USRS09

Traitement du Signal Embarqué

Matthias Puech

STMN I1 — ENJMIN, Cnam

Introduction au traitement du signal embarqué
- Présentation
- Microcontrôleurs et cartes
- Programmation de traitements temps réel

Représentation et traitement des signaux audio

Analyse et détection de transitoire

Synthèse par modèle physique

Traitement du signal embarqué

But de ce cours

- application des théories du signal
  (échantillonnage, filtrage, domaine temporel/fréquentiel…)
- les enjeux de la programmation embarquée
  (ressources limitées, programmation *bare metal*)
- réalisation d'une application audio sur microcontrôleur
  (un synthétiseur déclenché par impulsions sonores)

Traitement du signal embarqué

But de ce cours

- application des théories du signal
  (échantillonnage, filtrage, domaine temporel/fréquentiel...)
- les enjeux de la programmation embarquée
  (ressources limitées, programmation *bare metal*)
- réalisation d'une application audio sur microcontrôleur
  (un synthétiseur déclenché par impulsions sonores)

Evaluation

TP noté à rendre en bi-/trinôme à la fin des séances
⇝ `matthias.puech@lecnam.net`

Programmation embarquée sur microcontrôleur

Microcontrôleur (μC, uC, MCU)

Circuit intégré rassemblant les éléments essentiels d'un ordinateur

- unité de calcul et de contrôle (le coeur)
- mémoire vive
- périphériques
  (ex : port série, timers, DMA, flash...)
- entrées/sorties
  (ex : convertisseurs analogique ⟷ numérique (ADC/DAC))

Exemples d’utilisation

Electroménager & Internet of Things

- montres connectées, bracelets santé
- capteurs domotiques, four, machine à laver, thermostat...
- smartphones, tablettes

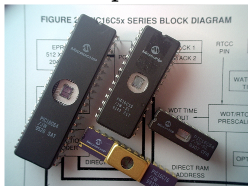
Automobile, Aeronautique

- une voiture moderne intègre ≈30 MCUs
- logiciel critique dans l’avionique (cf. reste du master)

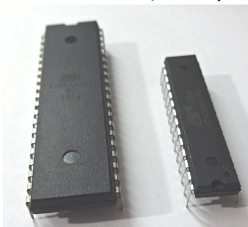
Un foyer moyen possède 4 CPUs et 30 MCUs.

Quelques MCUs fameux dans leur habitat naturel

- Microchip PIC16 (1990, 8 bits, 1MHz, quelques registres)

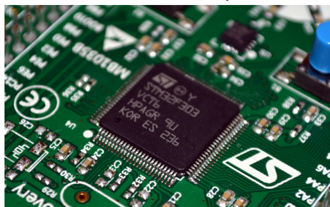


- Microchip PIC12 (1990, très utilisé en DIY)
- Atmel AVR (2000, 8/16 bits, base des Arduino)

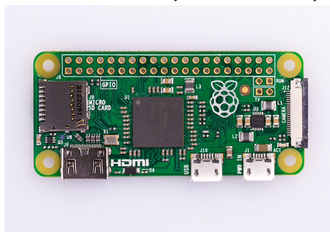


Quelques MCUs fameux dans leur habitat naturel

- ARM (2000) : Coeurs communs, différents constructeurs (ST, Texas Instruments, Microchip. . .)



- SoC Freescale, Broadcom, TI etc. (2010, Raspberry Pi)



1GHz, 64 bits, RAM 512 Mo

Le MCU, un compromis

L'intérêt

- forte intégration (une puce intègre toutes les fonctions)
- faible consommation électrique (1–500mW)
- faible coût (0.10–10€)
- généricité (par rapport au silicium dédié)

Le MCU, un compromis

L'intérêt

- forte intégration (une puce intègre toutes les fonctions)
- faible consommation électrique (1–500mW)
- faible coût (0.10–10€)
- généricité (par rapport au silicium dédié)

Les limitations

- peu de capacité de calcul (1–200 MHz)
- *très* faible stockage (1–512 Ko RAM, 1K–1M flash)

Un MCU n'est *pas* :

- un CPU (*Central Processing Unit*)
  (ex : Intel i7, Apple A11, ...)
  - puissance de calcul supérieure (GHz)
  - pas de mémoire embarquée (quelques registres)
  - pas de périphériques embarqués (bus externes)
- un FPGA (*Field-Programmable Gate Array*)
  (ex : Xilinx, Altera...)
  - circuit intégré reconfigurable
  - grand nombre de portes logiques généralistes
  - "programmation" en Verilog/VHDL

La famille des MCU STMicroelectronics STM32

STM32L0	Cortex M0+, 32MHz, 8Ko SRAM, 32-64Ko flash
STM32F1	Cortex M1, 24-72MHz, 4-96Ko SRAM, 16-1024Ko
STM32F4	Cortex M4 + FPU, <180MHz, 96-384Ko SRAM, 64-256Ko flash
...STM32F7	ARM Cortex-M7F, 216MHz, 512-1024Ko RAM, ...

La famille des MCU STMicroelectronics STM32

STM32L0 Cortex M0+, 32MHz, 8Ko SRAM, 32-64Ko flash

STM32F1 Cortex M1, 24-72MHz, 4-96Ko SRAM, 16-1024Ko

STM32F4 Cortex M4 + FPU, <180MHz, 96-384Ko SRAM, 64-256Ko flash

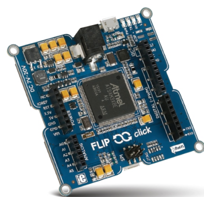
...STM32F7 ARM Cortex-M7F, 216MHz, 512-1024Ko RAM, ...

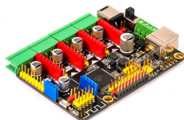
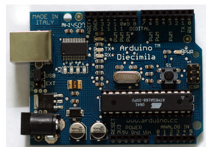
Chaque famille contient un jeu de périphériques embarqués :
(E/S numériques, ADC, DAC, USART, I2C, SPI, USB, DMA, timers...)

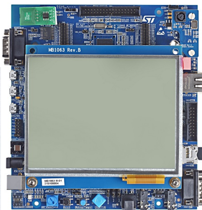
Les cartes d'essai

Si vous voulez étudier les MCUs ou faire du prototypage rapide, pas besoin de concevoir une carte à chaque fois, il existe de nombreuses *cartes d'essai*.

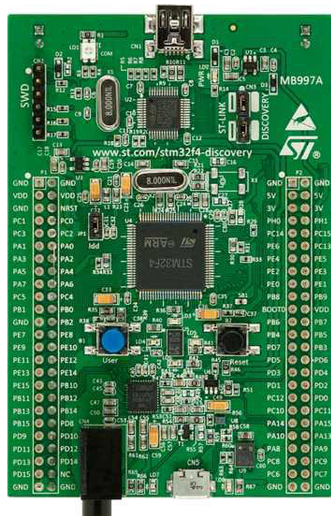
- produits d'appel pour les fabricants, donc très bon marché (≈20€)
- divers périphériques embarqués : capteurs, écrans, réseau, audio...
- programmation facile (interface USB)





La carte STM32F4-discovery



- STM32F407 : 72MHz, 192Ko RAM, 1Mo flash
- documentation et schéma dispo
- programmeur USB intégré
- périphériques externes :
  - port USB utilisateur
  - alimentation par USB ou externe (pile)
  - accéléromètre 3D
  - DAC audio + préampli casque
  - microphone MEMS
  - 4 LEDs utilisateur
  - 1 bouton utilisateur

Modèle de programmation

Le coeur ARM : une machine load/store

- $\approx$ 30 registres de 32 bits
  (ex : *R0-R12, SP, PC*)
- cycle *fetch/decode/execute*
  (avec pipelining basique)
- des instructions qui changent l'état des registres
  (ex : *add, bl*)
- des instructions qui lisent/écrivent dans la mémoire
  (ex : *ldr, str*)
- ...et c'est tout !

Modèle de programmation

Toute l'interaction avec les périphériques se fait par écriture dans des "cases mémoire" spéciales.

Exemple

Pour allumer les LEDs de la carte, il faut :

- écrire 0x00200000 à l'adresse 0x40021014,
- écrire 0x55550000 à l'adresse 0x48000000,
- écrire 0xFFFFFFFF à l'adresse 0x48001014.

(tout ça est documenté, pas besoin de se souvenir des adresses !)

Espace d'adressage

L'*espace d'adressage* est le *mapping* des adresses vers la RAM, la mémoire flash, les registres des périphériques...
(dans la documentation du MCU)

Modèle de programmation

Interruptions

cycle *fetch/decode/execute*. . . à une exception près :

- un évenement matériel peut survenir à tout moment
  (*interrupt request* ou *IRQ*)
- le processeur sauvegarde alors son “contexte” (là où il en est) et saute à une adresse donnée en mémoire (le code qui s’y trouve est l’*interrupt handler* ou *ISR*)
- une fois l’ISR exécuté, le processeur restaure son “contexte” est continue l’exécution là où il en était.

Application

- on configure un périphérique pour émettre une IRQ quand évenement
  (ex : message entrant sur le port USB, période d’un *timer* écoulé. . .)
- le handler exécute du code pour traiter l’interruption
  (ex : décodage du message et réponse, allumage d’une LED. . .)

Comment développer pour les ARM STM32 ?

Langages

C ou C++...
(ou tout langage compilant vers ARMv7)

Environnements de développement

Il en existe plusieurs (IDE, compilateur, debugger etc.) :

- Keil IDE / ArmCC (Keil),
- IAR Embedded Workbench (IAR),
- mBed (ARM)
- SW4STM/Eclipse/CubeMX/gcc (STMicroelectronics)
- ...
- `gcc/gdb/make/emacs` :)

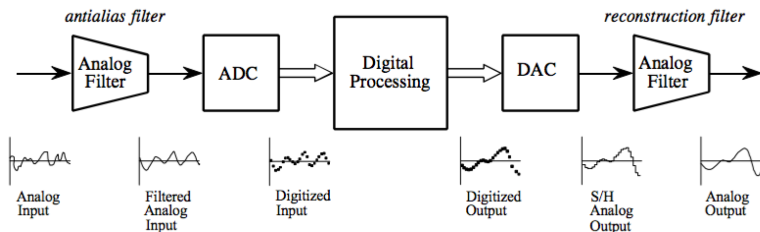
Librairies

…et plusieurs librairies d'abstraction pour accéder aux registres et configurer/interagir avec les périphériques :

- CMSIS (ARM) (paraphrase du manuel de référence)
- HAL (STMicroelectronics) (bibliothèque d'abstraction matérielle)
- mBed (ARM)
- Arduino (stm32duino)
- FreeRTOS (système d'exploitation embarqué temps réel)

De l'ADC au DAC

Un schéma courant de processeur de signal audio :
(ex : une pédale de guitare)



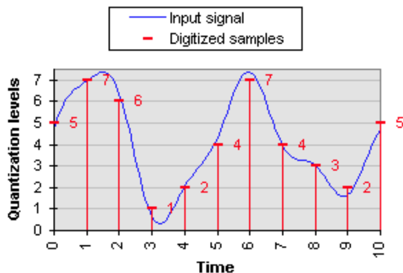
l'ADC (convertisseur analogique-numérique) échantillonne le signal analogique en un flux de valeurs numériques discrètes

le DAC (convertisseur numérique-analogique) reconstruit un signal analogique continu à partir d'un flux de valeurs numériques discrètes

entre les deux le DSP modifie arbitrairement le flux de valeurs

Échantillonnage et quantification

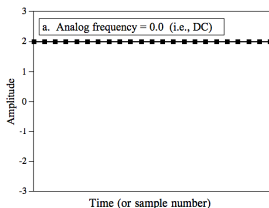
L'ADC découpe le signal dans le temps et dans l'amplitude

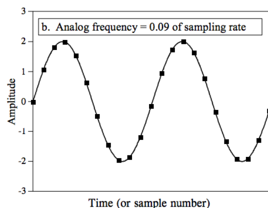


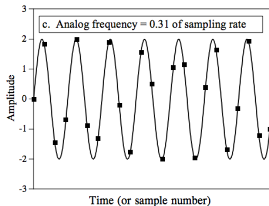
- la fréquence d'échantillonnage est notée f_S
  (ex : 32000Hz, 44100Hz, 48000Hz, 96000Hz)
- Nyquist : un flux de valeurs peut représenter un signal qui ne contient aucune composante fréquencielle $> f_S/2$
- le nombre de valeurs quantifiées est noté en bits
  (ex : 8 bits = 256 valeurs, 16 bits = 65536 valeurs)

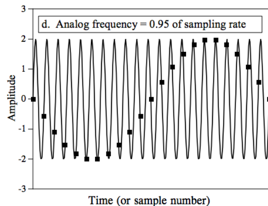
Effet de l'échantillonnage

Le signal analogique est échantillonné tous les $1/f_s$
(les valeurs intermédiaires sont ignorées)



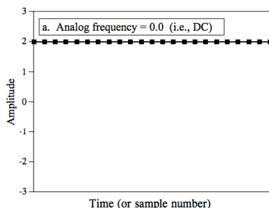


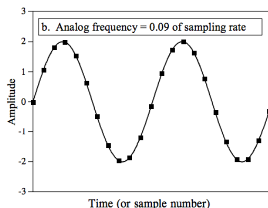


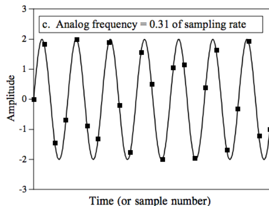


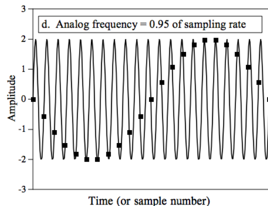
Effet de l'échantillonnage

Le signal analogique est échantillonné tous les $1/f_s$
(les valeurs intermédiaires sont ignorées)





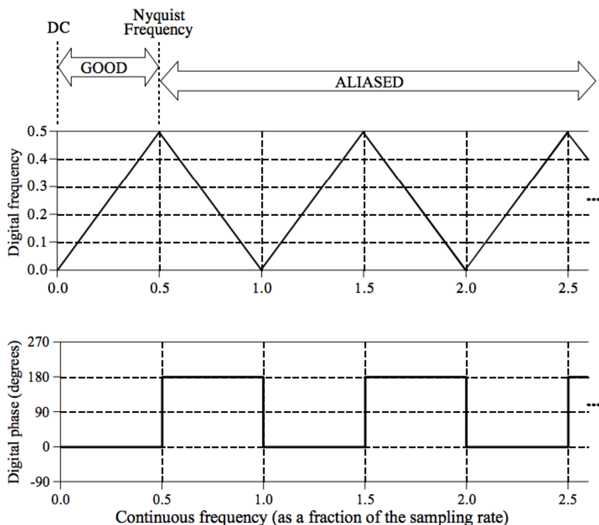




⇝ on peut confondre une sinusoïde de fréquence f avec une autre de fréquence $|f_S - f|$ (par exemple)

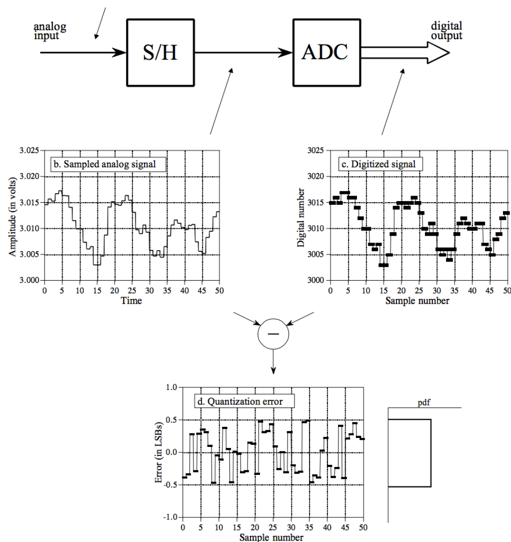
Effet de l'échantillonnage

⇝ c'est le phénomène de l'*aliasing* :



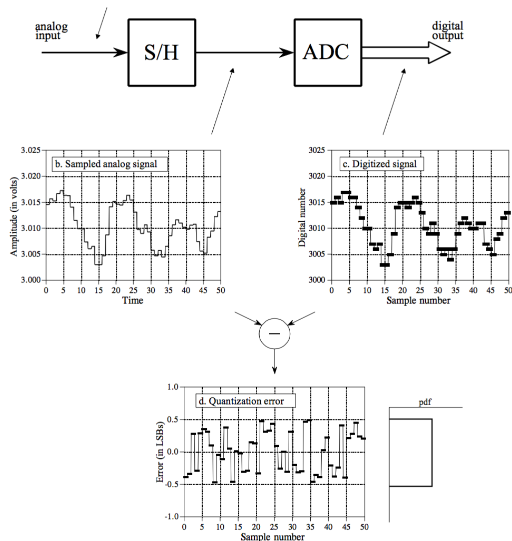
Effet de la quantification

L'ADC approxime la mesure à la valeur quantifiée la plus proche :



Effet de la quantification

L'ADC approxime la mesure à la valeur quantifiée la plus proche :



⇝ la quantification rajoute du bruit au signal

Traitement numérique du signal audio

Le but du jeu

Dans notre scénario (pédale de guitare) :

- on reçoit de l'ADC un échantillon f_S fois par secondes
- cet échantillon est une valeur numérique sur n bits
  (ex : 16 bits, i.e. `short` en C : valeurs dans $[-32768; 32767]$)
- -32768 représente la valeur minimum, 32767 la valeur maximum, 0 la valeur "au repos" (silence audio).
- on produit un échantillon que l'on envoit au DAC

Traitement numérique du signal audio

Le but du jeu

Dans notre scénario (pédale de guitare) :

- on reçoit de l'ADC un échantillon f_S fois par secondes
- cet échantillon est une valeur numérique sur n bits
  (ex : 16 bits, i.e. `short` en C : valeurs dans $[-32768; 32767]$)
- -32768 représente la valeur minimum, 32767 la valeur maximum, 0 la valeur “au repos” (silence audio).
- on produit un échantillon que l'on envoit au DAC

On peut modéliser cela par une fonction :

```
short Process(short in) { ... }
```

qui sera appelé par le système quand un nouvel échantillon sera nécessaire.

Traitement numérique du signal audio

Exemples

- traitement “silence”, qui ignore l’entrée et renvoit 0 tout le temps

```
short Process(short in) { return 0; }
```

Traitement numérique du signal audio

Exemples

- traitement “silence”, qui ignore l’entrée et renvoit 0 tout le temps
  ```
  short Process(short in) { return 0; }
  ```
- traitement “*bypass*”, qui envoit l’entrée vers la sortie sans modification :
  ```
  short Process(short in) { return in; }
  ```

Traitement numérique du signal audio

Exemples

- traitement “silence”, qui ignore l’entrée et renvoit 0 tout le temps

  ```
  short Process(short in) { return 0; }
  ```

- traitement “*bypass*”, qui envoit l’entrée vers la sortie sans modification :

  ```
  short Process(short in) { return in; }
  ```

- le traitement “attenuation” :
  (rapproche toutes les valeurs de zéro)

  ```
  short Process(short in) { return in/10; }
  ```

Traitement numérique du signal audio

Exemples

- traitement “silence”, qui ignore l’entrée et renvoit 0 tout le temps
  ```
  short Process(short in) { return 0; }
  ```
- traitement “*bypass*”, qui envoit l’entrée vers la sortie sans modification :
  ```
  short Process(short in) { return in; }
  ```
- le traitement “attenuation” :
  (rapproche toutes les valeurs de zéro)
  ```
  short Process(short in) { return in/10; }
  ```
- le traitement “amplification” (buggé) :
  ```
  short Process(short in) { return in*10; }
  ```

Traitement numérique du signal audio

Exemples

- traitement "silence", qui ignore l'entrée et renvoit 0 tout le temps
  ```
  short Process(short in) { return 0; }
  ```
- traitement "*bypass*", qui envoit l'entrée vers la sortie sans modification :
  ```
  short Process(short in) { return in; }
  ```
- le traitement "attenuation" :
  (rapproche toutes les valeurs de zéro)
  ```
  short Process(short in) { return in/10; }
  ```
- le traitement "amplification" (buggé) :
  ```
  short Process(short in) { return in*10; }
  ```
  (attention ! si $\texttt{in} * 10 > 32767$, on a overflow)

La ligne de retard naïve

Notre premier traitement non-trivial

On copie dans la sortie l'entrée *retardée* de R échantillons :

$$y[n] = x[n - R]$$

(y signal de sortie, x signal d'entrée, n le numéro de l'échantillon courant)

Applications

- c'est la base de tous les filtres numériques
  (ajouter à un signal des versions retardées de lui-même)
- effets d'echos et de réverbérations
  (simulation de l'effet de la vitesse finie du son)

La ligne de retard naïve

Exemples

- Retard de 1 échantillon ($R = 1$), i.e. $1/f_S$ secondes :
  (à $f_S = 48000$ KHz, cela fait 20 μs)

```
short buf;
short Process(short in) {
  short output = buf;
  buf = in;
  return output;
}
```

La ligne de retard naïve

Exemples

- Retard de 2 échantillon ($R = 2$) :

```
short buf1, buf2;
short Process(short in) {
  short output = buf2;
  buf2 = buf1;
  buf1 = in;
  return output;
}
```

La ligne de retard naïve

Exemples

- Retard de N échantillon :

```
short buf[N];
short Process(short in) {
  short output = buf[N-1];
  for (int i=N-2; i>=0; i--)
    buf[i] = buf[i+1];
  return output;
}
```

La ligne de retard naïve

Exemples

- Retard de N échantillon :

```
short buf[N];
short Process(short in) {
  short output = buf[N-1];
  for (int i=N-2; i>=0; i--)
    buf[i] = buf[i+1];
  return output;
}
```

Problème

le temps de calcul est proportionnel à la longueur du retard !

Une structure omniprésente : le *ring buffer*

Solution

Au lieu de décaler tous les échantillons à chaque itération, on :

- se donne un tableau t de taille N auquel on accède modulo N (ex : la case $N + 7 =$ la case 8)
- conserve un curseur c vers la prochaine case à écrire
- incrémente c à chaque écriture

Une structure omniprésente : le *ring buffer*

Solution

Au lieu de décaler tous les échantillons à chaque itération, on :

- se donne un tableau t de taille N auquel on accède modulo N (ex : la case $N + 7 =$ la case 8)
- conserve un curseur c vers la prochaine case à écrire
- incrémente c à chaque écriture

Opérations

write(v) écrit v sous le curseur c

read(n) lit la n-ième valeur écrite

Une structure omniprésente : le *ring buffer*

Solution

Au lieu de décaler tous les échantillons à chaque itération, on :

- se donne un tableau t de taille N auquel on accède modulo N (ex : la case $N + 7 =$ la case 8)
- conserve un curseur c vers la prochaine case à écrire
- incrémente c à chaque écriture

Opérations

write(v)	écrit v sous le curseur c
read(n)	lit la n-ième valeur écrite

Exemple

write(1) ; write(2) ; read(0) == 2
write(3) ; write(4) ; write(5) ;
read(0) == 5 ; read(3) == 2

La ligne de retard

Implémentation efficace avec *ring buffer*

- ```
  RingBuffer buf {N};
  short Process(short in) {
    short output = buf.read(N-1);
    buf.write(in);
    return output;
  }
  ```
  ```

La ligne de retard

Implémentation efficace avec *ring buffer*

- ```
  RingBuffer buf {N};
  short Process(short in) {
    short output = buf.read(N-1);
    buf.write(in);
    return output;
  }
  ```

Remarque

temps de calcul constant pour n'importe quel N.
  ```

Présentation du code source fourni

On fournit un carcan C++ qui contient la configuration :

- du microphone (`microphone.hh`),
- du DAC (`dac.hh`),
- de l'accéléromètre (`accelerometer.hh`)
- des LEDs et du bouton utilisateur (`{leds|button}.hh`)
- des horloges du système (`system.hh`)

Présentation du code source fourni

On fournit un carcan C++ qui contient la configuration :

- du microphone (`microphone.hh`),
- du DAC (`dac.hh`),
- de l'accéléromètre (`accelerometer.hh`)
- des LEDs et du bouton utilisateur (`{leds|button}.hh`)
- des horloges du système (`system.hh`)

Utilisation

1. lancer la machine virtuelle contenant la chaîne de compilation
2. brancher la carte
3. éditer le code
4. dans le répertoire `stm32-dsp-template`, taper `make flash` pour compiler et envoyer le code sur la carte
5. GOTO 3

Présentation du code source fourni

main.cc

- déclare la classe principale `struct Main`
  (et déclare un objet “`_`” de cette classe)
- initialise les périphériques
  (constructeurs des champs de `Main`)
- règle la fréquence d’échantillonnage f_S à 48kHz
- le constructeur de `Main` règle le volume du DAC
- contient la boucle principale (méthode `Process()`)

- contient code appelé toutes les millisecondes
  (méthode `onSysTick()`)

Présentation du code source fourni

La boucle principale

```
void Process(short *in, ShortFrame *out, size_t size) {
  accel_.ReadAccelData(&d);
  for (size_t i=0; i<size; i++) {
    out[i].l = in[i];
    out[i].r = in[i];
  }
}
```

Remarques

- cette méthode est appelée 1500 fois par secondes
- les échantillons sont traités par paquets de `size` (=32)
  (`in` et `out` sont des tableaux de taille `size`)
- on lit dans `in` les samples du micro ; on écrit dans `out` les samples pour le DAC (stéréo).
- à chaque appel on lit les données de l'accéléromètre dans `d`

Présentation du code source fourni

Structures de donnée utilisées

- un sample stéréo est représenté par une *frame* de deux samples `l` et `r` :

```
struct ShortFrame {
  short l;
  short r;
};
```

- les données de l'accéléromètre sont représentés par 3 valeurs (une par axe) :

```
struct AccelData {
  short x;
  short y;
  short z;
};
```

Introduction au traitement du signal embarqué

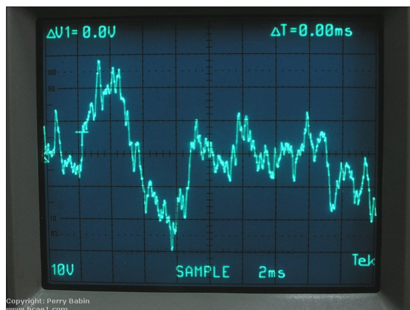
Représentation et traitement des signaux audio
- Représentation des signaux
- Traitement par bloc
- Filtrage dans le domaine temporel

Analyse et détection de transitoire

Synthèse par modèle physique

Le signal audio analogique

Mesure de pression/voltage au cours du temps



- centré autour de 0
  (pas de composante continue dans la mesure)
- précision infinie en temps et en amplitude
  (fonction de $\mathbb{R}$ dans $\mathbb{R}$)
- en pratique, borné par les limites du capteur
  (par convention, appelons -1 le min et 1 le max)

Représentation des signaux audio

PCM : *Pulse Code Modulation*

Le codage par défaut dans 99% des applications

- le signal est échantillonné à période régulière
- chaque échantillon est converti en un code de longeur fixe
  (indépendemment des précédents et des suivants)

Représentation des signaux audio

PCM : *Pulse Code Modulation*

Le codage par défaut dans 99% des applications

- le signal est échantillonné à période régulière
- chaque échantillon est converti en un code de longeur fixe
  (indépendemment des précédents et des suivants)
- la valeur des échantillons est bornée
  (par convention, $\min = -1$, $\max = 1$)

+ accès aléatoire en temps
+ pas de conversion de/vers un ADC/DAC

– débit très élevé
  (ex : 16 bits/44.1kHz/stéréo = 1.4Mbit/s)

Représentation des signaux audio

PCM : *Pulse Code Modulation*

Le codage par défaut dans 99% des applications

- le signal est échantillonné à période régulière
- chaque échantillon est converti en un code de longeur fixe
  (indépendemment des précédents et des suivants)
- la valeur des échantillons est bornée
  (par convention, $\min = -1$, $\max = 1$)

+ accès aléatoire en temps
+ pas de conversion de/vers un ADC/DAC

– débit très élevé
  (ex : 16 bits/44.1kHz/stéréo = 1.4Mbit/s)

Alternatives

- PDM : *Pulse Density Modulation* (Sony “Direct Stream Digital”)
- Modèles à prédiction linéaire (FLAC)
  (le codage d’un échantillon dépend des précédents)

Représentation des signaux audio

Quel codage pour l'amplitude des échantillons ?

$$k_T : \mathbb{R} \to T$$

Représentation des signaux audio

Quel codage pour l'amplitude des échantillons ?

$$k_T : \mathbb{R} \to T$$

Codage régulier

Les valeurs sont réparties linéairement entre le min et le max
$\rightsquigarrow$ nombres à virgule fixe, représentés par `{u|}int{8|16|32}_t`

Représentation des signaux audio

Quel codage pour l'amplitude des échantillons ?

$$k_T : \mathbb{R} \to T$$

Codage régulier

Les valeurs sont réparties linéairement entre le min et le max
⇝ nombres à virgule fixe, représentés par `{u|}int{8|16|32}_t`

Exemple (Codage "Q1.15")

$T = \texttt{int16_t}$

- $k_{\texttt{int16_t}}(-1) = -32768$ (amplitude minimum)
- $k_{\texttt{int16_t}}(0) = 0$ (amplitude médiane, "au repos")
- $k_{\texttt{int16_t}}(1) \simeq 32767$ (amplitude maximum)

Représentation des signaux audio

Quel codage pour l'amplitude des échantillons ?

$$k_T : \mathbb{R} \to T$$

Codage régulier

Les valeurs sont réparties linéairement entre le min et le max
⇝ nombres à virgule fixe, représentés par `{u|}int{8|16|32}_t`

Exemple (Codage "Q1.15")

$T = \texttt{int16_t}$

- $k_{\texttt{int16_t}}(-1) = -32768$ (amplitude minimum)
- $k_{\texttt{int16_t}}(0) = 0$ (amplitude médiane, "au repos")
- $k_{\texttt{int16_t}}(1) \simeq 32767$ (amplitude maximum)

+ calcul entier implémenté sur tous les MCU
  (code portable : pas besoin de FPU)
+ résolution constant (plus simple)
– attention à l'*overflow* quand on dépasse les min/max

Représentation des signaux audio

Codage (quasi-)exponentiel

Les valeurs sont répartie exponentiellement :
plus de valeurs près de 0 que près de grands nombres
$\rightsquigarrow$ nombres à virgule flottante, représentés par `float` ou `double`

Exemple (Codage flottant 32-bits)

$T = \texttt{float}$

- $k_{\texttt{float}}(-1) = -1$ (amplitude minimum)
- $k_{\texttt{float}}(0) = 0$ (amplitude médiane, “au repos”)
- $k_{\texttt{float}}(1) = 1$ (amplitude maximum)

Représentation des signaux audio

Codage (quasi-)exponentiel

Les valeurs sont répartie exponentiellement :
plus de valeurs près de 0 que près de grands nombres
⇝ nombres à virgule flottante, représentés par `float` ou `double`

Exemple (Codage flottant 32-bits)

$T = \texttt{float}$

- $k_{\texttt{float}}(-1) = -1$ (amplitude minimum)
- $k_{\texttt{float}}(0) = 0$ (amplitude médiane, “au repos”)
- $k_{\texttt{float}}(1) = 1$ (amplitude maximum)

+ énorme *headroom* : on peut largement dépasser -1/1
  (peu de risque d'*overflow*)
+ peu de perte d'information quand on divise

– résolution variable
  (erreurs d'arrondi plus complexes à comprendre)

Calcul à virgule fixe

Les processeurs usuels ne supportent pas les calculs à virgule fixe
(que les calculs entiers, et (parfois) flottants)

⇝ On peut émuler les nombres à virgule fixe avec des entiers !

Calcul à virgule fixe

Les processeurs usuels ne supportent pas les calculs à virgule fixe
(que les calculs entiers, et (parfois) flottants)

⇝ On peut émuler les nombres à virgule fixe avec des entiers !

Les formats Q

Q$m.n$ désigne le codage sur $m+n$ bits
avec m bits avant la virgule et n bits après (signé)

- La partie m est en complément à deux (ou si $m=0$ alors c'est n)
- 2^{m+n} valeurs comprises dans $[-2^{m-1}; 2^{m-1}-2^{-n}]$
  (résolution de 2^{-n})

Calcul à virgule fixe

Les processeurs usuels ne supportent pas les calculs à virgule fixe
(que les calculs entiers, et (parfois) flottants)

⇝ On peut émuler les nombres à virgule fixe avec des entiers !

Les formats Q

Q$m.n$ désigne le codage sur $m+n$ bits
avec m bits avant la virgule et n bits après (signé)

- La partie m est en complément à deux (ou si $m = 0$ alors c'est n)
- 2^{m+n} valeurs comprises dans $[-2^{m-1}; 2^{m-1} - 2^{-n}]$
  (résolution de 2^{-n})

Exemples

Q0.16	65536 valeurs $\in [-0.5; 0.4999847412]$
Q1.15	65536 valeurs $\in [-1; 0.9999694824]$
Q15.1	65536 valeurs $\in [-16384.0; 16383.5]$
Q16.0	les entiers 16 bits signés $\in [-32768; 32767]$

Calcul à virgule fixe

Conversion $\mathbb{R} \longleftrightarrow$ Qm.n

$$k_{\mathrm{Q}m.n}(x) = \lfloor 2^n x \rfloor \qquad k^{-1}_{\mathrm{Q}m.n}(y) = 2^{-n} y$$

Opérations mathématiques

- $k_{\mathrm{Q}(m+1).n}(x \pm y) = k_{\mathrm{Q}m.n}(x) \pm k_{\mathrm{Q}m.n}(y)$
  (additions et soustractions comme sur les entiers ;
  attention : il faut un bit de plus dans le résultat !)
- $k_{\mathrm{Q}m.n}(x \times y) = k_{\mathrm{Q}m.n}(x) \times k_{\mathrm{Q}m.n}(y) \times 2^{-n}$
  (après une multiplication, il faut diviser par 2^n ;
  attention : il faut $m + n$ bits pour stocker $x \times y$!)
- exercice : division entière (= arrondi)

Calcul à virgule fixe

Conversion $\mathbb{R} \longleftrightarrow$ Qm.n

$$k_{\mathrm{Q}m.n}(x) = \lfloor 2^n x \rfloor \qquad k^{-1}_{\mathrm{Q}m.n}(y) = 2^{-n} y$$

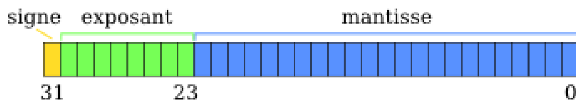
Opérations mathématiques

- $k_{\mathrm{Q}(m+1).n}(x \pm y) = k_{\mathrm{Q}m.n}(x) \pm k_{\mathrm{Q}m.n}(y)$
  (additions et soustractions comme sur les entiers ;
  attention : il faut un bit de plus dans le résultat !)
- $k_{\mathrm{Q}m.n}(x \times y) = k_{\mathrm{Q}m.n}(x) \times k_{\mathrm{Q}m.n}(y) \times 2^{-n}$
  (après une multiplication, il faut diviser par 2^n ;
  attention : il faut $m+n$ bits pour stocker $x \times y$!)
- exercice : division entière (= arrondi)

Exemples

```
int16_t q_add(int16_t a, int16_t b) {return a + b; }
int16_t q_mul(int16_t a, int16_t b) {
  return (int32_t)a * (int32_t)b  » N;
}
```

Rappel : Nombres flottants IEEE 754 32 bits



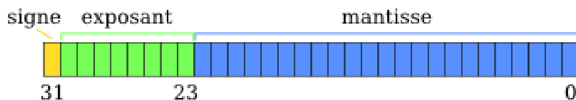
signe s 0 ⇝ positif, 1 ⇝ négatif

exposant e $\in [-128; 127]$

mantisse m $\in [1; 1,9999998808]$

$$k^{-1}_{\texttt{float}}(s, e, m) = (-1)^s \times m \times 2^e$$

Rappel : Nombres flottants IEEE 754 32 bits



signe s : $0 \rightsquigarrow$ positif, $1 \rightsquigarrow$ négatif

exposant e : $\in [-128; 127]$

mantisse m : $\in [1; 1,9999998808]$

$$k^{-1}_{\texttt{float}}(s, e, m) = (-1)^s \times m \times 2^e$$

Représentation à résolution variable

L'écart entre deux `floats` successifs est proportionnel à (une approximation du) nombre représenté
$\rightsquigarrow$ codage *quasi-exponentiel*

Rappel : Nombres flottants IEEE 754 32 bits

Remarques

- il y a des exceptions à cet encodage
  (± 0, $\pm\infty$, `NaN`, nombres denormaux)
- 1/4 de tous les `float` sont dans $[-1; 1]$
- nombres représentables $\in [1.175 \times 10^{-38}; 3.403 \times 10^{38}]$
- la résolution varie :
  - 1.4×10^{-45} autour de 0
  - 0.6×10^{-7} autour de ± 1
  - 2×10^{31} autour de $\pm\infty$

Avantages / Dangers

\+ division et multiplications par 2^n sans perte d'information
(sauf aux limites des nombres représentables)

! ne pas tester l'égalité de deux flottants

! ne pas ajouter un très petit nombre et un très grand

En pratique, comment choisir un codage ?

- ce qui arrive du microphone est en Q1.15
- ce qui sort vers le DAC est en Q1.15
- *conseil* : entre les deux, travailler en `float`

```
void Process(short *in,
             ShortFrame *out,
             size_t size) {
  for (int i=0; i<size; i++) {
    float input = (float)in[i] / 32768.0f;
    float output = ...;
    out[i] = (short)(output * 32768.0f);
  }
}
```

Traitement par bloc

Problème

- tous les samples ne demandent pas le même temps de calcul
  (ex : un `if` dont la condition dépend de l'entrée)
- si un sample met $> 1/f_S$ sec à être calculé, *buffer underrun*
- certains calculs peuvent être faits moins rapidement que f_S
  (ex : lecture/calcul des paramètres "lents")

Traitement par bloc

Problème

- tous les samples ne demandent pas le même temps de calcul
  (ex : un `if` dont la condition dépend de l'entrée)
- si un sample met $> 1/f_S$ sec à être calculé, *buffer underrun*
- certains calculs peuvent être faits moins rapidement que f_S
  (ex : lecture/calcul des paramètres “lents”)

Solution

Traitement par *blocs* :

- le microphone accumule S samples
  (dans notre cas, $S = 32$)
- puis on calcule S samples de sortie
- puis le DAC lit les S

\+ amortit le temps de calcul sur S samples

– latence accrue : S/f_S (contre)

Exemple : l'accéléromètre

la lecture des données prend ≈ 2ms ;
trop long pour être fait à chaque sample !

leadsto une lecture par bloc

```
void Process(short *in, ShortFrame *out, size_t
 size) {
  accel_.ReadAccelData(&d);

  for (size_t i=0; i<size; i++) {
    out[i].l = ...;
    out[i].r = ...;
  }
}
```

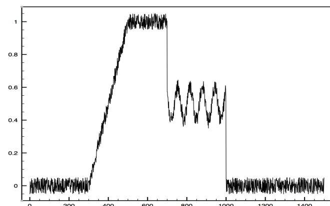
Problème du bruit dans les contrôles

Dans un système audio embarqué,

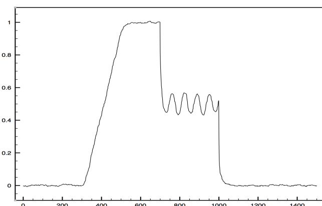
- les paramètres de traitement sont acquis par un ADC
  (ex : potentiomètre, capteur de température...)
  ⇝ ce que l'on capture est *signal + bruit*
- ces paramètres évoluent beaucoup plus lentement que l'audio
  (peu de signaux >1kHz)
- le bruit peut causer des perturbations audibles
  (ex : potentiomètre de volume bruité = crachotements)

⇝ on veut filtrer le signal entrant

Exemple



filtrage



Caractérisation d'un filtre dans le temps

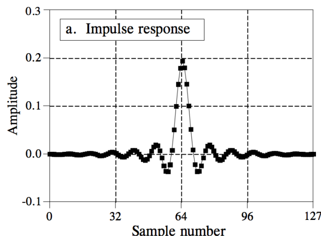
Réponse impulsionnelle réponse à la fonction de dirac :

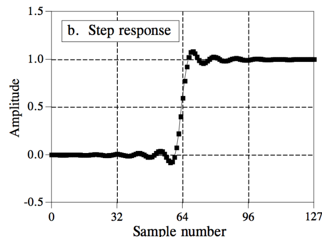
$$\delta[n] = \begin{cases} 1 \text{ si } n = 0 \\ 0 \text{ sinon} \end{cases}$$

Réponse indicielle réponse à la fonction échelon :

$$H[n] = \begin{cases} 1 \text{ si } n \geq 0 \\ 0 \text{ sinon} \end{cases}$$

Exemple



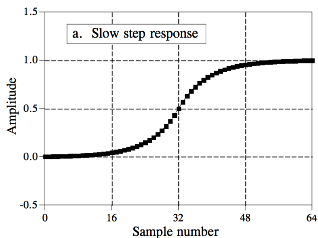


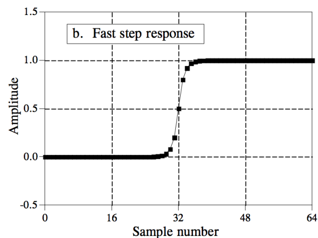
Caractérisation d'un filtre dans le temps

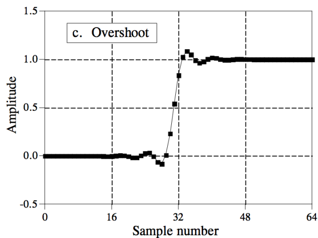
La réponse indicielle nous indique comment le filtre réagit à un mouvement brusque

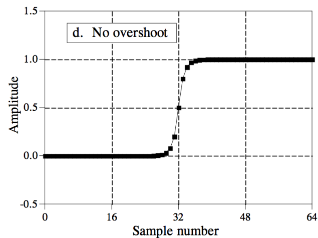
POOR

GOOD

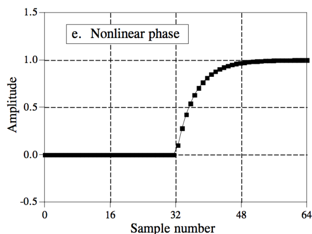


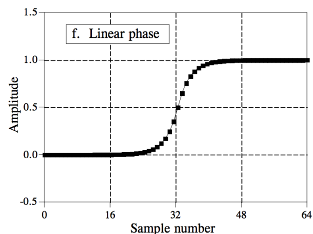






Caractérisation d'un filtre dans le temps



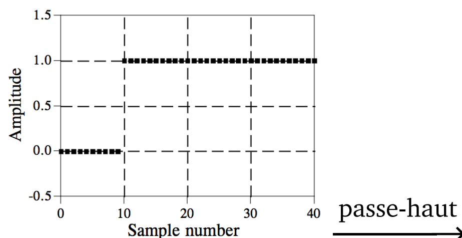


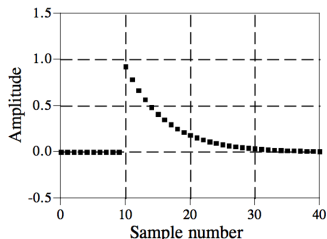
Caractérisation d'un filtre dans le temps

Effets des filtres dans le temps

passe-bas "lisse" le signal entrant / en fait la moyenne (cf. exemples ci-dessus)

passe-haut "élastique" qui rammène le signal vers 0 (plus ou moins vite)





Le filtre passe-bas à un pôle

$$\boxed{y[n] = y[n-1] + C(x[n] - y[n-1])}$$

C est le *coefficient* du filtre

$x[n] - y[n-1]$ est l'*erreur*
(différence entre l'entrée et la sortie)

le mouvement de $y[n]$ est proportionnel à l'erreur

Le filtre passe-bas à un pôle

$$\boxed{y[n] = y[n-1] + C(x[n] - y[n-1])}$$

C est le *coefficient* du filtre

$x[n] - y[n-1]$ est l'*erreur*
(différence entre l'entrée et la sortie)

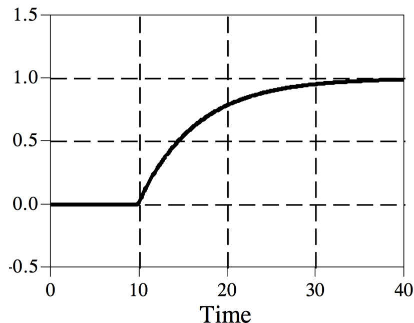
le mouvement de $y[n]$ est proportionnel à l'erreur

Remarques

- si $C = 0$, alors $y[n] = y[n-1]$
  (la sortie est immobile)
- si $C = 1$, alors $y[n] = x[n]$ (la sortie suit exactement l'entrée)
- si $0 < C < 1$, $y[n]$ "traîne" derrière $x[n]$

Caractérisation du passe-bas à un pôle

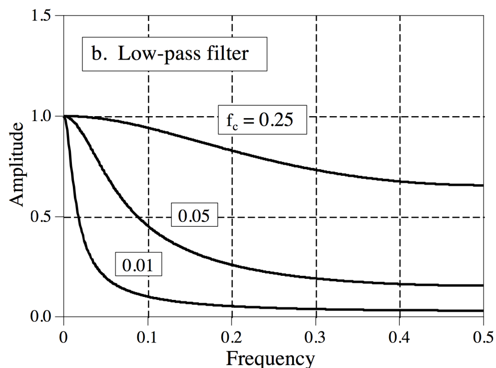
Réponse impulsionnelle



réponse exponentielle, infinie
⇝ c'est la simulation d'un filtre analogique RC

Caractérisation du passe-bas à un pôle

Réponse en fréquence



pente de -6dB/octave
⇝ un mauvais filtre “en fréquence”

Introduction au traitement du signal embarqué

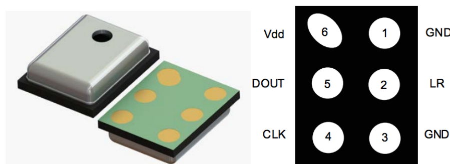
Représentation et traitement des signaux audio

Analyse et détection de transitoire
- Parenthèse : Microphone PDM et filtrage CIC
- Détection de transitoires

Synthèse par modèle physique

Le microphone sur notre carte

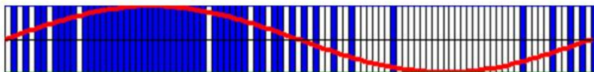
ST-Microelectronics MP45DT02-M



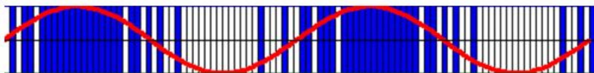
- microsystème électromécanique (MEMS)
  (composant mécanique sur silicium, puce montée en surface)
- interface purement numérique
  (pas besoin d'un ADC)
- entrée horloge à 3.2 MHz CLK
- sortie numérique DOUT, 1 bit par période d'horloge
- codage de l'audio en *Pulse Density Modulation* (PDM)

Pulse-density modulation

Encodage d’un signal analogique en signal binaire dans lequel l’amplitude est codée par la relative densité de 1 par rapport aux 0



An example of PDM of 100 samples of one period of a sine wave. 1s represented by blue, 0s represented by white, overlaid with the sine wave.



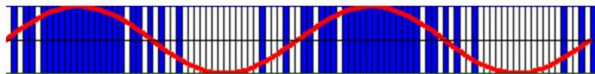
A second example of PDM of 100 samples of two periods of a sine wave of twice the frequency

- que des 1 (resp. 0) $\rightsquigarrow$ amplitude maximale (resp. minimale)
- autant de 0 que de 1 en moyenne $\rightsquigarrow$ amplitude médiane

Pulse-density modulation

Encodage d’un signal analogique en signal binaire dans lequel l’amplitude est codée par la relative densité de 1 par rapport aux 0

An example of PDM of 100 samples of one period of a sine wave. 1s represented by blue, 0s represented by white, overlaid with the sine wave.



A second example of PDM of 100 samples of two periods of a sine wave of twice the frequency

- que des 1 (resp. 0) $\rightsquigarrow$ amplitude maximale (resp. minimale)
- autant de 0 que de 1 en moyenne $\rightsquigarrow$ amplitude médiane

À ne pas confondre avec

Pulse Width Modulation (PWM) : amplitude codée par la largeur d’impulsion d’un signal rectangulaire.

Conversion PDM vers PCM

Filtrage

Pour évaluer la densité du signal PDM, on fait une “*moyenne*” des m dernières valeurs ⇝ filtre passe-bas

Phénomène de *bit growth*

il faut m bits pour stocker la moyenne de m bits

Conversion PDM vers PCM

Filtrage

Pour évaluer la densité du signal PDM, on fait une "*moyenne*" des m dernières valeurs $\rightsquigarrow$ filtre passe-bas

Phénomène de *bit growth*

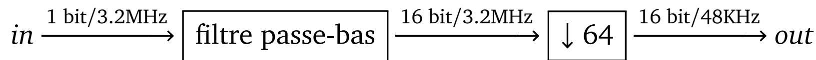
il faut m bits pour stocker la moyenne de m bits

Décimation ↓

Pour compenser la perte d'information, il y aura moins besoin d'échantillons PCM que d'échantillons PDM binaires (64 est un ratio usuel)

$$\text{signal 1 bit/3.2 MHz} \xrightarrow{\text{décimation}} \text{signal 16 bits/48kHz}$$

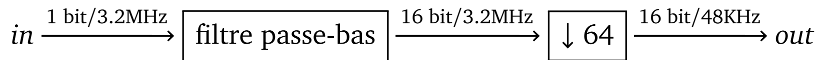
Conversion PDM vers PCM



Problèmes

- le filtrage doit se faire à 3.2MHz
  (coûteux)
- on jette 63 samples calculés

Conversion PDM vers PCM



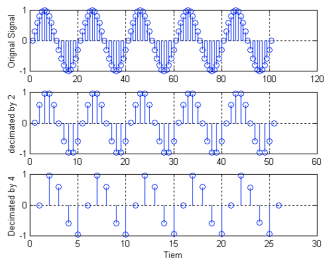
Problèmes

- le filtrage doit se faire à 3.2MHz (coûteux)
- on jette 63 samples calculés

⇝ peut-on faire mieux ?

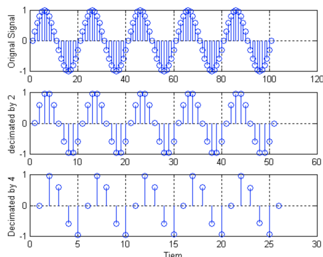
Décimation (ou sous-échantillonnage)

Division de la fréquence d'échantillonnage d'un signal par R, réalisée en ne gardant qu'un échantillon sur R.



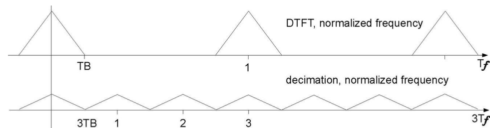
Décimation (ou sous-échantillonnage)

Division de la fréquence d'échantillonnage d'un signal par R, réalisée en ne gardant qu'un échantillon sur R.



Effet sur le spectre

Le spectre du signal original est répété R fois ;
on doit filtrer le signal original à $1/R$ pour éviter l'aliasing



Filtre moyenne

Comment calculer la moyenne glissante $y[n]$ d'un signal $x[n]$?

$$y[n] = \frac{1}{D} \sum_{i=0}^{D} x[n-i] \qquad \text{(sur } D \text{ points)}$$

Filtre moyenne

Comment calculer la moyenne glissante $y[n]$ d'un signal $x[n]$?

$$y[n] = \frac{1}{D}\sum_{i=0}^{D} x[n-i] \qquad \text{(sur } D \text{ points)}$$

Problème

Complexité $O(D)$ (impraticable pour $D > 10$…surtout à 3.2MHz !)

Filtre moyenne

Comment calculer la moyenne glissante $y[n]$ d'un signal $x[n]$?

$$y[n] = \frac{1}{D} \sum_{i=0}^{D} x[n-i] \qquad \text{(sur } D \text{ points)}$$

Problème

Complexité $O(D)$ (impraticable pour $D > 10$...surtout à 3.2MHz !)

Solution :

$$Dy[n] = x[n] + x[n-1] + \cdots + x[n-D]$$
$$Dy[n-1] = x[n-1] + \cdots + x[n-D] + x[n-D-1]$$

Filtre moyenne

Comment calculer la moyenne glissante $y[n]$ d'un signal $x[n]$?

$$y[n] = \frac{1}{D}\sum_{i=0}^{D} x[n-i] \qquad \text{(sur } D \text{ points)}$$

Problème

Complexité $O(D)$ (impraticable pour $D > 10\ldots$surtout à 3.2MHz !)

Solution :

$$Dy[n] = x[n] + x[n-1] + \cdots + x[n-D]$$
$$Dy[n-1] = x[n-1] + \cdots + x[n-D] + x[n-D-1]$$

Donc :

$$\boxed{y[n] = \frac{1}{D}(x[n] - x[n-D-1]) + y[n-1]}$$

Filtre moyenne

Comment calculer la moyenne glissante $y[n]$ d'un signal $x[n]$?

$$y[n] = \frac{1}{D}\sum_{i=0}^{D} x[n-i] \qquad \text{(sur } D \text{ points)}$$

Problème

Complexité $O(D)$ (impraticable pour $D > 10$...surtout à 3.2MHz !)

Solution :

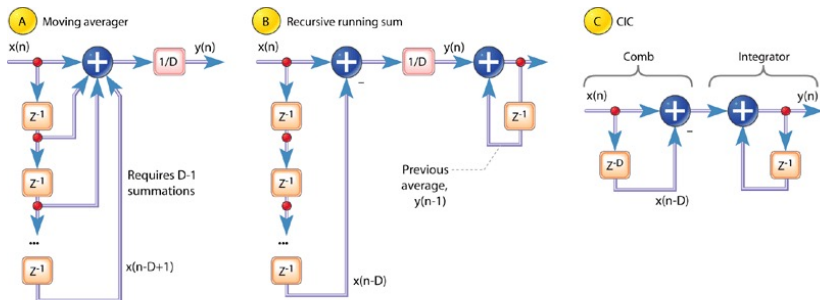
$$Dy[n] = x[n] + x[n-1] + \cdots + x[n-D]$$
$$Dy[n-1] = x[n-1] + \cdots + x[n-D] + x[n-D-1]$$

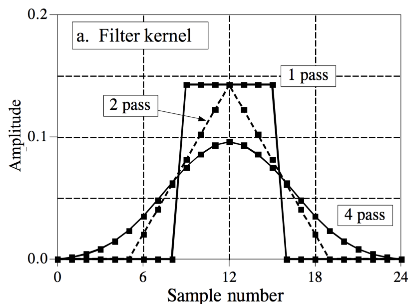
Donc :

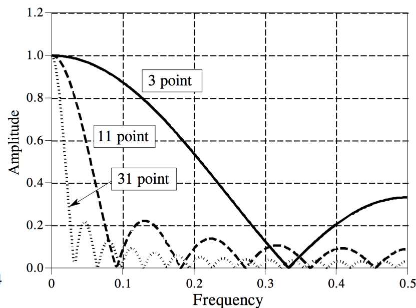
$$\boxed{y[n] = \frac{1}{D}(x[n] - x[n-D-1]) + y[n-1]}$$

$\rightsquigarrow$ moyenne glissante = filtre à réponse impulsionnelle infinie

Filtre moyenne



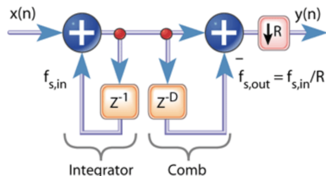




Le filtre CIC

Remarque

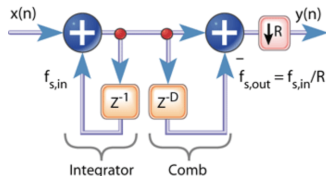
Filtre moyenne = composition d'un peigne et d'un intégrateur.
En rajoutant un décimateur on obtient :



Le filtre CIC

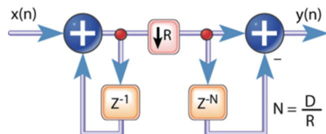
Remarque

Filtre moyenne = composition d'un peigne et d'un intégrateur.
En rajoutant un décimateur on obtient :



CIC : *Cascaded Integrator Comb filter* (Hogenauer, 1981)

On peut permuter l'intégrateur et la décimation ⇝ plus efficace



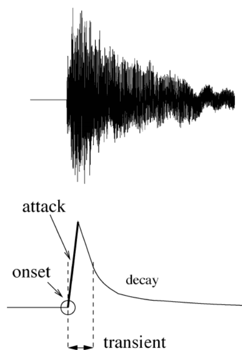
Filtrage du signal microphone, en pratique

- `microphone.hh: Process()`
  - les samples arrivent à 1.5KHz par blocs de 256 octets (débit binaire : 1500*256*8 = 3.2MHz)
  - on les passe à `PdmFilter` qui fait le filtrage
  - puis on appelle le code utilisateur, `callback_`
- `dsp.hh: PdmFilter`
  - le tableau précalculé `setbits` de 255 cases associe à tout octet le nombre de bits à 1 qu'il contient
  - première passe : conversion de chaque paquet de 8 bits
  - deuxième passe : on passe le résultat à `CicDecimator`
- `dsp.hh: CicDecimator`
  - intégrateur → décimation → peigne
  - nombre de passes du filtre configurable (paramètre de template `N`)

Détection de transitoires

Transitoire

Instants dans un signal qui marquent un changement brutal
(ex : début de note, changement brusque de timbre...)



Détecter les transitoires en temps réel, c'est déclencher un évenement quand elles surviennent.

Application

- déclenchement d'évenements synchrones avec un musicien (suivi de partition)
- détection de tempo (*beat-matching*)
- pitch shifting (modification en temps réel de la hauteur)

Algorithmes de détection de transitoires

Trois passes successives :

Pre-processing

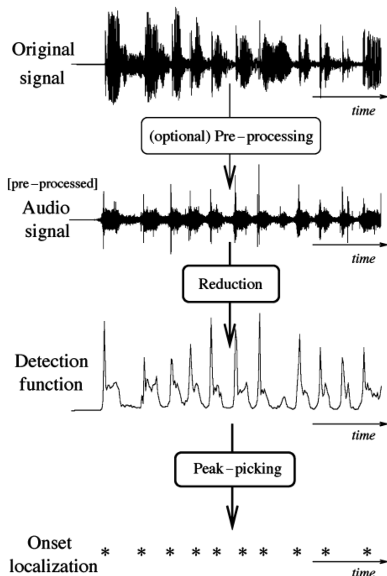
isole les fréquences intéressante
(ex : filtrage)

Réduction

transformation du signal en une mesure du *degré de changement*
(ex : enveloppe d'amplitude, dérivation)

Peak-picking

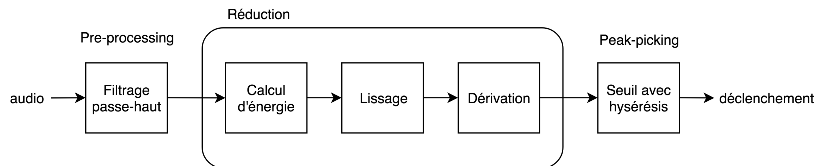
choix de temps discrets
(ex : fonction seuil, hystérésis)



Notre algorithme

On cherche à détecter des claquements de main :

- changement brusque de niveau sonore
- forte énergie dans les aigus



Pre-processing filtrage passe-haut
(élimine les faux-positifs dûs au *rumble*)

Réduction calcul de l'enveloppe du signal, puis dérivation
(nous donnera la *variation instantanée* du niveau sonore)

Peak-picking fonction seuil avec hystérésis
(évite la détection de plusieurs transitoires dans une seule attaque)

Pre-processing Filtrage passe-haut

Dérivée discrète

Un filtre passe-haut économique : prendre la “dérivée” d’un signal, c’est-à-dire la différence entre la valeur actuelle est la précédente.

Pre-processing Filtrage passe-haut

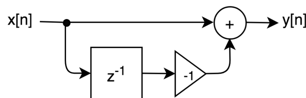
Dérivée discrète

Un filtre passe-haut économique : prendre la “dérivée” d’un signal, c’est-à-dire la différence entre la valeur actuelle est la précédente.

Caractérisation du filtre

$y[n] = x[n] - x[n-1]$

(i.e. convolution par {1 ; -1})



Pre-processing Filtrage passe-haut

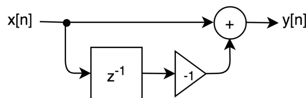
Dérivée discrète

Un filtre passe-haut économique : prendre la “dérivée” d’un signal, c’est-à-dire la différence entre la valeur actuelle est la précédente.

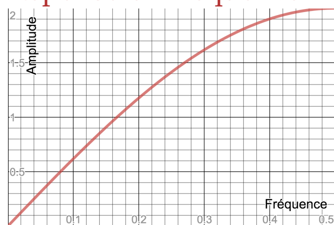
Caractérisation du filtre

$$y[n] = x[n] - x[n-1]$$

(i.e. convolution par {1 ; -1})



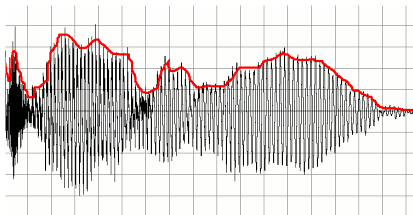
Réponse en fréquence



L'enveloppe d'amplitude d'un signal

Enveloppe d'amplitude

C'est le signal qui donne le niveau sonore perçu par l'oreille à chaque instant d'un signal audio



Remarques

- il est compris entre 0 (silence) et 1 (volume maximum) (pas de valeurs négatives)
- il suit les pics locaux du signal, mais pas ses oscillations du domaine audible (>20Hz)

Calcul d'enveloppe

En deux étapes :

calcul de l'énergie instantanée L'énergie instantanée d'un signal $x[n]$ compris entre -1 et 1 est :

$$E[n] = (x[n])^2$$

$E[n]$ est compris entre 0 et 1 (il est "rectifié")

lissage = filtrage passe-bas
(on utilisera le filtre à un pôle vu hier)

Calcul d'enveloppe

En deux étapes :

calcul de l'énergie instantanée
: L'énergie instantanée d'un signal $x[n]$ compris entre -1 et 1 est :

$$E[n] = (x[n])^2$$

$E[n]$ est compris entre 0 et 1 (il est "rectifié")

lissage
: = filtrage passe-bas
(on utilisera le filtre à un pôle vu hier)

Alternative : utiliser un filtre "non-symétrique" au coefficient plus fort en descente qu'en montée

Calcul d'enveloppe

En deux étapes :

calcul de l'énergie instantanée L'énergie instantanée d'un signal $x[n]$ compris entre -1 et 1 est :

$$E[n] = (x[n])^2$$

$E[n]$ est compris entre 0 et 1 (il est "rectifié")

lissage = filtrage passe-bas
(on utilisera le filtre à un pôle vu hier)

Alternative : utiliser un filtre "non-symétrique" au coefficient plus fort en descente qu'en montée

Dérivation

On s'intéresse aux *variations* de niveau, pas au niveau lui-même
⇝ dérivation du signal
(on utilisera le filtre passe-haut vu précédemment)

Peak-picking Seuil et hystérésis

Après réduction, on a un signal $r[n]$ qui est :

- ≈ 0 quand le volume est constant
- ≈ 1 quand le volume augmente brusquement
- ≈ -1 quand le l'audio diminue brusquement de volume

On cherche les pics de ce signal, qui correspondent à des transitoires.

Peak-picking Seuil et hystérésis

Après réduction, on a un signal $r[n]$ qui est :

- ≈ 0 quand le volume est constant
- ≈ 1 quand le volume augmente brusquement
- ≈ -1 quand le l'audio diminue brusquement de volume

On cherche les pics de ce signal, qui correspondent à des transitoires.

Fonction seuil

On se donne un seuil $0 < S < 1$;
Si $r[n-1] < S$ et $r[n] \geq S$, alors on marque n comme transitoire

Hystérésis

Problème

Si le signal est bruité, la fonction seuil peut détecter plusieurs transitoires dans une attaque (voir tableau)

Hystérésis

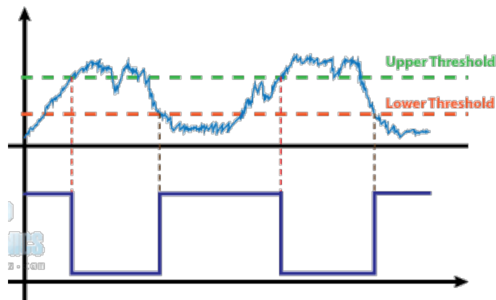
Problème

Si le signal est bruité, la fonction seuil peut détecter plusieurs transitoires dans une attaque (voir tableau)

Solution

Hystérésis : deux seuils de détections S_b et S_h

- Si $r[n-1] < S_b$ et $r[n] \geq S_b$, on *arme* la détection
- Si $r[n-1] < S_h$ et $r[n] \geq S_h$ et détection armée, on *désarme* la détection et on marque n comme transitoire

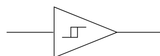


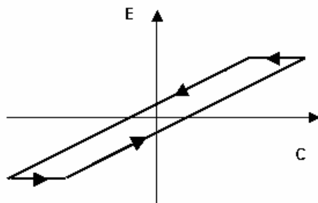
Hystérésis

Définition

(du grec hústeros, « après », « plus tard ») propriété d'un système dont l'évolution ne suit pas le même chemin selon qu'une cause extérieure augmente ou diminue.

Exemples

- la bascule de Schmitt en électronique 
- version continue : diagramme de cycle d'hystéresis (utile pour filtrer le mouvement d'un potentiomètre)



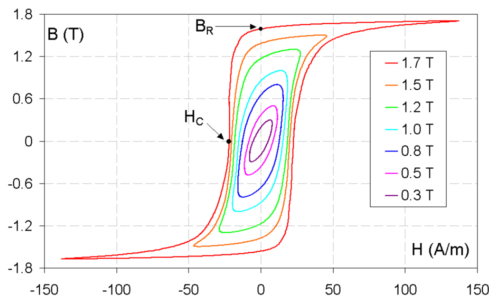
Hystérésis

Définition

(du grec hústeros, « après », « plus tard ») propriété d'un système dont l'évolution ne suit pas le même chemin selon qu'une cause extérieure augmente ou diminue.

Exemples

- dans la nature : aimantation des ferromagnétique vs. champ magnétique appliqué



Introduction au traitement du signal embarqué

Représentation et traitement des signaux audio

Analyse et détection de transitoire

Synthèse par modèle physique
- Parenthèse : DAC et Double buffering
- Modèlisation physique et guide d'ondes
- L'algorithme de Karplus-Strong

Synthèse par modèle physique

La puissance des machines permet de simuler les processus physiques qui créent les vibrations sonores

$$\text{modèles physiques} \longrightarrow \text{simulation} \longrightarrow \text{algorithmes}$$

Synthèse par modèle physique

La puissance des machines permet de simuler les processus physiques qui créent les vibrations sonores

modèles physiques $\longrightarrow$ simulation $\longrightarrow$ algorithmes

Les familles d'algorithmes classiques

- modèles source-filtre (voix humaine)
- synthèses modale (percussions notamment)
- guides d'onde (instruments à corde)

Les guides d'ondes

Digital waveguide synthesis [Julius O. Smith III, 1987] [1]

- un modèle simple de propagation des ondes stationnaires (simulation d'instruments à vent ou à cordes)
- n'utilise que des délais courts et des filtres

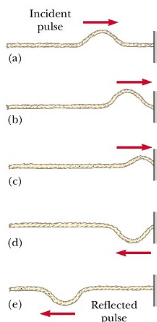


Figure 16.13 The reflection of a traveling wave pulse at the fixed end of a stretched string. The reflected pulse is inverted, but its shape is unchanged.

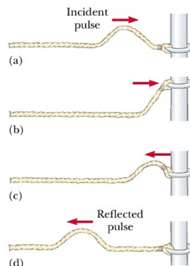
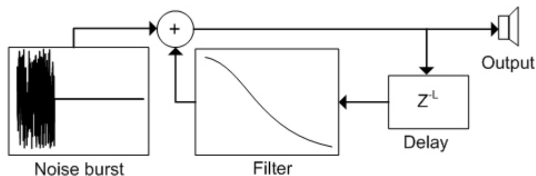


Figure 16.14 The reflection of a traveling wave pulse at the free end of a stretched string. The reflected pulse is not inverted.

1. https://ccrma.stanford.edu/~jos/swgt/swgt.html

L'algorithme de Karplus-Strong (1983)

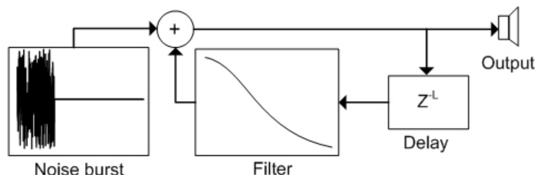
Un algorithme simple et populaire de synthèse de cordes pincées :
[Karplus & Strong, 1983]



- une impulsion courte de buit blanc (∼10ms)
- dans un délai avec réinjection (temps de délai $L = \frac{f_s}{f}$)
- un filtre passe-bas dans la boucle de réinjection

L'algorithme de Karplus-Strong (1983)

Un algorithme simple et populaire de synthèse de cordes pincées : [Karplus & Strong, 1983]



- une impulsion courte de buit blanc (∼10ms)
- dans un délai avec réinjection (temps de délai $L = \frac{f_s}{f}$)
- un filtre passe-bas dans la boucle de réinjection

DEMO