TP4 traitement et synthèse modulaire du signal audio
Annexe A : spécification des modules

notions : toutes les notions du cours

Ce document présente les spécifications des modules de traitement pour le TP4 “synthèse et traitement modulaire du signal audio”. **Pensez à amener un casque audio doté d’une prise minijack en séance !**

Table des matières

Première partie

Annexes 1 : spécification des modules

1 Les générateurs

1.1 Générateurs simples

Module `GenSineBasic`. Générateur sinusoïdal basique (pour se mettre en jambes)

Nom	Desc	Commentaires
Ports d'entrée	$\varnothing$	
Ports de sortie	1	l'échantillon généré à chaque pas
Fréquence	f	fréquence du signal généré
Amplitude	amp	amplitude du signal généré

La méthode `exec()` élabore le nouvel échantillon e suivant la formule $e = amp * sin(2 * PI * f * nbStep/SAMPLE\_FREQ)$, avec $nbStep$ l'indice du pas de temps - c'est à dire le nombre d'appels de la fonction d'exécution du module depuis le début, puis écrit e sur le port de sortie. Il faut stocker $nbStep$ dans l'objet.

La méthode `reset()` remet n à zéro.

1.2 Générateurs contrôlables

Module `GenSine`. Générateur sinusoïdal contrôlable.

Nom	Desc	Commentaires
Ports d'entrée	$\varnothing$, 1 ou 2	Le port 1, si il existe, contrôle la fréquence. Le port 2, si il existe, contrôle l'amplitude.
Ports de sortie	1	l'échantillon généré à chaque pas
Fréquence	f	fréquence du signal généré. Si le module a un ou deux ports d'entrée, elle est mise à jour à chaque pas à partir du port d'entrée 1. Sinon, elle est initialisée à la création et ne changera jamais.
Amplitude	amp	amplitude du signal généré. Si le module a deux ports d'entrée, elle est mise à jour à chaque pas à partir du port d'entrée 2. Sinon, elle est initialisée à la création et ne changera jamais.

La classe a trois constructeurs :

- `GenSine(String name, double f, double amp)` : dans ce cas, le module n'a aucun port d'entrée. Il est équivalent à un module `GenSineBasic`.
- `GenSine(String name, double amp)` : dans ce cas, le module a un unique port d'entrée, qui contrôle la fréquence du sinus généré ; l'amplitude est constante.
- `GenSine(String name)` : dans ce cas, le module a 2 ports d'entrées, qui contrôlent respectivement la fréquence et l'amplitude du sinus généré.

Pour cet oscillateur, il est nécessaire de stocker en attribut la *phase* du sinus (un `double`, initialisé à 0).

La méthode `exec()` :
— met à jour la fréquence f et l'amplitude amp à partir des ports d'entrée si ils existent
— calcule la nouvelle phase par la relation $phase = phase + 2 * PI * f / SAMPLE\_FREQ$
— élabore le nouvel échantillon e suivant la formule $e = amp * sin(phase)$
— écrit e sur le port de sortie.

La méthode `reset()` remet la *phase* à zéro - et bien sûr met 0. dans tous les ports d'entrée et de sortie du module.

Module `GenSquare`. Générateur carré contrôlable.

Cet oscillateur fonctionne sur le même principe que le `GenSine`, si ce n'est qu'il génère un signal carré.

Comme pour le `GenSine` est également nécessaire de stocker en attribut la *phase* (un `double`, initialisé à la valeur 0.).

La méthode `exec()` :
— met à jour la fréquence f et l'amplitude amp à partir des ports d'entrée si ils existent
— calcule la nouvelle phase par la relation $phase = phase + 2 * PI * f * SAMPLE\_FREQ$
— élabore le nouvel échantillon e suivant le principe suivant : si la valeur du sinus de la phase est positive, e vaut amp ; sinon, e vaut $-amp$. Autrement dit :

```
1       double e ;
2       if( ( (int) (phase / Math.PI) )% 2 == 0 ) {
3           e = amp;
4       } else {
5           e = -amp;
6       }
```

— écrit e sur le port de sortie.

La méthode `reset()` remet la *phase* à zéro.

Autres générateurs Vous pouvez compléter vos générateurs en ajoutant module `GenWhiteNoise` (générateur de bruit blanc) un module `GenSawtooth` (forme d'onde triangulaire), etc.

2 Conteneurs d'`AudioData`

Module `AudioDataReceiver`. Récepteur d'échantillon.

Un module `AudioDataReceiver` ne fait pas de traitement proprement dit : son rôle est de stocker les échantillons reçus dans son port d'entée dans un conteneur d'échantillons (une instance de la classe `AudioData`, fournie, voir le premier chapitre).

Après le calcul du patch, le module peut enregistrer ces échantillons dans un fichier audio, les jouer (envoi sur la carte son) ou encore visualiser la forme d'onde à l'écran.

Pour qu'un patch soit pertinent, c'est à dire pour qu'on puisse exploiter (écouter, enregistrer, afficher...) le signal qu'il produit, il faut qu'il possède au moins une instance de la classe `AudioDataReceiver`.

Un module `AudioDataReceiver` possède en attribut un conteneur d'échantillons, instance de la classe `AudioData`, créée par le constructeur de `AudioDataReceiver`.

Nom	Desc	Commentaires
Ports d'entrée	1	
Ports de sortie	1	copie de la valeur du port d'entrée (fonction *bypass*)
Paramètres	∅	

La méthode `exec()` :
- Ajoute l'échantillon présent le port d'entrée au conteneur d'échantillons `AudioData`.
- Copie cet échantillon dans le port de sortie (unique). Si ce port de sortie est connecté, la valeur de l'échantillon sera ainsi envoyé vers le module aval, sans modification. Ainsi, la classe permet de réaliser un *bypass* entre entrée et sortie. Elle peut donc être utilisée aussi bien "tout en bas" d'un patch pour récupérer ce qu'il génère, que "au mileu" d'un patch pour observer ce qui s'y passe.

La méthode `reset()` remet `nbStep` à zéro (retour au début du signal).

La classe `AudioDataProvider` possèdera également les méthodes suivantes, qui seront réalisées simplement par délégation à l'attribut `AudioData` :
- `void saveAudioDataToWavFile(String audioFileName)` : enregistre le signal contenu dans l'`AudioData` dans un fichier son nommé `audioFileName)`.
- `void displayAudioDataWaveform()` : affiche la forme d'onde à l'écran.
- `void playAudioData()` : envoie le signal sur la carte son.

Module `AudioDataProvider`. Lecteur d'échantillons.

Un module `AudioDataProvider` a pour rôle d'envoyer dans le patch un signal préexistant, par exemple chargé depuis un fichier. C'est donc une classe essentielle pour faire du traitement audio.

Son constructeur prend en paramètre un conteneur d'échantillon : une référence vers une instance de la classe `AudioData`, fournie, qui est stockée en attribut.

Le module a également un attribut `int nbStep` qui stocke l'indice de l'échantillon courant (compteur du nombre d'appels de la méthode `exec()`), initialisé à 0.

Nom	Desc	Commentaires
Ports d'entrée	0	
Ports de sortie	1	
Paramètres	`audioData`	une référence vers un conteneur d'échantillon `AudioData` qui sera lu par le module.

La méthode `exec()` :
- Incrémente `nbStep`
- Récupère la valeur de l'échantillon d'indice `nbStep` de l'`AudioData` - voir la méthode `double getSample(int id)` de la classe `AudioData`.
- Ecrit cet échantillon sur le port de sortie.

La méthode `reset()` remet `nbStep` à zéro (retour au début du signal).

3 Modules utilitaires

Module `Constant`. Sort une valeur constante.

Nom	Desc	Commentaires
Ports d'entrée	∅	
Ports de sortie	1	
Paramètres	double cste	la valeur constante sortie par ce module

Ce module a un port de sortie, et aucun port d'entrée.

Sa méthode `exec()` de ce module sort une valeur constante *cste*, qui est initialisée par le constructeur.

Sa méthode `reset()` ne fait rien.

Module `Mixer`. Additionneur de plusieurs signaux (mixage).

Nom	Desc	Commentaires
Ports d'entrée	nombre variable (>= 2)	Nombre de ports d'entrée variable, déterminé à la création
Ports de sortie	1	
Paramètres	∅	

Sa méthode `exec()` envoie sur le port de sortie la somme des valeurs des ports d'entrée.

Sa méthode `reset()` ne fait rien.

Module `Multiplier` et gain. Ce module permet à la fois de multiplier plusieurs signaux et d'appliquer un gain constant.

Nom	Desc	Commentaires
Ports d'entrée	nombre variable (>= 1)	Nombre de ports d'entrée variable, déterminé à la création
Ports de sortie	1	
Paramètres	gain (`double`)	Le gain global

Un module `Multiplier` a deux constructeurs :
- `Multiplier(int nbPortsEntree)` : le gain global est fixé à 1. Dans ce cas, `nbPortsEntree` doit être supérieur ou égal à 2.
- `Multiplier(int nbPortsEntree, double gain)` : le gain global est fixé à `gain`. Dans ce cas, `nbPortsEntree` doit être supérieur ou égal à 1.

La méthode `exec()` envoie sur le port de sortie le produit des valeurs des ports d'entrée et du gain.

La méthode `reset()` ne fait rien.

Module `Spliter`. Ce module génére en sortie plusieurs signaux identiques à son entrée.

Nom	Desc	Commentaires
Ports d'entrée	1	
Ports de sortie	nombre variable (>= 2)	Nombre de ports d'entrée variable, déterminé à la création
Paramètres	∅	

Sa méthode `exec()` envoie sur tous les ports de sortie une copie de la valeur du port d'entrée.

Sa méthode `reset()` ne fait rien.

4 Delai

Module `DelayBasic`. Delai constant

Nom	Desc	Commentaires
Ports d'entrée	1	
Ports de sortie	1	
Paramètres	delay	Le délai, en secondes (`double`). Constant et fixé à la création.

Sa méthode `exec()` envoie l'échantillon d'entrée retardé de `delay` échantillon. Pour cela, on s'appuiera sur le principe des buffers circulaires. Le module a en attribut :

— un tableau `circularBuffer` de `double`, de taille `(delay / SAMPLEFREQ)`,
— un entier `head` qui indique l'indice auquel ranger le nouvel échantillon entrant, initialisé à 0.

L'algorithme est alors :

```
1  public void exec() {
2      // on stocke le nouvel échantillon dans le buffer
3      circularBuffer[head] = getInputPortValue(0);
4      // l'échantillon à sortir est celui qui a été stocké il y a
5      // circularBuffer.length pas
6      double outputSample = circularBuffer[  (head+1) % delayLine.length  ] ;
7      // on sort l'échantillon
8      setAndSendOutputPortValue(0, outputSample);
9      // calcul de l'indice de la case dans laquelle stocker le prochain é
           chantillon
10     head = (head+1) % delayLine.length;
11 }
```

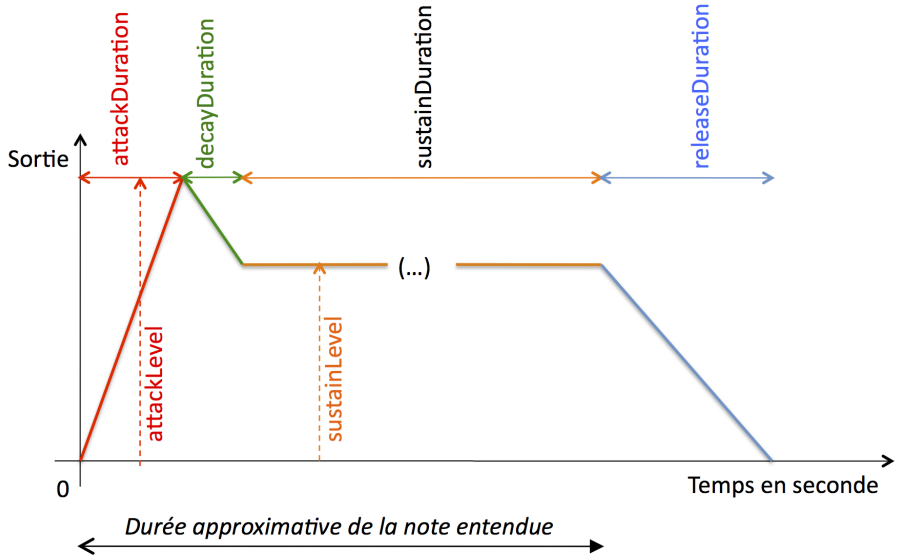
Sa méthode `reset()` remet `head` à zéro et réinitialise tous les échantillon stockés dans `circularBuffer` à 0..

5 Générateurs d'enveloppes

Un son "musical" a un début et une fin. Pour cela, il faut pouvoir faire varier l'amplitude du signal au cours du temps. Dans les systèmes de synthèse modulaire du signal audio, c'est le rôle des générateurs d'enveloppes. Il existe des variantes ; nous ne présenterons que le générateur d'enveloppe ADSR.

Module `EnvelopADSR`. Enveloppe de type *Attack, Decay, Sustain, Release*

Un générateur d'enveloppe ADSR suit les phases suivantes :
— la phase d'attaque : l'amplitude augmente jusqu'à une certaine valeur
— la phase de de descente (*decay*) : l'amplitude diminue jusqu'à une valeur stable durant la durée de la note
— la phase de maintien (*sustain*) : l'amplitude reste constante
— la phase de fin (*release* : l'amplitude retrouve lentement la valeur 0 (fin du son).



La durée de chaque phase est contrôlable, ainsi que le niveau d'attaque et le niveau de *sustain*. Tous ces paramètres sont fixés à la création.

Nom	Desc	Commentaires
Ports d'entrée	0	
Ports de sortie	1	
Durée de l'attaque	attackDuration	un `double` en secondes
Durée du decay	decayDuration	un `double` en secondes
Durée du sustain	sustainDuration	un `double` en secondes
Durée du release	releaseDuration	un `double` en secondes
Niveau d'attaque	attackLevel	un `double`
Niveau de sustain	sustainLevel	un `double`

Le module stockera en attribut le nombre de pas écoulés au moyen d'un `int nbStep`.

Il a deux constructeurs :

— `public ADSREnvelop(String name, double attackDuration, double decayDuration, double sustainDuration , double releaseDuration);`

— `public ADSREnvelop(String name, double noteDuration);` : ADSR d'une certaine durée, avec phases d'attaque et de fin par défaut. Fixe `attackDuration` à 0.1, `decayDuration` à 0.05, `releaseDuration` à 0.5 et calcule `sustainDuration` en fonction de `noteDuration`. `noteDuration` doit être supérieur à $0.1 + 0/05$.

Pour la méthode `exec()`, on peut observer que chaque partie est un segment de droite et on commence par calculer les positions de deux points de la droite. La méthode peut alors ressembler à ceci (ce code sera disponible en téléchargement sur le site, pour éviter que vous ayez à le recopier) :

```
1  @Override
2      public void execute() {
3          double timeSec = (double)sampleId/44100; // la durée écoulée depuis le
               début en secondes
4          if(timeSec >= attackDuration + decayDuration + sustainDuration +
               releaseDuration) {
5              sendOutput(0, 0.);
6              return;
7          }
8
```

```
9           double y1, y0, x0, x1;
10          if(timeSec < attackDuration) {
11              x0 = 0;
12              y0 = 0;
13              x1 = attackDuration;
14              y1 = attackLevel;
15          } else if (timeSec < (attackDuration + decayDuration)) {
16              x0 = attackDuration;
17              y0 = attackLevel;
18              x1 = attackDuration + decayDuration;
19              y1 = sustainLevel;
20          } else if(timeSec < (attackDuration + decayDuration + sustainDuration))
                {
21              x0 = attackDuration + decayDuration;
22              y0 = sustainLevel;
23              x1 = attackDuration + decayDuration + sustainDuration;
24              y1 = sustainLevel;
25          } else {
26              x0 = attackDuration + decayDuration + sustainDuration;
27              y0 = sustainLevel;
28              x1 = attackDuration + decayDuration + sustainDuration +
                    releaseDuration ;
29              y1 = 0;
30          }
31
32          double output = (y1-y0)/(x1-x0) * ( timeSec - x0) + y0;
33          sendOutput(0, output);
34          sampleId++;
35      }
```

La méthode `reset()` remet simplement `sampleId` à zéro.

6 Filtres FIR

On s'intéresse maintenant au filtrage. Un seul module sera spécifié : le module `BandPassFIlterFIR`, basé sur une fenêtre dont on contrôle la fréquence de résonance et la largeur de bande. Avec vos connaissances en traitement numérique du signal, vous pourrez ensuite implanter d'autres filtres.

La classe abstraite `FIRAbstract`. Comme tous les filtres FIR partagent la même structure (ils se calculent tous de la même manière, dès lors qu'on connait leurs coefficients), on implantera d'abord une classe abstraite `FIRAbstract`.

Nom	Desc	Commentaires
Ports d'entrée	1	
Ports de sortie	1	
Coefficients	`coeffArray`	Les coefficients du filtre FIR. Un tableau de (N+1) valeurs `double` . N est l'ordre du filtre. **Attention :** N doit être pair. Ce tableau pourra être déclaré `protected` pour faciliter l'implantation des sous-classes concrètes. Les valeurs de ce tableau sont également, bien sûr, la réponse impulsionnelle du filtre.

Le tableau `coeffArray` est initialisé à la création (par le constructeur de la sous-classe, donc).

Le module a en attribut :
— le tableau `coeffArray`
— un tableau `inputSamples` de taille `coeffArray.length`, qui stockera les $N+1$ précédents échantillons et sera mis à jour à chaque pas. Vous pouvez utiliser le principe du buffer circulaire (comme dans le module `DelayBasic`).

La méthode `exec()` calcule l'échantillon en sortie en fonction de l'entrée, c'est à dire la convolution du signal d'entrée et de la réponse impulsionnelle du filtre, ou encore : $(\sum_{i=0}^{N} coeffArray[i] * sample[i])$ où $sample[i]$ est le sample entré dans le module il y a i échantillons.

La méthode `reset()` remet à zéro les échantillons du tableau `coeffArray`.

Module `BandPassFIR`. Filtre passe bande contrôlé. Sous-classe de `FIRAbstract`.

Nom	Desc	Commentaires
Ports d'entrée	1 à 3	Le premier port correspond au signal à filtrer. Si ils existent, les ports 2 et 3 contrôlent respectivement la fréquence de résonance et la largeur de bande (en Hz).
Ports de sortie	1	
Fréquence de résonance	f_r (en Hz)	Initialisé dans le constructeur, ou mis à jour à chaque pas en fonction de la valeur du port d'entrée 2 si il existe.
Largeur de bande	*largeur* (en Hz)	Initialisé dans le constructeur, ou mis à jour à chaque pas en fonction de la valeur du port d'entrée 3 si il existe.

La classe a trois constructeurs :
— `BandPassFIR(String name, double f_c, double largeur)` : dans ce cas, le module a un seul port d'entrée. Les coefficients du filtre sont calculés une fois pour toute dans le constructeur en fonction de f_c et *largeur* (en Hz).
— `BandPassFIR(String name, double largeur)` : dans ce cas, le module a 2 ports d'entrée. Le second port d'entrée contrôle la fréquence de résonance f_r. Les coefficients du filtre sont recalculés à chaque pas.
— `BandPassFIR(String name)` : dans ce cas, le module a 3 ports d'entrée. Le port d'entrée 2 contrôle la fréquence de résonance f_r et le port d'entrée 3 contrôle la largeur de bande. Les coefficients du filtre sont recalculés à chaque pas.

Le module est doté d'une méthode `void updatecoeffArray(double fc, double largeur);` qui met à jour les coefficients du filtre (c'est à dire le tableau `coeffArray` hérité de la super-classe. Son algorithme peut-être le suivant (ce code sera mis en ligne sur le site du module pour éviter que vous n'ayez à le recopier) :

```
1  public void updatecoeffArray(double f_r, double largeur){
2          //Normalize f_r and largeur so that pi is equal to the Nyquist angular
               frequency
3          f_r = f_r/SAMPLEFREQ;
4          largeur = largeur/SAMPLEFREQ;
5
6          double f_1 = f_r-largeur; // freq coupure à gauche
7          double f_2 = f_r+largeur; // freq coupure à droite
8
9          double omega_1 = 2*Math.PI*f_1;
10         double omega_2 = 2*Math.PI*f_2;
11         int middle = (coeffArray.length -1) / 2 ;    //rappel : length impaire
12
13         // Calculate taps
14         // Due to symmetry, only need to calculate half the window
15         for ( int i=0 ; i< middle ; i++) {
```

```
16              double val1 = Math.sin( omega_1 * (i-middle) ) / ( Math.PI * (i-
                    middle) ) ;
17              double val2 = Math.sin( omega_2 * (i-middle) ) / ( Math.PI* (i-
                    middle) ) ;
18
19              // hamming windowing
20              double weight= 0.54 - 0.46* Math.cos( ( 2. * Math.PI * i ) /
                    coeffArray.length   ) ;
21
22              //weight= 1.0 ;
23              coeffArray[i]= ( val2 - val1 ) * weight ;
24              coeffArray[coeffArray.length-i-1]=  coeffArray[i] ;
25
26          }
27          coeffArray[middle]= 2.0 * ( f_2 - f_1 ) ;
28
29          //Scale filter to obtain an unity gain at center of passband
30          double realSum= 0,imagSum= 0  ;
31          for( int i = 0 ; i < coeffArray.length ; i ++ ) {
32              double argExp= -2. * Math.PI * i * f_r ;
33              realSum+= Math.cos( argExp  ) * coeffArray[i] ;
34              imagSum+= Math.sin( argExp  ) * coeffArray[i] ;
35          }
36
37          double sum= Math.sqrt( realSum * realSum + imagSum * imagSum ) ;
38          for(int i = 0 ; i < coeffArray.length ; i ++) {
39              coeffArray[i] = coeffArray[i] / sum ;
40          }
41          //System.out.println("BandPassFilterControlled fc == " + f_r + " norm.
                factor == " + sum ) ;
42      }
```

La méthode `exec()` met à jour f_r et *largeur* si les ports correspondants existent, puis recalcule les coefficients du filtre si nécessaire. Elle appelle ensuite la méthode de la super classe pour calculer la sortie.

Autre filtres FIR En vous appuyant sur le cours TNS, vous pourrez implanter d'autres filtres FIR (`HighPassFIR`, `BandPassFIR` par exemple...), et d'autres type de filtres.

7 D'autres modules ?

Un programme de synthèse et traitement modulaire du signal comprend en général plus de 100 types de modules. Moyenne RMS, filtres basés sur une transformée de fourrier, autres générateurs, module de synthèse granulaire... Maintenant que le squelette est en place, vous pouvez expérimenter !