



Ecole Polytechnique de Tunisie

Module: Traitement~numérique~du~signal

\large{Projet}

\large{{\color{red}{Débruitage~des~signaux~aléatoires~par~filtrage~de~Wiener~}}}}

Ce~travail~est~fait~par:~{\color{blue}{Sofien~Resifi,~Ahmed~Belkhir~\&~Omar~Khaled}}

Aperçu~général~sur~le~filtre~de~Wiener

Dans ce mini-projet, on s'intéresse au débruitage des signaux aléatoires en utilisant un filtre de Wiener de type RIF

Partie~préliminaire

{\color{green}{Question~1}}

\large{ {\color{red}{R}} \text{ est la matrice d'autocorrélation} }

$$\{\large{R=E[\mathbf{x}(n)\mathbf{x}(n)^T]}\}$$

\large{ {\color{red}{P}} \text{ est la matrice d'intercorrélation} }

$$\{\large{P=E[\mathbf{x}(n)d(n)]}\}$$

{\color{green}{Question~2}}

$$\{\large{\hat{d}(n)=\sum_{l=0}^{L-1} h_l x(n-l)} \text{ avec } h_{opt}=[h_0, h_1, \dots, h_{L-1}]\}$$

{\large \hat{d}(n)=h_{opt} \circledast x(n)}

$$\{\large \hat{d}(n)=\mathbf{h}_{opt}^T \mathbf{x}(n) \text{ avec } \mathbf{x}(n)=[x(n), \dots, x(n-L+1)]\}$$

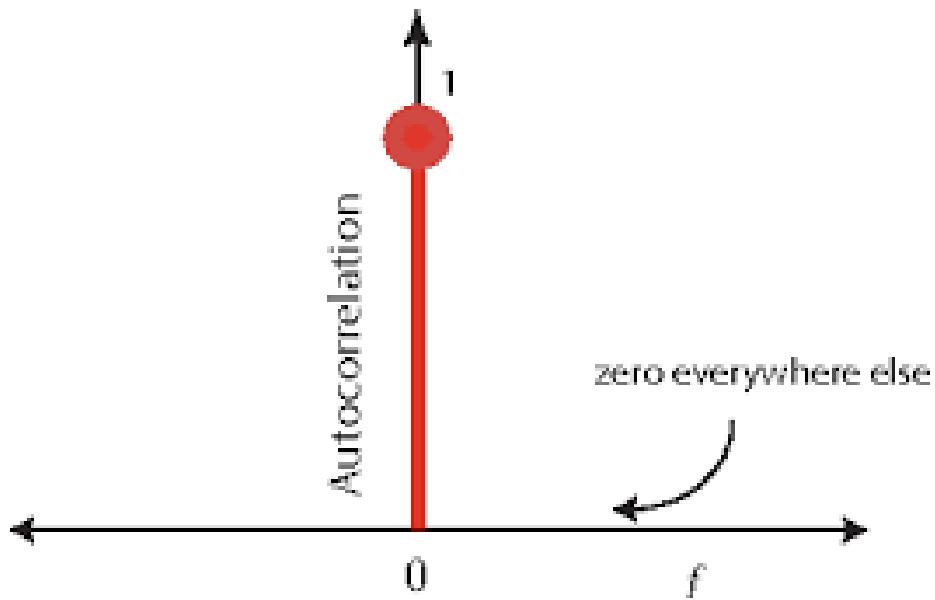
{\color{green}{Question~3}}

$$R = E[\mathbf{x}(n)\mathbf{x}(n)^T] = E[(\mathbf{d}(n) + \mathbf{u}(n))(\mathbf{d}(n) + \mathbf{u}(n))^T] = E[\mathbf{d}(n)\mathbf{d}(n)^T + E[\mathbf{u}(n)\mathbf{u}(n)^T] = R_d + R_u$$

{\color{red}{Car d(n) et u(n) sont indépendants}}

$$p = E[\mathbf{x}(n)d(n)] = E[(\mathbf{d}(n) + \mathbf{u}(n))d(n)] = E[\mathbf{d}(n)d(n)] \begin{aligned} p &= \begin{pmatrix} r_d(0) & \dots & r_d(L) & \dots & r_d(L-1) \end{pmatrix} \\ &\quad \begin{pmatrix} r_{\{X\}}(0) & \dots & r_{\{X\}}(1) & \dots & r_{\{X\}}(L-1) \end{pmatrix} \end{aligned}$$
$$R_{\{X\}} = \begin{pmatrix} r_{\{X\}}(0) & \dots & r_{\{X\}}(1) & \dots & r_{\{X\}}(L-1) \end{pmatrix}$$
$$r_{\{d\}}(0) = \begin{pmatrix} r_d(0) & \dots & r_d(L) & \dots & r_d(L-1) \end{pmatrix}$$
$$r_{\{d\}}(1) = \begin{pmatrix} r_d(1) & \dots & r_d(L+1) & \dots & r_d(2L-1) \end{pmatrix}$$
$$r_{\{d\}}(L) = \begin{pmatrix} r_d(L) & \dots & r_d(2L-1) & \dots & r_d(0) \end{pmatrix}$$
$$r_{\{d\}}(L-1) = \begin{pmatrix} r_d(L-1) & \dots & r_d(0) & \dots & r_d(L-2) \end{pmatrix}$$
$$r_{\{u\}}(0) = \begin{pmatrix} r_u(0) & \dots & r_u(L) & \dots & r_u(L-1) \end{pmatrix}$$
$$r_{\{u\}}(1) = \begin{pmatrix} r_u(1) & \dots & r_u(L+1) & \dots & r_u(2L-1) \end{pmatrix}$$
$$r_{\{u\}}(L) = \begin{pmatrix} r_u(L) & \dots & r_u(2L-1) & \dots & r_u(0) \end{pmatrix}$$
$$r_{\{u\}}(L-1) = \begin{pmatrix} r_u(L-1) & \dots & r_u(0) & \dots & r_u(L-2) \end{pmatrix}$$

\large{Voici l'autocorrelation d'un bruit blanc gaussien}



\large{Pour un bruit blanc Gaussien $r_u(\tau) = \sigma^2 \delta(\tau)$ }

\large{implique r_u est nulle partout sauf en $\tau=0$ où $r_u(0) = \sigma^2$ }

\large{Donc on en déduit que R_u a la forme suivante:} \begin{equation*} R_u = \begin{pmatrix} \sigma^2 & 0 & \cdots \\ 0 & \sigma^2 & \cdots \\ \vdots & \vdots & \ddots & \vdots & \sigma^2 \end{pmatrix} \end{equation*}

{\color{green}{Question~4}}

\large{on a RSB_{db} = 10 \log_{10} (\frac{E[d^2(n)]}{E[u^2(n)]})}

\large et~on~a RSB'_{db} = 10\log_{10}(\frac{E[d^2(n)]}{E[u^2(n)]})

\large Donc~pour~calculer~RSB'_{db}~il~faut~determiner~d'(n)~et~u'(n)

d'(n) = h_{opt} \circledast \mathbf{d}(n) = h_{opt}^T \mathbf{d}(n)

u'(n) = h_{opt} \circledast \mathbf{u}(n) = h_{opt}^T \mathbf{u}(n)

\implies E[d'^2(n)] = E[(h_{opt})^T \mathbf{d}(n)(h_{opt})^T \mathbf{d}(n)^T] = E[h_{opt}^T \mathbf{d}(n) \mathbf{d}(n)^T h_{opt}] = \mathbf{d}^T h_{opt}

\implies E[u'^2(n)] = E[(h_{opt})^T \mathbf{u}(n)(h_{opt})^T \mathbf{u}(n)^T] = E[h_{opt}^T \mathbf{u}(n) \mathbf{u}(n)^T h_{opt}] = \mathbf{u}^T h_{opt}

\large Donc~on~obtient:

$$\boxed{RSB'_{db} = 10\log_{10}(\frac{\mathbf{d}^T h_{opt}}{\mathbf{u}^T h_{opt}})}$$

{\color{green}{Question~5}}

J(h) = E[e^2(n)] = E[(d(n)-h)^2] = E[d^2(n)] - 2E[d(n)h] + E[h^2] = r_d(0) - 2\mathbf{d}^T \mathbf{p} + \mathbf{p}^T \mathbf{p}

\nabla J(h) = -2\mathbf{p} + 2R\mathbf{h}

\nabla J(h_{opt}) = 0

\implies R\mathbf{h}_{opt} = \mathbf{p}

$$\boxed{J_{min} = r_d(0) - \mathbf{p}^T \mathbf{p}}$$

Implémentation

Commençons tout d'abord par importer les modules nécessaires:

```
In [1]: from IPython.display import clear_output
!pip install soundfile
clear_output()
```

```
In [2]: from numpy import random
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import norm
from scipy.linalg import toeplitz
import wave
import soundfile as sf
import IPython.display as ipd
from tqdm import tqdm
from scipy import signal
import scipy as sc
import warnings
warnings.filterwarnings('ignore')
clear_output()
```

{\color{green}{Question~1}}

Générer $N = 10^3$ échantillons d'un signal de référence

$$d(n) = \sin(\omega_0 n + \phi(n))$$

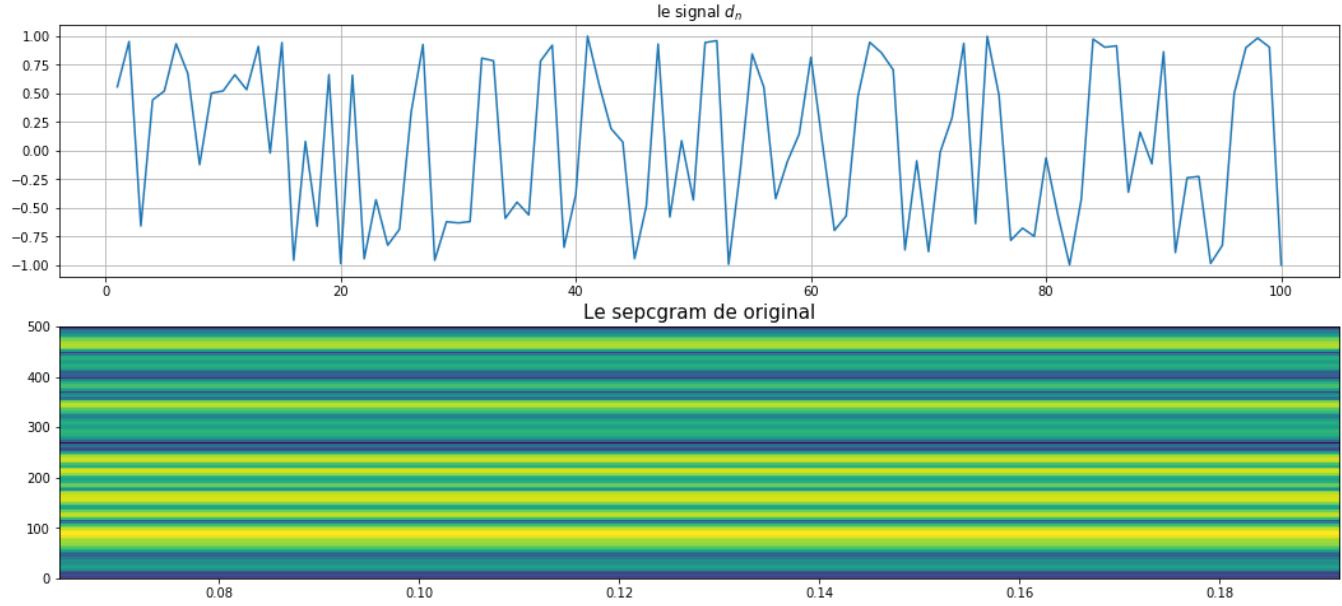
$$n = 1, \dots, N, \quad \omega_0 = 0.05\pi \quad \text{où } \phi(n) \sim U([0, 2\pi])$$

Pour cela on a définie la fonction `d` qui va créer le signal `d_n`:

```
In [3]: N=1000
w0=0.05*np.pi
phi= random.uniform(low=0,high=np.pi*2,size=N)
def d(n):
    return np.sin(w0*n+phi[n-1])
lis=np.arange(1,N+1)
d_n=d(lis)
```

On va visualiser le signal `d(n)` obtenu en fonction du temps ainsi que son spectrogramme:

```
In [4]: plt.subplots(figsize=(18,8))
plt.subplot(211)
sns.lineplot(x=lis[:100], y=d_n[:100])
plt.grid()
plt.title("le signal $d_n$")
plt.subplot(212)
plt.title("Le sepcgram de original", fontsize=15)
specgram=plt.specgram(d_n[:100], Fs=1000)
```



{\color{green}{Question~2}}

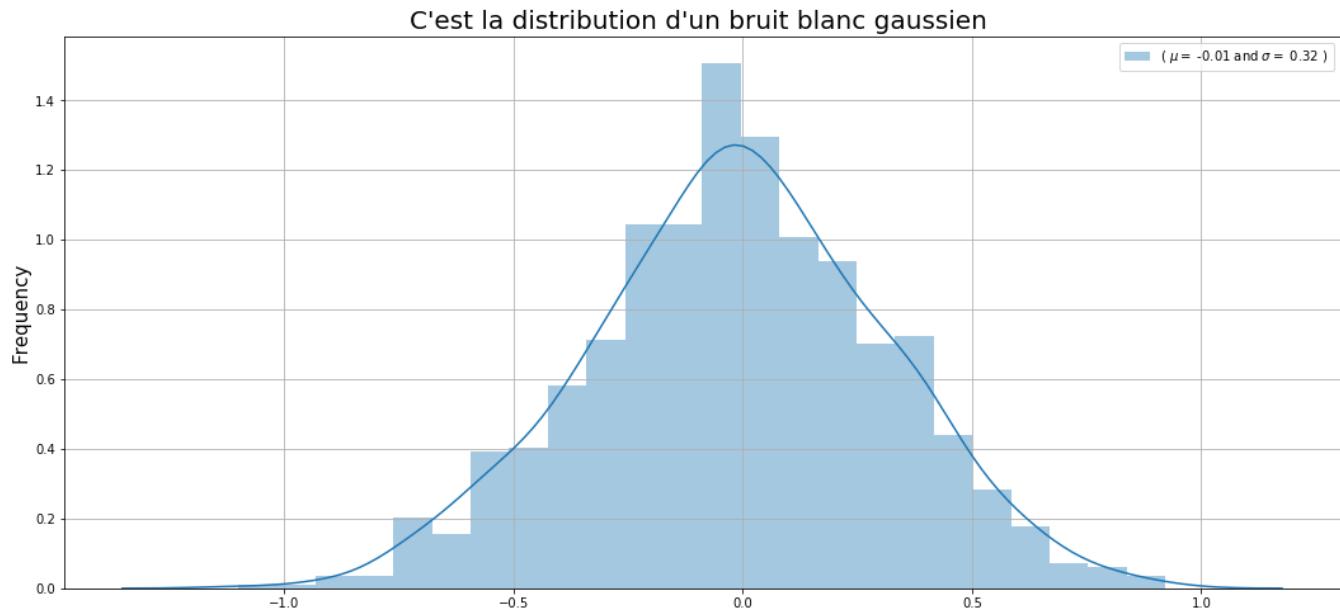
Générer le signal bruité $x(n)$ en utilisant un bruit blanc du type gaussien centré de puissance $\sigma^2=0.1$

Pour cela, on a crée la fonction `create_noise` suivante:

```
In [5]: def create_noise(N):
    mu, sigma = 0, np.sqrt(0.1)
    u_n = np.random.normal(mu, sigma, N)
    return u_n
u_n=create_noise(N)
```

On peut visualiser la distribution du bruit blanc gaussien:

```
In [6]: plt.subplots(figsize=(18,8))
sns.distplot(u_n)
(mu, sigma) = norm.fit(u_n)
plt.legend(['( $\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)],
          loc='best')
plt.title("C'est la distribution d'un bruit blanc gaussien ", fontsize=20)
plt.ylabel("Frequency", fontsize=15)
plt.grid()
```



On voit bien que la distribution est centrée en 0 et de variance $\sigma=0.33$

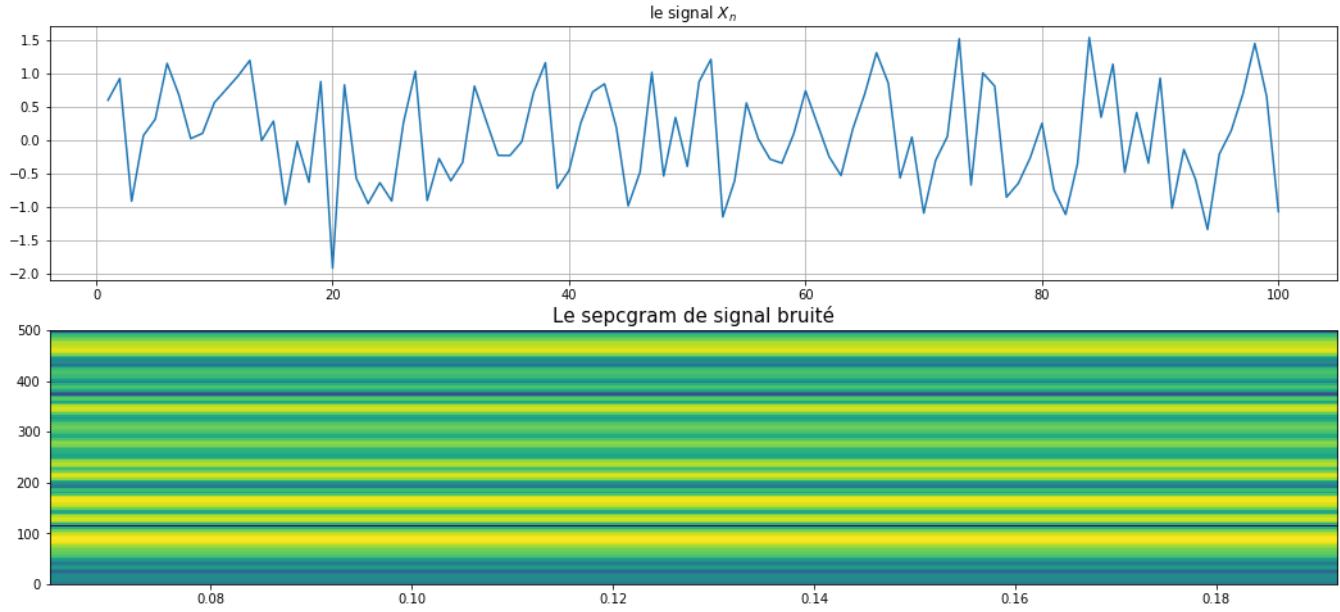
Maintenant on va créer le signal bruité à partir de d_n qui est le signal d'entrée et u_n bruit blanc gaussien

$$\boxed{X_n=d_n+u_n}$$

```
In [7]: X_n=d_n+u_n
```

De même, on peut visualiser le signal bruité X_n obtenu ainsi que son spectrogramme:

```
In [8]: plt.subplots(figsize=(18, 8))
plt.subplot(211)
sns.lineplot(x=lis[:100], y=X_n[:100])
plt.grid()
plt.title("le signal $X_n$")
plt.subplot(212)
plt.title("Le specgram de signal bruité", fontsize=15)
specgram=plt.specgram(X_n[:100], Fs=1000)
```



\large{Le signal est maintenant bruité.}

{\color{green}{Question~3}}

\large{Ici on va calculer R et P afin de calculer h_{opt}}

Pour cela, on utilise la fonction `create_R_P` suivante qui, étant donné le signal original, et le signal bruité, elle retourne h_{opt} du filtre optimal.

```
In [9]: def create_R_P(X_n, d_n, L, N):
    vect=np.correlate(X_n, X_n, mode='full')
    vect1=np.correlate(X_n, d_n, mode='full')
    R_pre=vect[N-1:N+L-1]
    P=vect1[N-1:N+L-1]
    R=toeplitz(R_pre, R_pre)
    return R, P
```

(On va choisir l'ordre $L=3$ pour ce filtre comme dans l'énoncé)

```
In [10]: L=3 # ordre de filtre
R,P=create_R_P(X_n,d_n,L,N)
```

```
In [11]: h_opt=np.dot(np.linalg.inv(R),P)
```

```
In [12]: h_opt
```

```
Out[12]: array([ 8.19982888e-01,  4.85705981e-03, -7.17373452e-04])
```

{\color{green}{Question~4}}

\large{Ici~on~va~créer~une~fonction~qui~s'appelle~filter~qui~retourne~le~résultat~du~filtre}

$$\{\large{\hat{d}(n)=h_{opt} \circledast x(n) }\}$$

```
In [13]: def filter(X_n, h_opt):
    return np.convolve(X_n, h_opt, mode='same')

d_hat=filter(X_n, h_opt)
```

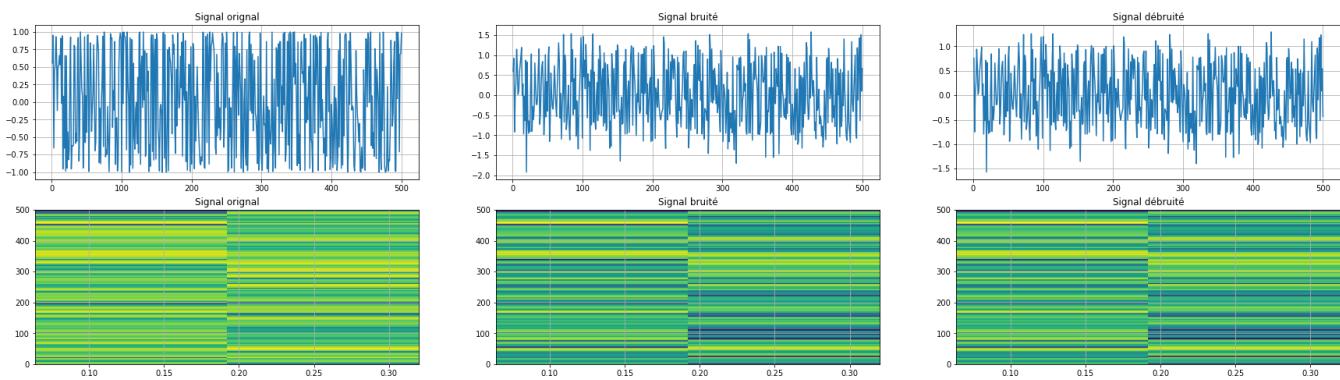
Etant donné le signal bruité et le vecteur h_{opt} , cette fonction filter retourne un signal débruité.

{\color{green}{Question~5}}

\large{Maintenant~on~va~tracer~le~signal~{\color{red}{original}}, le~signal~{\color{red}{bruité}}~puis~le~signal~{\color{red}{débruité}}~et~leur~spectrogrammes~respectifs}

```
In [14]: L=3#ordre de filtre
R,P=create_R_P(X_n,d_n,L,N)
h_opt=np.dot(np.linalg.inv(R),P)
d_hat=filter(X_n,h_opt)
```

```
In [15]: plt.subplots(figsize=(30,8))
plt.subplot(232)
plt.title("Signal bruité")
plt.plot(lis[:500],X_n[:500])
plt.grid()
plt.subplot(233)
plt.title("Signal débruité")
plt.plot(lis[:500],d_hat[:500])
plt.grid()
plt.subplot(231)
plt.title("Signal original")
plt.plot(lis[:500],d_n[:500])
plt.grid()
plt.subplot(235)
plt.title("Signal bruité")
specgram=plt.specgram(X_n[:500], Fs=1000)
plt.grid()
plt.subplot(236)
plt.title("Signal débruité")
specgram=plt.specgram(d_hat[:500], Fs=1000)
plt.grid()
plt.subplot(234)
plt.title("Signal original")
specgram=plt.specgram(d_n[:500], Fs=1000)
plt.grid()
```



On remarque bien que le signal débruité est plus proche du signal original que le signal bruité.

On a confirmé ainsi que notre filtre est fonctionnel!

{\color{green}{Question~6}}

On va déterminer les valeurs du Rapport Signal à Brui (RSB) avant et après débruitage ainsi que la valeur de J_{min} .

```
In [16]: RSB=10*np.log10(((np.array(d_n)**2).mean())/((np.array(u_n)**2).mean()))
RSB1=10*np.log10(((np.array(filter(d_n,h_opt)))**2).mean()/(np.array(filter(u_n,h_opt)))**2).mean()
print("La valeur du RSB avant débruitage est: {}".format(RSB))
print("La valeur du RSB après débruitage est: {}".format(RSB1))
```

La valeur du RSB avant débruitage est: 6.758508413061951
 La valeur du RSB après débruitage est: 6.755741131517791

La valeur du RSB est nettement améliorée. On a donc réussi à réduire la proportion du bruit dans le nouveau signal!

\large{Maintenant on va calculer J_{min} :

```
In [17]: R_d=np.correlate(d_n,d_n,mode='full')
r_d0=R_d[N-1]
```

$$\boxed{J_{min} = r_d(0) - \mathbf{h}_{opt}^T \mathbf{p}}$$

```
In [18]: jmin=r_d0-np.dot(h_opt.T,P)
print("la valeur de l'erreur Jmin est: {}".format(jmin))
```

la valeur de l'erreur Jmin est: 83.99405132834966

{\color{green}{Question~7}}

Dans le travail précédent, on a utilisé l'ordre $L=3$. Mais quel impact le choix de l'ordre aura-t-il sur la qualité du débruitage?

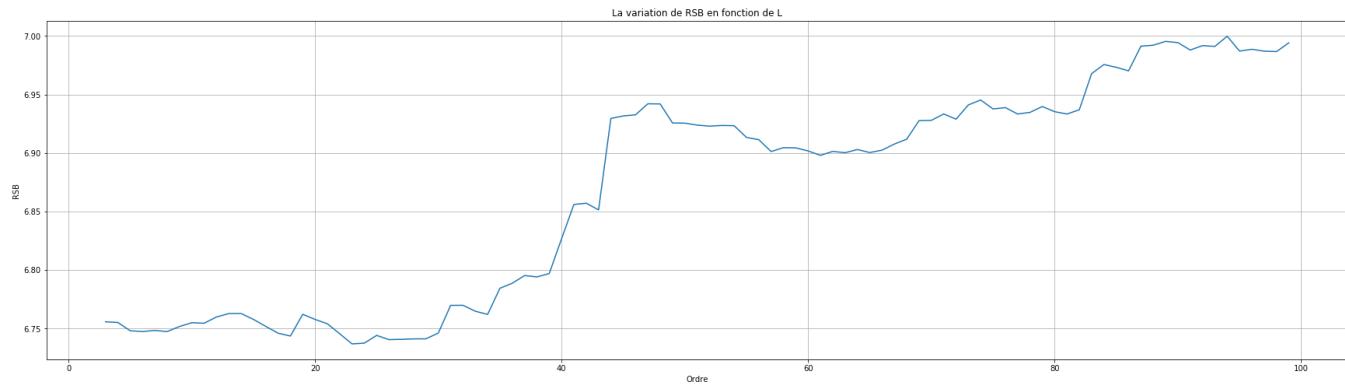
\large{On va observer les variations du RSB et de J_{min} en fonction de l'ordre du filtre:}

```
In [19]: def calcul_hopt(X,d,L,N):
    R,P=create_R_P(X,d,L,N)
    h_opt=np.dot(np.linalg.inv(R),P)
    return h_opt,P
```

```
In [20]: list_L=[i for i in range(3,100)]
list_hopt=[calcul_hopt(X_n,d_n,L,N)[0] for L in list_L ]
list_p=[calcul_hopt(X_n,d_n,L,N)[1] for L in list_L ]
```

```
In [21]: RSB_list = []
for i in range(len(list_L)):
    h_opt=list_hopt[i]
    L=list_L[i]
    RSB1=10*np.log10(((np.array(filter(d_n,h_opt)))**2).mean() / ((np.array(filter(u_n,h_opt)))**2).mean())
    RSB_list.append(RSB1)
```

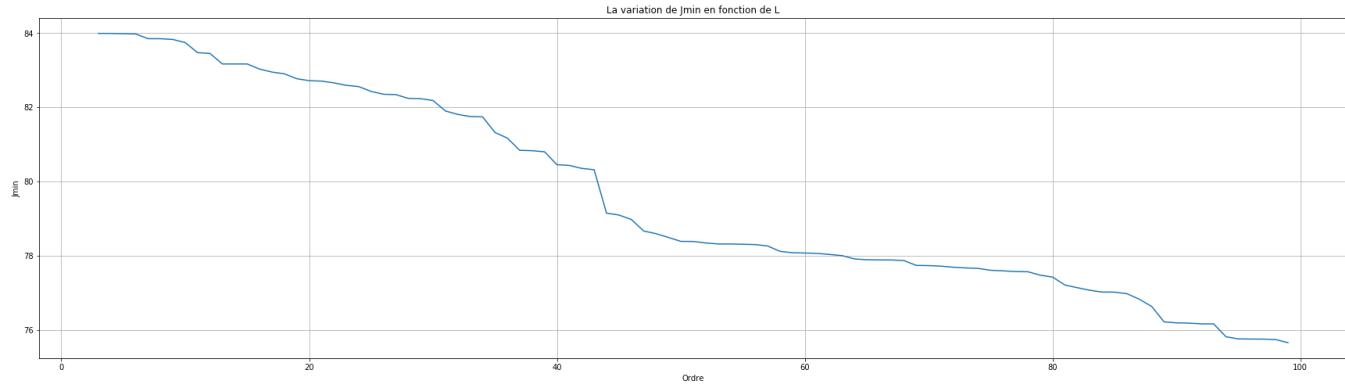
```
In [22]: plt.subplots(figsize=(30,8))
plt.title("La variation de RSB en fonction de L")
sns.lineplot(list_L,RSB_list)
plt.xlabel("Ordre")
plt.ylabel("RSB")
plt.grid()
```



On voit clairement que la RSB est croissante en fonction de l'ordre du filtre. Cependant, à partir d'un certain ordre, aux alentours de 40, l'amélioration du RSB n'est pas très importante.

```
In [23]: R_d=np.correlate(d_n,d_n,mode='full')
jmin_list=[]
for i in range(len(list_L)):
    h_opt=calcul_hopt(X_n,d_n,list_L[i],N)[0]
    p=calcul_hopt(X_n,d_n,list_L[i],N)[1]
    jmin=r_d0-np.dot(h_opt.T,p)
    jmin_list.append(jmin)
```

```
In [24]: plt.subplots(figsize=(30,8))
plt.title("La variation de Jmin en fonction de L")
plt.plot(list_L,jmin_list)
plt.xlabel("Ordre")
plt.ylabel("Jmin")
plt.grid()
```



La même remarque quant aux variations de l'erreur: plus on augmente l'ordre, plus on obtient des résultats meilleurs (erreur plus faible) et ayant un ordre supérieur à 40 n'aura pas d'impact important sur la qualité du débruitage.

La recherche de l'ordre de filtre est un problème d'optimisation:

On va choisir un ordre L_{opt} qui maximise le RSB (et minimise J_{min})

```
In [25]: L_opt=list_L[RSB_list.index(max(RSB_list))]
```

L_{opt} sera utile dans la partie bonus de ce rapport.

Le débruitage du signal bruité:

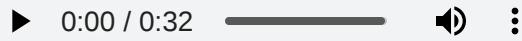
{\color{green}{Question~8}: La première méthode de débruitage}

{La première méthode de débruitage consiste à débruiter la totalité du signal c'est-on va considérer le signal comme un seul morceau:}

Ecouteons tout d'abord le morceau musical original:

```
In [26]: #listen to a sound
pth ="jonasz_lucille_extrait.wav"
ipd.Audio(pth)
```

Out[26]:



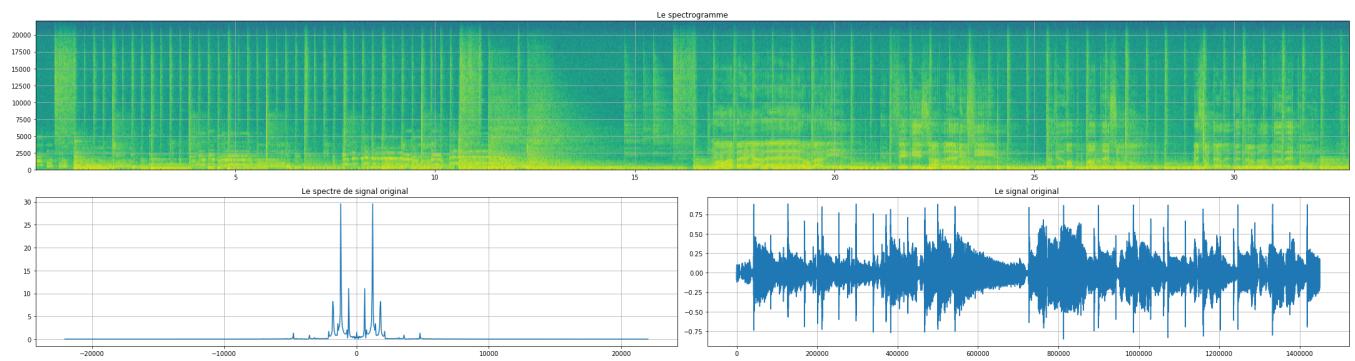
Pour travailler sur ce fichier son, on va utiliser le module soudfile pour extraire le vecteur contenant les valeurs des échantillons:

```
In [27]: signal, Fs =sf.read("jonasz_lucille_extrait.wav")
```

```
In [28]: f=np.linspace(-Fs/2,Fs/2,1024)
TFD=sc.fft(signal,1024)
y=abs(np.fft.fftshift(TFD))
```

Traçons maintenant le spectrogramme du morceau, son évolution temporelle, ainsi que son spectre:

```
In [29]: plt.subplots(figsize=(30,8))
plt.subplot(223)
plt.plot(f,y)
plt.title("Le spectre de signal original ")
plt.grid()
plt.subplot(211)
specgram=plt.specgram(signal, Fs=Fs)
plt.title("Le spectrogramme")
plt.grid()
plt.subplot(224)
plt.plot(signal)
plt.title("Le signal original ")
plt.grid()
plt.tight_layout()
```



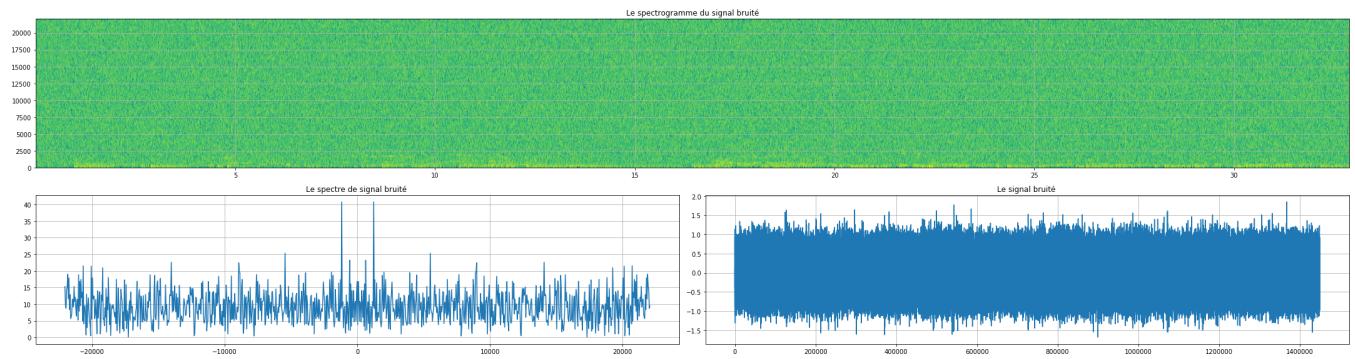
Maintenant on va créer du bruit pour l'ajouter à ce signal:

```
In [30]: u_n = create_noise(len(signal))
```

```
In [31]: signal_bruite=signal+u_n #c'est le signal bruité
```

```
In [32]: TFD1=sc.fft(signal_bruite,1024)
y_bruit=abs(np.fft.fftshift(TFD1))
```

```
In [33]: plt.subplots(figsize=(30,8))
plt.subplot(223)
plt.plot(f,y_bruit)
plt.title("Le spectre de signal bruité ")
plt.grid()
plt.subplot(211)
specgram=plt.specgram(signal_bruite, Fs=F)
plt.title("Le spectrogramme du signal bruité")
plt.grid()
plt.subplot(224)
plt.plot(signal_bruite)
plt.title("Le signal bruité ")
plt.grid()
plt.tight_layout()
```



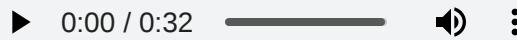
On remarque bien que le signal bruité présente un spectre contenant trop de fréquences et son specgramme est très bruité. Même l'évolution temporelle semble bruitée.

On va enregistrer le fichier audio dans un fichier pour travailler dessus facilement par la suite:

```
In [34]: sf.write('signal_bruit.wav',signal_bruite,F)
```

```
In [35]: pth1 ="signal_bruit.wav"
ipd.Audio(pth1)
```

Out[35]:



En écoutant le signal aussi, il n'est pas clair du tout. C'est comme le bruit qu'on entend lorsqu'on écoute une station radio sur une fréquence légèrement différente de la bonne fréquence ou lorsque la réception n'est pas bonne.

Pour débruyter ce signal, on aura besoin de calculer h_{opt} du filtre optimal.

Si on essaie de le calculer sur le fichier son en entier, cela va prendre trop de temps. On va donc le calculer sur le 1/4 du fichier:

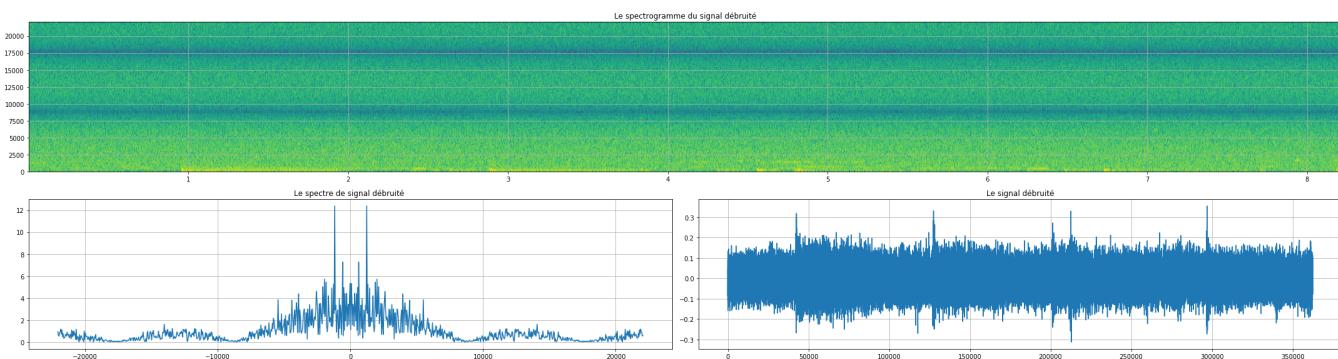
```
In [36]: L=5
N=int(len(signal)/4)
h_opt,P=calcul_hopt(signal_bruite[:N],signal[:N],L,N)
```

On a calculé h_{opt} . Appliquons le filtre sur le signal bruité et visualisons son spectrogramme, son spectre ainsi que son évolution temporelle:

```
In [37]: Signal_debrui=filter(signal_bruite[:N],h_opt)
```

```
In [38]: TFD2=sc.fft(Signal_debrui,1024)
y_debruit=abs(np.fft.fftshift(TFD2))
```

```
In [39]: plt.subplots(figsize=(30,8))
plt.subplot(223)
plt.plot(f,y_debruit)
plt.title("Le spectre de signal débruité ")
plt.grid()
plt.subplot(211)
specgram=plt.specgram(Signal_debrui, Fs=F)
plt.title("Le spectrogramme du signal débruité")
plt.grid()
plt.subplot(224)
plt.plot(Signal_debrui)
plt.title("Le signal débruité")
plt.grid()
plt.tight_layout()
```



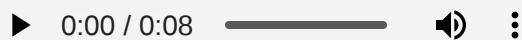
Le signal débruit est nettement meilleur que celui bruité. Ceci est clair d'après les 3 figures (spectrogramme, spectre et évolution temporelle).

Essayons d'écouter le fichier maintenant:

```
In [40]: sf.write('Signal_debrui_TP.wav',Signal_debrui,Fs)
```

```
In [41]: pth1 ="Signal_debrui_TP.wav"
ipd.Audio(pth1)
```

Out[41]:



Afin de mesurer l'amélioration qu'on a faite, on peut comparer les valeurs de RSB avant et après débruitage:

```
In [42]: # Avant débruitage
RSB=10*np.log10(((np.array(signal)**2).mean())/((np.array(u_n)**2).mean()))
print(RSB)

-8.560420853510928
```

```
In [43]: RSB1=10*np.log10(((np.array(Signal_debrui)**2).mean())/((np.array(filter(u_n,h_opt)))**2).mean())
print(RSB1)

1.6646671047201584
```

On constate que le Raport Signal à Bruit s'est nettement amélioré. On a réussi donc de réduire la proportion du bruit dans le signal débruité.

\large{Maintenant on va calculer J_{min}}

```
In [44]: R_d=np.correlate(signal[:N],signal[:N],mode='full')
r_d0=R_d[N-1]
```

$$\boxed{J_{\min} = r_d(0) - \mathbf{h}_{\text{opt}}^T \mathbf{p}}$$

```
In [45]: jmin=r_d0-np.dot(h_opt.T,P)
print("La valeur de l'erreur quadratique moyenne Jmin est: {}".format(jmin))

La valeur de l'erreur quadratique moyenne Jmin est: 2392.157630080044
```

Etudions l'impact de l'ordre du filtre sur le RSB et sur Jmin.

Pour cela, on va tracer leurs évolutions en fonction de l'ordre L du filtre:

Comme le calcul de RSB et de Jmin prend trop de temps vu que le fichier est volumineux, on va voir l'évolution jusqu'à l'ordre 10:

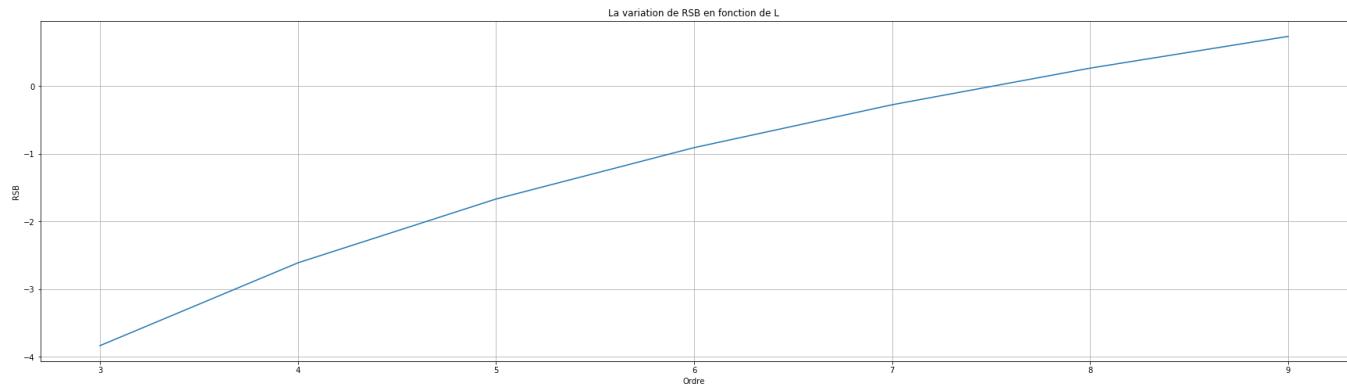
```
In [46]: list_L=[i for i in range(3,10)]
list_hopt=[]
list_p=[]
for i in tqdm(range(len(list_L))):
    h_opt,P=calcul_hopt(signal_bruite[:N],signal[:N],list_L[i],N)
    list_hopt.append(h_opt)
    list_p.append(P)
```

100% |██████████| 7/7 [14:20<00:00, 122.89s/it]

```
In [47]: RSB_list =[]
for i in tqdm(range(len(list_L))):
    h_opt=list_hopt[i]
    L=list_L[i]
    RSB1=10*np.log10(((np.array(filter(signal,h_opt)))**2).mean() / ((np.array(filter(u_n,h_opt)))**2).mean())
    RSB_list.append(RSB1)
```

100% |██████████| 7/7 [00:00<00:00, 21.46it/s]

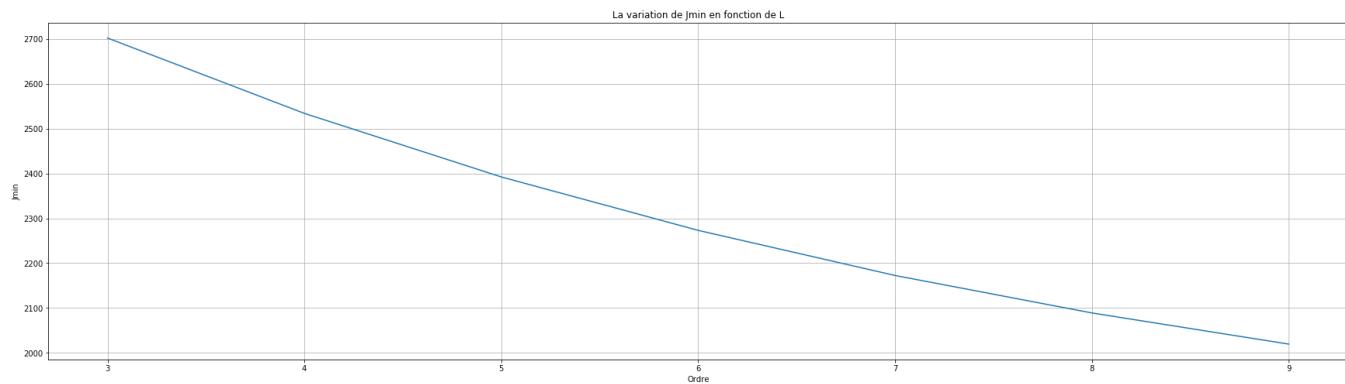
```
In [48]: plt.subplots(figsize=(30,8))
plt.title("La variation de RSB en fonction de L")
sns.lineplot(list_L,RSB_list)
plt.xlabel("Ordre")
plt.ylabel("RSB")
plt.grid()
```



Le rapport signal à bruit est nettement amélioré en augmentant l'ordre du filtre. Qu'en est-il de l'erreur Jmin?

```
In [49]: R_d=np.correlate(d_n,d_n,mode='full')
jmin_list=[]
for i in range(len(list_L)):
    h_opt=list_hopt[i]
    p=list_p[i]
    jmin=r_d0-np.dot(h_opt.T,p)
    jmin_list.append(jmin)
```

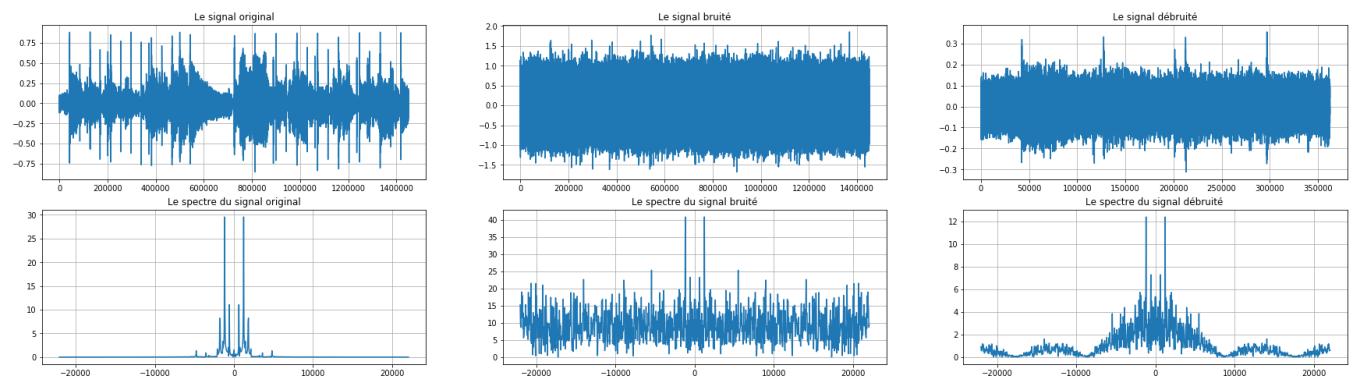
```
In [50]: plt.subplots(figsize=(30,8))
plt.title("La variation de Jmin en fonction de L")
plt.plot(list_L,jmin_list)
plt.xlabel("Ordre")
plt.ylabel("Jmin")
plt.grid()
```



De même, plus l'ordre du filtre est élevé, plus le débruitage est meilleur avec une erreur moyenne plus faible.

{\large{\color{red}{Récapitulatif}}}

```
In [51]: plt.subplots(figsize=(30,8))
plt.subplot(231)
plt.plot(signal)
plt.title("Le signal original")
plt.grid()
plt.subplot(232)
plt.plot(signal_bruite)
plt.title("Le signal bruité")
plt.grid()
plt.subplot(233)
plt.plot(Signal_debrui)
plt.title("Le signal débruité")
plt.grid()
plt.subplot(234)
plt.plot(f,y)
plt.title("Le spectre du signal original")
plt.grid()
plt.subplot(235)
plt.plot(f,y_bruit)
plt.title("Le spectre du signal bruité")
plt.grid()
plt.subplot(236)
plt.plot(f,y_debruit)
plt.title("Le spectre du signal débruité ")
plt.grid()
```



Ces figures résument l'effet du filtre sur le signal bruité. Le filtre nous a permis d'obtenir un signal plus proche du signal original.

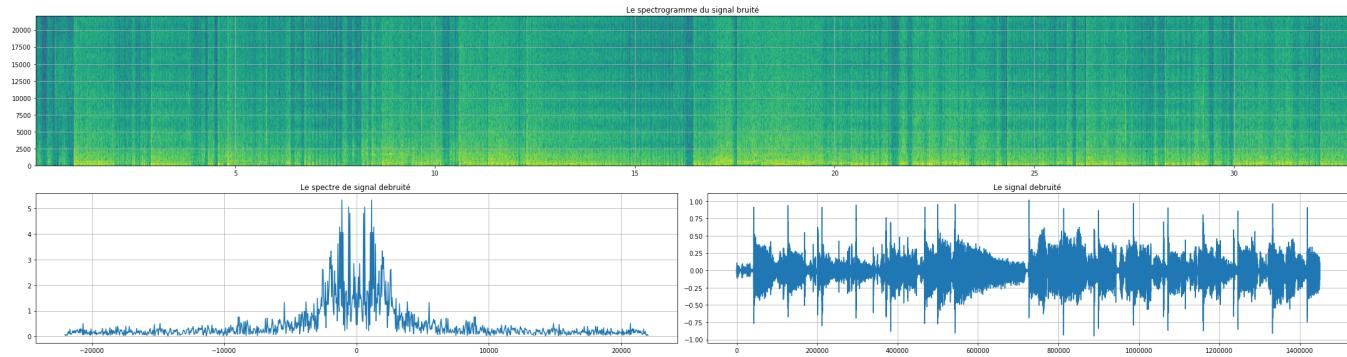
\color{green}{Question~9}: La~deuxième~méthode~de~débruitage}

Dans cette question, nous allons découper le morceau musical de 32s en des fenêtres d'analyse de 3ms chacune et on va appliquer le filtre sur chaque fenêtre à part en recalculant les coefficients de h_{opt} à chaque fois puis reconstituer le signal de nouveau:

```
In [52]: Signal_debrui_squence=[]
for i in tqdm(range(0,len(signal),135)):
    if i+135< len(signal):
        X=signal_bruite[i:i+135]
        d=signal[i:i+135]
        h_opt,_=calcul_hopt(X,d,15,len(X))
    else:
        X=signal_bruite[i:]
        d=signal[i:i+135]
        h_opt,_=calcul_hopt(X,d,15,len(X))
    Signal_debrui_squence.extend(filter(X,h_opt))
clear_output()
```

```
In [53]: TFD2=sc.fft(Signal_debrui_squence,1024)
y_debruit_sequence=abs(np.fft.fftshift(TFD2))
```

```
In [54]: plt.subplots(figsize=(30,8))
plt.subplot(223)
plt.plot(f,y_debruit_sequence)
plt.title("Le spectre de signal débruité ")
plt.grid()
plt.subplot(211)
specgram=plt.specgram(Signal_debrui_squence, Fs=F)
plt.title("Le spectrogramme du signal bruité")
plt.grid()
plt.subplot(224)
plt.plot(Signal_debrui_squence)
plt.title("Le signal débruité ")
plt.grid()
plt.tight_layout()
```



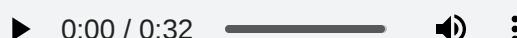
On remarque bien que le spectre, le spectrogramme et l'évolution temporelle sont désormais plus proches du signal original que le signal débruité par la première méthode en calculant le filtre optimal pour le signal entier.

Confirmons ceci en écoutant le morceau musical débruité:

```
In [55]: sf.write('Signal_debrui_squence_TP.wav',Signal_debrui_squence,Fs)
```

```
In [56]: pth1 ="Signal_debrui_squence_TP.wav"
ipd.Audio(pth1)
```

Out[56]:



En écoutant le morceau débruité par la deuxième méthode (en découpant le signal en des fenêtres d'analyse), on peut remarquer que la qualité de débruitage s'est nettement améliorée!

Interprétation:

L'analyse du signal par fenêtres est plus efficace car il tient compte de la non stationnarité du signal dans le morceau musical. Chaque 3ms a son h_{opt} approprié. Lorsqu'on calcule h_{opt} sur le signal en entier, on ne peut pas tenir en compte les particularités de certains morceaux très courts.

Maintenant, étudions l'effet de l'ordre L du filtre sur la qualité de débruitage:

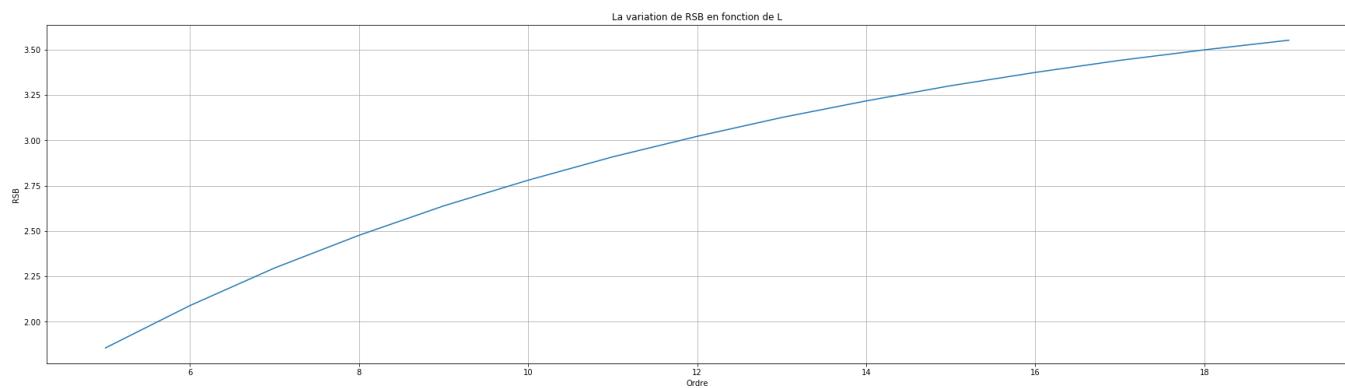
```
In [57]: def RSB(signal,signal_bruite,L):
    RSB_squence=[]
    for i in tqdm(range(0,len(signal),135)):
        if i+135< len(signal):
            X=signal_bruite[i:i+135]
            d=signal[i:i+135]
            h_opt,_=calcul_hopt(X,d,L,len(X))
            u=u_n[i:i+135]
        else:
            X=signal_bruite[i:]
            d=signal[i:]
            h_opt,_=calcul_hopt(X,d,L,len(X))
            u=u_n[i:]
            sequence=filter(X,h_opt)
            RSB_squence.append(10*np.log10(((np.array(sequence)**2).mean())/(np.array(filter(u,h_opt)))**2).mean()))
    return np.array(RSB_squence).mean()
RSB1=RSB(signal,signal_bruite,L)
clear_output()
RSB1
```

Out[57]: 2.6383936168956583

```
In [58]: list_L=[i for i in range(5,20)]
RSB_list=[]
for i in tqdm(range(len(list_L))):
    RSB_list.append(RSB(signal,signal_bruite,list_L[i]))
    clear_output()
```

100% |██████████| 15/15 [00:25<00:00, 1.70s/it]

```
In [59]: plt.subplots(figsize=(30,8))
plt.title("La variation de RSB en fonction de L")
sns.lineplot(list_L,RSB_list)
plt.xlabel("Ordre")
plt.ylabel("RSB")
plt.grid()
```



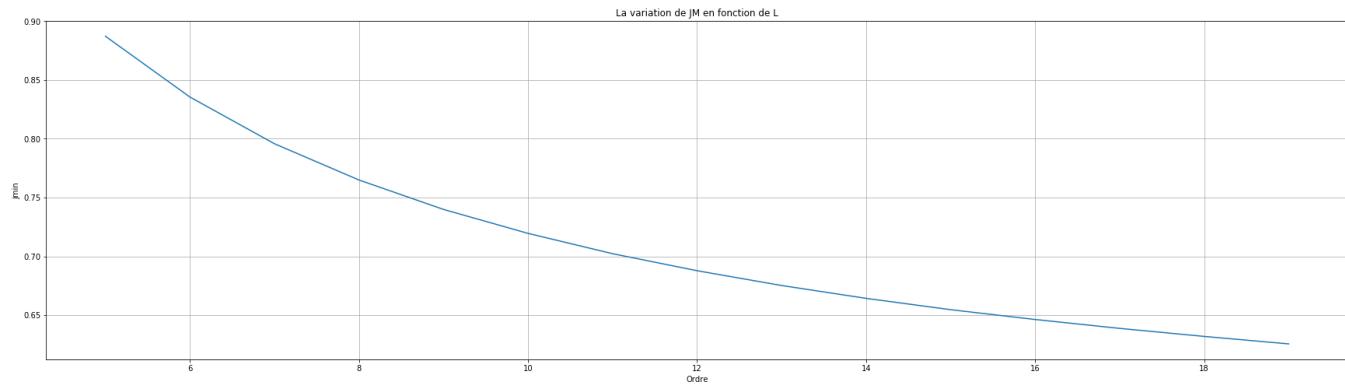
Comme dans la question 7, on voit que le rapport signal à bruit est croissant en fonction de l'ordre du filtre.

```
In [60]: def jmin(signal,signal_bruite,L):
    jmin_list1=[]
    for i in tqdm(range(0,len(signal),135)):
        if i+135< len(signal):
            X=signal_bruite[i:i+135]
            d=signal[i:i+135]
            R_d=np.correlate(d,d,mode='full')
            h_opt,P=calcul_hopt(X,d,L,len(d))
        else:
            X=signal_bruite[i:]
            d=signal[i:]
            R_d=np.correlate(d,d,mode='full')
            h_opt,P=calcul_hopt(X,d,L,len(d))
        jmin1=R_d[len(d)-1]-np.dot(h_opt.T,P)
        jmin_list1.append(jmin1)
    return np.array(jmin_list1)
```

```
In [61]: jmin_list=[]
for i in tqdm(range(len(list_L))):
    jmin_list.append(jmin(signal,signal_bruite,list_L[i]).mean())
    clear_output()
```

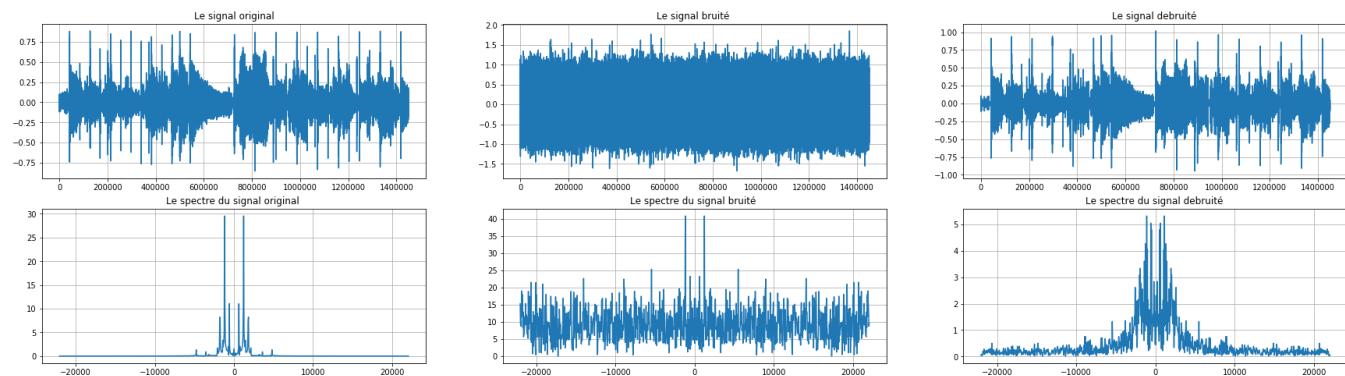
100% |██████████| 15/15 [00:13<00:00, 1.11it/s]

```
In [62]: plt.subplots(figsize=(30,8))
plt.title("La variation de JM en fonction de L")
sns.lineplot(list_L,jmin_list)
plt.xlabel("Ordre")
plt.ylabel("jmin")
plt.grid()
```



De même, plus l'ordre L du filtre est élevé, plus Jmin est faible et, par conséquent, plus la qualité de débruitage est meilleure.

```
In [63]: plt.subplots(figsize=(30,8))
plt.subplot(231)
plt.plot(signal)
plt.title("Le signal original")
plt.grid()
plt.subplot(232)
plt.plot(signal_bruite)
plt.title("Le signal bruité ")
plt.grid()
plt.subplot(233)
plt.plot(Signal_debrui_squence)
plt.title("Le signal débruité ")
plt.grid()
plt.subplot(234)
plt.plot(f,y)
plt.title("Le spectre du signal original ")
plt.grid()
plt.subplot(235)
plt.plot(f,y_bruit)
plt.title("Le spectre du signal bruité ")
plt.grid()
plt.subplot(236)
plt.plot(f,y_debruit_sequence)
plt.title("Le spectre du signal débruité ")
plt.grid()
```



Bonus:

Une autre approche pour le débruitage du morceau en entier

Dans les circonstances réelles, dans la réception, le filtre n'a pas accès au signal original. Sinon on n'aura plus besoin de ce filtre de débruitage.

Mais on peut envoyer une séquence pilote que l'émetteur et le récepteur connaissent. On envoie cette séquence pilote pour calculer h_{opt} du filtre. Les coefficients de h_{opt} obtenus tiennent en compte la particularité du canal d'émission et son bruit.

Dans notre cas, on va considérer le signal sinusoïdal de la 1ère question comme signal de référence (signal pilote) pour calculer h_{opt} qu'on va utiliser pour débruiter le morceau musical.

\color{green}{Question~8}: La première méthode de débruitage

{La première méthode de débruitage consiste à débruiter la totalité du signal, c'est-à-dire qu'on considère le signal comme un seul morceau}

\large{Maintenant on va calculer h_{opt2} pour $L=L_{\text{opt}}$ }

```
In [64]: h_opt2, _ = calcul_hopt(X_n, d_n, L_opt, 1000)
```

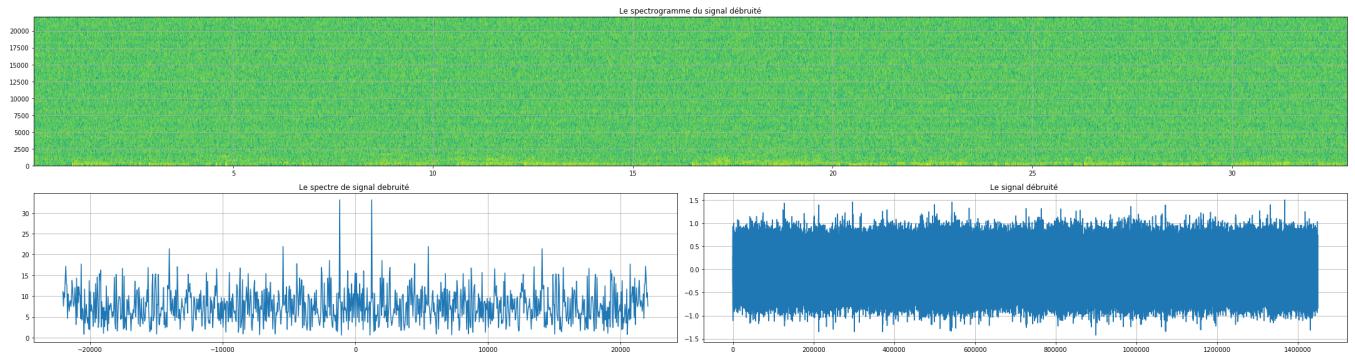
\large{Ici on va débruité le signal en utilisant notre filtre}

```
In [65]: Signal_debrui=filter(signal_bruite, h_opt2)
```

\large{Maitenant on va traçer le signal débruité puis sa TFD et son spectrogramme}

```
In [66]: TFD2=sc.fft(Signal_debrui, 1024)
y_debruit=abs(np.fft.fftshift(TFD2))
```

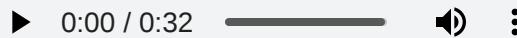
```
In [67]: plt.subplots(figsize=(30,8))
plt.subplot(223)
plt.plot(f,y_debruit)
plt.title("Le spectre de signal débruité ")
plt.grid()
plt.subplot(211)
specgram=plt.specgram(Signal_debrui, Fs=F)
plt.title("Le spectrogramme du signal débruité")
plt.grid()
plt.subplot(224)
plt.plot(Signal_debrui)
plt.title("Le signal débruité ")
plt.grid()
plt.tight_layout()
```



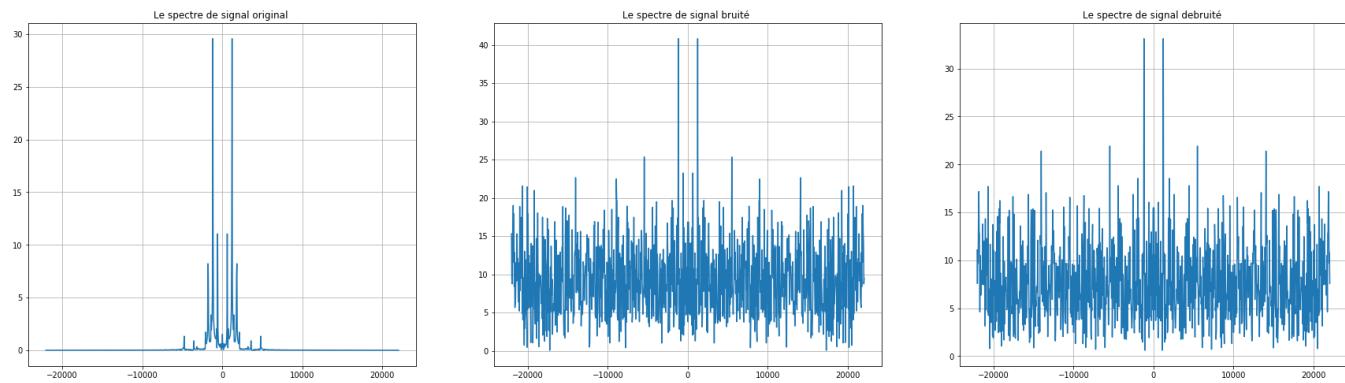
```
In [68]: sf.write('Signal_debrui.wav',Signal_debrui,Fs)
```

```
In [69]: pth1 ="signal_debrui.wav"
ipd.Audio(pth1)
```

Out[69]:



```
In [70]: plt.subplots(figsize=(30,8))
plt.subplot(131)
plt.plot(f,y)
plt.title("Le spectre de signal original ")
plt.grid()
plt.subplot(132)
plt.plot(f,y_bruit)
plt.title("Le spectre de signal bruité ")
plt.grid()
plt.subplot(133)
plt.plot(f,y_debruit)
plt.title("Le spectre de signal débruité ")
plt.grid()
```



L'utilisation du filtre calculé à partir de la séquence pilote donne des résultats plus ou moins acceptables, malgré que le calcul de `h_opt` est effectué sur un morceau totalement "nouveau" pour le filtre.

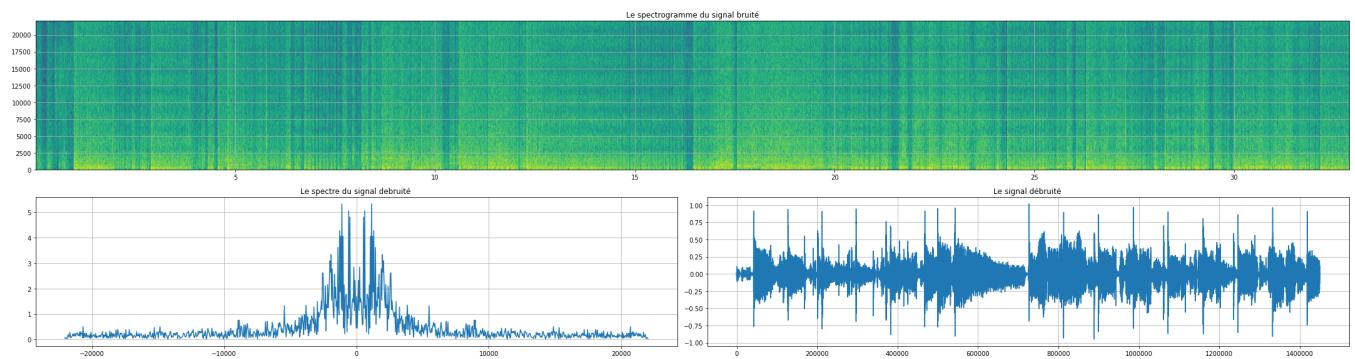
{\color{green}{Question~9}:La~deuxième~méthode~de~débruitage}

{Maintenant~on~va~débruiter~le~signal~mais~d'une~manière~séquentielle:~chaque~séquence~de~taille~3ms}
 {Après~tout~calcule~fait~on~a~trouvé~que~3ms~correspond~à~un~signal~qui~contient~135~valeurs}

```
In [71]: Signal_debrui_squnce2=[]
for i in tqdm(range(0,len(signal),135)):
    if i+135< len(signal):
        X=signal_bruite[i:i+135]
    else:
        X=signal_bruite[i:]
    Signal_debrui_squnce2.extend(filter(X,h_opt2))
clear_output()
```

```
In [72]: TFD3=sc.fft(Signal_debrui_squnce,1024)
y_debruit_sequence=abs(np.fft.fftshift(TFD3))
```

```
In [73]: plt.subplots(figsize=(30,8))
plt.subplot(223)
plt.plot(f,y_debruit_sequence)
plt.title("Le spectre du signal débruité ")
plt.grid()
plt.subplot(211)
specgram=plt.specgram(Signal_debrui_squence, Fs=F)
plt.title("Le spectrogramme du signal bruité")
plt.grid()
plt.subplot(224)
plt.plot(Signal_debrui_squence)
plt.title("Le signal débruité ")
plt.grid()
plt.tight_layout()
```



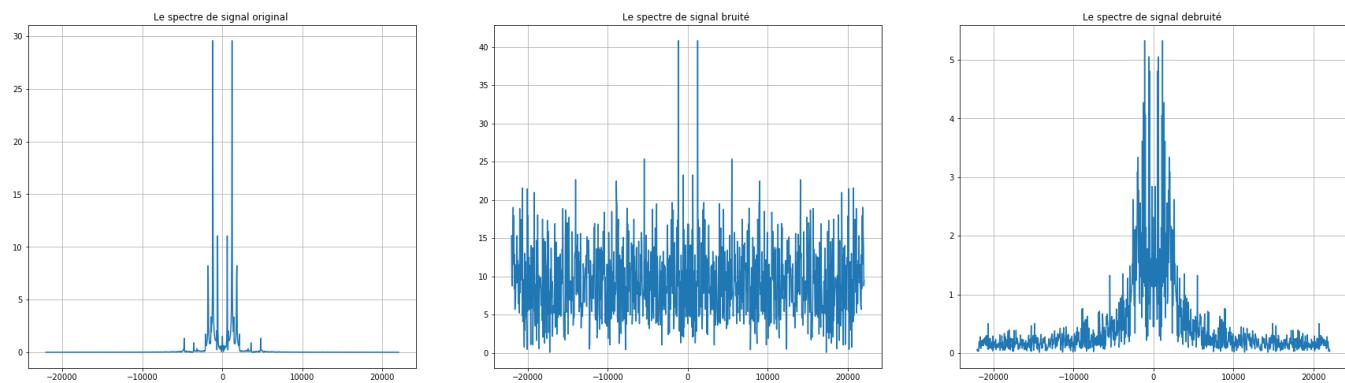
```
In [74]: sf.write('Signal_debrui_squence.wav',Signal_debrui_squence,Fs)
```

```
In [75]: pth1 ="Signal_debrui_squence.wav"
ipd.Audio(pth1)
```

Out[75]:



```
In [76]: plt.subplots(figsize=(30,8))
plt.subplot(131)
plt.plot(f,y)
plt.title("Le spectre de signal original ")
plt.grid()
plt.subplot(132)
plt.plot(f,y_bruit)
plt.title("Le spectre de signal bruité ")
plt.grid()
plt.subplot(133)
plt.plot(f,y_debruit_sequence)
plt.title("Le spectre de signal débruité ")
plt.grid()
```



Lorsqu'on utilise le filtre sur les différentes fenêtres mais avec le même h_{opt} , on n'a pas tenu en compte des particularités de chaque fenêtre de 3ms et de sa non stationnarité. C'est pour cela qu'on ne remarque pas qu'il y a amélioration de la qualité de débruitage.

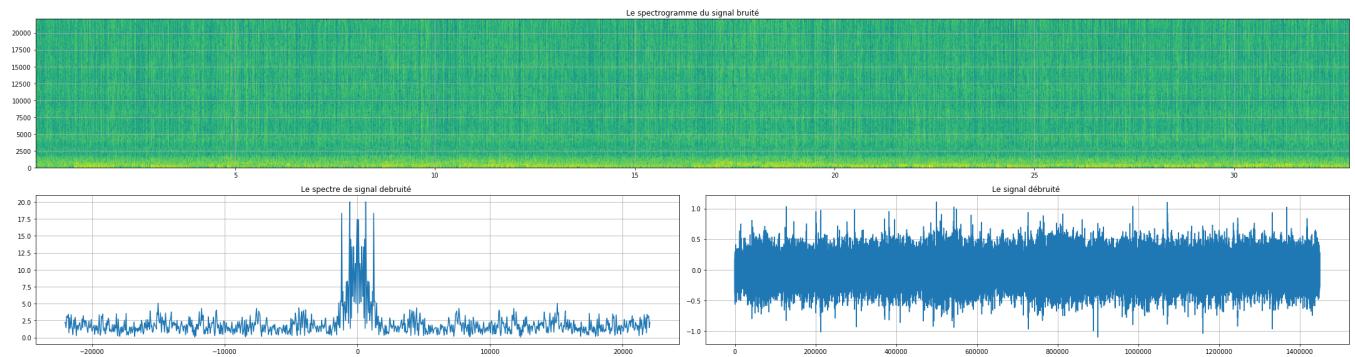
Le~filtre~prédefini~sur~scipy

La librairie scipy offre des filtres prédéfinis comme le filtre de Weiner. Essayons d'appliquer ce filtre prédéfini sur le signal:

```
In [77]: pre_defini=sc.signal.wiener(signal_bruit, mysize=25)
```

```
In [78]: TFD4=sc.fft(pre_defini,1024)
y_debruit_predefinie=abs(np.fft.fftshift(TFD4))
```

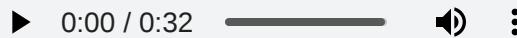
```
In [79]: plt.subplots(figsize=(30,8))
plt.subplot(223)
plt.plot(f,y_debruit_predefinie)
plt.title("Le spectre de signal débruité")
plt.grid()
plt.subplot(211)
specgram=plt.specgram(pre_defini, Fs=F)
plt.title("Le spectrogramme du signal bruité")
plt.grid()
plt.subplot(224)
plt.plot(pre_defini)
plt.title("Le signal débruité")
plt.grid()
plt.tight_layout()
```



```
In [80]: sf.write('pre_defini.wav',pre_defini,Fs)
```

```
In [81]: pth1="pre_defini.wav"
ipd.Audio(pth1)
```

Out[81]:



Filtre~de~Wiener~sur~les~images

Le filtrage inverse est une technique de restauration pour la déconvolution, c'est-à-dire que lorsque l'image est floue par un filtre passe-bas connu, il est possible de récupérer l'image par filtrage inverse ou filtrage inverse généralisé. Cependant, le filtrage inverse est très sensible au bruit additif. L'approche consistant à réduire une dégradation à la fois nous permet de développer un algorithme de restauration pour chaque type de dégradation et de les combiner simplement. {color:red}{Le filtrage~de~Wiener}} exécute un compromis optimal entre le filtrage inverse et le lissage du bruit. Il supprime le bruit additif et inverse le flou simultanément.

Le filtrage de Wiener est optimal en termes d'erreur quadratique moyenne. En d'autres termes, il minimise l'erreur quadratique moyenne globale dans le processus de filtrage inverse et de lissage du bruit. {color:blue}{Le-filtrage~de~Wiener~est~une~estimation~linéaire~de~l'image~d'origine}}. L'approche est basée sur un cadre stochastique. Le principe d'orthogonalité implique que le filtre de Wiener dans le domaine de Fourier peut être exprimé comme suit:

$$\boxed{\hat{D} = \frac{H^*(f_1, f_2)}{|H(f_1, f_2)|^2 + K} B(f_1, f_2)} \text{ avec } K = \frac{S_{xx}(f_1, f_2)}{S_{uu}(f_1, f_2)}$$

{et B(f₁, f₂) c'est la transformee de Fourier de l'image bruite}

où S_{xx}(f₁, f₂) et S_{uu}(f₁, f₂) sont respectivement les spectres de puissance de l'image d'origine et du bruit additif, et est le filtre de flou. Il est facile de voir que le filtre de Wiener a deux parties distinctes, une partie de filtrage inverse et une partie de lissage du bruit. Il effectue non seulement la déconvolution par filtrage inverse (filtrage passe-haut) mais supprime également le bruit avec une opération de compression (filtrage passe-bas).

```
In [82]: import os
import numpy as np
from numpy.fft import fft2, ifft2
from scipy.signal import gaussian, convolve2d
import matplotlib.pyplot as plt
```

Maintenant on va créer une fonction qui va ajouter à l'image original un bruit blanc additive

```
In [83]: def add_gaussian_noise(img, sigma):
    gauss = np.random.normal(0, sigma, np.shape(img))
    noisy_img = img + gauss
    noisy_img[noisy_img < 0] = 0
    noisy_img[noisy_img > 255] = 255
    return noisy_img
```

La fonction h_filter va créer la fonction transfer d'un filtre gaussien

```
In [84]: def h_filter(kernel_size = 3):
    h = gaussian(kernel_size, kernel_size / 3).reshape(kernel_size, 1)
    h = np.dot(h, h.transpose())
    h /= np.sum(h)
    return h
```

wiener_filter dans cette fonction on va implementé la formule encadré ci dessus

```
In [85]: def wiener_filter(img, kernel, K):
    kernel /= np.sum(kernel)
    dummy = np.copy(img)
    dummy = fft2(dummy)
    kernel = fft2(kernel, s = img.shape)
    kernel = np.conj(kernel) / (np.abs(kernel) ** 2 + K)
    dummy = dummy * kernel
    dummy = np.abs(ifft2(dummy))
    return dummy
```

```
In [86]: def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])
```

In [87]:

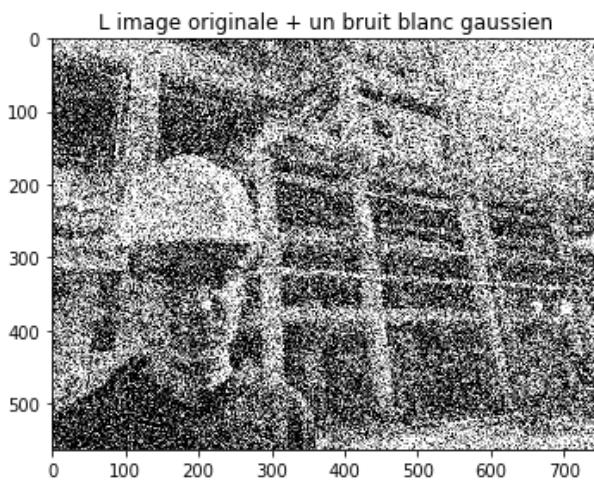
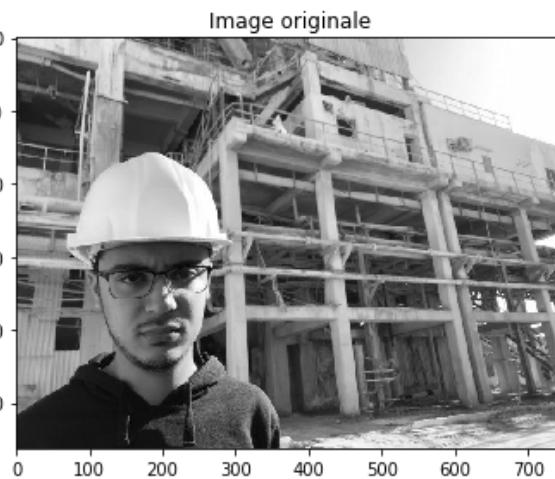
```
# Load image and convert it to gray scale
file_name = os.path.join('the_happy_ahmed.jpg')
img = rgb2gray(plt.imread(file_name))

# Add Gaussian noise
noisy_img = add_gaussian_noise(img, sigma = 100)

# Apply Wiener Filter
kernel = h_filter(5)
filtered_img = wiener_filter(noisy_img, kernel, K = 20)

# Display results
display = [img,noisy_img, filtered_img]
label = ['Image originale','L image originale + un bruit blanc gaussien', 'Débruitage par le filtre de Wiener']

plt.subplots(figsize=(12,8))
plt.subplot(223)
plt.imshow(display[1], cmap = 'gray')
plt.title(label[1])
plt.subplot(211)
plt.imshow(display[0], cmap = 'gray')
plt.title(label[0])
plt.subplot(224)
plt.imshow(display[2], cmap = 'gray')
plt.title(label[2])
plt.tight_layout()
```



Conclusion:

Ce projet a été une bonne opportunité pour découvrir les filtres sur le plan pratique et de les appliquer sur les différents signaux afin de voir leurs effets. Le filtre de Weiner a été publié en 1949. Qu'en est il des autres filtres plus sophistiqués et plus récents?