Traitement embarqué de signaux sonores

Feuille de TP

Le but de ce TP est d'implémenter un petit synthétiseur portable sur les cartes STM32F4-Discovery, modélisant le son d'un instrument à cordes pincées. Une note sera déclenché par un claquement de mains (détécté par le microphone), et sa hauteur sera déterminée par l'inclinaison de la carte (détectée par l'accéléromètre embarqué). Nous fournissons le code source de base, qui configure et donne une interface pour utiliser les périphériques embarqués. L'algorithme de synthèse que nous allons développer est l'algorithme de Karplus-Strong [1].

On écoutera le son produit par la prise casque ; le volume est fixé en software, à une valeur acceptable. Attention cependant à *toujours* enlever le casque de vos oreilles avant de modifier le code source : un seul oubli peut être fatal à votre ouïe.

À la fin de la session, envoyez l'ensemble de votre code source à `matthias.puech@lecnam.net`, dans une archive `.zip` portant le(s) nom(s) de leur(s) auteur(s). L'évaluation tiendra compte des fonctionnalités implémentées, de leur correction, et de la qualité et lisibilité du code source.

1 Buffer circulaire et ligne de retard simple

Pour se familiariser avec l'environnement de développement, construisons d'abord une simple ligne de retard à l'aide d'un buffer circulaire. Une ligne de retard prend le son entrant et le ressort inchangé après un temps de retard donné.

1.1 Prise en main

1. Téléchargez l'archive des sources. [2] Dans un terminal, placez-vous dans le répertoire décompressé, et tapez `vagrant up` pour démarrer la machine virtuelle.
2. Quand elle a démarré, tapez `vagrant ssh` pour s'y connecter en SSH. Branchez maintenant la carte en USB.
3. Une fois connectée, allez dans le répertoire `/vagrant`. Dans la marchine virtuelle, ce répertoire est synchronisé avec le répertoire où se trouvent les sources sur l'hôte. Tapez `make flash` pour envoyer le programme à la carte.
4. Le programme par défaut est téléchargé sur la carte ; il fait clignoter les quatre LEDs, et redirige l'entrée micro sur la sortie audio. Vérifiez avec un casque et vos yeux que c'est bien le cas.
5. Faisons maintenant une modification triviale et ré-envoyons le programme modifié à la carte : ouvrez le fichier `main.cc` ; dans la boucle principale (fonction `Process`), divisez le sample de la voix de gauche par 8 :

```
out[i].l = in[i] / 8;
```

6. Recommencez l'étape 4. et vérifiez au casque que la voix de gauche a bien un volume plus faible que celle de droite.

1. `https://en.wikipedia.org/wiki/Karplus-Strong_string_synthesis`
2. `https://gitlab.cnam.fr/gitlab/puechm/stm32-dsp-template/repository/archive.zip`.

1.2 Buffer circulaire

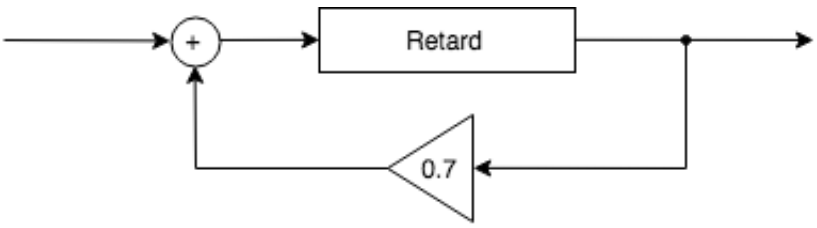
1. Écrire une classe `RingBuffer` implémentant un buffer circulaire de taille 48000, et contenant les méthodes :
   — `void Write(short x)` qui écrit la valeur `x` dans le buffer, et
   — `short Read(int n)` qui renvoit la `n`-ième valeur écrite (`n` < 48000).
2. Testez son implémentation dans la boucle principale :
   — déclarez un `RingBuffer buffer` dans la classe principale ;
   — écrivez dans `buffer` chaque sample reçu du micro (`in[i]`), et
   — envoyez au DAC (`out[i]`) le 48000-ième sample écrit dans `buffer` ;
   — vérifiez qu'il y a bien maintenant un décalage d'une seconde entre ce que le micro capte et la sortie audio.
3. Que se passe-t-il quand vous réduisez ou augmentez la fraction du buffer lue (l'argument de `Read`) ? Comment calculer la durée du décalage en fonction de cette taille en sample ? [3]

2 Ligne de retard avancée

Apportons maintenant deux améliorations à notre retard, qui nous serviront dans l'implémentation de l'algorithme de Karplus-Strong : une boucle de rétroaction, et une lecture avec interpolation linéaire.

2.1 Boucle de rétroaction

1. Réinjectez maintenant une portion de la sortie dans l'entrée :



   Pour cela, enregistrez la dernière valeur de la sortie dans une variable d'état, multipliez-la par 0.7 (le *coefficient de feedback*) pour l'atténuer, et enfin sommez cette valeur à l'entrée. Écoutez l'effet de ce changement sur la ligne de retard. Que se passe-t-il si vous changez la valeur 0.7 en 0.3 ? 1.0 ? plus de 1.0 ? (attention aux oreilles)
2. Vous venez de sommer deux samples de type `short` ; il y a donc potentiellement *overflow*. Modifiez le code précédent pour que la somme soit *saturante* (si la valeur est supérieure à la plus grande valeur représentable, elle reste à ce maximum ; idem pour le minimum). Testez et écoutez la différence.
3. Placez le coefficient de feedback à 0.9. Qu'entendez-vous si vous réduisez le temps de délai à 100ms ? puis 50ms ? puis 25ms ?
4. Isolez ce retard avec boucle de rétroaction dans une classe `FeedbackDelay` ayant une méthode publique `float Process(float input, float feedback, int delay)` : elle prend en paramètre le sample d'entrée, le coefficient de feedback et le temps de délai (en samples), et renvoit le sample de sortie.

3. On rappelle que la fréquence d'échantillonnage est de 48kHz

2.2 Retard variable et interpolation

1. Modulons maintenant le temps de retard dans le temps. Faites décrire au temps de retard une dent de scie de fréquence f très basse (0.2Hz par exemple) entre 1 et 48000.[4] Vous devrez entendre trois artefacts différents :
   (a) le changement de pitch de l'entrée (effet *Doppler*) ;
   (b) une discontinuité (un *click*) au moment de la remise à zéro de la phase ;
   (c) une distortion déplaisante, dûe à l'erreur d'arrondi quand vous passez `phase` (un `float`) à la fonction `Read` (qui attend un `int`). C'est celle-ci que l'on va éliminer maintenant :
2. Implémentez dans la classe `RingBuffer` une méthode `short ReadLinear(float n)` renvoyant la `n`-ième valeur écrite, avec interpolation linéaire. Appelez cette méthode à la place de `Read` dans votre ligne de retard, et constatez l'élimination de l'artefact (c).
3. Écrivez une classe `OnePoleLP` implémentant un filtre passe-bas à un pôle, et ayant une seule méthode :

   ```
   float Process(float input, float coefficient)
   ```

   Ajoutez une instance de ce filtre à la classe `Main`, et appliquez-le à chaque tour la boucle principale à la composante x [5] lue par l'accéléromètre, avec un coefficient de 0.0001.
4. Modulez le temps de retard avec la valeur de la composante x filtrée (il faudra la *scaler* pour qu'elle reste dans les bornes acceptables pour le temps de retard. De la même façon, modulez le feedback par exemple avec la composante y lue par l'accéléromètre, également filtrée.

3 Détection de transitoire

Nous allons maintenant implémenter l'algorithme de détection des transitoires. Pour cela nous avons besoin d'une boucle principale "vierge" ; sauvegardez donc votre fonction `Process` actuelle, et revenez à la version minimale fournie.

1. Dans la boucle principale, envoyez au DAC du bruit blanc au lieu des samples d'entrée. Pour synthétiser du bruit blanc, utilisez le générateur pseudo-aléatoire `short Random::Short()` défini dans `dsp.hh`. Testez et écoutez (attention aux oreilles).
2. Modifiez le code précédent pour ne générer du bruit que lorsque le bouton a été appuyé depuis plus de 100ms (4800 samples).[6] Dans le cas contraire, générer du silence (samples à 0)
3. Isolez toute la génération de ces impulsions de bruit dans une classe `NoiseBurst` ayant deux méthodes :
   — `short Process()` qui renvoit le sample suivant (aléatoire ou zéro), et
   — `void Trigger(int length)` qui déclenche l'émission de samples aléatoires pendant `length` samples.
   Nous allons nous servir de ce générateur d'impulsion pour notifier la détection d'une transitoire dans le signal d'entrée.

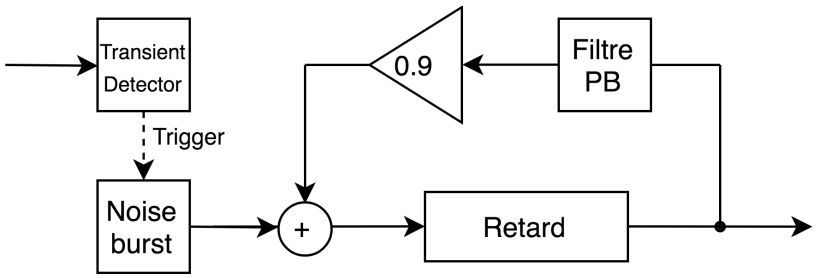
4. Pour ceci, ajoutez une variable d'état `float phase` initialisée à zéro, et incrémentez-la de f à chaque tour de boucle ; quand elle dépasse 48000, réinitialisez-là à 0.
5. Attention, les 3 composantes x, y et z lue sur l'acceleromètre sont de type `short`, et ont une valeur comprise entre -32768 (accélération négative de 2 g sur cet axe) et 32767 (accélération positive de 2 g)
6. La fonction `button_.read()` renvoit l'état du bouton.

4. Implémentez dans la boucle principale l'algorithme de détection de transitoires vu en cours sur le signal entrant ; celui-ci appellera la méthode `Trigger(200)` du générateur d'impulsion. Testez et ajustez les paramètres du détecteur de façon à déclencher une impulsion de bruit quand on claque des mains à proximité du micro.
5. Isolez l'algorithme de détection dans une classe `TransientDetector` comprenant une unique méthode publique `bool Process(float input)`. Celle-ci renvoit `true` quand une transitoire a été détéctée.

4 L'algorithme de Karplus-Strong

Réunissions finalement toutes les pièces du puzzle, pour former notre synthétiseur de cordes pincées déclenché par transitoires.

1. Ajoutez un filtre passe-bas `OnePoleLP` dans la boucle de rétroaction du délai de la classe `FeedbackDelay`. On passera le coefficient du filtre `lp` comme un argument supplémentaire de sa méthode publique : `float Process(float input, float feedback, float lp, int delay)`.
2. Faites passer l'impulsion de bruit dans le retard, et envoyez la sortie du retard dans le DAC, de façon à obtenir le flot de signal suivant :



3. Réglez les paramètres du système : 50 samples de bruit générés à chaque impulsion, feedback à 0.95, coefficient du filtre passe-bas 0.9, temps de délai de 100 samples. Déclenchez une impulsion ; quelle est la fréquence de la note que vous entendez ? Quel est le rapport entre le temps de délai et la fréquence de la note entendue ? Jouez avec les autres paramètres pour comprendre leur incidence sur le timbre.
4. Le Do 5 a une fréquence de 130.81 Hz ; le Ré au dessus une fréquence de 146.83 Hz, et le Mi 164.81 Hz. Dans la boucle principale, changez la note jouée à chaque impulsion, et suivre la séquence suivante : Do, Do, Do, Re, Mi, Re, Do, Mi, Re, Re, Do.
5. Servons-nous de l'accéléromètre pour faire des vibratos (petites modulations de la fréquence de la note). Pour cela, multipliez dans votre boucle principale le temps de délai choisi par $1 + 0.01x$, où x est la composante x de l'accéléromètre, variant entre -1 et 1.