Traitement du Signal Temps Réel

Sujet BE: Autotune

Olivier Perrotin & Thomas Hueber

DURÉE : $4 \times 4\,\text{h}$
DATE DE RENDU : 10 Janvier 2025

Introduction

1 But du BE

Le cours magistral ($2 \times 2\,\text{h}$) a abordé différents aspects liés à la conception d'un « système temps-réel » :

- Définitions d'un système temps-réel ($\neq$ système à exécution rapide)
- Modèles théoriques de conception (synchronous/scheduled, etc.)
- Choix du hardware (DSP, GPU, etc.)
- Techniques d'implémentation logicielle (bonnes pratiques, risque d'inversion de priorité, etc.)
- Audio sur PC (« mille-feuille » logiciel, API audio, etc.) et traitement audio temps-réel (modèle producteur-consommateur, buffer underrun/overrun, overlap-add, buffer circulaire).

Un focus particulier a été mis sur les systèmes dits « soft », destinés à une implémentation sur un OS standard. Dans le cadre du BE TSTR ($4 \times 4\,\text{h}$), vous mettrez en pratique certains des points abordés dans le cours, au travers de **l'implémentation d'un effet audio de type *autotune* par synthèse additive**.

2 Evaluation

Vous effectuerez une démonstration de votre programme en 5 min lors de la seconde partie de la 4$^{\text{e}}$ séance. Vous fournirez un fichier ZIP nommé `nom1_nom2_tstr_2024.zip`, à envoyer à olivier.perrotin@grenoble-inp.fr) contenant :

- Un `README` précisant la marche à suivre pour compiler et exécuter votre programme
- Votre code `duplex.cpp`, nettoyé et commenté pour chaque question du BE.
- Un répertoire `files/` contenant les enregistrements effectués, et dont le nom de chaque fichier est explicite et est proprement référencé dans le rapport (le nom doit contenir le nom du fichier original le cas échéant, le numéro de la question, et le type de donnée (par ex. in, out, f_0)). Les enregistrements audio sont à fournir en format `wav` et non en fichiers binaires.
- Un rapport *court* (max 4 pages, nommé `nom1_nom2_tstr_2024.pdf`) détaillant vos choix d'implémentation et les résultats obtenus. Les instructions pour lesquelles une réponse est attendue dans le rapport (réponse à une question, tracé d'une figure, etc.) sont indiquées en bleu dans les consignes suivantes. Vous êtes encouragés à y indiquer aussi les divers problèmes rencontrés.

Ce projet sera implémenté en standard C sous Linux, **sur les machines de l'école**, en s'appuyant sur l'API `RtAudio` développée par G. Paul Scavone.

> Dans votre rapport, accordez un soin particulier à *l'orthographe*, ainsi qu'aux *figures* (titres des axes explicites, texte lisible, usage efficace des couleurs, etc.).

3 Ressources

Les ressources du BE sont dans le dossier `BE_tstr_2024_v1` sur Chamilo. Vous y trouverez :

- L'API `RtAudio` `rtaudio-6.0.1.tar`
- Un répertoire `c/` contenant des fonctions en C qui vous seront utiles, et qui sont décrites en Annexe 2.
- Un répertoire `python/` contenant un script Python permettant de tracer des signaux enregistrés en format binaire par votre programme.
- Un répertoire `fichiers_audio_test/` contenant des fichiers audio à tester à l'entrée de votre système.

> Source et doc complète de `RtAudio` : `https://www.music.mcgill.ca/~gary/rtaudio/`

Prise en main de l'API RtAudio

1 Fonctionnement de base de l'API

1. Compilez RtAudio sur Linux en suivant les étapes suivantes
   a) Téléchargez l'API `RtAudio` `rtaudio-6.0.1.tar` et décompressez-là dans le répertoire de votre choix.
   b) Activez l'environnement conda `rtaudio`
   c) Compilez la librairie principale (`./configure` puis `make all`).
   d) Compilez les programmes d'exemples (`cd tests` puis `make all`).
2. Lancez d'abord `audioprobe` puis `duplex`. Que font ces deux programmes ?
3. Étudiez le code source `duplex.cpp` qui permet la gestion d'un flux audio E/S :
   a) Identifiez les paramètres de configuration du flux audio (fréquence d'échantillonnage, nombre de buffers internes, taille des buffers, etc.).
   b) Identifiez la fonction `openStream()`. Quel est son rôle ? Quel est le rôle du 7^{e} argument, appelé `userData` dans la doc ?
   c) Identifiez la fonction de callback audio `inout()`. Que fait cette fonction par défaut ?
   d) Observez son prototype et remarquez que trois des arguments de la fonction sont de type `void`. Expliquez pourquoi, et expliquez la méthode pour accéder à ces arguments (ligne 84 du code original). Appliquez cette méthode aux buffers d'entrée / sortie et remplacer la ligne `memcpy()` par une boucle qui copie élément par élément le buffer d'entrée dans le buffer de sortie.
   e) Quel lien faites-vous entre l'argument `data` du callback `inout()` et les arguments de la fonction `openStream()` ? Pour faire passer plusieurs paramètres du programme principal (`main`) à la fonction de callback audio, il est possible d'utiliser des structures de données. Implémentez cette solution pour faire passer à la fois la taille du buffer et la fréquence d'échantillonnage à `inout()`.
   f) Par la suite, nous travaillerons avec des doubles. Configurez l'API `RtAudio` pour fonctionner avec ce format (cf. flag `RTAUDIO_FLOAT64`).

> Dans ce BE, vous implémenterez une version temps-réel en C/C++ de l'autotune en partant du code source `duplex.cpp`. Dans le fichier « `c/somefunc_2024.cpp` » de l'archive `BE_tstr_2024_v1`, vous trouverez un ensemble de fonctions destinées à vous faciliter l'implémentation, listées en Annexe 2. Pour simplifier la compilation et utiliser le `MakeFile` existant, vous ne travaillerez que dans `duplex.cpp` en y ajoutant les fonctions nécessaires et leurs prototypes.

Boîte à outil pour le débugage

2 Lecture d'un fichier – tester des entrées spécifiques

Afin de tester votre programme sur des entrées contrôlées, et pouvoir reproduire des erreurs rencontrées, il est de bon usage de pouvoir utiliser en entrée du système des fichiers audio pré-enregistrés. Un ensemble de fichiers tests vous sont fournis dans le répertoire « `fichiers_audio_test/` » de l'archive `BE_tstr_2024_v1` à la fois en format `.wav` pour pouvoir les écouter facilement sur votre machine, mais aussi en format binaire (`.bin`, sans en-tête) au format double (64 bits), et donc facilement lisible avec `fread()`.

4. Chargez un des fichiers en vous aidant de l'algorithme 1 en Annexe 1. Fonctions utiles : `fopen()`, `fread()`, `fseek()`, `ftell()`, `fclose()`. Où placez-vous le chargement du fichier audio dans le programme, et pourquoi ?
5. Il s'agit maintenant de simuler une entrée temps-réel où, à chaque appel de la fonction callback `inout()` vous allez copier dans le buffer d'entrée de taille N les N échantillons suivants du fichier audio. En conservant la copie du buffer d'entrée vers le buffer de sortie, vous devriez alors entendre la lecture de votre fichier à l'appel de `duplex`.
6. Adaptez ensuite votre code pour que le fichier audio soit joué en boucle.

3 Écriture d'un fichier – observer l'évolution des variables

Il existe peu d'outils simples en C qui permettent d'observer et tracer l'évolution des variables à un instant donné. Une solution est d'écrire les variables dans un fichier pendant l'exécution, puis de les tracer avec un programme annexe. La fonction `write_buff_dump()` qui vous est fournie (cf. Annexe 2.1) permet de remplir progressivement un buffer appelé `buffer_dump` avec la variable ou le tableau de votre choix au cours de l'exécution du programme.

7. Initialisez un tableau `buffer_dump` dans votre programme, avec la taille de votre choix. Fonctions utiles : `malloc()`, `calloc()`, `free()`. Faites-vous l'initialisation dans le `main()` ou le `inout()` et pourquoi ?
8. Copiez le buffer d'entrée du callback `inout()` dans le buffer `buffer_dump` au fur et à mesure de l'exécution en utilisant `write_buff_dump()`.
9. Écrivez le buffer `buffer_dump` sur le disque dans un fichier binaire. Fonctions utiles : `fopen()`, `fwrite()`, `fclose()`. Faites-vous l'écriture dans le `main()` ou le `inout()` et pourquoi ?
10. Le script « `python/plot_dump.py` » permet de lire le fichier binaire écrit et de tracer les données. Adaptez le nom du fichier à lire dans le script, et personnalisez l'affichage de la figure selon vos besoins. Ce même script écrit le signal sur le disque en format `.wav` afin de vous permettre de l'écouter, dans le cas où le signal enregistré est audio. Pour avoir `matplotlib`, utilisez l'environnement conda `pytorch`.
11. Répétez la procédure avec un deuxième buffer pour aussi écrire le signal de sortie de la fonction `inout()`.

Pour chaque lancement du programme, l'entrée et la sortie du programme sont maintenant écrites dans des fichiers.

- Dans la suite du BE, il vous sera régulièrement demandé de joindre ces fichiers pour vérifier votre implémentation. Il vous sera aussi parfois demandé d'écrire des variables intermédiaires de votre programme pour les observer.
- Au-delà des consignes, il est aussi vivement encouragé d'écrire et observer les variables de votre programme qui vous semblent nécessaire pour vous aider dans le débugage de votre code.

Effet *Autotune* par synthèse additive

4 Analyse du signal

Fréquence fondamentale : Nous avons vu en cours qu'une méthode d'estimation de f_0 utilise le deuxième maximum de l'auto-corrélation du signal à analyser.

12. Implémentez la fonction d'auto-corrélation sur le buffer d'entrée de `inout()` puis extrayez-en le deuxième maximum pour obtenir f_0. Vous pouvez vous aider de l'algorithme 2 fourni en Annexe 1. Utilisez les différents fichiers audio de test fournis et tracez f_0 mesuré en fonction du temps (en secondes) avec `plot_dump.py` après l'avoir écrit dans un fichier. Joignez à votre rapport vos tracés de f_0 ainsi que les fichiers contenant les signaux d'entrée correspondants. Commentez vos résultats.
13. Pour réduire les erreurs de détection, limitez la recherche de la fréquence fondamentale à un intervalle correspondant à la tessiture vocale de votre choix.
14. Testez l'analyse de f_0 pour plusieurs tailles de buffer. Fournir les tracés correspondants et commentez. Pourquoi un buffer circulaire serait utile ici ? Vous implémenterez le buffer circulaire à la fin du BE. Choisissez en attendant la taille de buffer que vous trouvez la plus adaptée.

Analyse harmonique : Pour effectuer une synthèse additive, il est nécessaire de connaître les fréquences et amplitudes des harmoniques à générer.

15. Soit `n_fft` le nombre de points sur lequel vous calculez une Transformée de Fourier Discrète (TFD). Quelle est la valeur du bin de la TFD le plus proche de f_0 mesuré ?
16. Calculer la TFD du signal (cf. Annexe 2.4) et extrayez les valeurs de fréquence et d'amplitude pour chaque harmonique. Notez qu'une FFT vous donne la TFD à un facteur `n_fft` près. (Par exemple, l'application de la FFT à un cosinus donne deux pic d'amplitude `n_fft/2`). Normalisez la sortie de la FFT en conséquence pour obtenir l'amplitude réelle de la TFD.

5 Synthèse du signal

Il s'agit maintenant d'écrire un nouveau signal dans le buffer de sortie.

> Par défaut, les valeurs -1 et 1 du buffer de sortie correspondent au niveau maximal d'amplitude en valeur absolue de votre carte son. Pour protéger vos oreilles :
> - Vérifiez à ne pas envoyer des valeurs allant au-delà de cet intervalle dans le buffer de sortie.
> - L'amplitude [-1,1] peut être très forte en fonction de votre matériel sonore. Au premier lancement de votre programme, attendez d'entendre le niveau sonore avant de mettre le casque sur vos oreilles.

Première implémentation : La synthèse additive génère un signal par l'addition de cosinus d'amplitudes et fréquences correspondant à chaque harmonique du signal à reconstruire (cf. équation dans le cours).

17. Implémentez la synthèse additive à partir des fréquences et amplitudes des harmoniques relevées précédemment et envoyez le résultat sur le buffer de sortie. Vous percevez normalement de nombreux clics dans le signal. Écrivez le signal de sortie dans un fichier, tracez-le et joignez-le au rapport. A l'observation du signal, comment expliquez-vous les clics entendus ?

Ajout de la phase : L'information de phase permet de rendre cohérentes les trames de signal contiguës.

18. Rappelez l'équation de la phase de l'harmonique de la trame courante en fonction de la fréquence et la phase de l'harmonique de la trame précédente.

19. Inclure la phase dans la synthèse additive. Écrivez le signal de sortie dans un fichier et joignez-le au rapport. Tracez ensuite ce signal de sortie et commentez-le. Les transitions entre chaque trame sont-elles parfaites ?

6 Autotune

Entre l'analyse et la synthèse, il vous est maintenant possible de modifier la valeur de f_0 (et des harmoniques) pour réaliser l'effet de votre choix. L'autotune consiste à arrondir f_0 à des valeurs discrètes sur une gamme en demi-tons (gamme musicale occidentale). Un demi-ton correspond à $1/12$ d'une octave sur une échelle logarithmique. Le passage de f_0 en `Hz` en à f_0 en demi-tons (`ST`) et inversement se fait par :

$$f_{0ST} = 12 \log_2 (f_{0Hz}) \tag{6.1}$$

$$f_{0Hz} = 2^{\frac{f_{0Hz}}{12}} \tag{6.2}$$

20. Implémentez un autotune en passant f_0 en `ST`, en arrondissant la valeur, et en repassant en `Hz`. Pour des effets plus forts, arrondissez à 3, 4 ou 6 demi-tons.

7 Fenêtres glissantes

Analyse : Comme vu en question 14, l'implémentation d'un buffer circulaire permet de mesurer un f_0 lissé tout en conservant un buffer d'entrée de taille réduite.

21. Implémentez un buffer circulaire sur lequel vous effectuerez l'estimation de f_0. Tracez la courbe de f_0 obtenue avec différentes tailles de buffer et commentez. Joignez à chaque fois le fichier contenant le signal d'entrée correspondant.

Synthèse : La technique d'*overlap-add* permet de supprimer complètement les discontinuités entre deux trames de sorties contiguës.

22. Appliquez une fenêtre de Hanning (cf. Annexe 2.3) sur votre buffer de synthèse, et implémentez un *overlap-add* pour remplir le buffer de sortie.

> Vous ferez votre démonstration finale avec un fichier de votre choix comme entrée, ainsi qu'avec l'entrée microphone.

Annexes

1 Quelques algorithmes

Algorithm 1: Lire un fichier binaire

```
// Ouvrir un fichier
Ouverture avec fopen()
// Calculer sa taille
Mettre le pointeur de fichier à la fin du fichier avec fseek()
Lire la position du pointeur de fichier avec ftell()
Remettre le pointeur de fichier au début du fichier avec fseek()
// Lire un fichier
Lecture avec fread()
```

Algorithm 2: Demie auto-corrélation (indices positifs)

```
// Pour chaque élément de la corrélation (indices positifs)
for n = 0 ; n < N_y ; n++ do
    y[n] = 0 ;
    // Calcule la corrélation
    for k = n ; k < N_x ; k++ do
        y[n] = y[n] + x[k] × x[k − n]
    end
end
```

2 Fonctions fournies

Dans le fichier « `c/somefunc_2024.cpp` » de l'archive `BE_tstr_2024_v1`.

2.1 Écrire dans un buffer

`int write_buff_dump(double* buff, const int n_buff, double* buff_dump, const int n_buff_dump, int* ind_dump);`

Copie le tableau `buff` de taille `n_buff` dans `buff_dump` de taille `n_buff_dump`. La copie commence à l'indice `ind_dump` du tableau `buff_dump` qui est incrémenté au fur et à mesure. Quand `ind_dump` atteint `n_buff_dump`, alors on ne peut plus écrire dans `buff_dump`.

`double* buff` :	Tableau à copier.
const int `n_buff` :	Nombre d'éléments du tableau à copier (si `n_buff` = 1, cela revient à un passage par adresse de la variable `buff`).
`double* buff_dump` :	Tableau qui reçoit la copie.
`const int n_buff_dump` :	Taille du tableau qui reçoit la copie.
`int* ind_dump` :	Tête d'écriture du tableau qui reçoit la copie. Celle-ci est *passée par adresse* pour être mise à jour à l'appel de la fonction.

2.2 Mesurer un temps d'exécution

```
double get_process_time();
```

Retourne l'horloge du système.

2.3 Appliquer une fenêtre de Hanning sur un signal

```
static double *hanning(double *w, const int leng);
```

Remplit le tableau `w` avec une fenêtre de Hanning de taille `leng`. Avant l'appel de la fonction, `w` est un tableau vide. Après l'exécution, il contient la fenêtre.

2.4 Calculer une transformée de Fourier avec l'algorithme FFT

```
int get_nextpow2(int n);
```

Retourne la puissance de 2 juste supérieure à `n`.

```
int fftr(double *x, double *y, const int m);
int fft(double *x, double *y, const int m);
```

Calcule la transformée de Fourier d'un signal réel (`fftr`) ou d'un signal complexe (`fft`) avec une FFT.

- `double *x` : La partie réelle du signal. Ces valeurs seront remplacées par la partie réelle de la transformée de Fourier après exécution.
- `double *y` : Le partie imaginaire du signal. Ces valeurs seront remplacées par la partie imaginaire de la transformée de Fourier après exécution.
- const int m : Taille de la TFD.

```
int ifft(double *x, double *y, const int m);
```

Calcule la transformée de Fourier inverse d'un signal avec l'algorithme FFT.

- `double *x` : La partie réelle de la transformée de Fourier. Ces valeurs seront remplacées par la partie réelle du signal après exécution.
- `double *y` : Le partie imaginaire de la transformée de Fourier. Ces valeurs seront remplacées par la partie imaginaire du signal après exécution.
- `const int m` : Taille de la TFD.