Traitement numérique du signal

1. Dispositif matériel et logiciel

Ce document montre comment utiliser une carte d'acquisition Eurosmart (Sysam PCI ou Sysam SP5) pour effectuer des opérations de traitement du signal avec un environnement de programmation Python.

Le module python `pycanum`, développé par l'auteur, permet de piloter la carte d'acquisition directement depuis un script python. Il est ainsi possible de traiter les signaux juste après leur acquisition, ou même en temps réel lors d'une acquisition en flot continu. Le module permet aussi la synthèse de signaux via le convertisseur numérique-analogique de la carte, ou par la sortie audio de l'ordinateur.

Les modules python utilisés pour le traitement du signal sont `numpy`, `matplotlib`, et `scipy.signal`.

Pour générer des signaux afin de tester les différentes techniques, nous utilisons trois méthodes :

- ▷ Un générateur de fonctions basse fréquence (GBF) de laboratoire pour les signaux simples.
- ▷ Un générateur de signal audio périodique, intégré au module `pycan`.
- ▷ Le logiciel de synthèse audio Pure Data, avec un programme permettant de générer des sons comportant des harmoniques.
- ▷ La lecture d'un fichier audio avec un lecteur multimédia.

Pour les trois dernières méthodes, le signal est récupéré sur une des deux voies de la sortie audio de l'ordinateur, au moyen d'un câble jack-RCA et d'un adaptateur RCA-banane.

Les caractéristiques données dans le texte sont celles de la carte Sysam SP5. Sauf mention contraire, les exemples fonctionnent également sur la carte Sysam PCI, mais avec des caractéristiques maximales inférieures (fréquence d'échantillonnage, mémoire, etc).

Certaines fonctions du module `pycanum` ne sont pas décrites dans ce document, ou sont mentionnées sans être détaillées. Pour plus d'informations, consulter CAN Eurosmart : interface pour Python.

2. Numérisation et analyse d'un signal

2.a. Échantillonnage et numérisation

L'interface avec la carte d'acquisition est établie de la manière suivante :

```
import pycanum.main as pycan
sys = pycan.Sysam("SP5")
```

Cela crée un objet de la classe `Sysam`. Les différentes fonctions sont accessibles par cet objet.

La carte Sysam SP5 comporte 4 convertisseurs analogique-numériques (CAN) 12 bits pouvant échantillonner jusqu'à 10 MHz. Dans le mode d'acquisition standard, les

échantillons sont mémorisés dans la carte puis transférés à l'ordinateur par la liaison USB.

Pour configurer les CAN, il faut tout d'abord préciser les entrées à utiliser (numérotées de 0 à 7) et la tension maximale qui sera envoyée sur ces entrées, de la manière suivante :

```
sys.config_entree(voies,calibres,diff=[])
```

L'argument `voies` est la liste des entrées, `calibres` est la liste des tensions maximales. Les calibres des amplificateurs sont sélectionnés en fonction de ces valeurs. Il y a quatre calibres : 10, 5, 2 et 0.2 Volts. L'argument optionnel `diff` permet d'utiliser un ou plusieurs CAN en mode différentiel. Par exemple, le canal 0 peut être utilisé en mode différentiel en précisant `diff=[0]`.

La deuxième étape est la configuration de l'échantillonnage, qui consiste à choisir une période d'échantillonnage en microsecondes (mimimum 0.1) et un nombre de points :

```
sys.config_echantillon(techant,nbpoints)
```

Le nombre de points est le nombre d'échantillons pour chaque voie à numériser. Le nombre total d'échantillons est limité par la mémoire interne de la carte (2^{18} échantillons).

Pour faire l'acquisition, il faut exécuter :

```
sys.acquerir()
```

Cette fonction déclenche l'acquisition de manière logicielle et retourne lorsque celle-ci est terminée. Pour obtenir un déclenchement par seuil à partir d'une source, il faut utiliser la fonction `config_trigger` ou `config_trigger_externe`.

La récupération des instants et des valeurs des tensions se fait de la manière suivante :

```
t = sys.temps()
u = sys.entrees()
```

Chacune de ces fonctions renvoit un tableau `numpy` comportant autant de lignes que de voies numérisées. Par exemple, les tableaux `t[0]` et `u[0]` contiennent les instants et les tensions pour la première voie. Les instants des différentes voies sont identiques si celles-ci sont sur des canaux différents.

Voici un exemple complet, qui montre comment importer le module et ouvrir l'interface avec la carte. Une acquisition est faite sur l'entrée EA0 avec une fréquence d'échantillonnage de 10 kHz. Le nombre de points est 10000, soit une durée totale de 1 s.

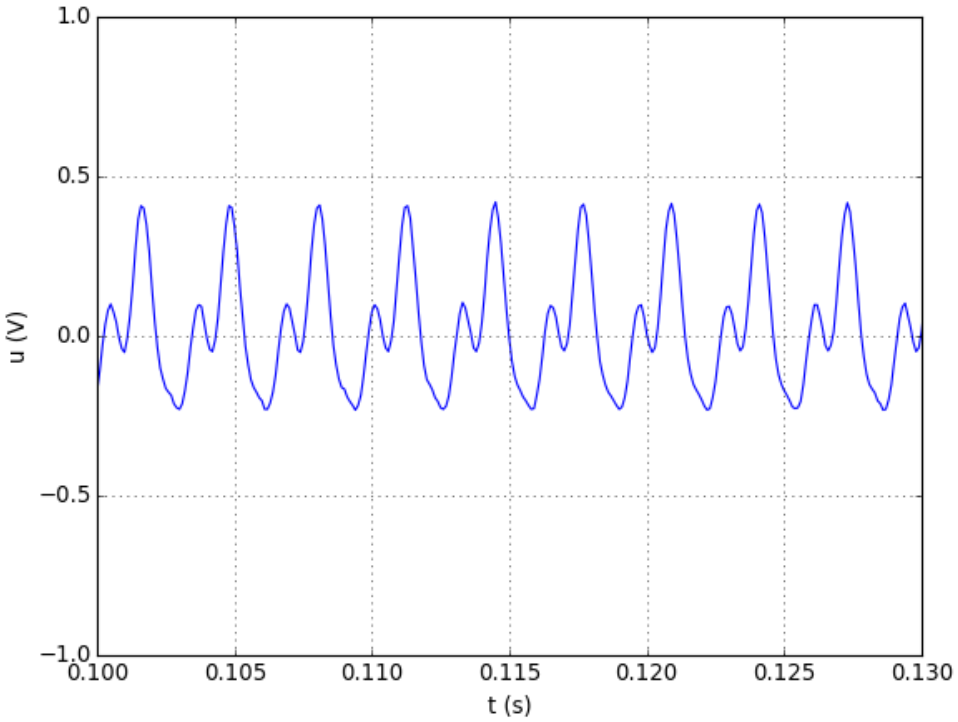
```
import pycanum.main as pycan
sys = pycan.Sysam("SP5")
sys.config_entrees([0],[2])
fechant = 10000.0
techant = 1.0/fechant
nechant = 10000
sys.config_echant(techant*1e6,nechant)
sys.acquerir()
t = sys.temps()
u = sys.entrees()
sys.fermer()
```

Pour enregistrer les données dans un fichier texte, on peut utiliser la fonction suivante :

```
import numpy
numpy.savetxt('signal-1.txt',[t[0],u[0]])
```

Comme exemple de signal numérisé, on considère un signal périodique d'environ 311 Hz comportant 3 harmoniques, généré sur la sortie audio par un programme Pure Data. L'une des deux voies de la sortie audio est reliée à l'entrée EA0 de la carte d'acquisition. L'acquisition d'une durée de 1 seconde permettra de faire une analyse fréquentielle au Hz près. Pour la représentation temporelle, on se limite à une fenêtre de quelques dizaines de millisecondes :

```
[t0,u0] = numpy.loadtxt('signal-1.txt')
figure()
plot(t0,u0)
xlabel("t (s)")
ylabel("u (V)")
fenetre = 0.03
axis([0.1,0.1+fenetre,-1,1])
grid()
```

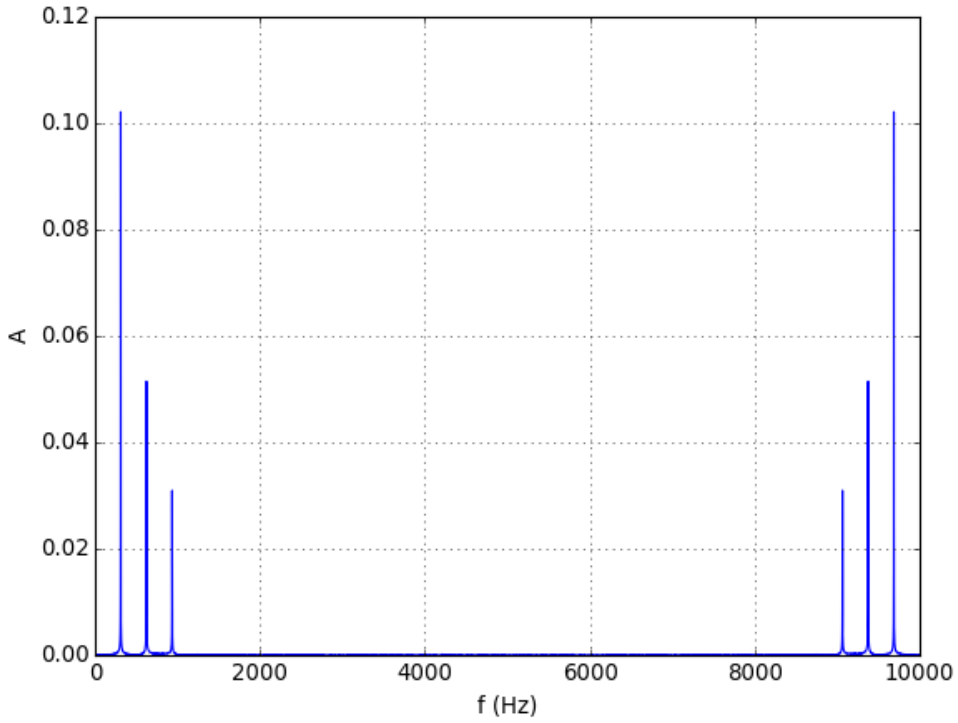


2.b. Analyse spectrale

L'analyse spectrale d'un signal numérique, ou analyse fréquentielle, se fait par la transformée de Fourier discrète (TFD). Nous utilisons pour cela la fonction `numpy.fft.fft`, qui calcule la TFD avec l'algorithme de transformée de Fourier rapide. Pour construire

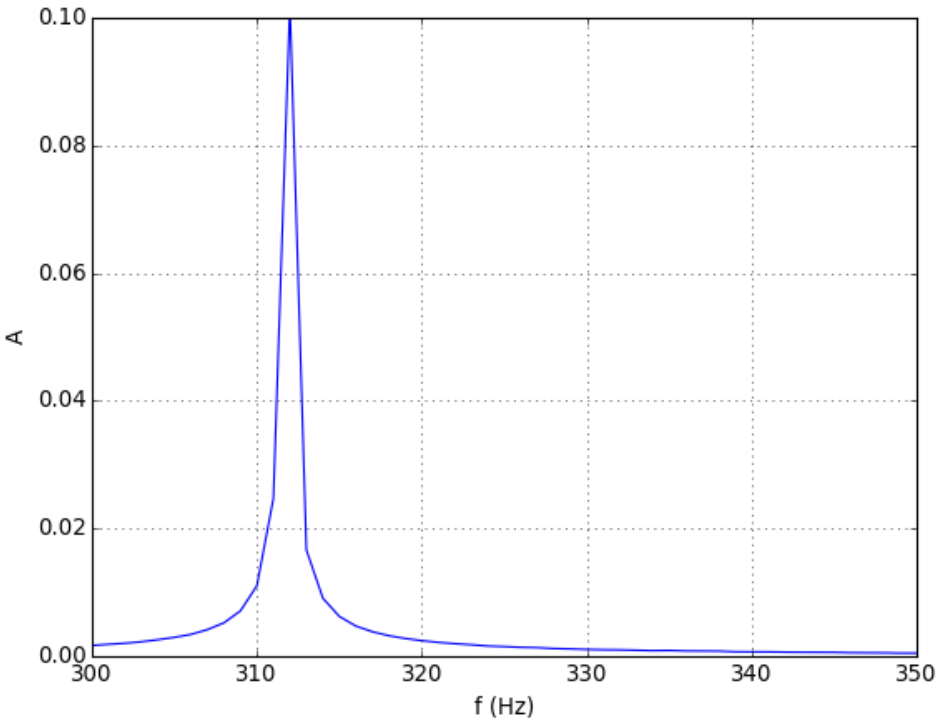
l'échelle de fréquence, il faut attribuer la fréquence d'échantillonnage à la plus grande fréquence du spectre. La résolution fréquentielle du spectre est donc égale à l'inverse de la durée totale du signal, soit $1/(N_e T_e)$, où N_e est le nombre d'échantillons.

```
import numpy.fft
Ne = len(u0)
Te = t0[1]-t0[0]
spectre = numpy.absolute(numpy.fft.fft(u0))/Ne
frequences = numpy.arange(Ne,)*1.0/(Te*Ne)
figure()
plot(frequences,spectre)
xlabel("f (Hz)")
ylabel("A")
grid()
```



On reconnaît le spectre du signal analogique entre 0 et la fréquence de Nyquist 5000 Hz, et son spectre image entre 5000 Hz et 10000 Hz. On peut regarder en détail la première harmonique pour obtenir la fréquence du signal :

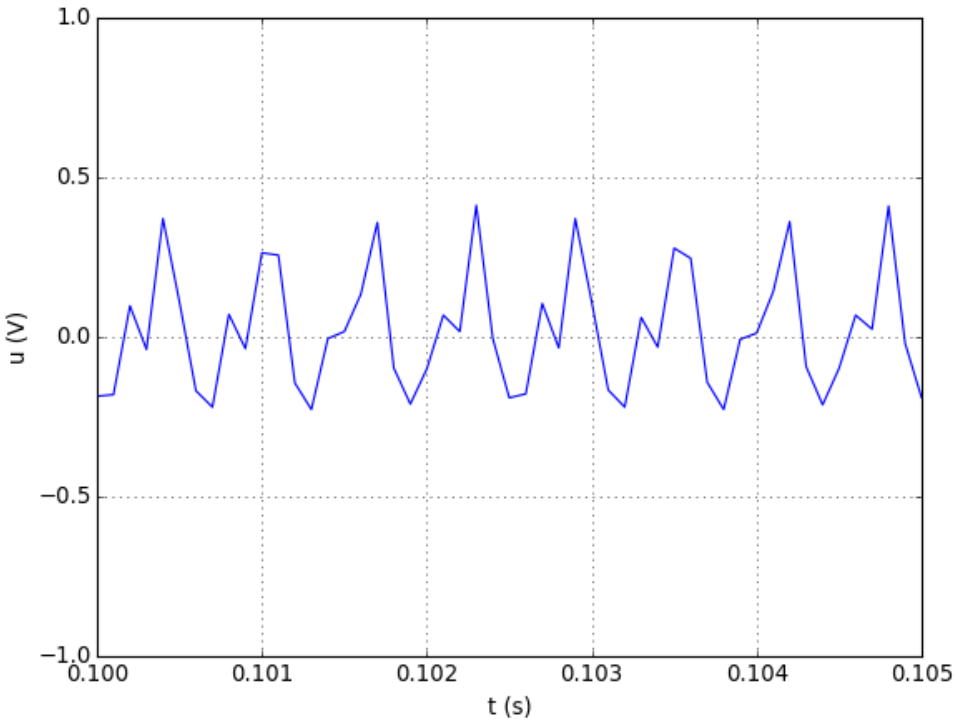
```
figure()
plot(frequences,spectre)
xlabel("f (Hz)")
ylabel("A")
grid()
axis([300,350,0,0.1])
```



La fréquence peut être déterminée au Hertz près, car la durée totale analysée est de 1 seconde.

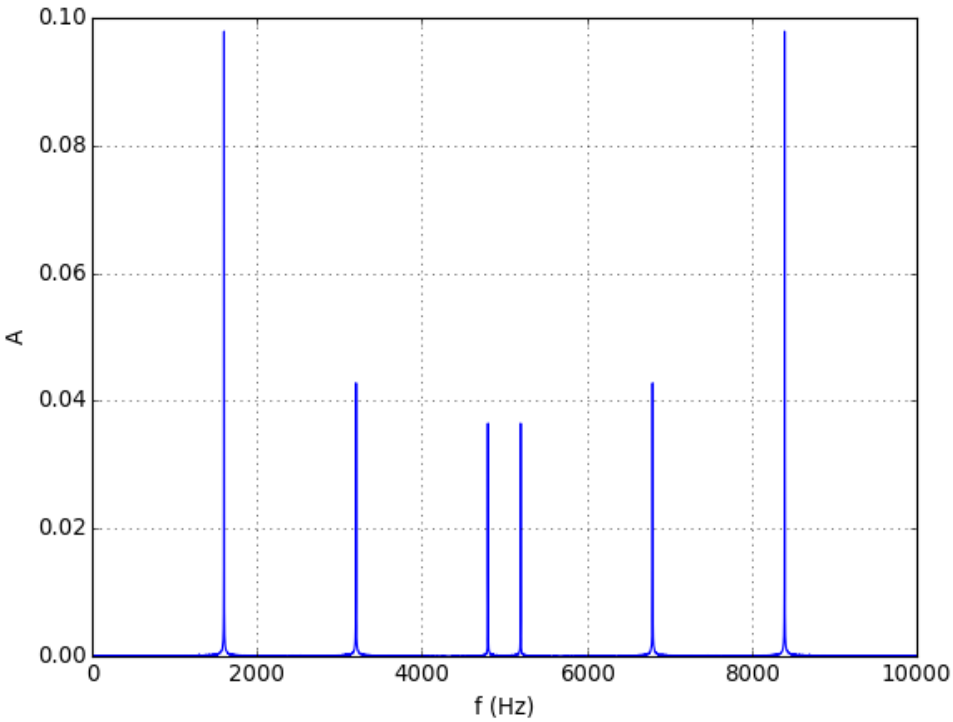
L'acquisition précédente a été faite en respectant la condition de Nyquist-Shannon pour l'harmonique de rang 3. Il est intéressant d'augmenter la fréquence du signal pour que son harmonique de rang 3 soit juste en dessous de la fréquence de Nyquist, par exemple avec une fréquence fondamentale de 1600 Hz :

```
[t0,u0] = numpy.loadtxt('signal-2.txt')
figure()
plot(t0,u0)
xlabel("t (s)")
ylabel("u (V)")
fenetre = 0.005
axis([0.1,0.1+fenetre,-1,1])
grid()
```



La représentation temporelle du signal obtenue en reliant les points par des segments (interpolation linéaire) est mauvaise. Voyons le spectre :

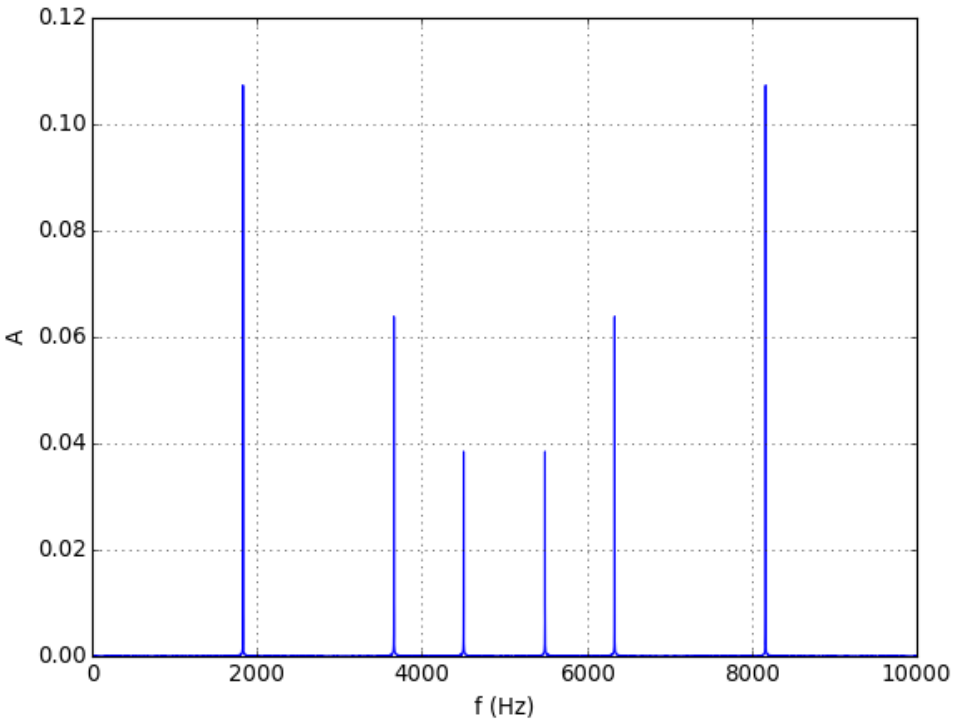
```
Ne = len(u0)
Te = t0[1]-t0[0]
spectre = numpy.absolute(numpy.fft.fft(u0))/Ne
frequences = numpy.arange(Ne,)*1.0/(Te*Ne)
figure()
plot(frequences,spectre)
xlabel("f (Hz)")
ylabel("A")
grid()
```



Le spectre est correct, car la condition de Nyquist-Shannon est respectée. Le théorème de Nyquist-Shannon montre qu'il est en théorie possible de reconstruire le signal, même si l'interpolation linéaire n'est pas satisfaisante. Une manière d'obtenir une bonne reconstruction du signal est de prolonger la transformée de Fourier discrète avec des zéros puis de calculer la transformée inverse, mais cette méthode ne fonctionne que sur le signal entier, pas sur des blocs consécutifs. Dans un système temps-réel (par exemple un lecteur CD), la reconstruction se fait par interpolation.

Voyons le cas où la condition de Nyquist-Shannon n'est pas respectée :

```
[t0,u0] = numpy.loadtxt('signal-3.txt')
Ne = len(u0)
Te = t0[1]-t0[0]
spectre = numpy.absolute(numpy.fft.fft(u0))/Ne
frequences = numpy.arange(Ne,)*1.0/(Te*Ne)
figure()
plot(frequences,spectre)
xlabel("f (Hz)")
ylabel("A")
grid()
```



On observe le phénomène de repliement de spectre : l'harmonique de rang 3, dont la fréquence est supérieure à la fréquence de Nyquist, se replie dans la gamme $[0, 5000]\ Hz$. Dans ce cas, la reconstruction du signal conduirait à l'apparition d'une fréquence qui n'est pas multiple de la fréquence fondamentale.

3. Filtrage numérique

3.a. Conception d'un filtre

Relation de récurrence et fonction de transfert en Z

On considère un signal numérique défini par une suite de nombres réels x_n. La fréquence d'échantillonnage est notée f_e. La période d'échantillonnage est $T_e = 1/f_e$.

Le filtrage numérique linéaire consiste à obtenir un nouveau signal y_n en utilisant une relation de récurrence de la forme suivante :

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

Les coefficients a_n et b_n sont réels. D'une manière générale, un échantillon de la sortie est obtenu en faisant une combinaison linéaire des N échantillons précédents de l'entrée et des $M-1$ échantillons précédents de la sortie.

Cette relation est invariante dans le temps car les coefficients sont constants. Elle est causale car le calcul de la sortie à l'instant n se fait avec les valeurs de l'entrée pour des instants antérieurs. Les filtres fonctionnant en temps réels dans les DSP (*digital signal processor*) sont bien sûr causals.

Pour étudier la réponse fréquentielle du filtre, on considère une entrée sinusoïdale. En utilisant la représentation complexe, le signal d'entrée s'écrit dans ce cas :

$$x_n = \exp(i2\pi f n T_e) = z^n \tag{2}$$

où l'on a introduit une variable complexe z définie par :

$$z = \exp(i2\pi f T_e) \tag{3}$$

L'argument de ce nombre complexe de module unité varie de 0 à π lorsque la fréquence de la sinusoïde varie de 0 à $f_e/2$.

La relation de récurrence étant linéaire, on peut s'attendre à une sortie de la forme suivante :

$$y_n = H(z)x_n \tag{4}$$

En reportant les expressions complexes de x_n et de y_n dans la relation de récurrence, on obtient en effet :

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots} \tag{5}$$

Cette fonction complexe de la variable z est la *fonction de transfert en Z* du filtre. En faisant le changement de variable défini par la relation (3), on obtient la réponse fréquentielle du filtre, dont on déduit le gain et le déphasage en fonction du rapport f/f_e.

La *réponse impulsionnelle* d'un filtre est la réponse à une entrée ne comportant qu'une seule valeur non nulle $x_0 = 1$. Lorsque le nombre de coefficients non nuls de la réponse impulsionnelle est finie, le filtre est dit à *réponse impulsionnelle finie*.

Un filtre est dit *stable* si sa réponse impulsionnelle tend vers 0 lorsque n tend vers l'infini. Une condition nécessaire et suffisante de stabilité est la suivante : tous les pôles de la fonction de transfert en Z doivent avoir un module strictement inférieur à 1.

Un filtre à réponse impulsionnelle finie (RIF) est toujours stable, puisque sa réponse impulsionnelle s'annule à partir d'un certain rang N.

Filtre à réponse impulsionnelle finie

Lorsque les coefficients a_k sont tous nuls, la relation de récurrence s'écrit :

$$y_n = \sum_{k=0}^{N-1} b_k x_{n-k} \tag{6}$$

La sortie à l'instant n est une combinaison linéaire des valeurs de l'entrée aux instants n, $n-1$, etc. Il s'agit d'un produit de convolution discret. On vérifie facilement que la réponse impulsionnelle est précisément la suite des coefficients b_0, b_1, ... b_{N-1}. Ce type de filtre est donc, par construction, à réponse impulsionnelle finie (filtre RIF).

Le type le plus utilisé de filtre RIF est le filtre à phase linéaire, dont les coefficients sont calculés par série de Fourier et par application d'un fenêtrage (par exemple Hamming). La fonction scipy.signal.firwin permet de faire ce calcul.

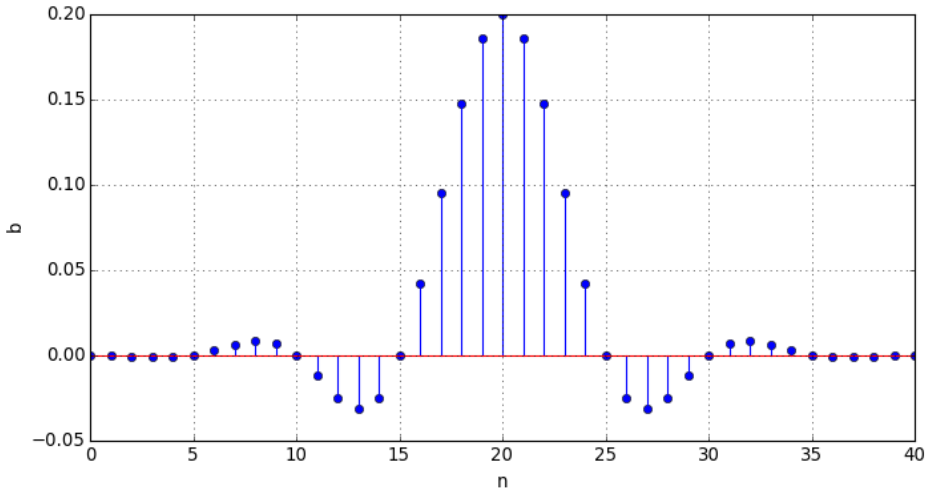
Voici par exemple le calcul d'un filtre passe-bas dont la fréquence de coupure est $0,1 f_e$. Le nombre de coefficients est choisi impair, égal à $N = 2P + 1$. Un fenêtrage de Hamming est appliqué :

```
import scipy.signal
P=20
b1 = scipy.signal.firwin(numtaps=2*P+1,cutoff=[0.1],window='hann',nyq=0.5)
```

L'argument `nyq=0.5` permet de préciser que la fréquence de Nyquist (moitié de la fréquence d'échantillonnage) est égale à 1/2, car la valeur par défaut est 1. De cette manière, la fréquence donnée dans la liste `cutoff` est bien relative à la fréquence d'échantillonnage.

Voici une représentation graphique des coefficients, c'est-à-dire de la réponse impulsionnelle :

```
from matplotlib.pyplot import *
figure(figsize=(10,5))
stem(b1)
xlabel("n")
ylabel("b")
grid()
```

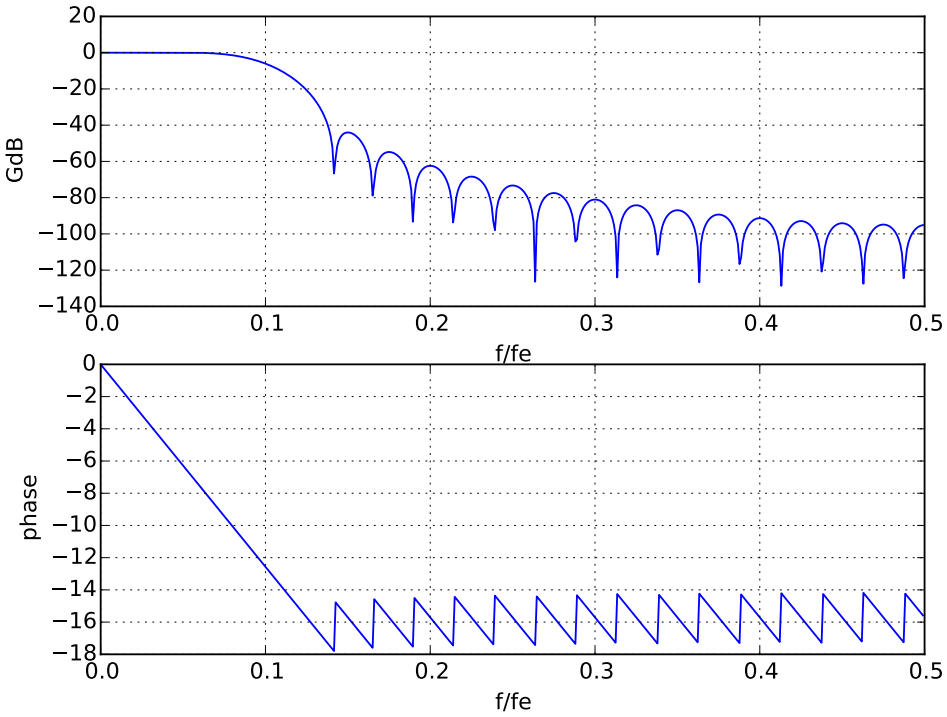


La réponse fréquentielle peut être obtenue avec la fonction `scipy.signal.freqz`, qui renvoit un tableau contenant les pulsations réduites comprises entre 0 et π (l'argument de la variable z) et un tableau contenant les valeurs de la fonction de transfert en Z.

```
w,h=scipy.signal.freqz(b1)
```

```
import numpy
figure()
subplot(211)
plot(w/(2*numpy.pi),20*numpy.log10(numpy.absolute(h)))
xlabel("f/fe")
ylabel("GdB")
grid()
subplot(212)
plot(w/(2*numpy.pi),numpy.unwrap(numpy.angle(h)))
```

```
xlabel("f/fe")
ylabel("phase")
grid()
```



Le retard de la sortie par rapport à l'entrée est constant, égal à PT_e. On voit en effet que l'échantillon prépondérant dans le calcul de la convolution se trouve au rang $n-P-1$. C'est la raison pour laquelle le déphasage varie linéairement avec la fréquence, une propriété importante de ce type de filtre, qui permet de transmettre des signaux dans la bande passante sans distorsion.

Filtre récursif

Le filtre est dit *récursif* lorsqu'il existe au moins un coefficient a_k non nul. Dans ce cas, un ou plusieurs échantillons antérieurs de la sortie sont utilisés pour caluler y_n :

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

La réponse impulsionnelle d'un tel filtre est infinie (RII), 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 stabilité est indispensable pour que le filtre soit utilisable. Dans ce cas, la réponse impulsionnelle tend très rapidement vers 0 (de manière exponentielle).

La fonction scipy.signal.iitfilter permet d'obtenir un filtre RII à partir de fonctions de transfert analogiques standard (Butterworth, Chebychev, etc). Voici par exemple comment obtenir un filtre passe-bande de Butterworth d'ordre 2 avec des fréquences de coupure $0,1f_e$ et $0,2f_e$.

```
b2,a2 = scipy.signal.iirfilter(N=2,Wn=[0.1*2,0.2*2],btype="bandpass",ftype="butter")
```

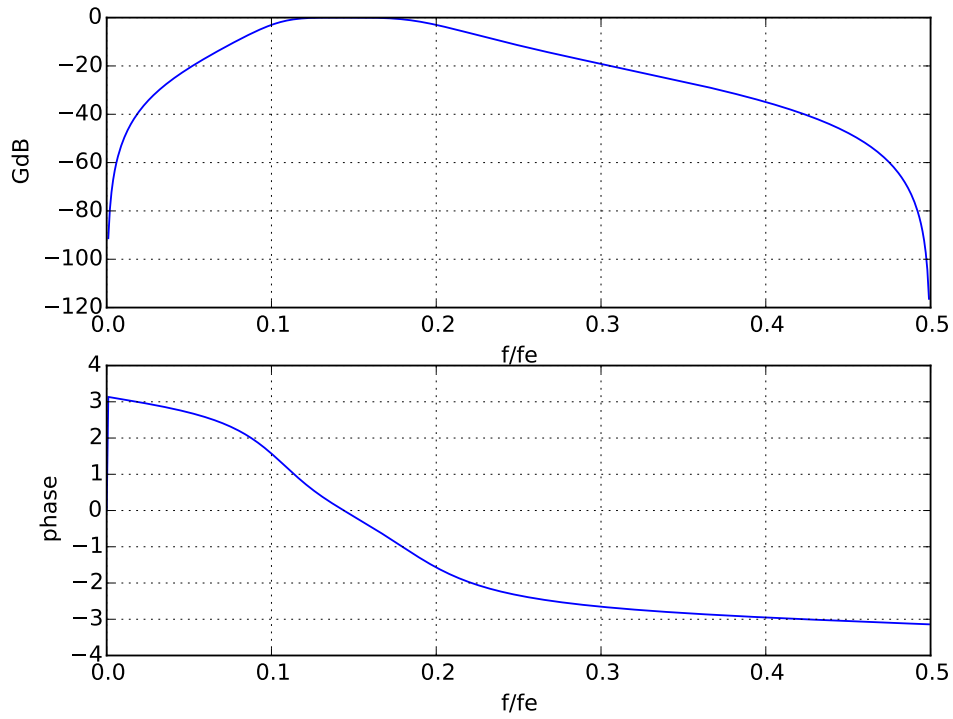
La fréquence de Nyquist est nécessairement 1, ce qui oblige à multiplier les fréquences de coupure par 2.

Voici les coefficients :

```
print(a2)
--> array([ 1.         , -1.94246878,  2.1192024 , -1.21665164,  0.4128016 ])

print(b2)
--> array([ 0.06745527,  0.        , -0.13491055,  0.        ,  0.06745527])
```

et la réponse fréquentielle :

```
w,h=scipy.signal.freqz(b2,a2)

figure()
subplot(211)
plot(w/(2*numpy.pi),20*numpy.log10(numpy.absolute(h)))
xlabel("f/fe")
ylabel("GdB")
grid()
subplot(212)
plot(w/(2*numpy.pi),numpy.unwrap(numpy.angle(h)))
xlabel("f/fe")
ylabel("phase")
grid()
```



3.b. Réalisation du filtrage

Dans le cas d'un filtre RIF, le filtrage s'obtient par convolution entre la réponse impulsionnelle et le signal. Cela peut être fait de la manière suivante :

```
y = scipy.signal.convolve(x,b,mode='same')
```

où `b` est la liste des coefficients b_k. Le mode `same` permet d'obtenir une liste de même taille que la liste initiale, ce qui est généralement souhaitable pour ne pas avoir à recalculer l'échelle de temps. Le produit de convolution appliqué par cette fonction n'est pas exactement celui de la relation causale (6), mais une convolution centrée, définie par :

$$y_n = \sum_{k=-P}^{P} b_{P-k} x_{n+k} \tag{8}$$

Dans ce cas, le filtre RIF à phase linéaire conduit à un déphasage nul. Pour le filtrage d'un signal entier stocké en mémoire, cela est plutôt avantageux, car le signal filtré se superpose au signal de départ. Ce type de filtrage RIF non causal est aussi utilisé pour les images. En revanche, il ne reproduit pas le résultat d'un filtrage en temps réel, pour lequel un décalage de P échantillons apparaît nécessairement entre l'entrée et la sortie. D'autre part, l'utilisation de la fonction `convolve` ne permet pas de traiter des blocs de signal successifs.

Pour appliquer un filtrage récursif (relation (7)), le plus simple et le plus efficace est d'utiliser la fonction `scipy.signal.lfilter`, qui opère de manière causale. Cette fonction ne calcule pas la somme (7) de manière directe, mais utilise une forme appelée *forme directe transposée de type II*. Cette forme est constituée des relations suivantes :

$$y_n = b_0 x_n + v_1 \tag{9}$$
$$v_1 = b_1 x_n - a_1 y_n + v_2 \tag{10}$$
$$v_2 = b_2 x_n - a_2 y_n + v_3 \tag{11}$$
$$\cdots \tag{12}$$

Les variables v_1, v_2, etc constituent les variables d'état du filtre. On remarque que le calcul de y_n fait appel à la valeur de v_1 calculée à l'instant $n-1$. Il faut tout d'abord calculer l'état initial des variables d'état :

```
zi = scipy.signal.lfiltic(b,a,y=[0],x=[0])
```

Les valeurs initiales de x et y sont complétées à zéro, mais il est possible de choisir un état initial quelconque. Les variables d'état sont renvoyées dans le tableau `zi`.

Voici comment se fait le filtrage d'un premier bloc de signal, en utilisant l'état initial des variables :

```
[y0,zf] = scipy.signal.lfilter(b,a,x0,zi=zi)
```

La fonction renvoit les échantillons filtrés `y0` et un tableau `zf` contenant les variables d'état. Il est ainsi possible de filtrer une deuxième bloc de signal (consécutif au premier), en utilisant ce tableau comme condition initiale :

```
[y1,zf] = scipy.signal.lfilter(b,a,x0,zi=zf)
```

Il est bien sûr possible d'utiliser cette fonction pour un filtre RIF, en posant `a=[1.0]`. On obtient ainsi un filtrage identique à celui d'un filtre temps-réel (convolution causale).

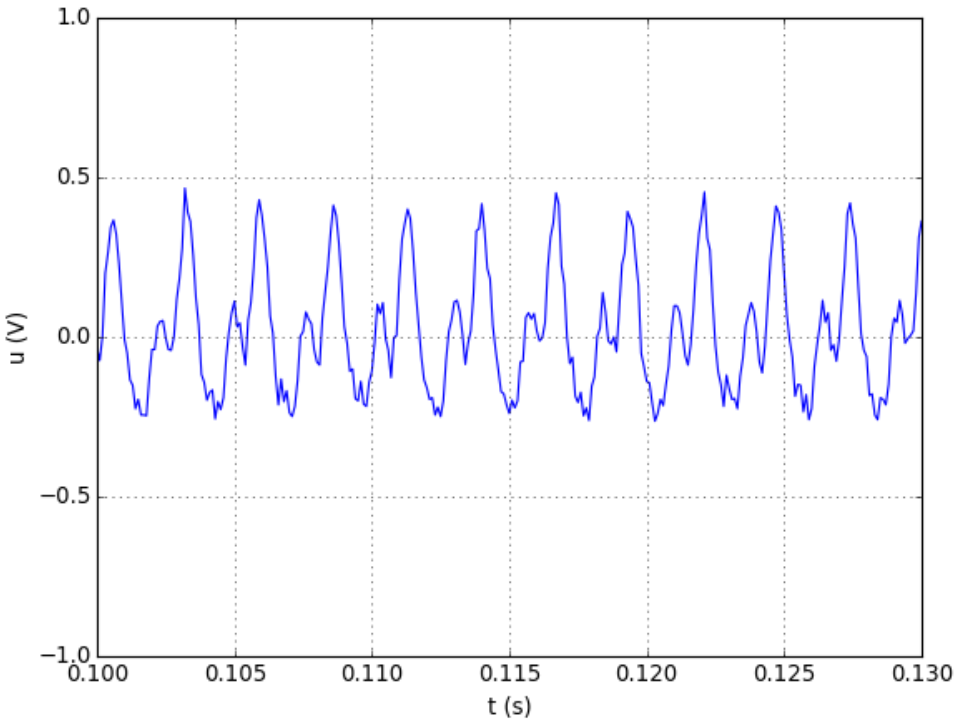
3.c. Exemples

Réduction du bruit

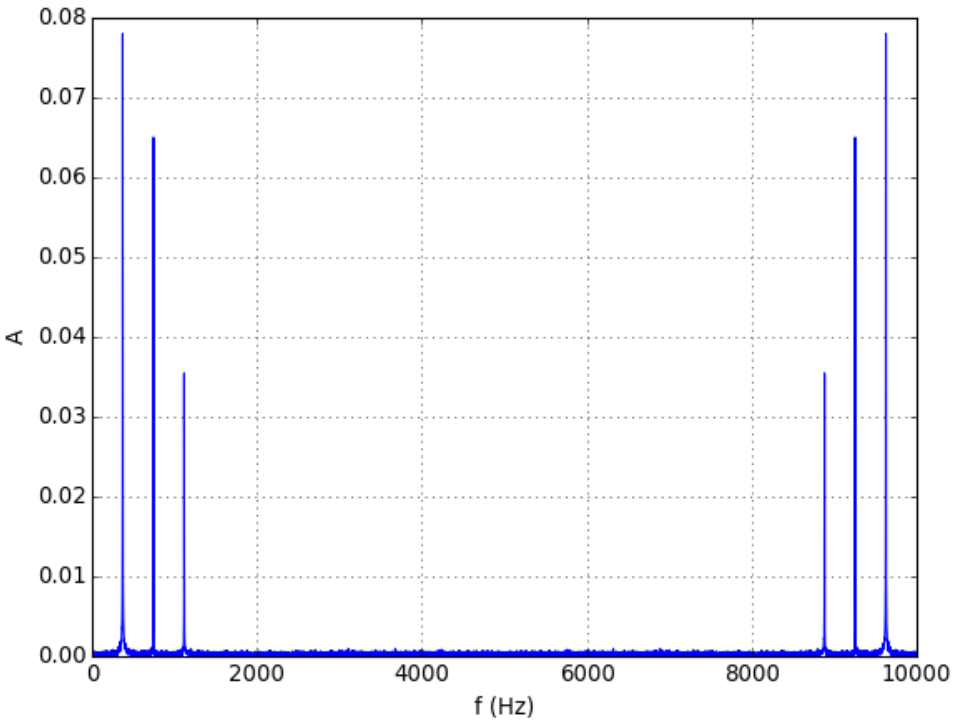
La réduction du bruit est une opération de filtrage très courante. Les capteurs délivrent en général des signaux bruités, et les circuits analogiques d'amplification introduisent aussi du bruit. Pour traiter le bruit sur le signal numérique, il faut échantillonner à une fréquence assez élevée. En effet, le bruit est en général caractérisé par une grande étendue spectrale, et la fréquence d'échantillonnage devra être assez élevée pour limiter le phénomène de repliement du spectre du bruit.

L'exemple ci-dessous est un signal périodique à 3 harmoniques généré avec un programme Pure Data qui permet d'ajouter un bruit d'amplitude variable. Sa fréquence est d'environ 372 Hz et la fréquence d'échantillonnage est de 10000 Hz.

```
[t0,u0] = numpy.loadtxt('signal-4.txt')
figure()
plot(t0,u0)
xlabel("t (s)")
ylabel("u (V)")
fenetre = 0.03
axis([0.1,0.1+fenetre,-1,1])
grid()
```

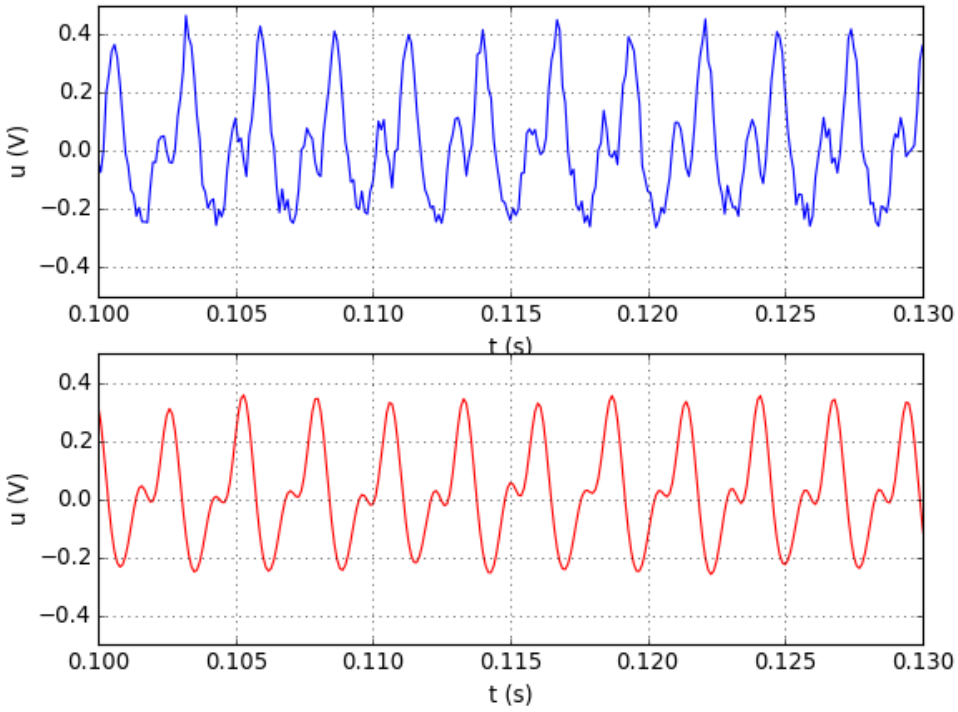



```
Ne = len(u0)
Te = t0[1]-t0[0]
spectre = numpy.absolute(numpy.fft.fft(u0))/Ne
frequences = numpy.arange(Ne,)*1.0/(Te*Ne)
figure()
plot(frequences,spectre)
xlabel("f (Hz)")
ylabel("A")
grid()
```



Pour réduire le bruit, nous utilisons le filtre RIF passe-bas défini plus haut, à 41 coefficients. Sa fréquence de coupure réduite est $f_c/f_e = 0,1$. Pour une fréquence d'échantillonnage de 10000 Hz, la fréquence de coupure réelle est 1000 Hz. Cela nous convient pour garder l'harmonique de rang 3 dans la bande passante. Voici le filtrage effectué avec la fonction `scipy.signal.lfilter` :

```
a1 = [1.0]
zi = scipy.signal.lfiltic(b1,a1,x=[0],y=[0])
[y0,zf] = scipy.signal.lfilter(b1,a1,u0,zi=zi)
figure()
subplot(211)
plot(t0,u0,'b')
xlabel("t (s)")
ylabel("u (V)")
grid()
fenetre = 0.03
axis([0.1,0.1+fenetre,-0.5,0.5])
subplot(212)
plot(t0,y0,'r')
xlabel("t (s)")
ylabel("u (V)")
grid()
axis([0.1,0.1+fenetre,-0.5,0.5])
```



On observe une légère ondulation de basse fréquence causée par le repliement des hautes fréquences du bruit. Le résultat est convenable mais pourrait être amélioré en augmentant la fréquence d'échantillonnage. Cependant, augmenter la fréquence d'échantillonnage nécessite d'augmenter parallèlement le nombre de coefficients du filtre, pour conserver la même sélectivité.

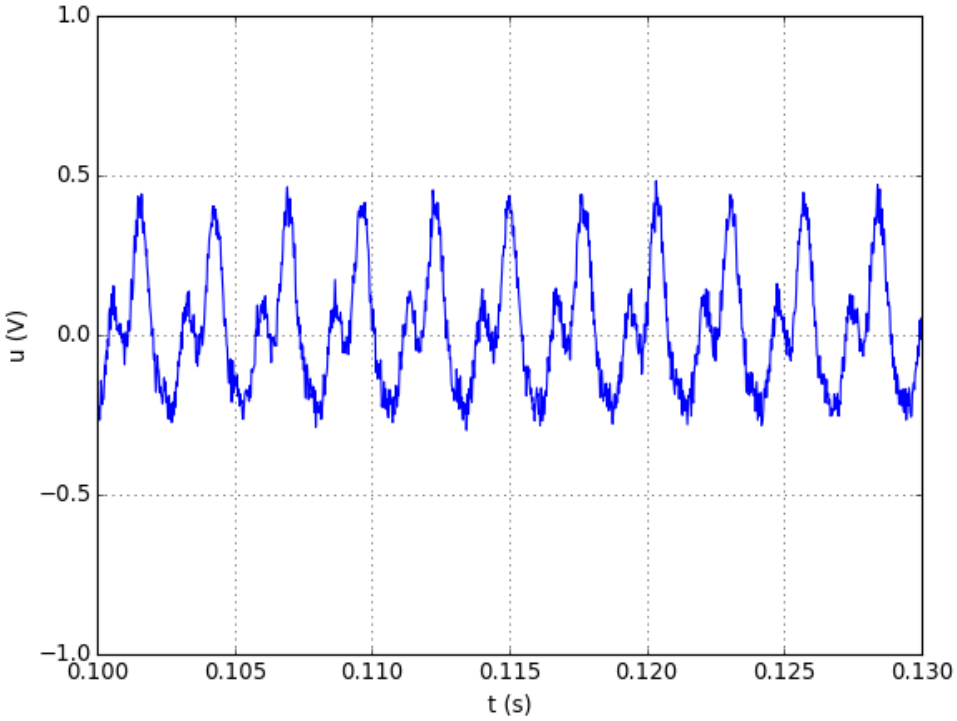
Sur-échantillonnage

Le sur-échantillonnage consiste à échantillonner le signal à une fréquence beaucoup plus élevée que la plus haute fréquence utile de son spectre. Par exemple, un signal audio dont la plus haute fréquence utile est à 20 kHz est échantillonné à 196 kHz. Cela permet de réduire l'effet du repliement des hautes fréquences présentes dans le signal, en particulier celles provenant du bruit. Nous pouvons appliquer cette technique au signal précédent, qui comporte beaucoup de bruit. Voici le script qui fait l'échantillonnage à 50 kHz, pour une durée de 1 seconde.

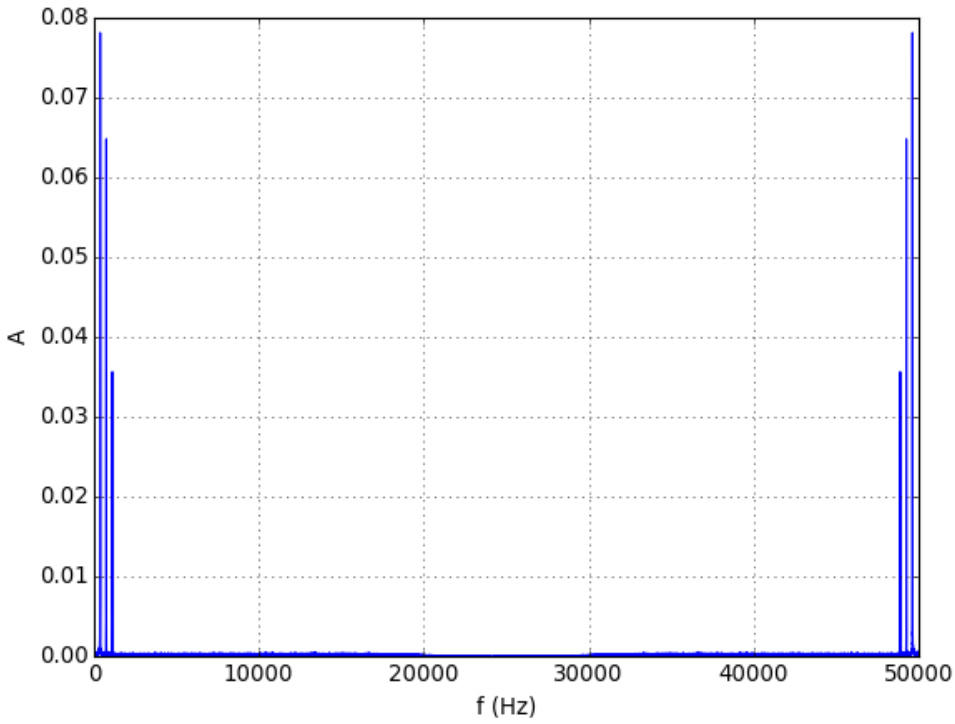
```
import pycanum.main as pycan
can = pycan.Sysam("SP5")
sys.config_entrees([0],[2])
fechant = 50000.0
techant = 1.0/fechant
nechant = 50000
sys.config_echant(techant*1e6,nechant)
sys.acquerir()
t = sys.temps()
u = sys.entrees()
sys.fermer()
```

Voici le signal et son spectre :

```
[t0,u0] = numpy.loadtxt('signal-5.txt')
figure()
plot(t0,u0)
xlabel("t (s)")
ylabel("u (V)")
fenetre = 0.03
axis([0.1,0.1+fenetre,-1,1])
grid()
```



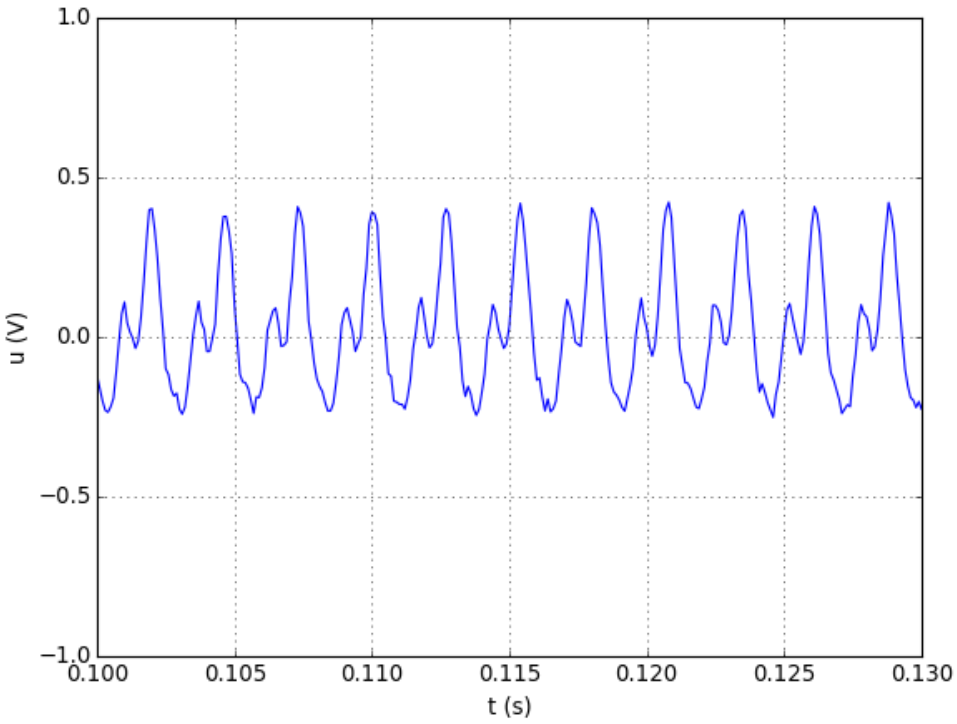

```
Ne = len(u0)
Te = t0[1]-t0[0]
spectre = numpy.absolute(numpy.fft.fft(u0))/Ne
frequences = numpy.arange(Ne,)*1.0/(Te*Ne)
figure()
plot(frequences,spectre)
xlabel("f (Hz)")
ylabel("A")
grid()
```



Le bruit semble avoir une bande spectrale limitée à 20 kHz, limite qui vient du logiciel de synthèse. Dans la réalité, la bande spectrale du bruit peut être beaucoup plus large.

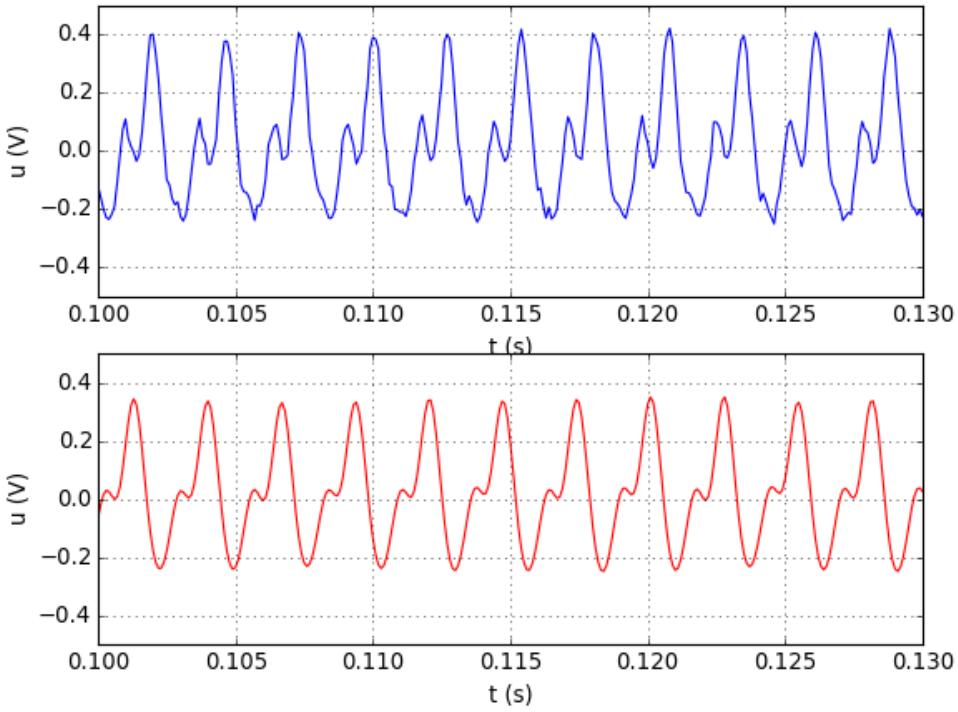
L'information utile du signal se trouve dans une bande spectrale réduite comparée à la bande $[0, 25]$ kHz. Considérons par exemple, que la bande utile est $[0, 5]$ kHz. Dans le domaine audio, une telle bande est largement suffisante pour transmettre la voix avec une très bonne qualité. Avant de réduire la fréquence d'échantillonnage d'un facteur 5, il faut effectuer un filtrage passe-bas à la fréquence de coupure de 5 kHz, ce qui peut se faire avec le filtre RIF passe-bas déjà utilisé plus haut. Voici donc le filtrage passe-bas suivi d'une réduction de la fréquence d'échantillonnage :

```
zi = scipy.signal.lfiltic(b1,a1,y=[0],x=[0])
[y0,zf] = scipy.signal.lfilter(b1,a1,u0,zi=zi)
y1 = y0[::5]
t1 = t0[::5]


figure()
plot(t1,y1)
xlabel("t (s)")
ylabel("u (V)")
fenetre = 0.03
axis([0.1,0.1+fenetre,-1,1])
grid()
```



La fréquence d'échantillonnage a été ramenée à 10 kHz. On peut réduire le bruit avec un filtre passe-bas comme plus-haut :

```
zi = scipy.signal.lfiltic(b1,a1,y=[0],x=[0]) # condition initiale
[y2,zf] = scipy.signal.lfilter(b1,a1,y1,zi=zi)
figure()
subplot(211)
plot(t1,y1,'b')
xlabel("t (s)")
ylabel("u (V)")
grid()
fenetre = 0.03
axis([0.1,0.1+fenetre,-0.5,0.5])
subplot(212)
plot(t1,y2,'r')
xlabel("t (s)")
ylabel("u (V)")
grid()
axis([0.1,0.1+fenetre,-0.5,0.5])
```



Le résultat est bien meilleur que dans le cas d'un échantillonnage primaire à 10 kHz, car nous avons échantillonné plus rapidement le bruit, ce qui a pour effet de réduire le repliement dans la bande utile des hautes fréquences du bruit. Le même résultat pourrait être obtenu avec un seul filtre de fréquence de coupure 1 kHz, mais comportant 80 coefficients.

4. Acquisition en mode permanent

4.a. Principe et fonctions

Dans le mode d'acquisition standard, les échantillons sont stockés dans la mémoire interne de la carte d'acquisition avant d'être transférés à l'ordinateur. Cela permet d'opérer à la fréquence d'échantillonnage maximale de 10 MHz mais limite le nombre de points acquis à 2^{18}. Dans de nombreuses applications, on peut se contenter d'une fréquence d'échantillonnage moins élevée tout en ayant besoin soit d'un nombre de points plus grand, soit d'un signal numérique en flot continu, sans interruption. Le mode d'acquisition permanent permet de faire cela, mais il n'est disponible que sur la carte Sysam SP5.

La configuration de l'échantillonnage en mode permament se fait avec l'appel suivant :

```
sys.config_echantillon_permanent(techant,nbpoints)
```

Cette fonction est similaire à la fonction `config_echantillon` déjà décrite. `techant` est la période d'échantillonnage en microsecondes. `nbpoints` est soit le nombre de points total à acquérir (pour chaque voie), soit la taille des paquets pour une acquisition sans fin en flot continu.

Pour une acquisition d'un grand nombre de points, par exemple 10^6, on peut utiliser l'appel suivant :

```
sys.acquerir_permanent()
```

qui retourne lorsque l'acquisition est terminée. Si l'acquisition est longue, il est préférable de la lancer sur une *thread* d'exécution parallèle, de la manière suivante :

```
sys.lancer_permanent(repetition=0)
```

Cette fonction retourne dès que l'acquisition est lancée, ce qui permet d'accomplir pendant l'acquisition des tâches de traitement ou de tracé du signal. Si `repetition=1`, l'acquisition est répétée dans fin, ce qui permet de faire un traitement en flot continu.

Il peut être utile de faire un filtrage pendant l'acquisition. Le filtrage se configure par :

```
sys.config_filtre(a,b)
```

où `a,b` sont les listes de coefficients a_k et b_k.

Lorsque l'acquisition est terminée, les tensions non filtrées et filtrées sont récupérées avec les fonctions suivantes :

```
x = sys.entrees()
y = sys.entrees_filtrees(reduction=1)
```

L'argument optionnel `reduction` est un entier qui permet de faire une réduction de la fréquence d'échantillonnage, effectuée après le filtrage.

Pendant une acquisition lancée sur thread parallèle, un paquet de données est lu avec la fonction suivante :

```
A = sys.paquet(premier,reduction=1)
```

L'argument `premier` est l'indice du premier point à lire, ou -1 si l'on veut récupérer un paquet complet contenant le nombre de points défini avec la fonction `config_echantillon_permanent`. Si N est le nombre de voies acquises, le tableau `A` contient les instants de ces N voies sur les N premières lignes, les tensions sur les N lignes suivantes, et les tensions filtrées sur les N dernières lignes.

4.b. Exemple 1 : spectrogramme audio

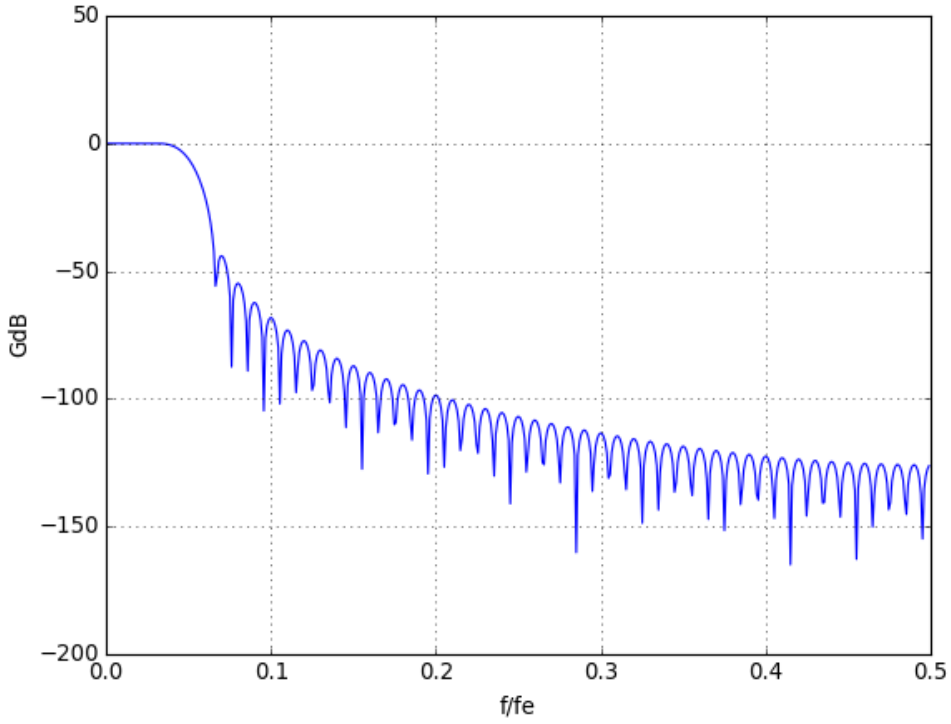
Pour faire l'analyse spectrale d'un signal non périodique, comme un signal audio, on est amené à utiliser une fenêtre dont la durée T est l'inverse de la résolution spectrale souhaitée. Par exemple avec $T = 10\,ms$ on obtient une résolution fréquentielle de 100 Hz. La fenêtre est déplacée sur l'axe du temps, en général par saut, ce qui donne une suite de spectres que l'on représente sous forme d'une image.

On considère un signal audio échantillonné à 40 kHz d'une durée de plusieurs dizaines de secondes. Le nombre d'échantillons à acquérir dépasse largement la limite de 2^{18} de la carte Sysam SP5, ce qui justifie l'emploi du mode permament.

Dans le cas d'un son musical ou de la voix, on peut se limiter à une analyse spectrale dans la bande $[0, 2]$ kHz. Nous allons donc procéder à un filtrage passe-bas de fréquence de coupure 2 kHz avant de réduire la fréquence d'échantillonnage d'un facteur 10. Le spectrogramme sera calculé sur ce signal réduit.

Voici la conception d'un filtre passe-bas RIF à 101 coefficients :

```
fechant = 40000.0
fc = 2000.0
b = scipy.signal.firwin(numtaps=101,cutoff=[fc/fechant],window='hann',nyq=0.5)
w,h=scipy.signal.freqz(b,[1.0])
figure()
plot(w/(2*numpy.pi),20*numpy.log10(numpy.absolute(h)))
xlabel("f/fe")
ylabel("GdB")
grid()
```

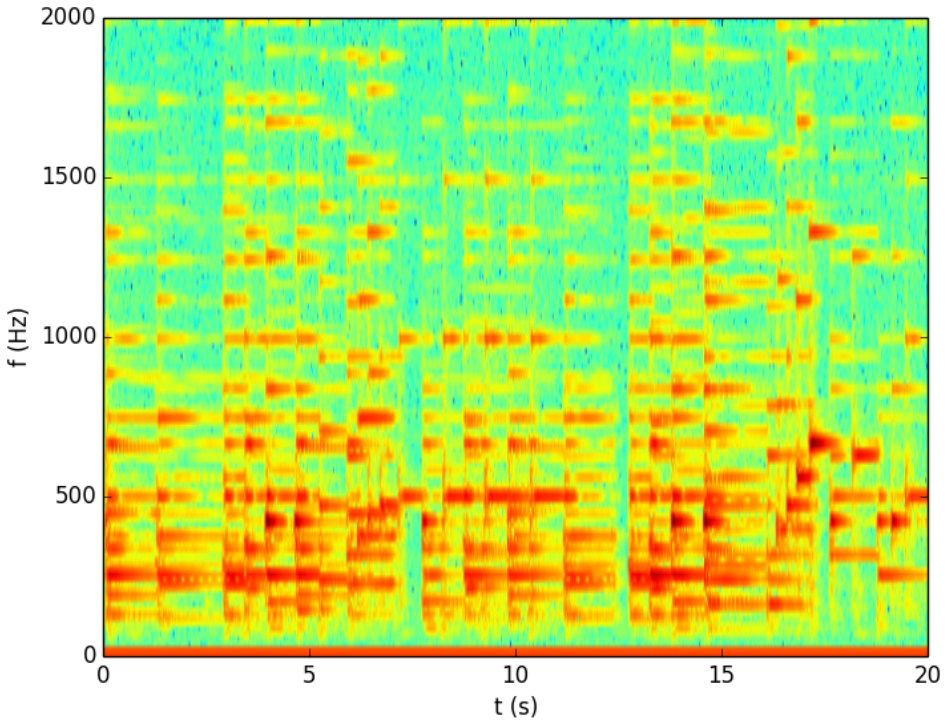


Voici le script faisant l'acquisition avec le filtrage. Les échantillons sont lus avec un facteur de réduction de 10, qui amène la fréquence d'échantillonnage finale à 4 kHz. La fonction

`matplotlib.pyplot.specgram` est utilisée pour calculer et tracer le spectrogramme. La fenêtre d'analyse a une largeur de 256 échantillons, soit une durée de 64 ms, qui donnera une résolution fréquentielle de 16 Hz environ. Le déplacement de la fenêtre se fait par sauts d'une durée égale à la moitié de celle de la fenêtre.

```
import pycanum.main as pycan
import numpy
from matplotlib.pyplot import *
import scipy.signal

sys = pycan.Sysam("SP5")
sys.config_entrees([0],[2.0])
fechant = 40000.0
techant = 1.0/fechant
duree = 20.0
ne = int(duree/techant)
sys.config_echantillon_permanent(techant*1e6,ne)
sys.config_filtre([1.0],b)
sys.acquerir_permanent()
x = sys.entrees_filtrees(reduction=10)
sys.fermer()
figure()
specgram(x[0],NFFT=256,Fs=4000,noverlap=128)
xlabel("t (s)")
ylabel("f (Hz)")
```

Voici le résultat pour un fichier son mp4 lu sur l'ordinateur, la sortie audio étant reliée à l'entrée EA0 de la carte d'acquisition. Il s'agit de l'enregistrement d'une musique jouée au piano, comportant essentiellement une succession d'accords.



On obtient ainsi une analyse temps-fréquence. La résolution temporelle est ici suffisante pour bien discerner les accords. Si l'on souhaite augmenter la résolution fréquentielle, il faudra augmenter la largeur de la fenêtre d'analyse, ce qui aura aussi pour effet de réduire la résolution temporelle. Il faut donc trouver un compromis entre ces deux résolutions, en fonction du type de son analysé.

4.c. Exemple 2 : filtrage et tracé en temps réel

Dans certains cas, il peut être utile de tracer les signaux en temps réel, par exemple pour étudier le comportement transitoire d'un filtre, lorsqu'une caractéristique du signal (amplitude ou fréquence) change rapidement. Lorsqu'on fait une expérience avec un ou plusieurs capteurs, il peut être utile d'effectuer un filtrage tout en suivant le résultat en temps réel.

On utilise pour cela le module `matplotlib.animate`, qui permet de rafraichir périodiquement des courbes tracées dans une fenêtre graphique. Le script présenté ci-dessous est prévu pour faire fonctionner un capteur à une fréquence d'échantillonnage de 1 kHz. Il effectue les opérations suivantes :

- ▷ Sur-échantillonnage à 1 kHz.
- ▷ Filtrage passe-bas anti-repliement puis réduction de la fréquence d'échantillonnage à 100 Hz.

La variable `duree_totale` fixe la durée totale de l'acquisition (sans limite). La variable `duree` fixe la durée de la fenêtre visualisée. La variable `interval` fixe la période de rafraichissement de cette fenêtre.

L'exemple ci-dessus montre qu'un filtre RIF à 101 coefficients convient pour une réduction d'un facteur 10.

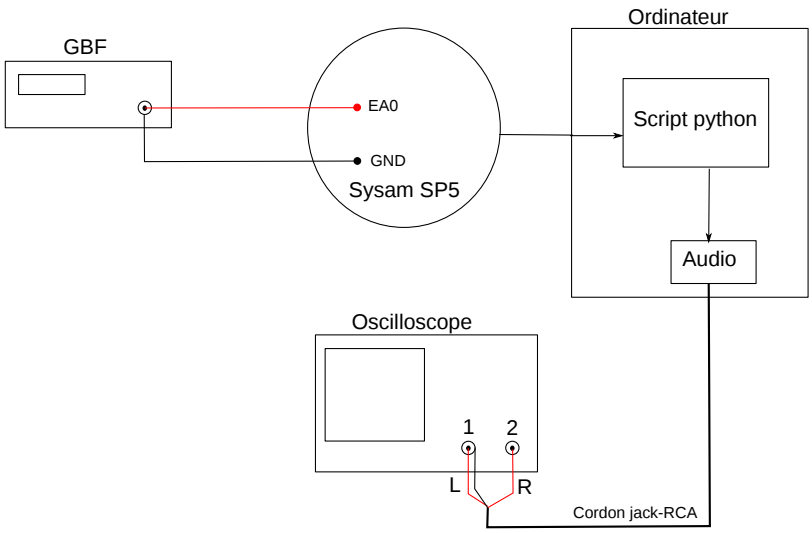
acquisitionPermanenteTrace.py

```
# -*- coding: utf-8 -*-

import pycanum.main as pycan
import math
from matplotlib.pyplot import *
import matplotlib.animation as animation
import numpy
import scipy.signal

sys=pycan.Sysam("SP5")
Umax = 2
sys.config_entrees([0],[Umax])
fe=1000.0 # fréquence de la numérisation
te=1.0/fe
duree_totale = 600.0
N = int(duree_totale*fe)
sys.config_echantillon_permanent(te*1e6,N)
reduction = 10 # réduction de la fréquence d'échantillonnage
fe_r = fe/reduction
te_r = te*reduction
duree = 10.0 # durée des blocs
longueur_bloc = int(duree/te_r) # taille des blocs traités
nombre_blocs = int(N*te/duree)
nombre_echant = nombre_blocs*longueur_bloc

#filtre passe-bas anti-repliement
fc = fe/reduction/2*0.8 # fréquence de coupure
b = scipy.signal.firwin(numtaps=101,cutoff=[fc/fe],nyq=0.5,window='hann')
sys.config_filtre([1],b)

sys.lancer_permanent()
n_tot = 0 # nombre d'échantillons acquis

fig,ax = subplots()
t = numpy.arange(longueur_bloc)*te_r
u = numpy.zeros(longueur_bloc)

line0, = ax.plot(t,u)
ax.grid()
ax.axis([0,duree,-Umax,Umax])
ax.set_xlabel("t (s)")
u = numpy.array([],dtype=numpy.float32)
t = numpy.array([],dtype=numpy.float32)

def animate(i):
    global ax,sys,t,u,line0,n_tot,longueur_bloc
    data = sys.paquet_filtrees(n_tot,reduction)
    t0 = data[0]
```

```
    u0 = data[1]
    n_tot += u0.size
    t = numpy.append(t,t0)
    u = numpy.append(u,u0)
    if n_tot >= nombre_echant:
        print(u"Acquisition terminée")
    i2 = n_tot-1
    if n_tot == 0:
        return
    if n_tot > longueur_bloc:
        i1 = i2-longueur_bloc
    else:
        i1 = 0
    line0.set_data(t[i1:i2],u[i1:i2])
    ax.axis([t[i1],t[i2],-Umax,Umax])

interval = 1 # intervalle de rafraichissement en s
ani = animation.FuncAnimation(fig,animate,frames=int(duree_totale/interval),repeat=Fa
show(block=True)
numpy.savetxt("data.txt",[t,u])
sys.fermer()
```

4.d. Exemple 3 : filtrage audio en temps réel

L'acquisition en mode permanent permet de faire le filtrage d'un signal audio en temps réel. Le module `pycanum` comporte une fonction pour rediriger des signaux vers la sortie audio. Afin d'étudier le filtre, nous faisons la numérisation d'une sinusoïde délivrée par un générateur de fonctions. Le filtrage est effectué en temps réel. Parallèlement, le signal d'origine et le signal filtré sont envoyés sur la sortie audio pour être observés à l'oscilloscope.



La configuration de l'échantillonnage se fait avec :

```
sys.config_echantillon_permanent(techant,nbpoints)
```

et le lancement d'une acquisition en flot continu se fait par :

```
sys.lancer_permanent(repetition=1)
```

Dans ce cas les données sont traitées par paquets et le nombre de points `nbpoints` défini ci-dessus est le nombre d'échantillons contenus dans un paquet. La commande suivante permet de récupérer le dernier paquet :

```
data = sys.paquet(-1)
```

Si aucun nouveau paquet n'est disponible, le tableau renvoyé est vide. Un tampon comportant 16 paquets permet de faire face à une diminution transitoire du rythme de lecture.

Pour effectuer le filtrage, on peut soit faire appel au filtre intégré du module `pycanum`, qui se programme avec la fonction `config_filtre`, soit utiliser la fonction `scipy.signal.lfilter`, comme montré ci-dessous.

Pour utiliser la sortie audio stéréo, il faut tout d'abord définir deux tampons, qui sont des objets de la classe `RingBuffer`. Par exemple :

```
tampon_x = pycan.RingBuffer(6,N)
```

définit un tampon comportant $2^6 = 64$ paquets de N échantillons.

Dans le cas présent, on définit un tampon pour contenir le signal non filtré et un second pour le signal filtré. Le déclenchement du flux audio de sortie se fait avec :

```
pycan.output_stream_start(fe,tampon_x,tampon_y)
```

où `fe` est la fréquence d'échantillonnage, qui doit bien sûr être égale à celle de la numérisation. Pour écrire un bloc dans un tampon, il faut exécuter :

```
tampon_x.write(x)
```

Voici le script complet, qui effectue un filtrage récursif biquadratique (filtre passe-bas) :

filtrageBIQUADSortieAudio.py

```
import pycanum.main as pycan
import scipy.signal
import time

fe=10000.0
fc = 1000
b,a = scipy.signal.iirfilter(N=2,Wn=[fc/fe*2],btype="lowpass",ftype="butter")
zi = scipy.signal.lfiltic(b,a,y=[0,0],x=[0,0]) # condition initiale

sys=pycan.Sysam("SP5")
Umax = 2.0
sys.config_entrees([0],[Umax])
te=1.0/fe
N = 200 # taille des paquets
tampon_x = pycan.RingBuffer(6,N)
tampon_y = pycan.RingBuffer(6,N)
duree = N*te
sys.config_echantillon_permanent(te*1e6,N)
sys.lancer_permanent(repetition=1) # lancement d'une acquisition sans fin
pycan.output_stream_start(fe,tampon_x,tampon_y)

k = 0
while k<3000:
    data = sys.paquet(-1,reduction=1)
    if data.size!=0:
        x = data[1]
        [y,zi] = scipy.signal.lfilter(b,a,x,zi=zi)
        tampon_x.write(x)
        tampon_y.write(y)
        k+=1
    time.sleep(duree*0.2)

pycan.output_stream_stop(0)
sys.stopper_acquisition()
time.sleep(1)
sys.fermer()
```

Les signaux sont traités par paquets de N échantillons (de 100 à 1000), ce qui conduit à un délai entre l'entrée et la sortie égal environ à N multiplié par la période d'échantillonnage. Il ne s'agit donc pas d'un véritable filtrage en temps réel (de type DSP), dont le délai est égal à la période d'échantillonnage. Cependant, les deux signaux visualisés à l'oscilloscope sont synchrones et leur déphasage est exactement celui qu'on aurait avec un véritable filtre temps réel.

5. Annexe : synthèse numérique de signaux

5.a. Utilisation du logiciel Pure Data

Pure Data (puredata.info) est un logiciel de programmation visuelle spécialisé dans le traitement du son. Sa fonction première est la synthèse de sons pour la composition musicale, mais il peut aussi traiter des images et des vidéos.

Pour la prise en main du logiciel et l'apprentissage de la synthèse audio, nous conseillons les liens suivants :

- ▷ Manuel FLOSS Pure Data
- ▷ Programming Electronic Music in Pd

Le programme syntheseHarmonique.pd permet de faire la synthèse de signaux périodiques avec 3 harmoniques, et avec la possibilité d'ajouter du bruit.

5.b. Synthèse par table

5.c. Sortie audio

Le module `pycanum` comporte des fonctions pour générer un signal périodique directement sur la sortie audio de l'ordinateur.