Fiche Ressource n°3 SAE22

Filtrage en temps réel avec Python et Scipy

1. Introduction au filtrage numérique

En électronique, un filtre numérique est un élément qui effectue un filtrage à l'aide d'une succession d'opérations mathématiques sur un signal discret. Comme un filtre analogique, il modifie le contenu spectral du signal d'entrée en atténuant ou éliminant certaines composantes spectrales indésirables. Contrairement aux filtres analogiques, qui sont réalisés à l'aide d'un agencement de composantes physiques (résistance, condensateur, inductance, …), les filtres numériques sont réalisés soit par des circuits intégrés dédiés, des processeurs programmables soit par logiciel dans un ordinateur.

Comme on l'a vu en cours et en TD, un filtre analogique est caractérisé dans le domaine temporel par une équation différentielle par exemple pour un circuit RC avec une tension d'entrée x(t) et de sortie y(t) :

$$x(t) - RC\frac{dy(t)}{dt} - y(t) = 0$$

Pour un signal discret avec une période d'échantillonnage T_E, en estimant la dérivée par :

$$\frac{dy(t)}{dt} \approx \frac{y(n.T_E) - y([n-1].T_E)}{T_E} \text{ qu'on note plus simplement } \frac{y_n - y_{n-1}}{T_E}$$

On obtient une équation aux différences :

$$x_n - RC\,\frac{y_n - y_{n-1}}{T_E} - y_n = 0 \rightarrow x_n - (\frac{RC}{T_E} + 1)\,y_n - \frac{RC}{T_E}\,y_{n-1} = 0$$

$$y_n = \frac{1}{(\frac{RC}{T_E} + 1)}\,x_n - \frac{RC}{T_E + RC}\,y_{n-1} \rightarrow y_n = b_0\,x_n - a_1\,y_{n-1}$$

De manière générale, pour un filtre d'ordre M, on obtient une équation aux différences de forme générale :

$$y_n = b_0\,x_n + b_1\,x_{n-1} + \ldots + b_1\,x_{n-N} - a_1\,y_{n-1} - a_2\,y_{n-2} - \ldots - a_M\,y_{n-M}$$

$$y_n = \sum_{k=0}^{N} b_k\,x_{n-k} - \sum_{k=1}^{M} a_k\,y_{n-k}$$

Il y a deux grandes familles de filtres numériques : les filtres à réponse impulsionnelle finie **FIR** en anglais (**Finite impulse response***)* et les filtres à réponse impulsionnelle infinie en anglais **IIR** (**Infinite Impulse Response**).

La sortie y_n d'un filtre FIR dépend uniquement de l'entrée x_n (les coefficients a_n sont donc nuls) contrairement aux filtres IIR qui sont des filtres récursifs, c'est-à-dire que la sortie du filtre dépend à la fois du signal d'entrée et des échantillons précédent de la sortie y_{n-i}. Les filtres IIR sont principalement la version numérique des filtres analogiques traditionnels : Butterworth, Tchebychev, Bessel et Elliptique.

Comme leur nom l'indique, la réponse impulsionnelle d'un filtre FIR se stabilisera ultimement à zéro alors qu'un filtre IIR possède une réponse impulsionnelle qui ne s'annule jamais définitivement ou qui converge éventuellement vers zéro à l'infini.

Pour obtenir la fonction de transfert H(f) pour les filtres analogiques, on prend la transformée de Fourier de l'équation différentielle qui caractérise le système. Sur le même principe, pour obtenir la fonction de transfert d'un filtre numérique, on prend la transformée en Z de l'équation aux différences. Sans rentrer dans les détails de cette transformée, on obtient :

$$H(z) = \frac{b_0 + b_1 . z^{-1} + \ldots + b_N . z^{-N}}{1 + a_1 z^{-1} + \ldots + a_M z^{-M}}$$

2. Filtrage avec Python et Scipy

La librairie *scipy.signal* permet de créer des filtre FIR ou IIR de type Butterworth, Tchebychev, Bessel et Elliptique. Pour minimiser le volume de connaissances à acquérir, on utilisera seulement les filtres IIR de butterworth en TP ou en SAÉ.

scipy.signal.butter permet de générer les coefficients a_n et b_n d'un filtre numérique dont la fonction de transfert approche celle d'un filtre analogique Butterworth, avec la structure suivante :

```
b,a = signal.butter(ordre du filtre, fréquence de coupure normalisée,
btype ='type de filtre').
```

La fréquence de coupure normalisée correspond à la fréquence de coupure divisée par $f_E/2$, f_E étant la fréquence d'échantillonnage.

A partir des coefficients a et b, la fonction *signal.freqz* permet de générer les valeurs complexes de la fonction de transfert h pour différentes valeurs de fréquence f ou plus précisément de pulsation ω en rad/échantillon avec la structure suivante :

`ω, h = signal.freqz(b, a)` avec

$$f = \frac{f_{E \times \omega}}{2\pi}$$

Ceci permet de tracer la fonction de transfert du filtre. Pour le tracé, la fonction matplotlib $semilogx$ permet d'effectuer le tracé avec une échelle semilog. L'échelle semilog est une échelle logarithmique par décade : de 1 à 10 puis de 10 à 100, puis de 100 à 1000, …

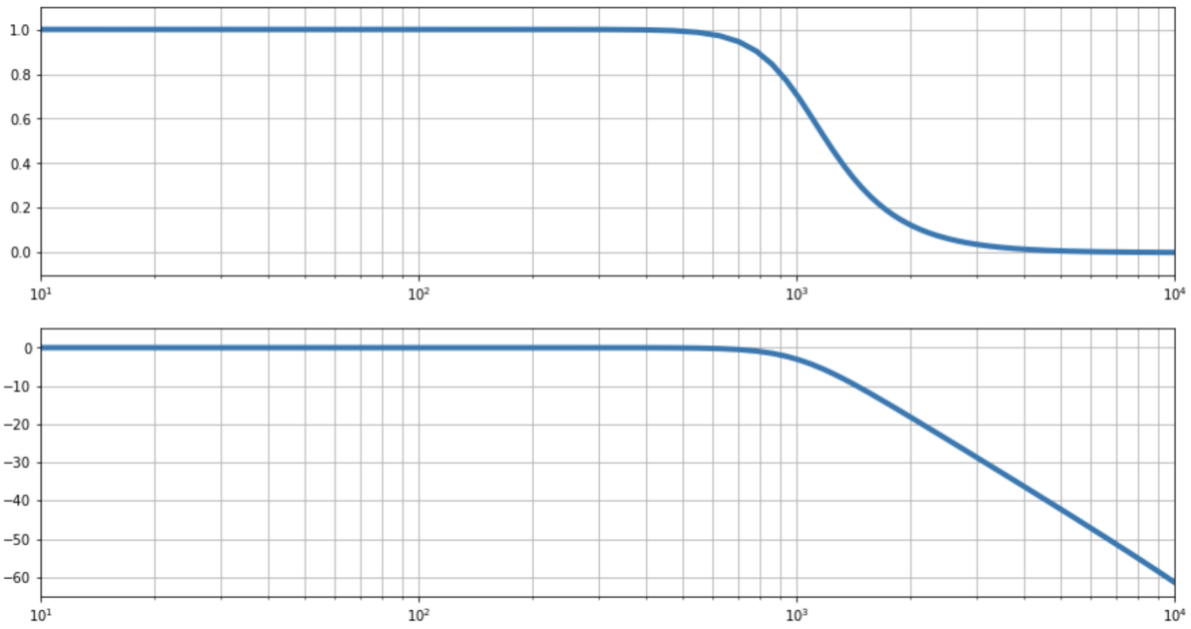
Ceci est illustré ci-dessous pour un filtre passe-bas d'ordre 3, de fréquence de coupure 1000Hz.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal

#Caractéristiques du filtre
fe=80000
nyquist=0.5*fe
fc=1000
fcn=fc/nyquist
order=3

#Création du filtre
b, a = scipy.signal.butter(order, fcn, btype='low', analog=False)
w, h = scipy.signal.freqz(b, a)
f=0.5 * fe * w / np.pi

#Création d'une figure avec 2 systèmes d'axes au format 2 lignes 1 colonne
fig, ax = plt.subplots(2,1, figsize = (15, 10))
ax[0].semilogx(f, (abs(h)), linewidth = 4)
ax[0].grid(True, which="both")
ax[1].semilogx(f, 20*np.log10(abs(h)),linewidth = 4)
ax[0].grid(True, which="both")
```



Une fois le filtre créé, on peut filtrer un signal avec la fonction signal.lfilter comme illustré ci-dessous :

```
b, a = scipy.signal.butter(ordre, fcn, btype='low', analog=False)
filtered=scipy.signal.lfilter(b, a, signal)
```

On peut de même créer un filtre passe-haut, passe-bande ou coupe-bande en modifiant le champ *btype* dans la fonction signal.butter.

3. Forme SOS « Second Order Filter »

Il existe différentes façons de réaliser un même filtre numérique par exemple la forme directe qui donne les coefficient a_n et b_n du filtre comme on l'a présenté dans la section précédente.

Un deuxième type de réalisation du filtre est la forme **SOS** "Second Order Filter" qui réalise le filtre par une succession de filtres du second ordre, par exemple 3 filtres d'ordre 2 pour un filtre d'ordre 6.

Certaines réalisations sont plus efficaces en termes de nombre d'opérations ou d'éléments de stockage nécessaires à leur mise en œuvre, et d'autres offrent des avantages tels qu'une stabilité numérique améliorée et une erreur d'arrondi réduite.

Il est généralement conseillé d'utiliser la forme SOS notamment pour les filtres passe-bande ou coupe-bande avec un ordre supérieur à 2, car il y a des risques d'instabilité.

L'option *output* dans la fonction *signal.butter* permet de spécifier le type de réalisation souhaité pour le filtre. Un exemple de mise en œuvre d'un filtre numérique IIR de Butterworth avec une réalisation de type SOS est illustrée ci-dessous :

```
fe = 100000
nyquist = 0.5 * fs
fc1 = 1000
fc2 = 1200
fcn1 = fc1 / nyquist
fcn2 = fc2 / nyquist
order = 8

#Création du filtre
filtre_sos = signal.butter(order, [fcn1,fcn2], btype='bandpass', analog = False, output = 'sos')

#Filtrage du signal
porteuse = signal.sosfilt(filtre_sos, signal)
```

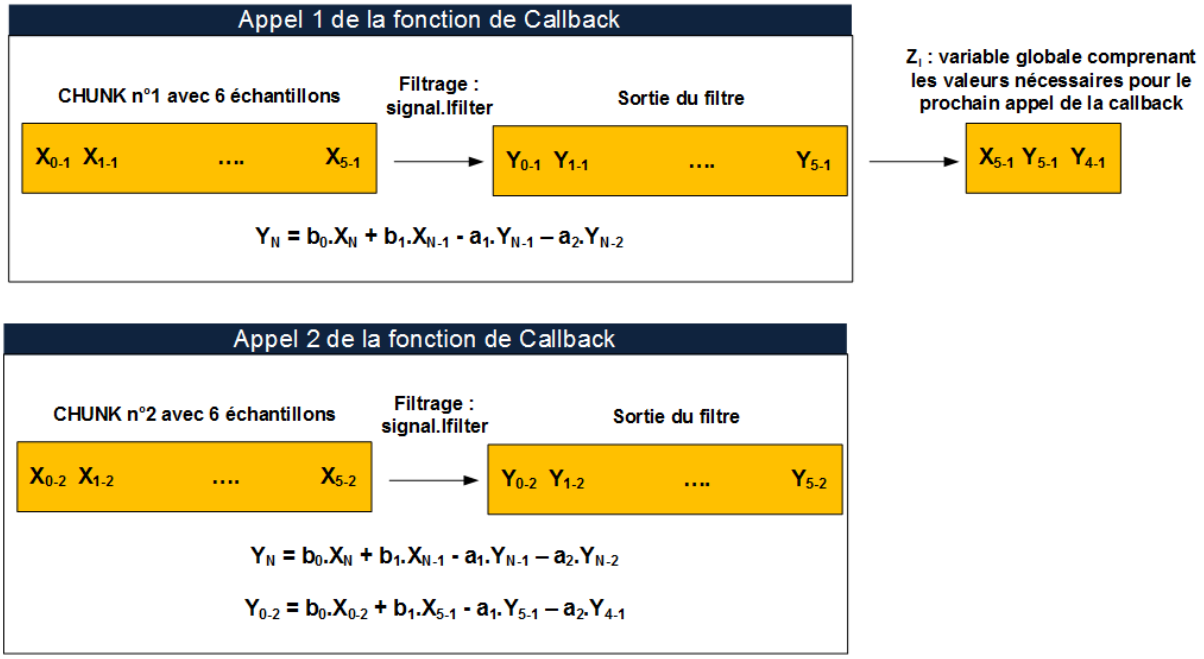
4. Filtrage en temps réel

Quand on filtre en temps réel, on ne travaille pas sur tout le signal d'un coup mais sur des morceaux chunk du signal qu'on traite dans le cadre de la SAE dans la fonction de callback.

Comme on l'a expliqué, un filtre numérique est défini par une équation aux différences :

$$y_n = b_0\, x_n + b_1\, x_{n-1} + \ldots + b_1\, x_{n-N} - a_1\, y_{n-1} - a_2\, y_{n-2} - \ldots - a_M\, y_{n-M}$$

Cependant à chaque appel de la fonction de callback, pour le premier échantillon du chunk, le filtre ne dispose pas des valeurs y_{n-i} et x_{n-i} puisqu'ils étaient donnés dans le précédent appel de la fonction de callback. Il faut donc stocker ces valeurs dans une variable globale pour que le filtre puisse avoir accès à ces valeurs à chaque appel de la fonction de callback. Ceci est illustré ci-dessous :



Pour le filtrage en temps réel, la structure du programme pour un filtre avec une forme SOS sera donc la suivante :

```
#Calcul des coefficients du filtre en forme SOS
coef_sos = signal.butter(order1, [fcn1,fcn2], btype='bandpass', analog = False, output = 'sos')

#Initialisation de la variable globale zi pour stocker les valeurs précédentes xn-i et yn-i
zi = signal.sosfilt_zi(sos1)


def callback(indata, outdata, frames, time, status):
    global zi1
    #Récupération du morceau "chunk" de signal voie gauche
    signal = indata[:,0]
    #Filtrage du chunk
    signal_filtre, zi = signal.sosfilt(coef_sos, signal, zi=zi1)
```