Conception et mise en oeuvre des filtres numériques

S.Gibet

Année 2019-2020

On s'intéresse ici à la conception de filtres numériques et à leur réponse fréquentielle, en utilisant le module *scipy signal*. On verra différentes manières de réaliser le filtrage numérique, que l'on appliquera ensuite sur des signaux réels. Le filtrage est réalisé soit avec des fonctions écrites en python, soit en utilisant les fonctions *signal.convolve* et *signal.lfilter*. Le programme *filtreLTI.py* est donné sur Moodle de façon à vous permettre de tester les différents filtres et leur visualisation.

1 Définitions

1.1 Relation de récurrence

On considère un signal numérique x_n obtenu avec la période d'échantillonnage T_e. Le filtrage numérique linéaire consiste à obtenir un nouveau signal numérique y_n en utilisant une relation de récurrence de la forme suivante :

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

Les coeficcients a_k et b_k sont réels. De manière générale, un échantillon de sortie est obtenu en faisant une combinaison linéaire des N échantillons précédents de l'entrée et des $M-1$ échantillons précédents de la sortie.

1.2 Réponse impulsionnelle

Une impulsion est un signal numérique défini par :

$x_0 = 1$
$x_n = 0$ si $n \neq 0$

La réponse impulsionnelle est la sortie y_n obtenue pour cette entrée impulsion.
La réponse impulsionnelle est dite finie si le nombre d'échantillons non nuls est fini.

1.3 Stabilité

Un filtre est stable si la réponse impulsionnelle tend vers zéro lorsque n tend vers l'infini. Un filtre à réponse impulsionnelle finie (FIR) est donc toujours stable.

Lorsqu'un filtre à réponse impulsionnelle infinie (IIR) est stable, sa réponse impulsionnelle décroit de manière exponentielle avec n, si bien qu'elle peut être considérée en pratique comme

finie. Un tel filtre est stable si et seulement si tous les pôles de sa fonction de transfert ont un module strictement inférieur à 1. Un filtre IIR instable a une réponse impulsionnelle qui présente soit un comportement oscillatoire, soit une divergence vers l'infini.

2 Conception d'un filtre

2.1 Filtre à réponse impulsionnelle finie

Lorsque les coefficients a_n sont tous nuls, la sortie s'exprime comme une combinaison linéaire des échantillons de l'entrée :

$$y_n = \sum_{k=0}^{N-1} b_k x_{n-k} \tag{2}$$

L'opération ainsi réalisée est un produit de convolution, c'est pourquoi on parle dans ce cas de filtrage par convolution. Ce type de filtrage est aussi employé en traitemetn d'image.
La réponse impulsionnelle est :

$$y_n = b_n \tag{3}$$

Autrement dit, les coefficients b_n constituent la réponse impulsionnelle du filtre. Cette réponse est dite finie, car le nombre d'échantillons non nuls en sortie est fini, et égal au nombre de coefficients b_n non nuls.

Le type le plus répandu de filtre à réponse impulsionnelle finie est le filtre FIR à phase linéaire, dont les coefficients sont obtenus par série de Fourier. On se contente dans ce TP d'utiliser la fonction *signal.firwin* pour calculer un tel filtre.

Dans le programme *filtreFIR.py*, vous trouverez un exemple de construction d'un filtre passe-bas FIR constitué de 41 paramètres, avec une fréquence de coupure 0.1 (relative à la fréquence d'échantillonnage). Les questions suivantes illustrent les définitions précédentes. Le programme est déjà donné. Il s'agira ici de bien comprendre la signification des paramètres, de les modifier et de visualiser les différentes sorties (réponse impulsionnelle, réponse fréquentielle).

1. Tracez la réponse impulsionnelle de ce filtre FIR. Il s'agit d'une fonction sinus cardinal dont le maximum est à l'indice $P = numtaps/2$. On peut s'attendre à un décalage de P échantillons entre l'entrée et la sortie.

2. Calculez la réponse fréquentielle et visualisez la. Cette réponse fréquentielle est obtenue avec la fonction *freqz*. Vous remarquerez que ces filtres FIR peuvent être très sélectifs si le nombre de coefficients est élevé. Par ailleurs la phase est linéaire par rapport à la fréquence dans la bande passante. Les signaux dont le spectre est dans la bande passante sont donc transmis sans distorsion, avec un décalage constant, égal ici à P.

3. Pour définir un filtre passe-haut ou passe-bande, il faut donner la valeur False à l'argument *pass_zero*. Définissez plusieurs filtres FIR avec des fonctions passe bas, passe-haut, passe-bande et passe-bas. Observez l'influence des fréquences de coupure, ainsi que la phase.

2.2 Filtre à réponse impulsionnelle infinie

Lorsque les coefficients a_k sont non nuls, on parle de filtre récursif :

$$y_n = \sum_{k=0}^{N-1} b_k x_{n-k} - \sum_{k=1}^{M-1} a_k y_{n-k} \tag{4}$$

La réponse impulsionnelle d'un tel filtre est infinie, c'est-à-dire qu'elle comporte un nombre infini d'échantillons non nuls (du moins en théorie). Il peut arriver que le filtre soit instable, c'est-à-dire que sa réponse impulsionnelle ne tende pas vers zéro.

La fonction *signal.iitfilter* permet d'obtenir un filtre IIR à partir de fonctions de transfert analogiques standard (Butterworth, Chebychev, etc). Voici par exemple comment obtenir un filtre passe-bas d'ordre 2 avec une fréquence de coupure fc/fe=0.1 :

```
b3,a3 = scipy.signal.iirfilter(N=2,Wn=[0.1*2],btype="lowpass",ftype="butter")

print(a3)
--> array([ 1.        , -1.1429805,  0.4128016])

print(b3)
--> array([ 0.06745527,  0.13491055,  0.06745527])
```

La fréquence de coupure relative a été multipliée par deux, car la fréquence de Nyquist est par convention égale à 1. Ce filtre comporte 3 coefficients b_n et 3 coefficients a_n (en comptant a0 = 1).

Les filtres IIR ont l'avantage de comporter peu de coefficients, d'introduire un faible déphasage (à peu près linéaire dans la bande passante), et d'être très proches des filtres analogiques. En revanche, ils ne permettent pas d'obtenir la très forte sélectivité des filtres FIR, car les filtres IIR d'ordre élevé sont instables. À nombre de coefficients identique, ils sont toutefois plus efficaces que les filtres FIR. Les filtres IIR sont très utilisés en traitement du son. L'inconvénient des filtres IIR est la possibilité d'obtenir un filtre instable lorsque l'ordre est élevé (à partir de l'ordre 6 environ). Pour savoir si un filtre est stable, il faut calculer ses pôles :

```
(zeros,poles,gain) = signal.tf2zpk(b3,a3)
```

et vérifier que leur module est strictement inférieur à 1 :

```
print(numpy.absolute(poles))
```

1. Déterminez la réponse fréquentielle du filtre IIR précédent (voir le code dans le programme $filtreFir$.
2. Modifiez les paramètres des filtres et observez leur réponse en fréquence.
3. Calculez les pôles et zéros du filtre IIR et tracez les ;
4. Vérifiez la valeur du module des pôles

3 Réalisation du filtrage

Le filtrage consiste à appliquer la relation de récurrence (1) pour calculer chaque échantillon du signal de sortie.

L'implémentation du filtrage en temps réel, sur un processeur de signal (DSP) ou sur microcontrôleur, pose des problèmes spécifiques liés aux ressources disponibles sur ce type d'unité (quantité de mémoire, vitesse de calcul, calculs en virgule flottante, etc).

On s'intéresse ici au filtrage réalisé en Python, sur le signal bruité suivant :

```
t = np.linspace(-1, 1, 201)
x = (np.sin(2*np.pi*0.75*t*(1-t) + 2.1) + 0.1*np.sin(2*np.pi*1.25*t + 1) + 0.18*np.cos(2*np.p
xn = x + np.random.randn(len(t)) * 0.08
```

3.1 Réalisation d'un filtrage FIR par convolution

L'opération à réaliser sur le signal est un produit de convolution :

$$y_n = \sum_{k=0}^{N-1} b_k x_{n-k} \tag{5}$$

Le nombre de coefficients $N = 2P+1$ a été défini par *numtaps* dans la fonction $firwin$. On remarque que les $2P$ premières valeurs de y_n ne peuvent pas être calculées avec cette somme. Il y deux manières de procéder pour ces $2P$ premiers échantillons : soit on les calcule en utilisant une partie des coefficients, soit on ne les filtre pas et on annule la sortie. Nous adoptons la seconde solution, plus facile à implémenter. Voici une fonction réalisant le filtrage :

```
def filtrage_convolution(x,b):
     N = len(b)
     ne = len(x)
     y = np.zeros(ne)
     for n in range(N-1,ne):
          accum = 0.0
          for k in range(N):
               accum += b[k]*x[n-k]
          y[n] = accum
     return y
```

En pratique, vous utiliserez la fonction *signal.convolve*.
Convolution centrée – Le mode *same* permet d'obtenir une sortie de même taille que l'entrée. Dans ce cas, Le signal de sortie n'est pas décalé par rapport au signal d'entrée. Cela est du au fait que la convolution appliquée est en fait centrée. Si $N = 2P+1$ est le nombre de coefficients, la relation utilisée est :

$$y_n = \sum_{k=-P}^{P} b_{P-k} x_{n+k} \tag{6}$$

Pour les P premiers et les P derniers points, une partie de la somme est calculée.
Ce mode de calcul de la convolution, appelé convolution centrée, convient bien au filtrage des images, où l'on veut que chaque pixel filtré coïncide avec le pixel d'origine, mais il ne correspond pas à la réalité du filtrage numérique en temps-réel, où les échantillons du futur ne sont évidemment pas disponibles lorsqu'on calcule y_n.

1. Filtrez le signal bruité xn donné ci-dessus par la fonction *signal.convolve*.
2. Visualisez le signal non bruité, le signal bruité et le signal bruité filtré.

3.2 Réalisation d'un filtrage IIR récursif

Pour un filtre IIR, on s'intéresse à l'application de la relation récursive entre l'entrée et la sortie suivante :

$$y_n = \sum_{k=0}^{N-1} b_k x_{n-k} - \sum_{k=1}^{M-1} a_k y_{n-k} \tag{7}$$

Pour un filtrage rapide, vous utiliserez la fonction *signal.lfilter*. Vous regarderez le programme permettant de contruire un filtre IIR butterworth et les 3 versions différentes du filtrage.