L1 Acoustique — Algorithmique

Projet F03 — Synthétiseur ⋆⋆⋆⋆

Cyril Desjouy

16 juillet 2024

1 Introduction

Le but de ce projet est de développer des outils permettant de faire de la synthèse sonore sous Python. Il s'agira de développer une suite de fonctions, qui, utilisées ensemble permettront notamment de synthétiser des signaux.

2 Préliminaire 1 : Algorithme de génération d'arpège

Selon Wikipedia, un arpège est *une série de notes émises successivement et qui, considérées ensemble, forment un accord.*

Nous nous concentrerons ici sur les accords majeurs (vous pourrez bien entendu étendre ce travail par vous même si vous le souhaitez). Les accords majeurs sont définis comme illustré au tableau suivant :

Note	**Notation anglo-saxonne**	**Accord**
Do	C	Do Mi Sol
Ré	D	Ré Fa# La
Mi	E	Mi Sol# Si
Fa	F	Fa La Do
Sol	G	Sol Si Ré
La	A	La Do# Mi
Si	B	Si Ré# Fa#

Table 1 – Liste de quelques accords majeurs

Écrire une fonction `arpeggiator(N, M, display=False)` :

- prenant en argument d'entrée :
  - un nombre de temps N,
  - un nombre de mesures M,
  - un booléen `display` précisant à la fonction s'il faut ou non afficher sous forme textuelle la pièce musicale,
- retournant une liste de listes (chaque sous liste représente une mesure) représentant une mélodie aléatoire sur M mesures de N temps en utilisant les notes d'un même accord,
- affiche la mélodie sous la forme décrite dans l'exemple ci-dessous pour $N = 5$ et $M = 8$.

```
Mesure 1 [D]- Ré Fa# Ré Fa# Ré
Mesure 2 [G]- Ré Ré Si Ré Ré
Mesure 3 [A]- La La Mi Do# Do#
Mesure 4 [F]- Do Do Fa Do Fa
Mesure 5 [E]- Sol# Sol# Mi Sol# Si
Mesure 6 [A]- Do# La Mi La Do#
Mesure 7 [C]- Sol Do Sol Sol Mi
Mesure 8 [D]- La Fa# Fa# Fa# Fa#
```

3 Préliminaire 2 : Algorithme de conversion de notes

Afin de pouvoir synthétiser des notes de musique, il faut au préalable connaître leurs fréquences fondamentales. Il est possible de calculer la fréquence d'une note comme suit :

$$f_n = f_0 \times 2^{n/12}, \tag{1}$$

où f_0 est une note de référence et n est le nombre de demi-tons au dessus de la note de référence [1]. Une octave comprend 12 demi-tons :

```
notes = ['do', 'do#', 'ré', 'ré#', 'mi', 'fa', 'fa#', 'sol', 'sol#', 'la', 'la#', 'si']
```

À titre d'exemple, pour calculer la fréquence du sol_3, il faut calculer le nombre de demi-tons qui le sépare d'une référence, prise ici comme étant le do_0 (32.7 Hz). Du do_0 au do_3, il y a 3×12 demi-tons. Puis entre le do_3 et le sol_3, il y a 7 demi-tons soit un total de 43 demi-tons entre do_0 et sol_3. Sa fréquence est alors : $f_{sol3} = 32.7 \times 2^{43/12}$ Hz.

Écrire une fonction `note2freq(X, N)` :
- prenant en argument d'entrée :
  - une chaîne de caractère représentant la note X,
  - un entier N représentant l'octave où est située la note X,
- et retournant la fréquence (en Hz) correspondant à la note X_N.

Tester cette fonction avec différentes notes et comparez vos résultats avec ceux du tableau disponible sur la page wikipedia suivante : fréquence d'une note

4 Prélimaire 3 : Données numériques représentant une note de musique

La méthode proposée ici consiste à synthétiser chaque note sous la forme d'un signal sinusoïdal pur $s_n(t)$ défini comme suit :

$$s_n(t) = \sin(2\pi f_n t), \tag{2}$$

où f_n est la fréquence correspondant à la note à synthétiser et t est le temps défini entre 0 et $t_{\max}$, $t_{\max}$ correspondant à la durée totale de la note. Afin d'éviter les discontinuités entre chaque note synthétisée, il est nécessaire de fenêtrer temporellement le signal $s_n(t)$ comme présenté à la figure 1. Le signal $s_n^w(t)$ fenêtré peut s'écrire :

$$s_n^w(t) = \sin(2\pi f_n t) \times T(t), \tag{3}$$

où $T(t)$ est ici une fenêtre temporelle. On choisira une fenêtre de type *Tukey* (Le module `scipy.signal` fournit la fonction `tukey`).

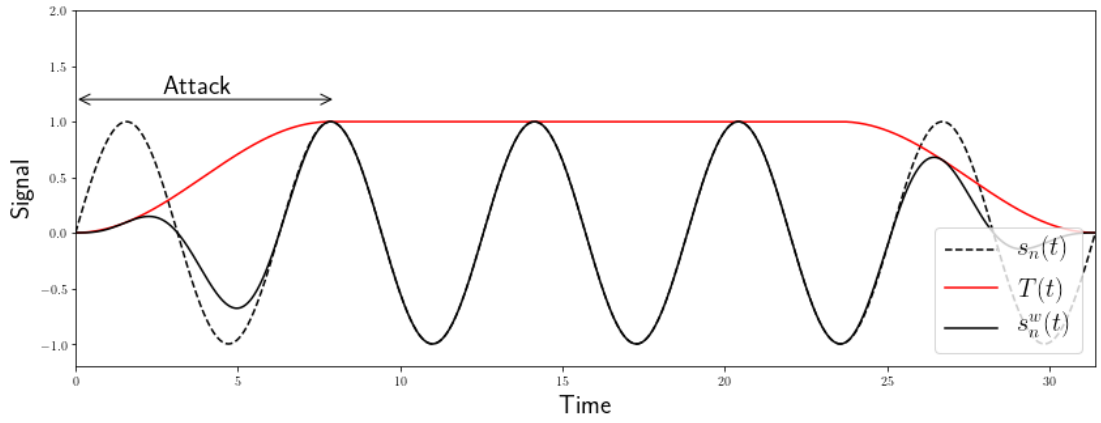

FIGURE 1 – Signaux brut $s_n(t)$ et fenêtré $s_n^w(t)$.

1. Vous pouvez par exemple prendre le Do_0 comme note de référence qui a pour fréquence $f_0 = 32.7$ Hz.

Écrire une fonction `freq2array(f, dt, tmax=1, alpha=0.8, display=False)` :
- prenant en argument d'entrée :
  - la fréquence de la note f,
  - le pas temporel entre chaque échantillon dt,
  - sa durée t_{max},
  - une argument optionnel α représentant le pourcentage de la fenêtre égal à 1 (Cf. documentation de la fonction `tukey`),
  - un argument booléen optionnel `display` précisant s'il faut ou non afficher une figure représentant les données numériques,
- et retournant un objet `ndarray` représentant la note de musique.

Puis :
- synthétiser une note à l'aide de cette fonction,
- et, avec l'option `display=True`, la fonction devra également tracer sur une même figure $s(t)$, $T(t)$, et $s(t)T(t)$.

5 Prélimaire 4 : *Wrapper* pour la création de l'array

Il peut être plus pratique d'envelopper (*to wrap* en anglais) les deux fonctions précédentes dans une unique fonction.

Écrire une fonction `note2array(X, N, dt, tmax=1, alpha=0.8)` qui utilise les fonctions `note2freq(X, N)` et `freq2array(f, dt, tmax=1, alpha=0.8)` pour retourner directement l'array représentant la note à partir de la note elle même et des paramètres dt, t_{max}, et α.

6 Prélimaire 5 : Assemblage des différentes notes

Maintenant que nous sommes en mesure de convertir une note textuelle (du style `do3`) en sigal numérique, il ne manque plus qu'à envisager une vue plus globale du problème.

Notre pièce musical finale sera constitué d'une unique signal audio, il va donc de soit que toutes les notes synthétisées doivent faire partie d'un unique objet de type `ndarray` afin d'obtenir un signal final global.

Dans un premier temps, nous allons faire un assemblage de notes de même durée, et avec le même paramètre α de fenêtrage. Plus tard nous améliorerons ce point.

Écrire une fonction `assemble(song, dt, tmax=1, alpha=0.8)` :
- prenant en argument d'entrée :
  - la liste de liste `song` regroupant les notes de musique,
  - le pas temporel entre chaque échantillon dt,
  - la durée t_{max} d'une note,
  - une argument optionnel α représentant le paramétrage de la fenêtre temporelle.
- et retournant un objet `ndarray` représentant la pièce musicale.

Cette fonction utilisera bien entendu les fonctions précédemment développées.

7 Essai # 1 : Première écoute

Il s'agit maintenant d'écrire nos données numériques dans un fichier audio. Le format audio de base est le format `wav`. Le module `scipy.io.wavfile` propose les fonctions `read` et `write` comme illustré ci-après.

Lecture/Ecriture de fichiers wav

```
import scipy.io.wavfile as wf

wf.write(fname, rate, data)          # fonction d'écriture
rate, data = wf.read(fname)          # fonction de lecture
```

Les arguments de ces deux fonctions sont :

— `fname` qui représente le chemin vers le ficher à lire ou à écrire,

— `data` qui représente les données à écrire ou les données lues,

— `rate` qui représente quant à lui la fréquence d'échantillonnage $f_e = 1/dt$ du signal, *i.e* l'inverse du pas temporel dt entre chaque échantillon.

Utilisez la fonction `write` afin d'écrire votre `ndarray` représentant cotre pièce musical dans un fichier `wav`. Pour cette première synthèse, vous pouvez utiliser la sortie de la fonction `arpeggiator` ou la suite de note ci-dessous qui se prête bien aux premiers tests.

Guns N' Roses - Sweet Child O' Mine

```
sweet_child = [['Ré3', 'Ré4', 'La3', 'Sol3'],
               ['Sol4', 'La3', 'Fa#4', 'La3']]
```

Note : Le fichier `wav` généré par votre fonction pourra être lu à l'aide du lecteur média de votre choix (vlc par exemple fonctionne à tous les coups, contrairement au lecteur média fournit avec Windows).

ATTENTION : Ajuster le volume sonore à un niveau relativement bas pour ce premier test car vous pourriez avoir de mauvaises suprises...

8 Essai #2 : Seconde écoute

Deux notions essentielles dans le domaine de l'audio sont la notion de quantification et la notion d'échantillonnage temporel.

Pour la notion de quantification, vous consulterez la documentation de la fonction `write` qui contient des informations cruciales pour l'écriture de fichier `wav`.

Pour la notion d'échantillonnage temporel, la chose fondamentale est de rester cohérent. Voici quelques ordres de grandeur.

Le la3, note du diapason, a une fréquence fondamentale d'environ 440 Hz, soit une période d'environ 2.27e-3 s. Si votre dt est plus grand que cela, n'espérez pas pouvoir entendre le la3. Avec une dizaine de points par période temporelle, le signal est correctement défini. Il faudrait donc un dt d'au plus 2.27e-4 s pour avoir une bonne description du la3.

Et votre dt, combien vaut il ?

Pour référence, la fréquence d'échantillonnage utilisée pour les cd audios est de 44100 Hz, soit un dt entre chaque échantillon temporel d'environ 2.27e-05 s. Chaque la3 est discrétisé avec une centaine d'échantillon à cette fréquence d'échantillonnage, tandis que le do9, à une fréquence d'environ 16.7 kHz, est discrétisé avec 2 échantillons. Mais bon, ... qui entend vraiment à presque 17 kHz à part les jeunes enfants ?

Si la première écoute n'était pas satisfaisante, modifier votre dt en conséquence. Le mieux est de l'ajuster à 1 / 44100, fréquence d'échantillonnage du cd audio.

9 Postliminaire 1 : Rendre le code plus flexible

Une première attention doit être apportée à la fonction `note2array`. Le fait de ne préciser qu'un argument t_{max}, un argument α restreint le champ des possibilités. Réfléchissez à une nouvelle structure pour cette fonction.

Il peut également être intéressant de fournir une fonction `oscillateur` à cette fonction `note2array` qui serait utilisée en lieu et place de la fonction sin et permettrait de personnaliser davantage la synthèse.

De plus, la symétrie de la fonction `tukey` limite également les effets d'attaque et de sustain qui pourraient être différencié. Au S1, nous avons recodé la fonction `tukey`, vous pouvez donc la personnaliser si vous le souhaitez.

10 Postliminaire 2 : Et après ?

Il s'agira finalement d'implémenter tout ce que vous trouverez utile pour améliorer votre module et vos fonctions. Par exemple :

— Prise en compte du rythme
— Addition, multiplication d'objets `Arpeggiator`
— Des méthodes de synthèse différentes [Cf. Figure 2] :
  — Triangle
  — Modulation d'amplitude
  — Ajout d'harmoniques
  — ...

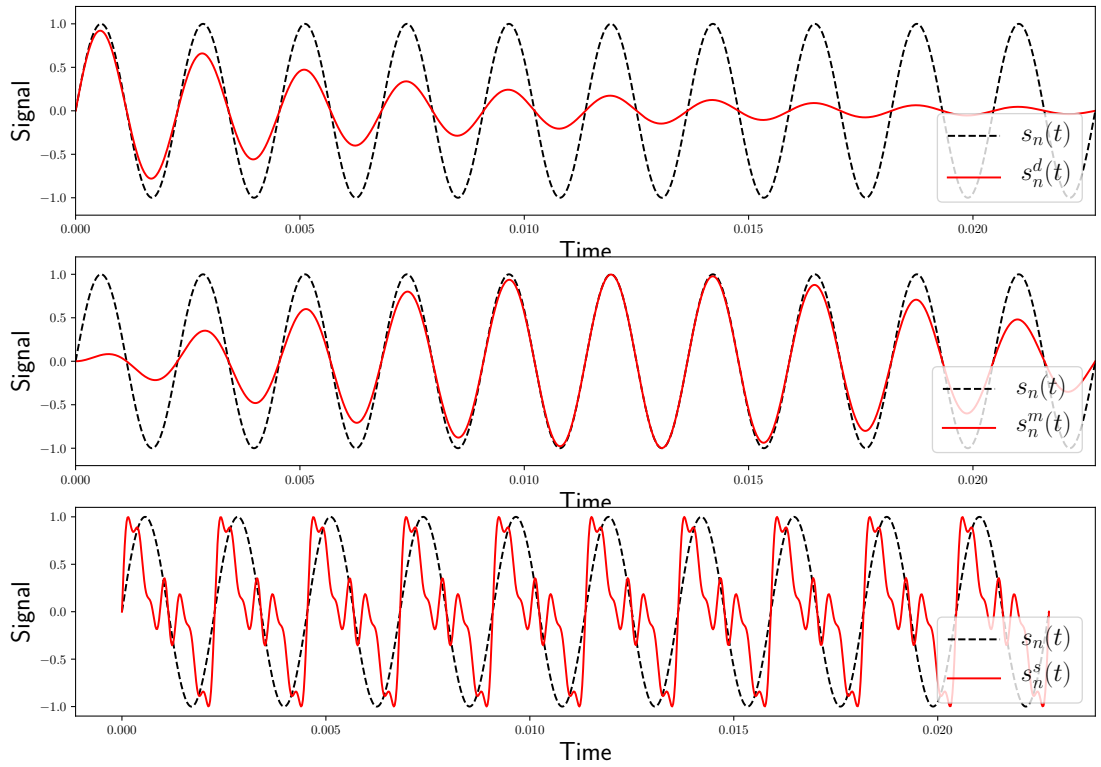

FIGURE 2 – Décroissance d'amplitude ($s_n^d(t)$), Modulation d'amplitude ($s_n^m(t)$), Ajout d'harmoniques ($s_n^s(t)$).