TP - Filtrage numérique

- Dans les séances précédentes, nous avons revu l'effet d'un filtrage analogique sur le spectre d'un signal (cas des filtres linéaires). Il peut être intéressant de remplacer une étape de filtrage analogique par une étape de **filtrage numérique** réalisée une fois le signal analogique converti en signal numérique.
  - Le filtrage numérique offre davantage de souplesse : on peut modifier les paramètres du filtres sans changer de composant dans le circuit, il n'y a plus de problème de mise en cascade des filtres (voir TP filtres en cascade).
  - Le filtrage numérique ne convient par contre pas pour des signaux de fortes puissances. Il est plus lourd à mettre en œuvre qu'un filtre analogique (conversion analogique numérique CAN / filtrage / conversion numérique analogique CNA).
- On suppose dans ce TP que le signal à filtrer a été correctement échantillonné c'est-à-dire en respectant le critère de Shannon-Nyquist :

$$f_e > 2.f_{max}$$

  En pratique, on ajoute un filtre passe-bas (fréquence de coupure en $f_e/2$) dit **filtre anti-aliasing** avant l'échantillonnage pour éviter les signaux parasites liés à un repliement de spectre.
- L'outil nous permettant d'analyser l'effet d'un filtrage est la **transformée de Fourier**. En pratique, elle est calculée par un logiciel (ordinateur / oscilloscope numérique) à partir d'un signal échantillonné : on parle alors de transformée de Fourier discrète (TFD).

Nous allons dans un premier temps, nous intéresser à la TFD transformée de Fourier discrète, nous verrons ensuite des exemples de filtres numériques.

De la TF à la TFD

- La transformée de Fourier $S(f)$ de la fonction $s(t)$ s'obtient en calculant l'intégrale :

$$S(f) = \int_{-\infty}^{+\infty} s(t) \exp(-i2\pi ft)\ dt$$

  - La composante $S(f)$ est un nombre complexe, on représente en général $|S(f)|$.
  - Les fréquences $f \in]-\infty, +\infty[$ dans le cas général mais les valeurs $f \geq 0$ suffisent pour décrire un signal réel.
- Dans le cas où l'acquisition a lieu sur une durée T_{acq}, on approxime le calcul de la TF :

$$S(f) \simeq \int_{0}^{+T_{acq}} s(t) \exp(-i2\pi ft)\ dt$$

  On observe alors un élargissement des raies : dans le cas d'un signal purement sinusoïdal de fréquence f_o, le spectre comporte une raie de largeur $\Delta f \simeq \frac{1}{T_{acq}}$

On considère cette fois que le signal $s(t)$ est un signal discret avec :

$$s_k = s(t_k)\ ;\ t_k = kT_e$$

On peut approximer la TF par la somme discrète suivante (méthode des rectangles) :

$$S(f) = T_e \sum_{k=0}^{N-1} s_k \exp(-i2\pi fkT_e)$$

En choisissant les fréquences dans la liste :

$$f_n = n\frac{1}{T_{acq}} = \frac{n}{NT_e}$$

On obtient la transformée de Fourier discrète (TFD) :

$$S_n = S_(f_n) = T_e \sum_{k=0}^{N-1} s_k \exp(-i2\pi nk/N)$$

Le programme recommande d'utiliser la fonction **rfft** de la bibliothèque **numpy.fft** pour calculer la TFD d'un signal (il s'agit d'une fft : fast Fourier transform). On peut obtenir la liste des fréquences à l'aide de **rfft.rfftfreq** ou générer une liste des $\frac{n}{NT_e}$ (voir les exemples).

Signal à analyser

1. Créer la liste des instants `t` ainsi que le signal `s1`.
   — échantillonné à $f_e = 4$kHz ;
   — sur une durée T_{acq} de 1s ;
   — comportant une composante de fréquence $f_1 = 50$Hz et d'amplitude 1 et d'une autre composant de fréquence f_2 =400Hz d'amplitude 0.5 correspondant à un bruit que l'on souhaite éliminer.
   — on notera `N` le nombre de points et `Te` le pas d'échantillonnage. On pourra utiliser :

   $$\text{np.arange}(\text{debut}, \text{fin}, \text{pas})$$

2. Tracer ce signal sur une durée correspondant à 2 périodes $T_1 = 1/f_1$.
3. Générer la TFD de `s1` à l'aide de l'instruction :

   $$\text{fourier1} = \text{np.fft.rfft}(\text{s1})$$

   Quel est le nombre de valeurs dans `fourier1` ? Vérifier que `fourier1` est constituée de nombres complexes.
4. Vérifier que pour `N` pair, `fourier1` comporte `N`/2 + 1 valeurs alors que pour `N` impair il en comporte (`N`+1)/2.
5. On donne ci-dessous un extrait de la notice de la fonction `numpy.fft.rfft`

   Notes

   When the DFT is computed for purely real input, the output is Hermitian-symmetric, i.e. the negative frequency terms are just the complex conjugates of the corresponding positive-frequency terms, and the negative-frequency terms are therefore redundant. This function does not compute the negative frequency terms, and the length of the transformed axis of the output is therefore `n//2 + 1`.

   When `A = rfft(a)` and fs is the sampling frequency, `A[0]` contains the zero-frequency term 0*fs, which is real due to Hermitian symmetry.

   If *n* is even, `A[-1]` contains the term representing both positive and negative Nyquist frequency (+fs/2 and -fs/2), and must also be purely real. If *n* is odd, there is no term at fs/2; `A[-1]` contains the largest positive frequency (fs/2*(n-1)/n), and is complex in the general case.

   If the input *a* contains an imaginary part, it is silently discarded.

   Justifier à l'aide de cet extrait que pour `N` pair, les valeurs de `fourier1` correspondent à la liste de fréquences :

   $$0, \frac{1}{\text{Tacq}}, \frac{2}{\text{Tacq}}, \ \ldots \ \frac{\text{N}}{2\text{Tacq}}$$

6. Générer la liste des fréquences correspondant au spectre calculé à l'aide de np.fft.rfftfreq(N,d=Te) avec N nombre de points dans le signal et Te temps d'échantillonnage.
7. Tracer alors le spectre du signal : il faut penser à prendre le module des valeurs calculées (np.abs()) et il est nécessaire de normaliser par 2/N (vous devez obtenir un pic d'amplitude 1 à f_1 et un pic d'amplitude 0.5 à f_2. Vérifier que la largeur des pics est de 2/Tacq.

Dans la suite, on étudie l'effet de différents filtres sur ce signal. Vous pouvez également récupérer des valeurs expérimentales dans le fichiersignal2.csv :

```
f = open("signal2.csv") # on oouvre le fichier avec les résultats
sep=";" # données séparées par un ; dans tableau1
data=f.readlines() #on lit toutes les lignes
f.close() #on referme le fichier


t1=[]
s1=[]

for ligne in data:
    ligne=ligne.strip().split(sep) #on sépare les différents élémen
ts
    ligne=list(map(float,ligne)) #on convertit chaque élément en fl
ottant
    t1.append(ligne[0]) #on rentre les valeurs dans les lites adapt
ées
    s1.append(ligne[1])
plt.plot(t1,s1)
plt.show()

N1=len(t1)
```

Étude d'un filtre à moyenne glissante

Pour éliminer la composante à haute fréquence (f_2), on peut choisir d'utiliser un filtre à moyenne glissante dans lequel la ième composante filtre1[i] sera la moyenne entre signal[i], signal[i-1]... jusqu'à signal[i-nfiltre] soit sur nfiltre valeurs.

1. Générer le signal filtre1 à partir de s1 en prenant nfiltre=10.
2. Tracer filtre1 en fonction de f.
3. Tracer le spectre de filtre1. Observer l'influence de nfiltre.

Comme on fait une moyenne sur 10 valeurs, toute variation du signal de fréquence supérieure à $f_e/10$ est gommée par le filtre. Il s'agit bien d'un filtre passe-bas supprimant les fréquences supérieures à $f_e/10$. Ici 10 mesures avec f_e=4kHz, cela fait une moyenne sur 10*0,25 ms soit une période du signal à 400Hz correspondant au bruit, d'où l'explication de la disparition totale du pic.

Filtre passe-bas d'ordre 1

On peut également partir des filtres analogiques connus pour réaliser un filtrage numérique. On considère par exemple la forme canonique de la fonction de transfert d'un filtre passe-bas d'ordre 1 :

$$\underline{H}(j\omega) = \frac{H_o}{1 + j\frac{\omega}{\omega_c}}$$

ω_c désignant la pulsation de coupure à -3dB du filtre.

1. Montrer, à partir de l'expression de la fonction de transfert que le signal d'entrée et le signal de sortie du filtre sont liés par la relation :

$$\frac{dv_s}{dt} + \omega_c v_s = H_o \omega_c v_e$$

2. En écrivant :

$$\frac{dv_s}{dt} = \frac{v_s[t] - v_s[t - T_e]}{T_e}$$

exprimer $v_s(t)$ en fonction de $v_s(t - T_e)$ et $v_e(t)$.

3. Écrire le programme associé en prenant une fréquence de coupure de l'ordre de 60Hz, on appellera filtre2 la fonction correspondante. Tracer filtre2 en fonction de t.
4. Définir la transformée de Fourier de filtre2 et tracer son spectre. Commenter.
5. Adapter la méthode à un filtrage passe bas du second ordre.

Filtrage dans le domaine fréquentiel

On peut également réaliser le filtrage dans le domaine fréquentiel. Comme on l'a déjà vu dans le premier TP, dans le cas d'un filtre linéaire de fonction de transfert $\underline{H}$, la réponse à une tension d'entrée :

$$u(t) = c_o + \sum_{n=1}^{\infty} c_n \cos(2\pi n f\ t + \varphi_n)$$

s'obtient en sommant les réponses à ses différentes composantes (en notant $\omega_n = n2\pi f$) :

$$s(t) = c_o |\underline{H}(0)| + \sum_{n=1}^{\infty} c_n |\underline{H}(j\omega_n)| \cos(2\pi n f\ t + \varphi_n + \arg(\underline{H}(j\omega_n))$$

En adaptant ce résultat au cas d'un signal non périodique, on peut ainsi :

1. Déterminer la transformée de Fourier du signal d'entrée (coefficients c_n et phases φ_n).
2. Multiplier ces valeurs par la $\underline{H}(j\omega_n)$.
3. Reconstruire le signal filtré à l'aide d'une transformée de Fourier inverse (voir numpy.fft.irfft).

numpy.fft.rfft

fft.rfft(*a, n=None, axis=- 1, norm=None*) [source]

Compute the one-dimensional discrete Fourier Transform for real input.

This function computes the one-dimensional *n*-point discrete Fourier Transform (DFT) of a real-valued array by means of an efficient algorithm called the Fast Fourier Transform (FFT).

Parameters: **a** : *array_like*

Input array

n : *int, optional*

Number of points along transformation axis in the input to use. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input along the axis specified by *axis* is used.

axis : *int, optional*

Axis over which to compute the FFT. If not given, the last axis is used.

norm : *{"backward", "ortho", "forward"}, optional*

> ***New in version 1.10.0.***

Normalization mode (see **numpy.fft**). Default is "backward". Indicates which direction of the forward/backward pair of transforms is scaled and with what normalization factor.

> ***New in version 1.20.0:*** The "backward", "forward" values were added.

Returns: **out** : *complex ndarray*

The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified. If *n* is even, the length of the transformed axis is `(n/2)+1`. If *n* is odd, the length is `(n+1)/2`.

Raises: **IndexError**

If *axis* is not a valid axis of *a*.

numpy.fft.rfftfreq

`fft.rfftfreq(n, d=1.0)` [source]

Return the Discrete Fourier Transform sample frequencies (for usage with rfft, irfft).

The returned float array *f* contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start). For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

Given a window length *n* and a sample spacing *d*:

```
f = [0, 1, ...,     n/2-1,     n/2] / (d*n)   if n is even
f = [0, 1, ..., (n-1)/2-1, (n-1)/2] / (d*n)   if n is odd
```

Unlike **fftfreq** (but like **scipy.fftpack.rfftfreq**) the Nyquist frequency component is considered to be positive.

Parameters: `n` : *int*
Window length.

`d` : *scalar, optional*
Sample spacing (inverse of the sampling rate). Defaults to 1.

Returns: `f` : *ndarray*
Array of length `n//2 + 1` containing the sample frequencies.

Examples

```
>>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5, -3, 4],
dtype=float)
>>> fourier = np.fft.rfft(signal)
>>> n = signal.size
>>> sample_rate = 100
>>> freq = np.fft.fftfreq(n, d=1./sample_rate)
>>> freq
array([  0.,  10.,  20., ..., -30., -20., -10.])
>>> freq = np.fft.rfftfreq(n, d=1./sample_rate)
>>> freq
array([  0.,  10.,  20.,  30.,  40.,  50.])
```