`, qui utilise $2^{16} = 65536$ niveaux, en un tableau d'entiers qui utilise 4 bits. Vous allez appeler le fichier avec des échantillons quantifiés avec 16 niveaux `mel_4b.wav`. Ecoutez le nouveau fichier. Ensuite, avec l'aide du script `read_and_plot_quant.py`, plotter les échantillons de `mel_4b.wav` (mélodie codé à 4 bits) et `mel.wav` (mélodie codé à 16 bits). Vous pouvez régler le zoom en tapant la valeur initiale et finale de zoom come 1re et 2è argument, respectivement. ATTENTION AVEC L'ECHELE DE DEUX COURBES A 4 ET A 16 BITS ! ! ! !

Question 2 : Est-ce que vous écoutez les deux "DO, RE, MI, FA, SOL, LA, SI" avec la même qualité qu'auparavant ? Pour quoi ? Qu'est-ce que la courbe verte représente ?