Exemples de filtres RIF

1. Introduction

Ce document présente des exemples de calcul de filtres (1D) à réponse inpulsionnelle finie.

La réponse impulsionnelle du filtre est calculée par la méthode de la série de Fourier. Cette méthode est expliquée dans Filtres à réponse impulsionnelle finie. Elle est ici rappelée sans justification.

2. Réponse impulsionnelle

2.a. Méthode de calcul

La réponse impulsionnelle est calculée à partir d'une fonction gain $G(x)$, où x est le rapport de la fréquence sur la fréquence d'échantillonnage $x = f/f_e$. Il s'agit d'une fonction paire qui doit être définie sur l'intervalle $[0, 1/2]$. Le gain est généralement défini pour donner un filtre idéal : sa valeur est 1 dans la bande passante, 0 dans la bande atténuée.

La fonction obtenue sur l'intervalle $[-1/2, 1/2]$ est prolongée par périodicité. Ses coefficients de Fourier g_k définissent une réponse impulsionnelle infinie. Pour la rendre finie, on la tronque au rang P. Il reste alors $2P + 1$ coefficients (k variant de $-P$ à P) qui définissent la réponse impulsionnelle finie :

$$h_k = g_{k-P} \quad (0 \leq k \leq 2P) \tag{1}$$

Le fait de tronquer la réponse impulsionnelle rend la réponse fréquentielle différente de celle définie au départ par la fonction $G(x)$. Il faut donc étudier le filtre RIF obtenu en calculant sa réponse fréquentielle, comme expliqué dans Introduction aux filtres numériques. Plus l'indice P est élevé, plus la réponse fréquentielle sera proche de la réponse idéale.

2.b. Filtres standard

On donne ci-dessous les coefficients de Fourier pour les filtres standard, définis par un gain $G(x)$ idéal ([1]).

Le filtre passe-bas est défini à partir d'un gain $G(x)$ valant 1 sur l'intervalle $[0, a]$, 0 ailleurs. Les coefficients de Fourier sont :

$$g_k = 2a \operatorname{sinc}(k2a) \tag{2}$$

où la fonction sinus cardinale est définie par :

$$\operatorname{sinc}(x) = \frac{\sin(\pi x)}{\pi x} \tag{3}$$

Le filtre passe-haut est défini par $G(x) = 1$ sur l'intervalle $[a, 1/2]$, $G(x) = 0$ ailleurs. Les coefficients de Fourier sont :

$$\begin{aligned} g_0 &= 1 - 2a \\ g_k &= -2a \operatorname{sinc}(k2a) \end{aligned} \tag{4}$$

Le filtre passe-bande est défini par $G(x) = 1$ sur l'intervalle $[a, b]$. Ses coefficients de Fourier sont :

$$\begin{array}{l} g_0 = 2(b-a) \\ g_k = \frac{\sin(k2b)-\sin(2ka)}{k\pi} \end{array} \tag{5}$$

Le filtre coupe bande est définie par $G(x) = 0$ sur l'intervalle $[a, b]$, $G(x) = 0$ ailleurs. Ses coefficients de Fourier sont :

$$\begin{array}{l} g_0 = 2(a-b)+1 \\ g_k = \frac{\sin(k2a)-\sin(2kb)}{k\pi} \end{array} \tag{6}$$

Un filtre de quadrature permet de déphaser un signal sinusoïdal de $\pi/2$. Il peut être obtenu à partir d'une réponse fréquentielle $G(x)$ impaire purement imaginaire. Ses coefficients de Fourier sont ([1]) :

$$\begin{array}{ll} g_k = 0 & \forall k \text{ pair} \\ g_k = \frac{2}{k\pi} & \forall k \text{ impair} \end{array} \tag{7}$$

2.c. Fenêtrage progressif

La troncature de la réponse impulsionnelle au rang P (fenêtrage rectangulaire) introduit des ondulations dans la bande passante de la réponse fréquentielle (aussi dans la bande atténuée). Pour réduire ces ondulations, on utilise un fenêtrage progressif. Les plus utilisés sont les fenêtrages de Hamming, Hann et Blackman. Par exemple, le fenêtrage de Hann consiste à multiplier les coefficients de Fourier par les coefficients suivants (pour $k \leq P$) :

$$w_k = \frac{1}{2}\left[1 + \cos\left(\frac{2\pi\, k}{N}\right)\right] \tag{8}$$

où $N = 2P + 1$.

Pour obtenir les fonctions de fenêtrage, on utilisera la fonction python `scipy.signal.get_window`.

3. Fonctions de calcul

```
import numpy
import math
import cmath
import scipy.signal
from matplotlib.pyplot import *
```

La fonction suivante fournit la réponse impulsionnelle du filtre. On doit fournir les coefficients de Fourier, l'indice P de troncature et le type de fenêtre à appliquer.

```
def filtreRIFgeneral(g,P,fenetre):
    N=2*P+1 # ordre du filtre
    liste_k = numpy.arange(start=-P,stop=P+1)
    n = liste_k.size
    h = numpy.zeros(n)
```

```
    for k in range(n):
        h[k] = g(liste_k[k])
    if fenetre!="rect":
        h = h*scipy.signal.get_window(fenetre,N)
    return h
```

La fonction suivante fournit la réponse impulsionnelle de filtres passe-bas (PBas), passe-haut (PHaut), passe-bande (PBande) et coupe-bande (CBande), construits à partir d'une fonction $G(x)$ dont le gain vaut 1 dans la bande passante et 0 dans la bande atténuée. Pour les filtres PBas et PHaut, il faut fournir la fréquence de coupure (divisée par f_e) `a`. Pour les filtres PBande et CBande, il faut fournir en plus la deuxième fréquence de coupure `b`. Un filtre de quadrature (Quad) est aussi calculé.

```
def filtreRIF(type,a,b,P,fenetre):
    if type=='PBas':
        def g(k):
            return 2*a*numpy.sinc(k*2*a)
    elif type=='Phaut':
        def g(k):
            if k==0:
                return 1-2*b
            else:
                return -2*a*numpy.sinc(k*2*a)
    elif type=='PBande':
        def g(k):
            if k==0:
                return 2*(b-a)
            else:
                return (numpy.sin(2*math.pi*k*b)-numpy.sin(2*math.pi*k*a))/(k*math.pi
    elif type=='CBande':
        def g(k):
            if k==0:
                return 2*(a-b)+1
            else:
                return (numpy.sin(2*math.pi*k*a)-numpy.sin(2*math.pi*k*b))/(k*math.pi
    elif type=='Quad':
        def g(k):
            if k%2==0:
                return 0.0
            else:
                return 2.0/(k*math.pi)
    else:
        def g(k):
            return 1.0
    return filtreRIFgeneral(g,P,fenetre)
```

La fonction suivante renvoit la réponse fréquentielle (gain et phase) d'un filtre RIF.

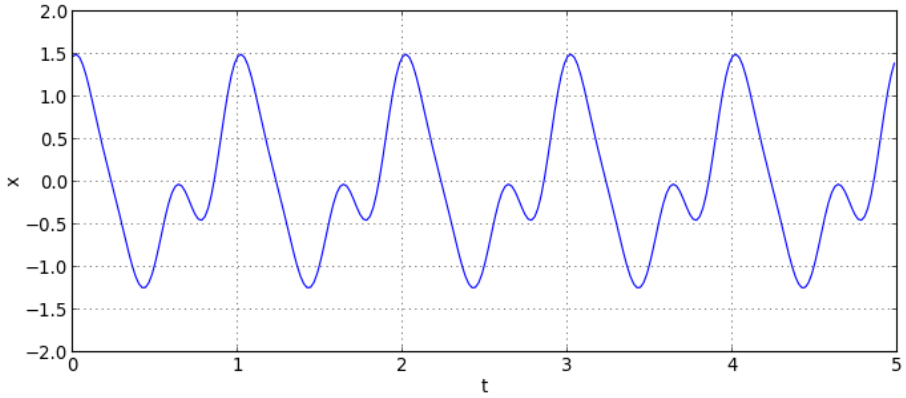
La fonction `numpy.unwrap` permet d'éviter les sauts de phase lorsque celle-ci effectue plusieurs tours (ce qui est très courant dans les filtres RIF).

```
def reponseFreq(h):
    N = h.size
    def Hf(f):
        s = 0.0
        for k in range(N):
            s += h[k]*numpy.exp(-1j*2*math.pi*k*f)
        return s
    f = numpy.arange(start=0.0,stop=0.5,step=0.0001)
    hf = Hf(f)
    g = numpy.absolute(hf)
    phi = numpy.unwrap(numpy.angle(hf))
    return [f,g,phi]
```

3.a. Filtre passe-bas

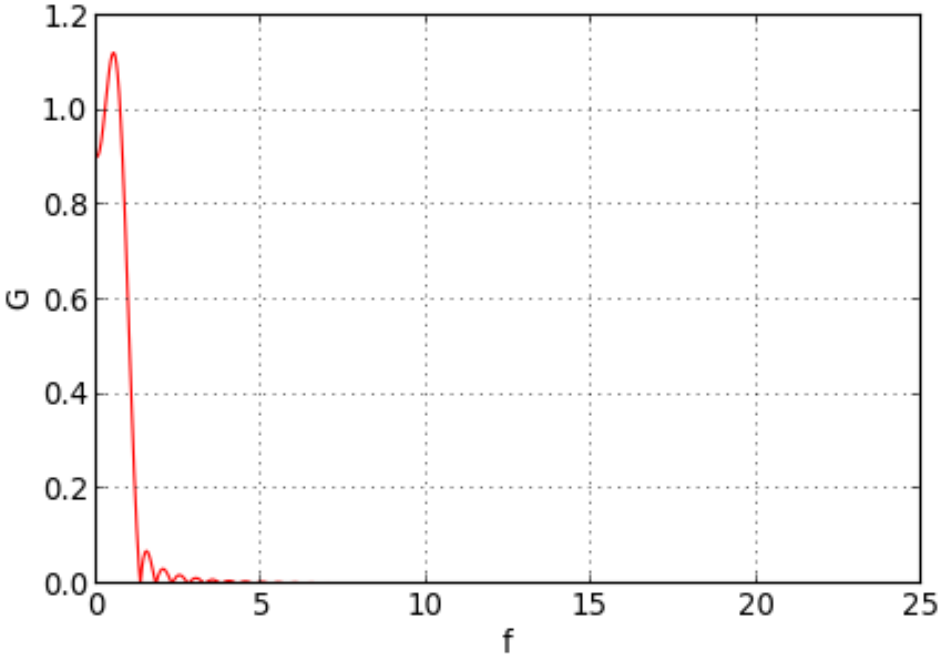
Pour tester le filtre, on échantillonne une fonction trigonométrique de période 1. La fréquence d'échantillonnage est de 50 points par période.

```
def signal(t):
    return math.cos(2*math.pi*t)+0.5*math.cos(2*2*math.pi*t-1.0)\
    +0.25*math.cos(3*2*math.pi*t+0.6)
fe = 50.0
te=1.0/fe
t = numpy.arange(start=0.0,stop=5.0,step=te)
n = t.size
x = numpy.zeros(n)
for k in range(n):
    x[k] = signal(te*k)
figure(figsize=(10,4))
plot(t,x,'b')
xlabel('t')
ylabel('x')
axis([0,5,-2,2])
grid()
```



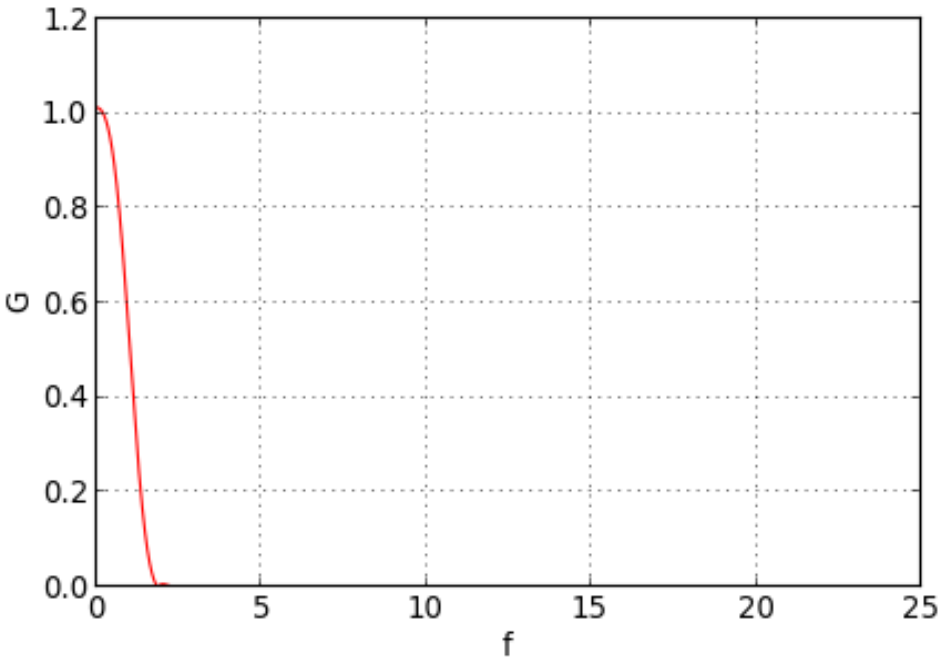
Voyons tout d'abord la réponse fréquentielle du filtre défini avec une troncature sans fenêtrage progressif :

```
fc = 1.0 # frequence de coupure
a = fc/fe
P = 50
h = filtreRIF("PBas",a,None,P,"rect")
[f,g,phi] = reponseFreq(h)
figure(figsize=(6,4))
plot(fe*f,g,'r')
xlabel('f')
ylabel('G')
grid()
```



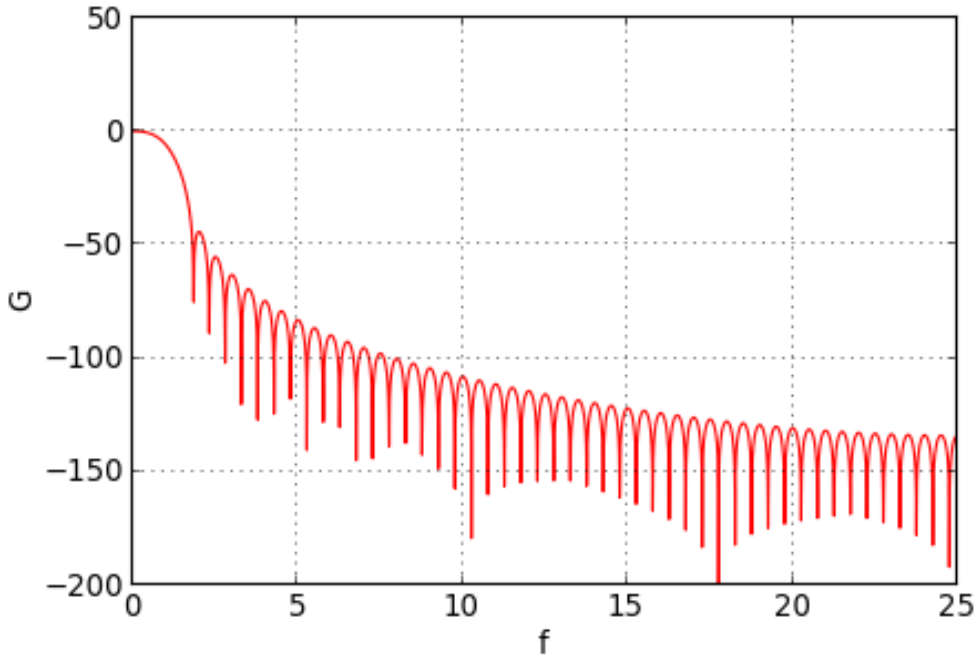
Il y a des ondulations très marquées dans la bande passante et dans la bande atténuée. Pour les réduire, on utilise un fenêtrage de Hann :

```
h = filtreRIF("PBas",a,None,P,"hann")
[f,g,phi] = reponseFreq(h)
figure(figsize=(6,4))
plot(fe*f,g,'r')
xlabel('f')
ylabel('G')
grid()
```



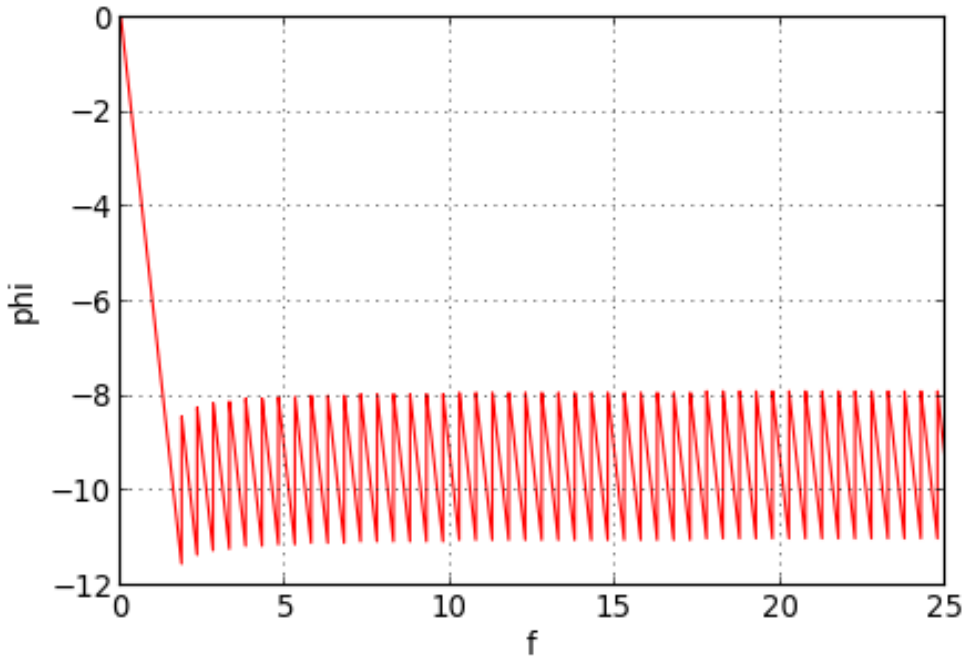
Pour mieux voir le gain dans la bande atténuée, on trace le gain en décibel :

```
figure(figsize=(6,4))
plot(fe*f,20*numpy.log10(g),'r')
xlabel('f')
ylabel('G')
grid()
```



Il y a toujours des ondulations dans la bande atténuée mais elles seront peu gênantes car situées en dessous de −50 décibels. On trace aussi le déphasage :

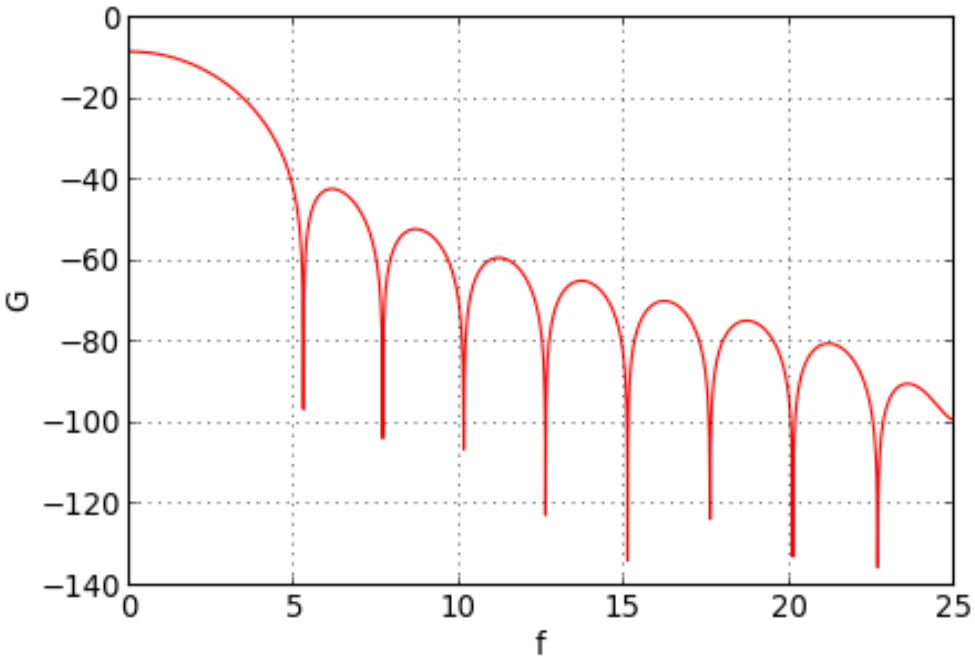
```
figure(figsize=(6,4))
plot(fe*f,phi,'r')
xlabel('f')
ylabel('phi')
grid()
```



On vérifie la linéarité de la phase dans la bande passante, qui permet de conserver la forme des signaux dont les harmoniques sont dans la bande passante.

Une question importante est celle du choix de l'indice de troncature P. L'ordre du filtre est $N = 2P + 1$. Plus P est élevé, plus la réponse est proche de la réponse idéale définie au départ par la fonction $G(x)$. Nous avons choisi ci-dessus un indice $P = 50$, ce qui donne un bon résultat. Voyons ce qu'il arrive si l'on abaisse l'indice P :

```
P2=10
h2 = filtreRIF("PBas",a,None,P2,"hann")
[f,g2,phi2] = reponseFreq(h2)
figure(figsize=(6,4))
plot(fe*f,20*numpy.log10(g2),'r')
xlabel('f')
ylabel('G')
grid()
```



Non seulement le filtre est beaucoup moins sélectif mais le gain dans la bande passante est inférieur à 1. Pour conserver une réponse correcte, il faut donc un indice P assez grand. Une autre solution consiste à abaisser la fréquence d'échantillonnage. En effet, cela revient à augmenter la fréquence de coupure réduite a et donc à tronquer le sinus cardinal plus loin. D'une manière générale, la troncature doit conserver au moins les premiers lobes du sinus cardinal. Par exemple, pour conserver au moins le maximum principal et le premier maximum secondaire, il faut :

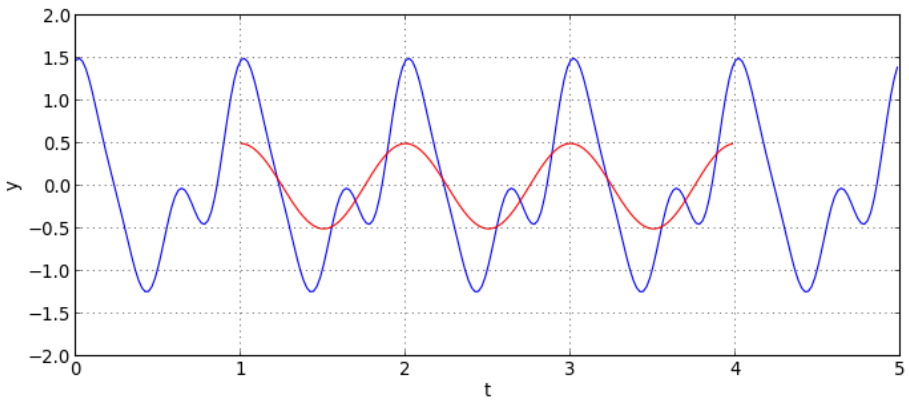
$$2Pa > 2 \tag{9}$$

Dans l'exemple ci-dessus $a = 1/50$; la condition est $P > 50$. On peut donc retenir comme règle qualitative que l'indice P doit être supérieur à l'inverse de a. Pour voir l'influence de P sur la réponse fréquentielle, voir la simulation Analyse spectrale et filtrage numérique.

Dans un filtre fonctionnant en temps réel, le choix d'un indice P pas trop élevé peut être un impératif de fonctionnement, car le temps de calcul de la convolution est proportionnel à P. Lorsque la fréquence de coupure réelle doit être abaissée, on devra aussi abaisser la fréquence d'échantillonnage pour que a ne soit pas trop faible.

Pour effectuer le produit de convolution entre le signal échantillonné et la réponse impulsionnelle, on peut utiliser la fonction `scipy.signal.convolve`. L'utilisation de l'option `mode='valid'` restreint le calcul de la convolution aux points valides. Ainsi le premier point de la convolution est calculé à partir des $2P+1$ premiers points de `x` et le dernier point est calculé à partir des $2P1+1$ derniers points de `x`.

```
y = scipy.signal.convolve(x,h,mode='valid')
ny = y.size
ty = numpy.zeros(ny)
for k in range(ny):
    ty[k] = P*te+te*k
figure(figsize=(10,4))
plot(t,x,'b')
plot(ty,y,'r')
xlabel('t')
ylabel('y')
axis([0,5,-2,2])
grid()
```

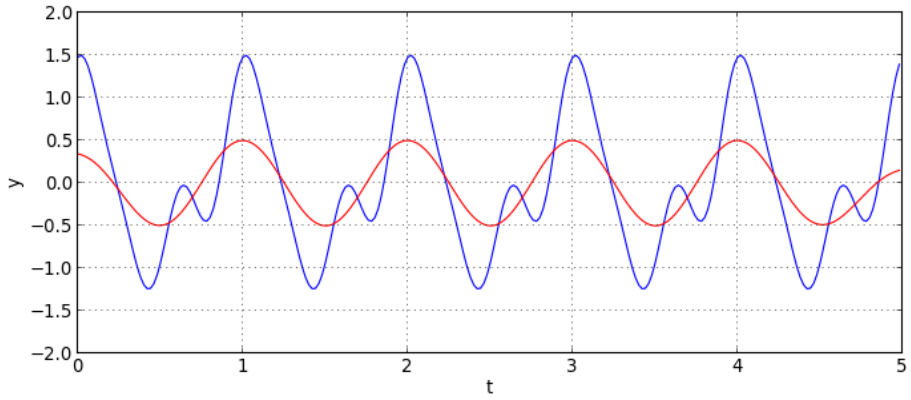


Le signal de sortie a été décalé pour être synchrone avec le signal d'entrée. Le retard $\tau = PT_e$ se manifeste alors par un délai sans sortie au début de l'échantillon et à la fin. Si ce délai est sans importance pour les signaux périodiques, il peut se traduire par une perte d'information dans le cas des signaux non périodiques. En pratique, il faut adopter un compromis entre la qualité du filtrage (qui augmente avec P) et un délai le plus court possible.

On peut aussi utiliser l'option `mode='same'` qui permet de calculer aussi la convolution pour les P premiers et P derniers points, en utilisant qu'une partie de la réponse impulsionnelle. Dans ce cas, le nombre de points n'est pas modifié par la convolution et on peut garder l'échelle de temps initiale.

```
y = scipy.signal.convolve(x,h,mode='same')
figure(figsize=(10,4))
```

```
plot(t,x,'b')
plot(t,y,'r')
xlabel('t')
ylabel('y')
axis([0,5,-2,2])
grid()
```

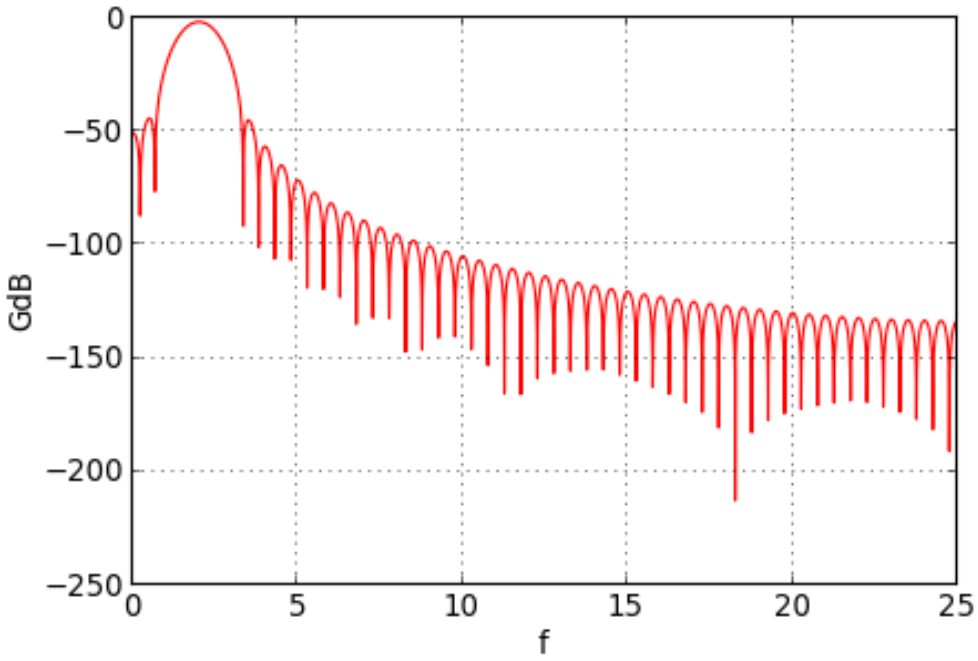


Bien sûr, le filtrage n'est pas correct pour les P premiers et P derniers points, mais il est parfois préférable de ne pas perdre de points dans le filtrage.

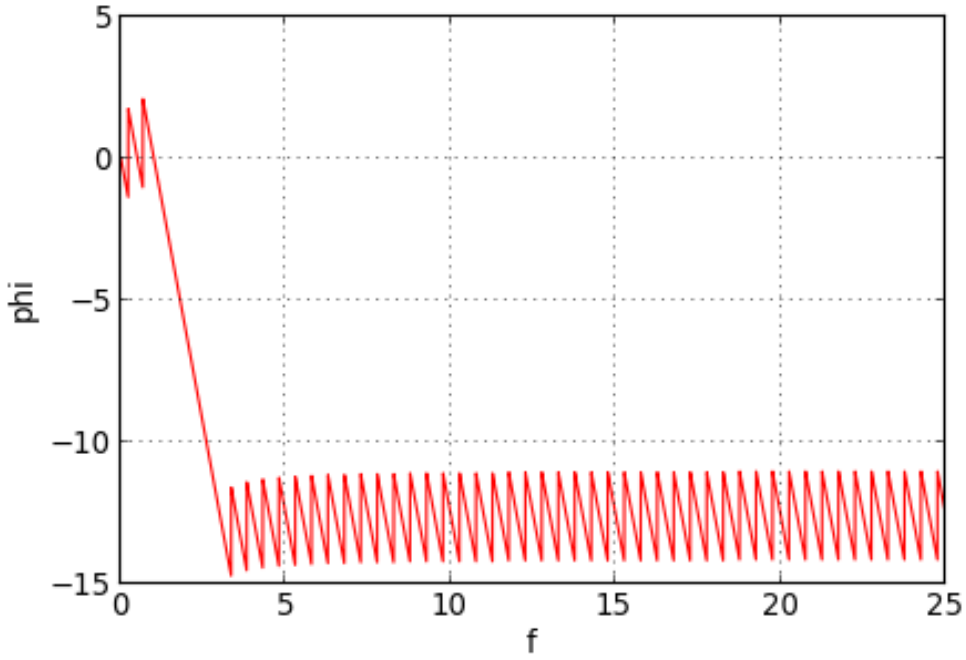
3.b. Filtre passe-bande

Définition du filtre, calcul de la réponse impulsionnelle et tracé du gain en décibel (en fonction de la fréquence réelle) :

```
fca = 1.5 # frequence de coupure basse
fcb = 2.5 # coupure haute
a = fca/fe
b = fcb/fe
P = 50
h = filtreRIF("PBande",a,b,P,"hann")
[f,g,phi] = reponseFreq(h)
figure(figsize=(6,4))
plot(fe*f,20*numpy.log10(g),'r')
xlabel('f')
ylabel('GdB')
grid()
```



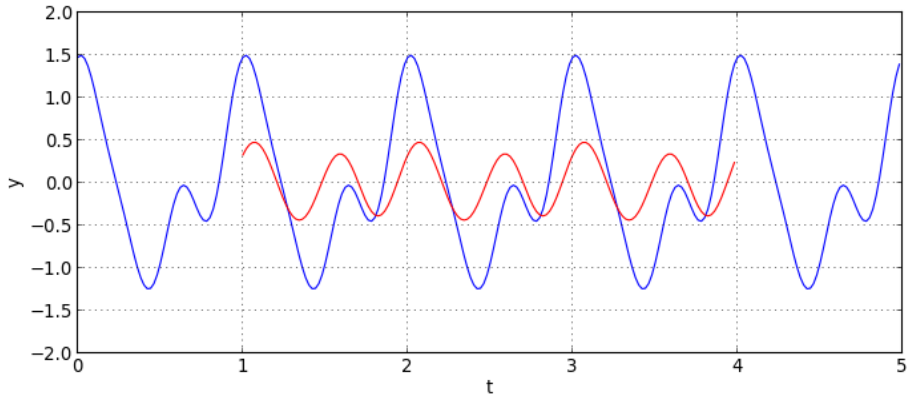

```
figure(figsize=(6,4))
plot(fe*f,phi,'r')
xlabel('f')
ylabel('phi')
grid()
```



Filtrage par convolution :

```
y = scipy.signal.convolve(x,h,mode='valid')
```

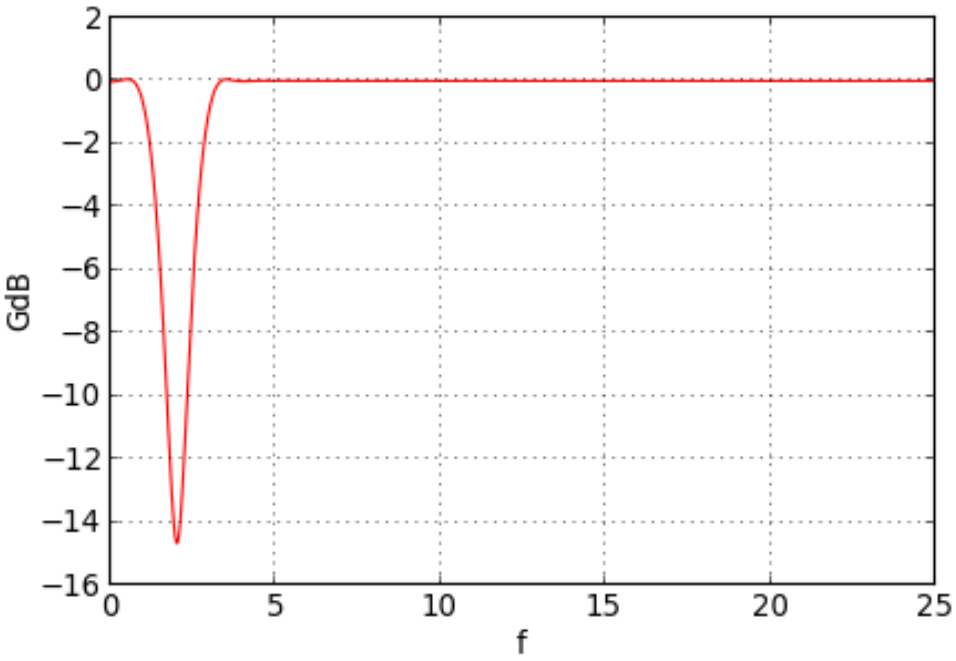
```
ny = y.size
ty = numpy.zeros(ny)
for k in range(ny):
    ty[k] = P*te+te*k
figure(figsize=(10,4))
plot(t,x,'b')
plot(ty,y,'r')
xlabel('t')
ylabel('y')
axis([0,5,-2,2])
grid()
```



3.c. Filtre coupe-bande

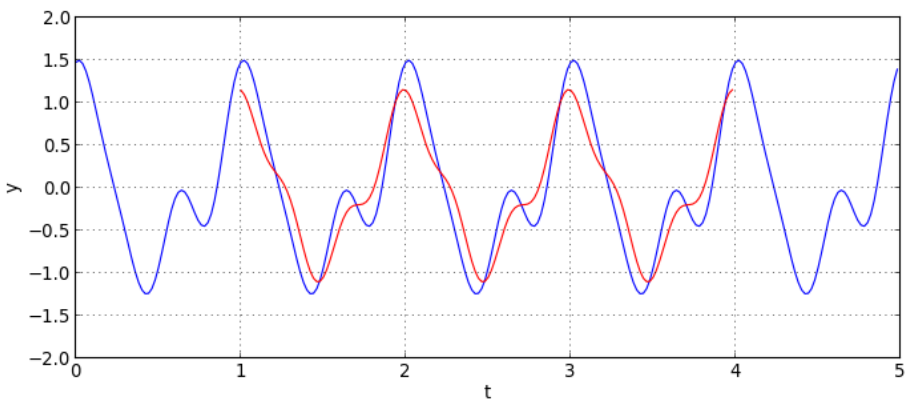
Définition du filtre, calcul de la réponse impulsionnelle et tracé du gain en décibel (en fonction de la fréquence réelle) :

```
fca = 1.5 # frequence de coupure basse
fcb = 2.5 # coupure haute
a = fca/fe
b = fcb/fe
P = 50
h = filtreRIF("CBande",a,b,P,"hann")
[f,g,phi] = reponseFreq(h)
figure(figsize=(6,4))
plot(fe*f,20*numpy.log10(g),'r')
xlabel('f')
ylabel('GdB')
grid()
```



Filtrage par convolution :

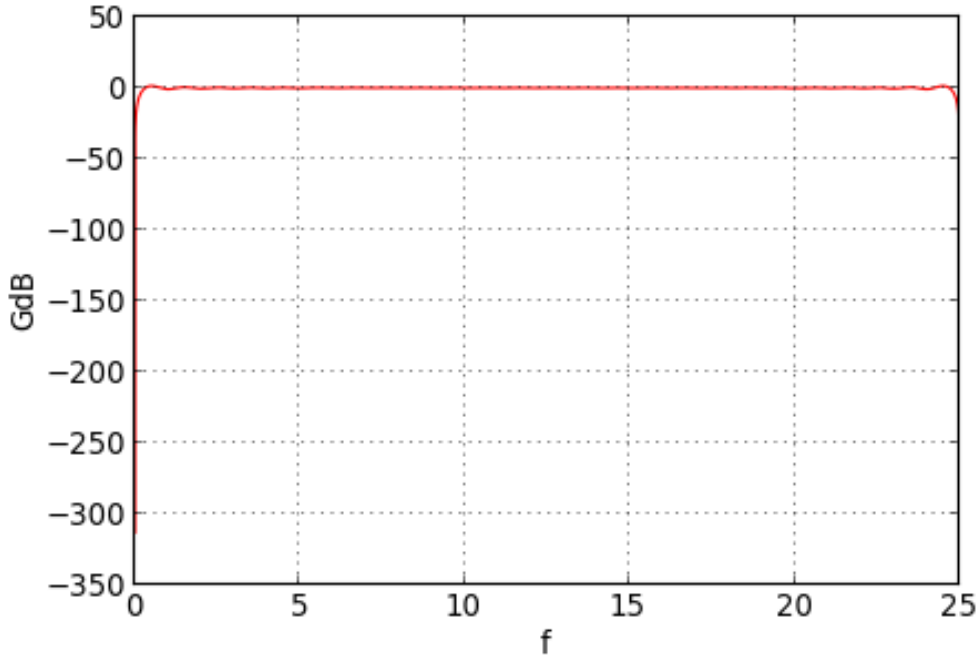
```
y = scipy.signal.convolve(x,h,mode='valid')
ny = y.size
ty = numpy.zeros(ny)
for k in range(ny):
    ty[k] = P*te+te*k
figure(figsize=(10,4))
plot(t,x,'b')
plot(ty,y,'r')
xlabel('t')
ylabel('y')
axis([0,5,-2,2])
grid()
```



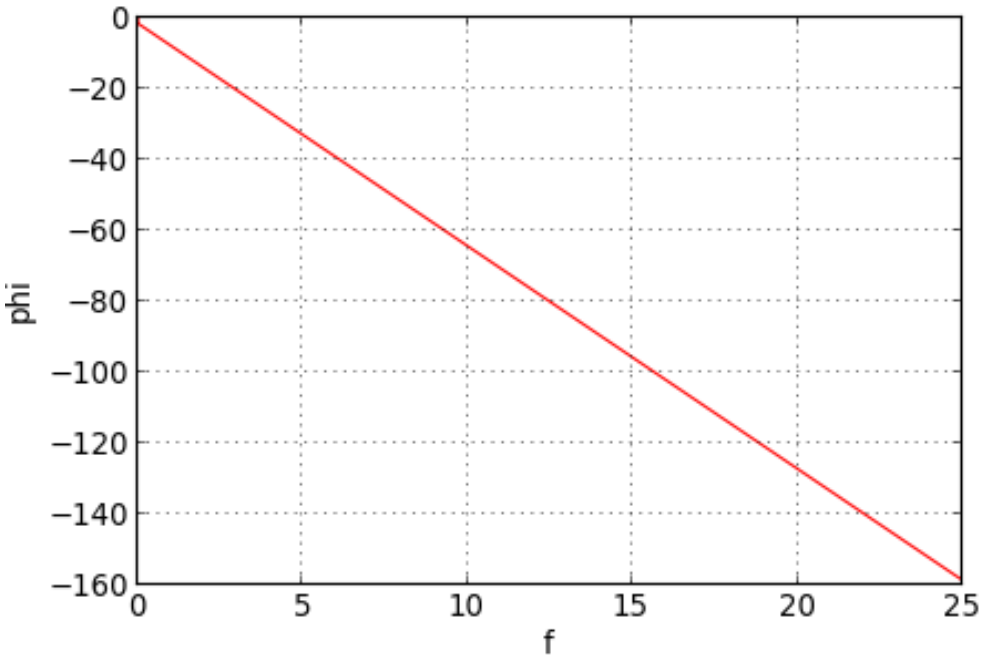
3.d. Filtre de quadrature

Le filtre de quadrature déphase de $\pi/2$ toutes les composantes spectrales du signal.

```
P=int(fe)
h = filtreRIF("Quad",None,None,P,"boxcar")
[f,g,phi] = reponseFreq(h)
figure(figsize=(6,4))
plot(fe*f,20*numpy.log10(g),'r')
xlabel('f')
ylabel('GdB')
grid()
```

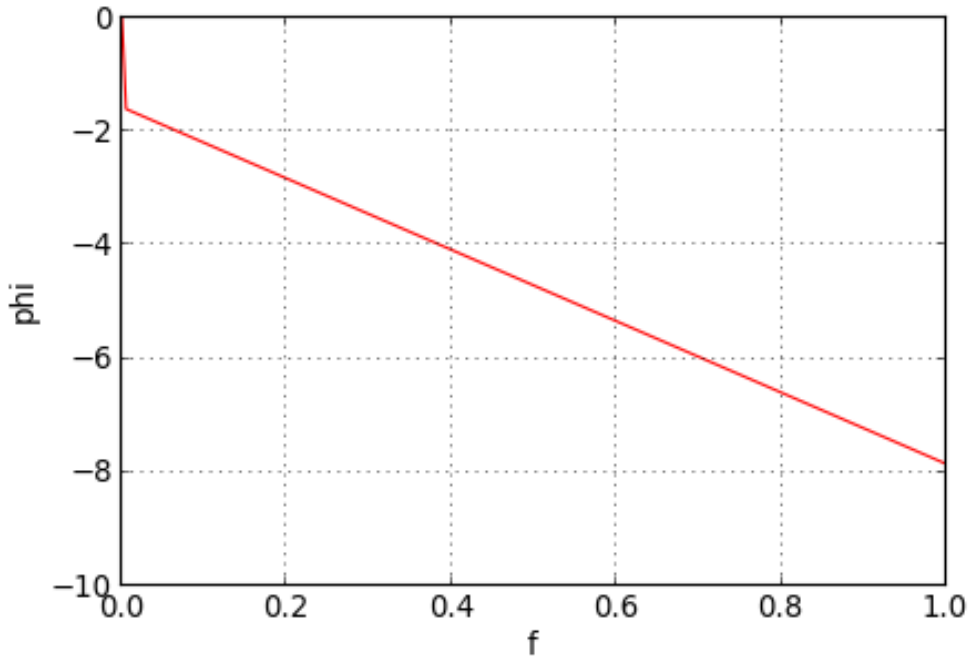



```
figure(figsize=(6,4))
plot(fe*f,phi,'r')
xlabel('f')
ylabel('phi')
grid()
```



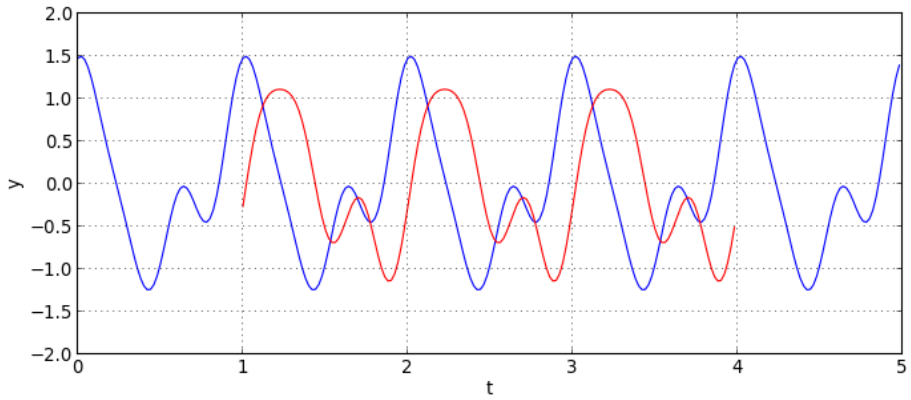
Voyons la phase en détail au voisinage de l'origine :

```
figure(figsize=(6,4))
plot(fe*f,phi,'r')
xlabel('f')
ylabel('phi')
axis([0,1,-10,0])
grid()
```



On constate que la phase n'est pas linéaire par rapport à la fréquence.

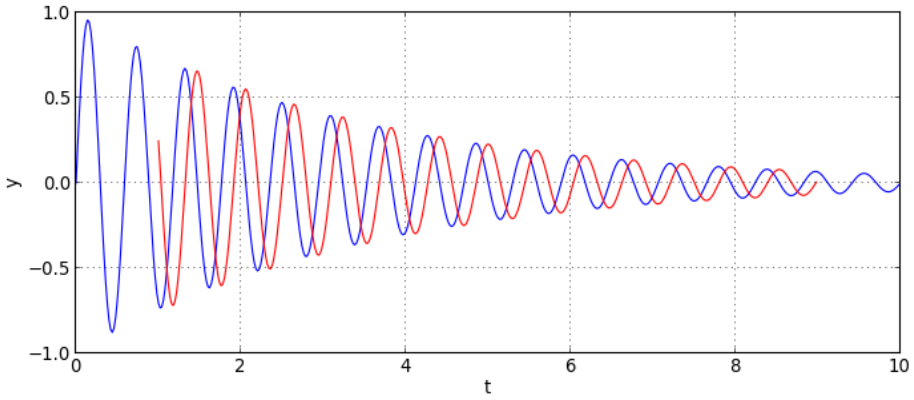
```
y = scipy.signal.convolve(x,h,mode='valid')
ny = y.size
ty = numpy.zeros(ny)
for k in range(ny):
    ty[k] = P*te+te*k
figure(figsize=(10,4))
plot(t,x,'b')
plot(ty,y,'r')
xlabel('t')
ylabel('y')
axis([0,5,-2,2])
grid()
```



Comme on peut le constater sur cet exemple, le filtre de quadrature déforme les signaux dans la bande passante. En effet, la phase de ce filtre n'est pas linéaire par rapport à la fréquence. Son application est plutôt réservée aux signaux sinusoïdaux ou quasi-sinusoïdaux :

```
f0 = 1.7
def signal(t):
    return numpy.sin(2*math.pi*f0*t)*numpy.exp(-t*0.3)
t = numpy.arange(start=0.0,stop=10.0,step=te)
x = signal(t)
y = scipy.signal.convolve(x,h,mode='valid')
ny = y.size
ty = numpy.zeros(ny)
for k in range(ny):
    ty[k] = P*te+te*k
figure(figsize=(10,4))
plot(t,x,'b')
plot(ty,y,'r')
xlabel('t')
ylabel('y')
axis([0,10,-1,1])
```

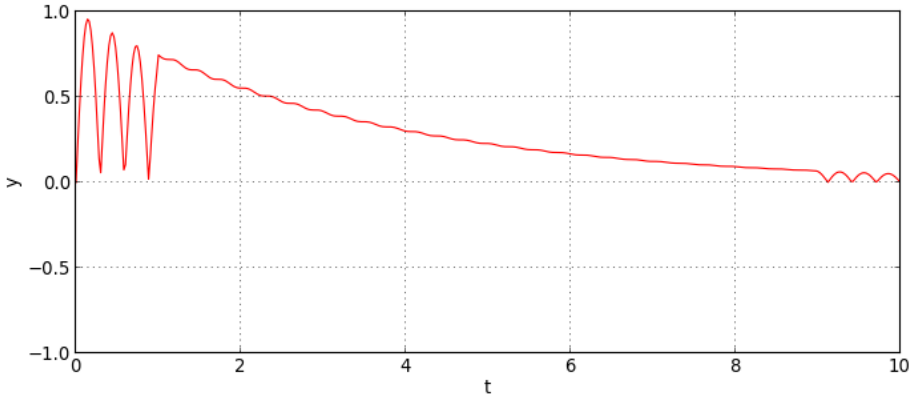
```
grid()
```



Remarque : l'échelle de temps du signal filtré est calculée de manière à annuler le décalage PT_e, ce qui donne bien un déphasage de $\pi/2$ entre les deux signaux. Pour un filtre temps-réel, le décalage PT_e vient se rajouter et la quadrature ne fonctionne pas.

Une application de ce filtre est la détection d'amplitude, obtenue en calculant la norme de $z = x + jy$:

```
y_full = numpy.zeros(x.size)
for k in range(ny):
    y_full[P+k] = y[k]
amplitude = numpy.sqrt(y_full*y_full+x*x)
figure(figsize=(10,4))
plot(t,amplitude,'r')
xlabel('t')
ylabel('y')
axis([0,10,-1,1])
grid()
```

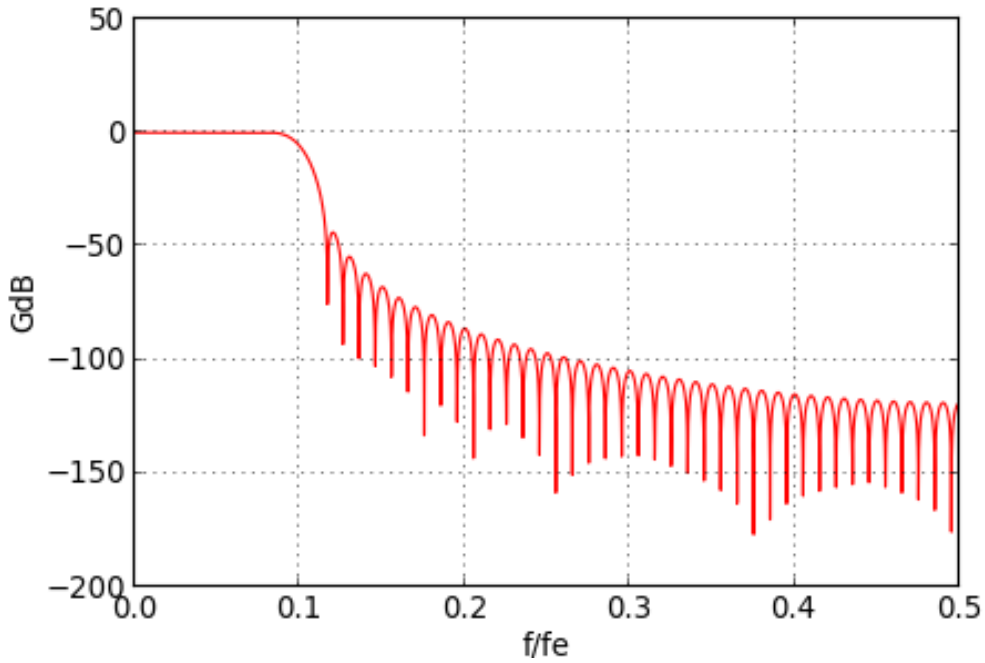


Bien sûr, la détection ne fonctionne qu'aux instants où la sortie est disponible.

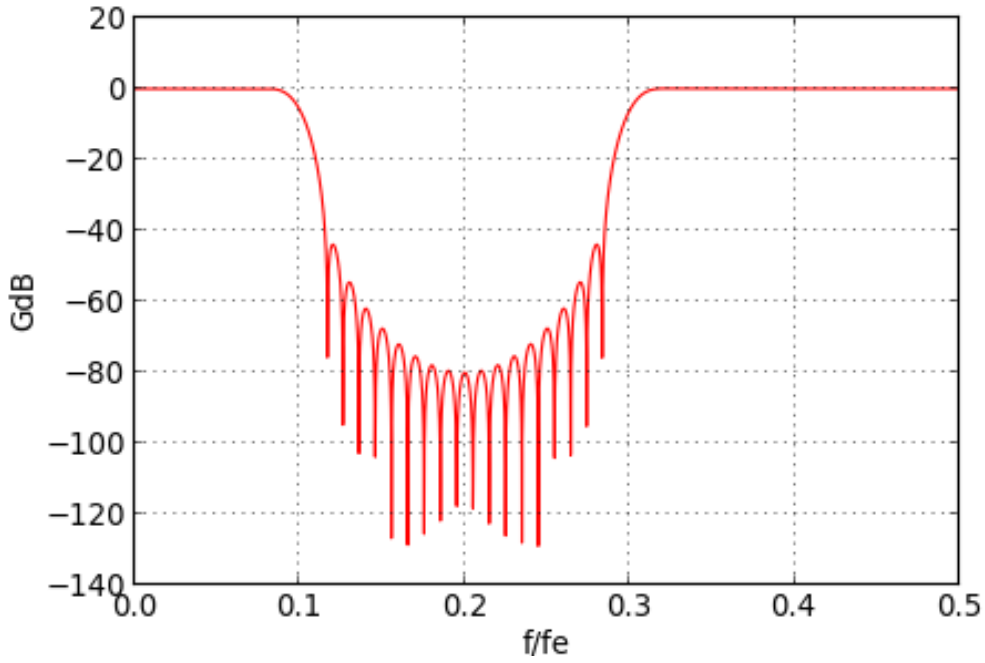
4. Utilisation de scipy.signal.firwin

La fonction `scipy.signal.firwin` permet de générer la réponse impulsionnelle d'un filtre défini par une liste de fréquences de coupures. Par défaut, la fréquence de Nyquist (moitié de la fréquence d'échantillonnage) est égale à 1. Pour reprendre la convention des exemples précédents, nous la fixons à 0.5.

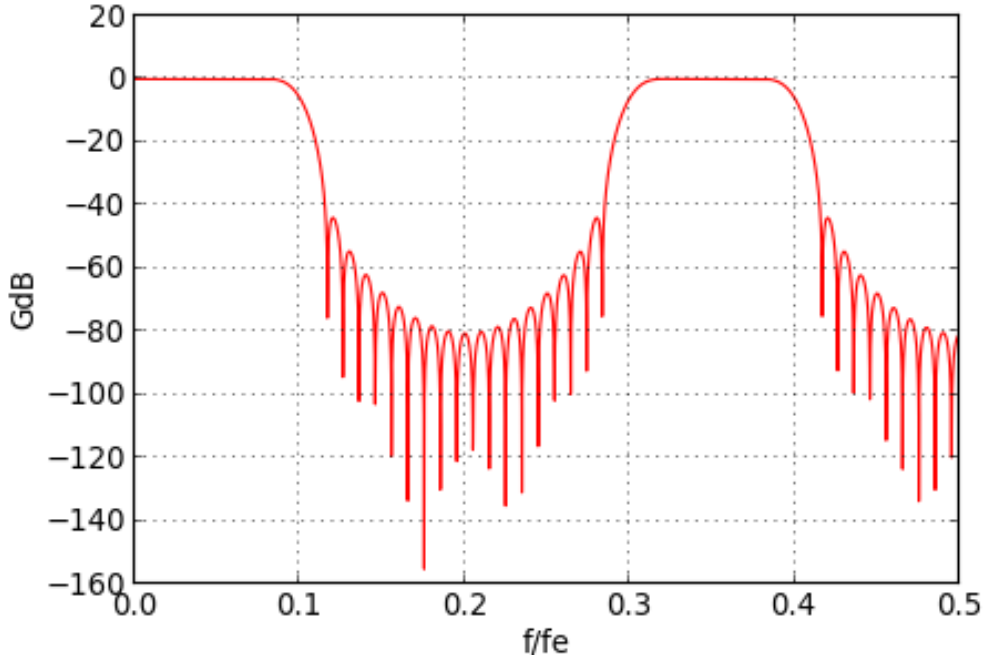
```
h = scipy.signal.firwin(numtaps=2*P+1,cutoff=[0.1],nyq=0.5,window='hann')
[f,g,phi] = reponseFreq(h)
figure(figsize=(6,4))
plot(f,20*numpy.log10(g),'r')
xlabel('f/fe')
ylabel('GdB')
grid()
```




```
h = scipy.signal.firwin(numtaps=2*P+1,cutoff=[0.1,0.3],nyq=0.5,window='hann')
[f,g,phi] = reponseFreq(h)
figure(figsize=(6,4))
plot(f,20*numpy.log10(g),'r')
xlabel('f/fe')
ylabel('GdB')
grid()
```




```
h = scipy.signal.firwin(numtaps=2*P+1,cutoff=[0.1,0.3,0.4],nyq=0.5,window='hann')
[f,g,phi] = reponseFreq(h)
figure(figsize=(6,4))
plot(f,20*numpy.log10(g),'r')
xlabel('f/fe')
ylabel('GdB')
grid()
```



Références

[1] E. Tisserand, J.F. Pautex, P. Schweitzer, *Analyse et traitement des signaux*, (Dunod, 2008)