TP4_CompressiveSensing

January 14, 2026

1 TP 4 - Echantillonnage compressif

Récemment (début années 2004-présent), de nouveaux concepts et théorèmes ont été développés et risquent de révolutionner à relativement court terme la fabrication de certains appareils de mesure numériques (microphones, imageurs, analyseurs de spectres,...). Ces nouvelles techniques sont couramment appelées échantillonnage compressif, "compressive sampling" ou encore "compressed sensing". Nous les décrivons rapidement ci-après. Vous trouverez un tutoriel bien plus complet (sous forme de présentation) à l'adresse http://users.ece.gatech.edu/~justin/ssp2007/ssp07-cs-tutorial.pdf

1.1 1. Le théorème de Shannon

Aujourd'hui, presque tous les appareils de mesure reposent sur le théorème de Shannon. Celui-ci (vous l'avez déjà vu en 2ème année) peut s'énoncer ainsi :

```
Soit $g:\mathbb{R}\to \mathbb{R}$ une fonction de $L^2(\mathbb{R})$. Si sa transformée de Four
```

Ce théorème est illustré sur les figures ci-après:

[16]:
```
# Imports et configuration du notebook
# - numpy: calcul numérique
# - math: fonctions mathématiques (non utilisé intensivement ici)
# - scipy.fftpack: DCT/IDCT utilisées pour la base cosinus
# - matplotlib.pyplot: tracés
import numpy as np
import math
import scipy.fftpack as fft
import matplotlib.pyplot as plt

# Permet de recharger automatiquement les modules modifiés sans relancer
# le kernel (pratique lors du développement et des tests interactifs)
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Les instruments de mesures qui reposent sur ce théorème sont donc construits suivant le principe :

Filtre passe-bas $\rightarrow$ Echantillonnage à une fréquence $f > 2f_M$ $\rightarrow$ Interpolation sinc

Pour beaucoup d'applications, ce principe présente deux défauts majeurs :

- Les signaux sont rarement naturellement à spectre borné, et on perd donc l'information haute-fréquence en effectuant un filtrage passe-bas.
- Pour beaucoup de signaux, il faut choisir une très haute fréquence d'échantillonnage pour obtenir un résultat satisfaisant. Ceci implique que les données à stocker ont une taille très importante et qu'il faut les compresser après coup (par exemple : jpeg).

1.2 2. L'échantillonnage compressif

1.2.1 2.1 Principe général

L'idée sous jacente à l'échantillonnage compressif est de réaliser la compression dès l'acquisition. Supposons que le signal $x \in \mathbb{R}^n$ que l'on souhaite mesurer s'écrive comme une combinaison linéaire de la forme :

$$x = \sum_{i=1}^{m} \alpha_i \psi_i \tag{1}$$

où $\psi_i \in \mathbb{R}^n,\ i = 1..m$, sont des "fonctions de base" (en traitement d'images, ces fonctions pourraient être des ondelettes, en traitement du son, des ondelettes ou des atomes de Fourier, pour certaines applications, on pourrait imaginer des splines...} et $\alpha_i \in \mathbb{R}$ sont des coefficients. On peut réécrire l'équation (1) sous la forme matricielle condensée :

$$x = \Psi\alpha \quad \text{où} \quad \alpha = \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{pmatrix} \quad \text{et} \quad \Psi = \left(\psi_1, \psi_2, ..., \psi_m\right).$$

Pour pouvoir reconstruire tous les éléments de $\mathbb{R}^n$, on suppose généralement que la matrice Ψ est une matrice surjective (ainsi, la famille des $(\Psi_i)_i$ est génératrice), ce qui implique que $m \geq n$. Dans le langage du traitement d'image, on dit alors que Ψ est un frame (une base si $m = n$).

L'échantillonnage compressif repose sur l'hypothèse suivante : les signaux x que l'on souhaite mesurer sont parcimonieux, c'est-à-dire que la majorité des coefficients α_i dans (1) sont nuls ou encore que

$$\#\{\alpha_i \neq 0, i = 1..m\} \ll n.$$

On va voir que cette hypothèse permet - dans certains cas - de réduire drastiquement le nombre de mesures par rapport au théorème de Shannon avec en contre-partie, le besoin de résoudre un problème d'optimisation pour reconstruire la donnée. L'objectif de ce TP est de résoudre le problème d'optimisation résultant.

Le principe de l'acquisition du signal x est le suivant :

- On effectue un petit nombre $p \ll n$ de mesures linéaires du signal x inconnu. On note ces mesures y_i, et comme elles sont linéaires par rapport à x, il existe pour chaque i un vecteur $a_i \in \mathbb{R}^n$ tel que

  $$y_i = \langle a_i, x \rangle, i = 1..p.$$

  On peut aussi écrire cette opération de mesure sous la forme condensée :

  $$y = Ax \quad \text{où} \quad y = \begin{pmatrix} y_1 \\ \vdots \\ y_p \end{pmatrix} \quad \text{et} \quad A = \begin{pmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_p^T \end{pmatrix}.$$

- On reconstruit le signal x en résolvant le problème contraint suivant :

$$(2) \qquad \text{Trouver } \alpha^\star \text{ solution de: } \min_{\alpha\in\mathbb{R}^m, A\Psi\alpha=y} \|\alpha\|_0$$

où $\|\cdot\|_0$ est la norme de comptage, aussi appelée norme l^0 définie par :

$$\|\alpha\|_0 = \#\{\alpha_i \neq 0, i = 1..m\}.$$

Autrement dit, l'idée est la suivante : on cherche $\alpha^\star$, le signal le plus parcimonieux dans le frame Ψ, parmi les signaux qui peuvent donner lieu aux mesures y. Après avoir trouvé $\alpha^\star$, on recouvre $\tilde{x}$, une approximation du signal x en calculant $\tilde{x} = \Psi\alpha^\star$.

1.3 2.2. Simplification du problème d'optimisation

Le problème précédent est un problème combinatoire NP-complet, ce qui signifie que trouver α peut demander un temps exponentiel en fonction de n, la dimension du signal. Pour le résoudre en pratique, il est souvent remplacé par :

$$(3) \qquad \text{Trouver } \alpha^* \in \arg\min_{\alpha\in\mathbb{R}^m, A\Psi\alpha=y} \|\alpha\|_1$$

où $\|\alpha\|_1 = \sum_{i=1}^m |\alpha_i|$ est la norme l^1 de α. On peut dans certains cas montrer que les solutions de (2) et de (3) sont identiques.

Un appareil de mesure n'étant jamais parfait, il est impossible de mesurer exactement $y_i = \langle a_i, x\rangle$. Le vecteur y est bruité et la contrainte $A\Psi\alpha = y$ est trop forte. Elle est donc généralement relaxée et le problème devient :

$$(4) \qquad \text{Trouver } \alpha^* \in \arg\min_{\alpha\in\mathbb{R}^m} \|\alpha\|_1 + \frac{\sigma}{2}\|A\Psi\alpha - y\|_2^2.$$

Si σ tend vers 0, la solution du problème (4) tend vers une solution du problème (3). C'est le problème (4) que nous allons résoudre dans ce TP. Dans la suite , on notera F la fonction :

$$F(\alpha) = \|\alpha\|_1 + \frac{\sigma}{2}\|A\Psi\alpha - y\|_2^2.$$

Pour conclure cette introduction à l'échantillonnage compressif, notons que de façon similaire au théorème de Shannon, on dispose d'une condition de reconstruction exacte :

Supposons que :

$x = \sum_{i=1}^m \alpha_i\psi_i \in \mathbb{R}^n$ avec $\|\alpha\|_0 = k$.

On effectue p mesures linéaires de x avec $p \geq C \cdot k \cdot \log(n)$, où $C = 20$.

On choisit les coefficients de la matrice $A \in \mathcal{M}_{p,n}$ de façon aléatoire (e.g. on peut choisir les coefficients $a_{i,j}$ de A de façon indépendante suivant une loi normale.)

```
Alors, la résolution du problème (3) permet de reconstruire $x$ <b>exactement</b> avec une très
```

L'expérience a montré qu'en pratique, il suffit en général de $p = 2k$ mesures pour reconstruire le signal exactement en grande dimension !

1.4 3. Préliminaires théoriques

Commençons par remarquer que les problèmes (3) et (4) sont convexes (contraintes convexes et fonctions convexes) tandis que le problème (2) ne l'est pas. En revanche, aucun des trois problèmes n'est différentiable.

Q1. Soit $J(\alpha) = \frac{\sigma}{2}\|A\Psi\alpha - y\|_2^2$. Calculez $\nabla J(\alpha)$.

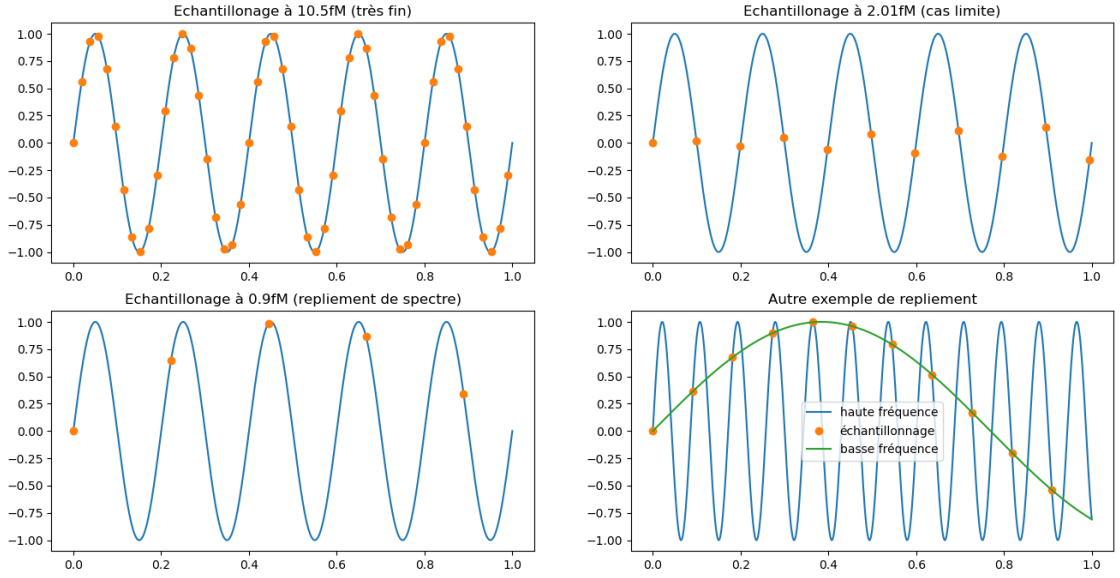
$$\sigma A\Psi^T(A\Psi\alpha - y)$$

Q2. Montrer que la fonction J est de classe C^1 à gradient Lipschitz et montrer que $\sigma|||A|||^2|||\Psi|||^2$ est un majorant de L, la constante de Lipschitz de ∇J. On rappelle que

$$|||B|||^2 = \sup_{x\neq 0}\frac{\|Bx\|_2^2}{\|x\|_2^2} = \sup_{x\neq 0}\frac{(B^TBx, x)}{(x,x)} = \sup_{\lambda\in Sp(B^TB)}(\lambda)$$

fait sur papier

[17]:
```
# Affichage illustratif du théorème de Shannon (figure et démonstration visuelle)
# La fonction shannon() produit des graphiques d'exemple pour comprendre
# l'idée du théorème d'échantillonnage de Shannon.
from Shannon import shannon
shannon()
```



Dans la suite on prendra

$$L = 2 * \sigma * \text{np.sum}(A * A)$$

Démonstration :

Effectivement on verra que α se décompose en $\alpha = \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix}$ et que $\Psi\alpha = \Psi_1\alpha_1 + \Psi_2\alpha_2$ où Ψ_1 et Ψ_2 sont des isométries. Ainsi $\|\Psi\alpha\| \leq \|\alpha_1\| + \|\alpha_2\|$ On utilisera ensuite

$$\begin{aligned} \|\Psi\alpha\|^2 &\leq \left(\|\alpha_1\| + \|\alpha_2\|\right)^2 \\ &\leq \|\alpha_1\|^2 + \|\alpha_2\|^2 + 2\|\alpha_1\|\|\alpha_2\| \\ &\leq 2\left(\|\alpha_1\|^2 + \|\alpha_2\|^2\right) = 2\|\alpha\| \end{aligned}$$

On peut en conclure que

$$|||\Psi||| \leq 2.$$

Pour $|||A|||$, on utilise

$$sup_{\lambda \in Sp(A^T A)}(\lambda) \leq \sum_{\lambda \in Sp(A^T A)} \lambda = tr(A^T A) = \sum_{i,j} A_{i,j}^2$$

On obtient donc au final $L \leq 2 * \sigma * \text{np.sum}(A * A)$.

Construction de l'algorithme

On note α^k l'itéré courant. En appliquant le lemme de Nesterov à la fonction J, on a:

$$\forall \alpha \in \mathbb{R}^m,\ J(\alpha) \leq J(\alpha^k) + \langle \nabla J(\alpha^k), \alpha - \alpha^k \rangle + \frac{L}{2}\|\alpha - \alpha^k\|_2^2.$$

En posant $\phi(\alpha, \alpha^k) = J(\alpha^k) + \langle \nabla J(\alpha^k), \alpha - \alpha^k \rangle + \frac{L}{2}\|\alpha - \alpha^k\|_2^2 + \|\alpha\|_1$, on a alors:

$$\forall \alpha \in \mathbb{R}^m,\ F(\alpha) = J(\alpha) + \|\alpha\|_1 \leq \phi(\alpha, \alpha^k),$$

avec : $\phi(\alpha^k, \alpha^k) = F(\alpha^k)$.

Cette inégalité motive alors l'algorithme de descente suivant:

$$\alpha^{k+1} = \arg\min_{\alpha \in \mathbb{R}^m} \phi(\alpha, \alpha^k).$$

Q3. Montrer que l'algorithme s'écrit de façon équivalente sous la forme:

$$\alpha^{k+1} = \text{prox}_{\frac{1}{L}\|\cdot\|_1}\left(\alpha^k - \frac{1}{L}\nabla J(\alpha^k)\right).$$

$$0 \in \partial F(x) \Leftrightarrow x = prox_{th}(x - t\nabla f(x))$$

avec

$$t = 1/L$$

et $h = ||.||$

Q4. En déduire la formule analytique donnant α^{k+1} en fonction de α^k.

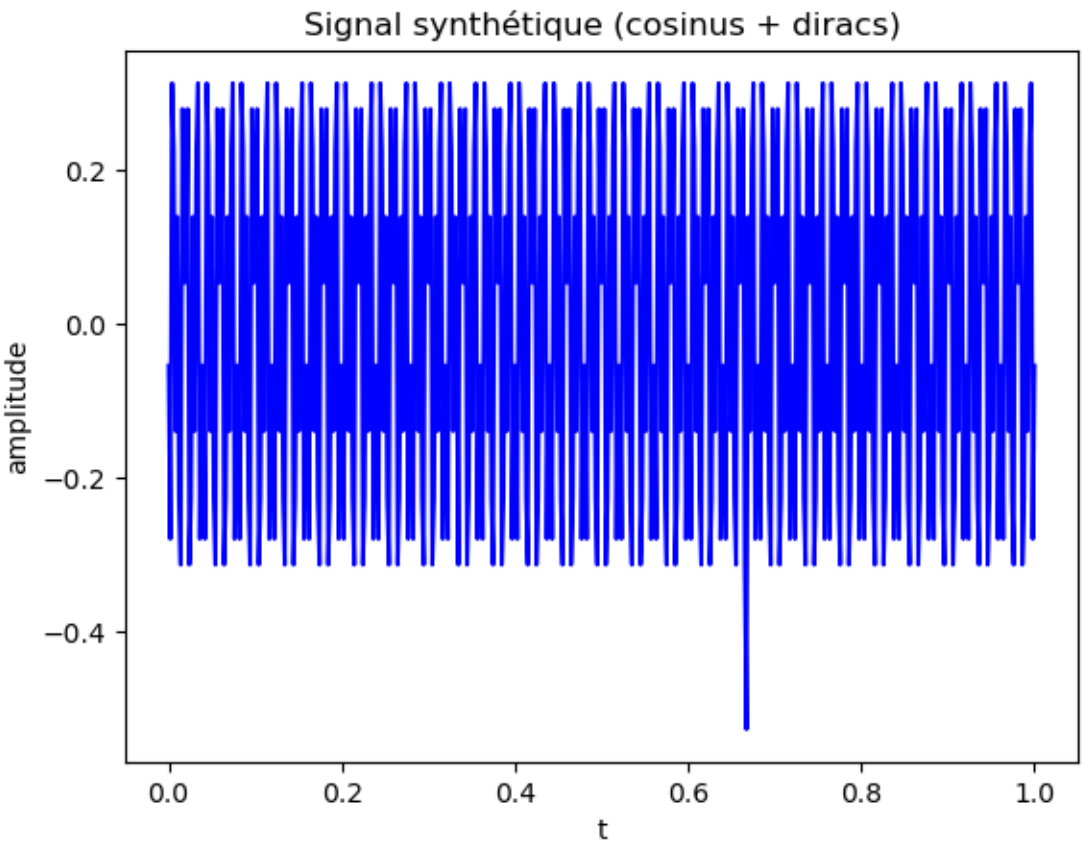
Votre réponse ici

Q5. De quelle quantité la fonction coût $F(\alpha)$ décroît-elle à chaque itération ?

Votre réponse ici

1.5 4. Partie expérimentale

[18]:
```
## Initialisations et génération d'un signal synthétique
# - On crée un signal composé de quelques cosinus (IDCT) et de deux diracs.
# - Ce signal est parcimonieux dans le frame "I + cosinus".

n=500                # Taille de l'échantillon
t=np.linspace(0,1,n) # Abscisses pour affichage

## Generation du signal
x=np.zeros(n)
# vecteur temporaire pour placer un coefficient dans la base cosinus
tmp=np.zeros(n)
# On ajoute deux cosinus (via l'IDCT orthonormée)
tmp[350]=4
x+=fft.idct(tmp,norm='ortho')
# un autre cosinus
tmp=np.zeros(n)
tmp[150]=-3
x+=fft.idct(tmp,norm='ortho')
# On ajoute deux diracs (éléments de la base canonique)
x[int(n/3)]+=0.2
x[int(2*n/3)]+=-0.3
# Visualisation rapide
plt.plot(t,x,'b')
plt.title('Signal synthétique (cosinus + diracs)')
plt.xlabel('t')
plt.ylabel('amplitude')
plt.show()

## Mesure du signal (A est la matrice d'observation)
p=2    # nombre de mesures (très petit ici pour l'illustration)
np.random.seed(42)
A=np.random.randn(p,n) # matrice de mesure aléatoire
y=A.dot(x)         # mesures (observations)
# y contient les données observées utilisées pour la reconstruction
```



Le code *Generesignal.py* génère un signal discret x qui peut être vu comme une combinaison linéaire de cosinus à différentes fréquences et de diracs. Ce signal n'est pas parcimonieux dans la base canonique des diracs (car il faut à peu près n diracs pour représenter un cosinus) et il n'est pas parcimonieux dans la base des sinus (il faut faire une combinaison linéaire de n cosinus pour représenter un dirac).

Par contre, ce signal est parcimonieux dans un frame qui est l'union de la base canonique et de la base des cosinus. Dans ce frame, il suffit en effet de 4 coefficients non nuls pour reconstruire parfaitement le signal.

On choisira donc le frame représenté par une matrice $\Psi = \begin{pmatrix} I & C \end{pmatrix} \in \mathcal{M}_{n,2n}(\mathbb{R})$ où C est une base de cosinus à différentes fréquences.

1.5.1 4.1. Implémentation de l'itération proximale

Q6 Implémentez l'opérateur linéaire Ψ et son adjoint Ψ^*.

Pour Ψ, vous vous servirez de la fonction `idct` de Python dans la libraire `scipy.fftpack` qui calcule la transformée en cosinus discret d'un vecteur. Vous ferez attention à préciser `norm='ortho'` dans les options de la `idct` pour que `dct` soit bien l'opération inverse de `idct`.

Pour Ψ^*, vous utiliserez le fait que la `idct` est une isométrie quand on précise `norm='ortho'` dans les options de `idct`. C'est-à-dire que `idct` et `dct` sont adjoints.

[19]:
```
## Linear operator Psi and its adjoint PsiT
# Psi(alpha) = a1 + IDCT(a2) where alpha = [a1 (dirac), a2 (cosinus)]
# PsiT(x) returns [x ; DCT(x)] which is the adjoint (DCT is adjoint of IDCT
# when using norm='ortho').

def Psi(alpha):
    """Applique Psi à alpha et renvoie le signal x.

    alpha: vecteur de taille 2n (première moitié coefficients diracs, deuxième␣
 ↪moitié cosinus)
    """
    alpha = np.asarray(alpha)
    a1 = alpha[:n]            # partie Dirac
    a2 = alpha[n:]            # partie cosinus

    # IDCT orthonormée (reconstruction de la composante cosinus)
    x2 = fft.idct(a2, norm="ortho")

    return (a1 + x2)


def PsiT(x):
    """Adjoint de Psi : renvoie [x, DCT(x)]."""
    x = np.asarray(x)

    # Première partie : identité (coefficients dirac)
    a1 = x.copy()

    # Deuxième partie : DCT orthonormée (coefficients cosinus)
    a2 = fft.dct(x, norm="ortho")

    # On concatène les deux blocs pour former un vecteur de taille 2n
    return np.concatenate([a1, a2])

# Petit test unitaire pour vérifier l'adjonction: <Psi(alpha), x> == <alpha,␣
 ↪PsiT(x)>
n=100
np.random.seed(12)
alpha=np.random.randn(2*n)
x=np.random.randn(n)
print('Inner product check: ',np.sum(Psi(alpha)*x),'=',np.sum(alpha*PsiT(x)))
```

```
Inner product check:  -0.25681865786765595 = -0.2568186578676559
```

Dans ce TP, on va chercher à reconstruire un signal unidimensionnel $x : [0,1] \to \mathbb{R}$ de la forme :

$$x(t) = \sum_{k=0}^{n} \alpha_k \delta_{k/n}(t) + \sum_{k=0}^{n} \alpha_{k+n} \cos\left(\frac{2k\pi}{n} t\right)$$

on a donc $m = 2n$. Dans le code ci-dessous, on génère un signal x de cette forme avec $\alpha = 0$ sauf pour 4 coefficients.

Qu'essaye t'on de faire avec le code qui est donné dans les dernières lignes de la cellulle précédente ?

on verifie que ça adjointe bien

Q7 Implémentez l'algorithme de calcul du prox de la norme L^1. la fonction `prox(alpha,s)` doit calculer le résultat de $\text{prox}_{s\|\bullet\|_1}(\alpha)$.

[20]:
```
def prox(alpha,s) :
    """Proximal operator (soft-thresholding) for s * ||.||_1.

    prox(alpha, s) = sign(alpha) * max(|alpha|-s, 0)
    """
    alpha = np.asarray(alpha)
    return np.sign(alpha) * np.maximum(np.abs(alpha) - s, 0.0)

# Tests simples pour vérifier le comportement du prox
n=2
np.random.seed(12)
alpha=np.random.randn(2*n)
print(prox(alpha,0.1)) # [ 0.37298583 -0.58142588  0.1424395  -1.60073563]
print(prox(alpha,0.3)) # [ 0.17298583 -0.38142588  0.         -1.40073563]
print('Psi test output (sample):',Psi(alpha)) # affiche Psi(alpha)
```

```
[ 0.37298583 -0.58142588  0.1424395  -1.60073563]
[ 0.17298583 -0.38142588  0.         -1.40073563]
Psi test output (sample): [-0.55818526  0.69260643]
```

Q7 Implémentez l'algorithme proximal dans la fonction `RestoreX(A,y,sigma,nit)` avec les notations suivantes: * `A` est la matrice d'échantillonnage. * `y` est le vecteur de mesures. * `sigma` est un paramètre du modèle. * `nit` est le nombre d'itérations qui est fixé ici (pas de critère d'arrêt).

Cette fonction rendra `alpha,x,CF` avec

- `alpha` est la solution approximative du problème (4).
- `x` est donné par `Psi(alpha)`.
- `CF` est la fonction coût à chaque itération de l'algorithme.

On initialisera `alpha` comme étant le vecteur nul.

[21]:
```
def J(alpha,sigma, A, y, Psi):
    """Partie quadratique J(alpha) = (sigma/2) * ||A Psi(alpha) - y||^2."""

    return((sigma/2)* np.linalg.norm(A@Psi(alpha)-y, 2)**2)


def RestoreX(A,y,sigma,nit)  :
    """Algorithme proximal pour résoudre
```

```
    min_alpha  ||alpha||_1 + (sigma/2) ||A Psi(alpha) - y||_2^2

    - A : matrice d'observation (p x n)
    - y : mesures (p,)
    - sigma : paramètre du modèle
    - nit : nombre d'itérations fixes

    Retourne: (alpha, x, CF) où x = Psi(alpha) et CF est la liste des coûts
    à chaque itération.
    """

    n = A.shape[1]
    alpha = np.zeros(2*n)  # initialisation

    # Lipschitz constant L approximée par 2*sigma*sum(A^2) (conforme au cours)
    L= np.sum(A*A)*2*sigma

    k=0
    CF =[]

    while k<nit :
        # calcul du gradient de la partie quadratique via l'adjoint PsiT
        r = A@Psi(alpha) - y
        grad = sigma*PsiT(A.T @ r)

        # pas proximal: soft-thresholding après un pas de gradient
        alpha = prox(alpha-(1/L)*grad ,1/L)
        x =Psi(alpha)
        k= k+1

        # stockage du coût total F = J + ||alpha||_1
        CF.append(J(alpha, sigma, A, y, Psi)+np.linalg.norm(alpha,1))

    return (alpha,x, np.asarray(CF))


# verification des calculs et tests unitaires
n=500       #Taille de x
p=80        #Nombre de mesures
alpha_target=np.zeros(2*n)
alpha_target[[166,333,650,850]]=[0.2,-0.3,-3,4]
x_target=Psi(alpha_target)
np.random.seed(42)
A=np.random.randn(p,n) #La matrice de mesure
y=A.dot(x_target)      #Les mesures
```

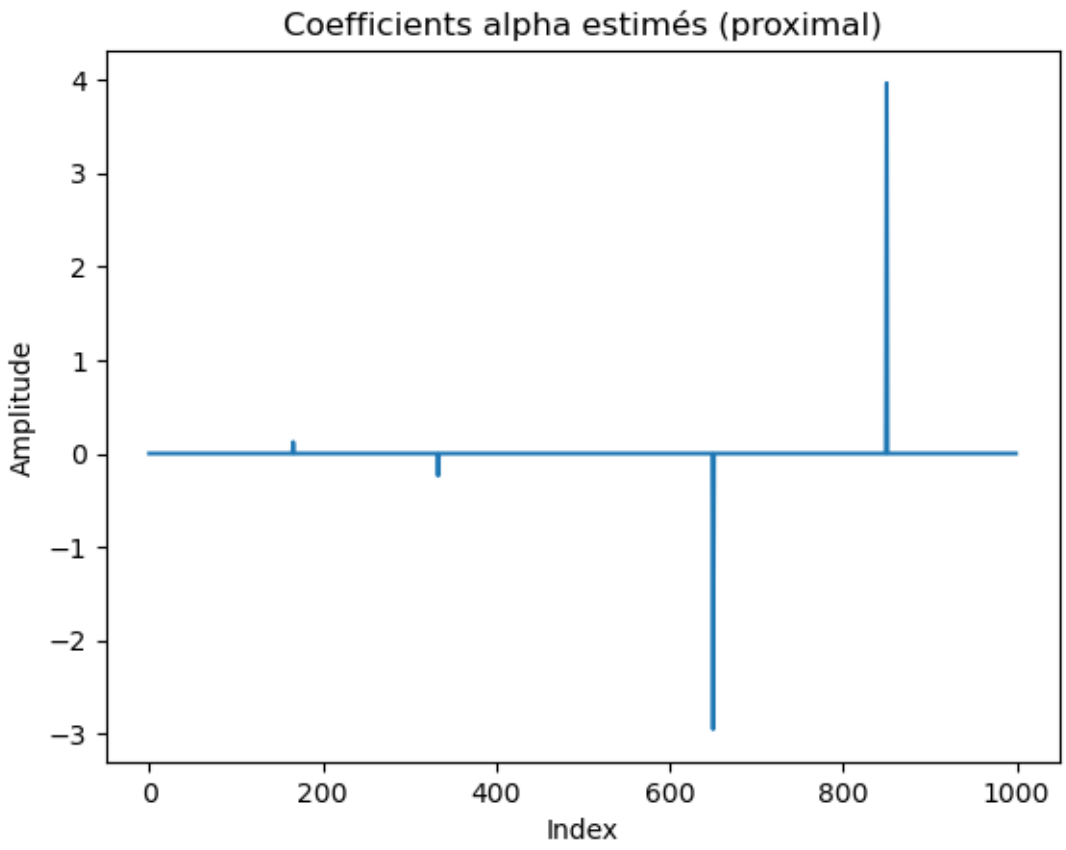
```
alpha,x,CF=RestoreX(A,y,np.pi,3)
print('(A,y) :',np.linalg.norm(A),np.linalg.norm(y))
print('shapes :', alpha.shape,x.shape,CF.shape)
print('CF :',CF)
print('True ?=?',np.allclose(Psi(alpha),x))
print('(alpha,x) :',np.linalg.norm(alpha),np.linalg.norm(x))
#(A,y) : 200.0215692529301 49.433723156789824
#shapes : (1000,) (500,) (4,)
#CF : [3838.54396488 3728.53381694 3622.05099603 3518.97398107]
#True ?=? True
#(alpha,x) : 0.062000521746624085 0.08768165277722285
```

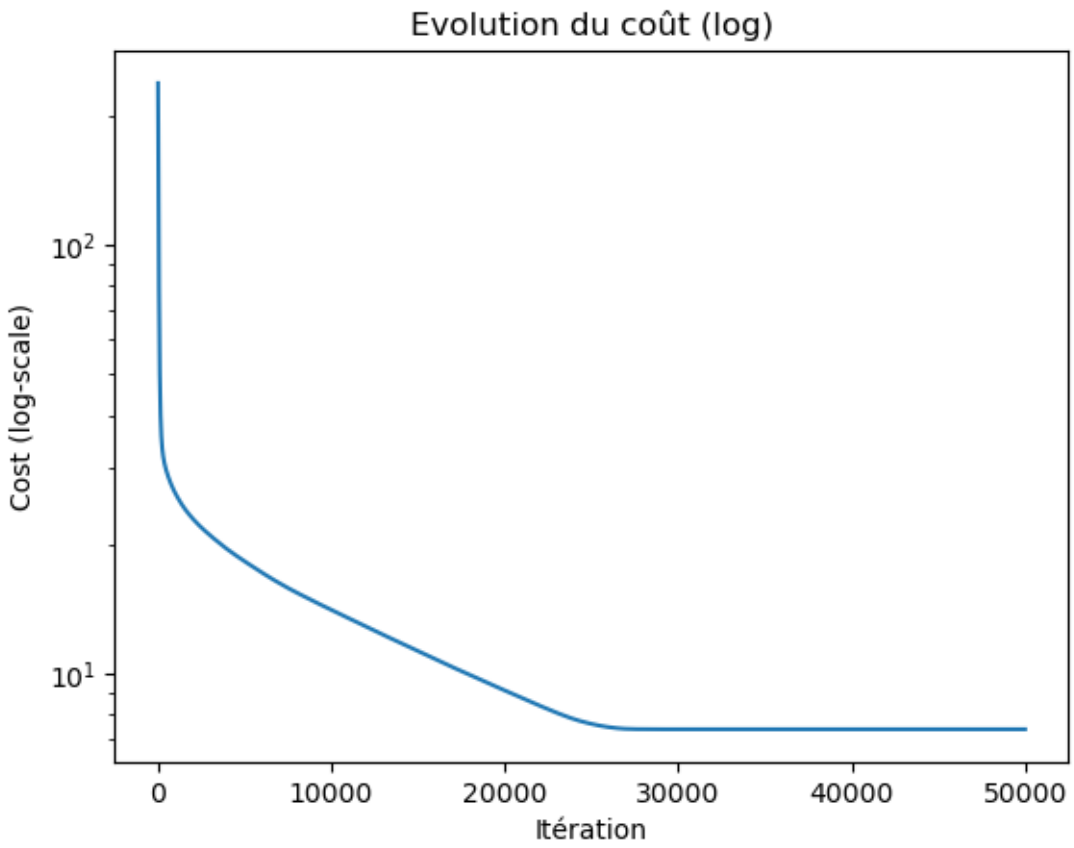
```
(A,y) : 200.02156925293014 49.433723156789824
shapes : (1000,) (500,) (3,)
CF : [3728.53381694 3622.05099603 3518.97398107]
True ?=? True
(alpha,x) : 0.062000521746624064 0.08768165277722283
```

Q8. Testez votre algorithme ! Les paramètres `sigma` et `nit` sont à choisir par vous-même (il faut en pratique beaucoup d'itérations pour converger). Vous pourrez observer la façon dont la suite α^k se comporte au fur et à mesure des itérations.

[ ]:

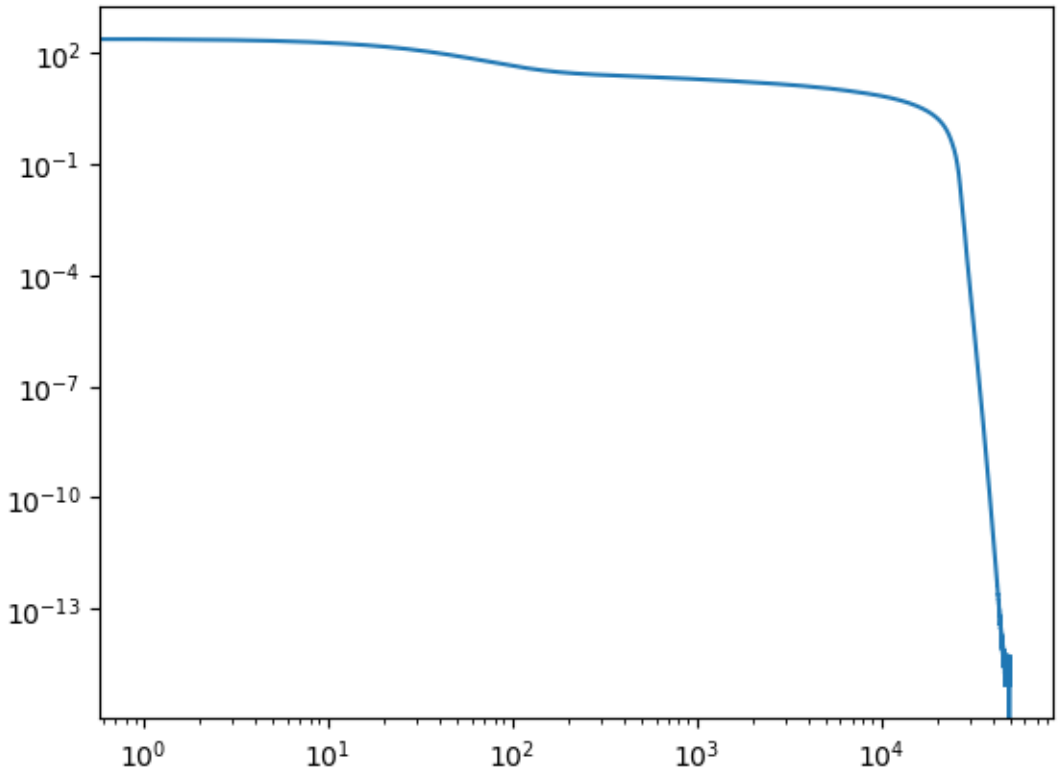
[22]:
```
# Test final : lancer la reconstruction sur (A,y) et afficher les coefficients␣
↪alpha
# Ici, on montre l'évolution de alpha après beaucoup d'itérations (nit grand)
(alpha,xtilde,CF)=RestoreX(A,y,0.2,50000)
plt.figure()
plt.plot(alpha)
plt.title('Coefficients alpha estimés (proximal)')
plt.xlabel('Index')
plt.ylabel('Amplitude')
plt.show()

# Affichage du coût (log) pour vérifier la décroissance
plt.figure()
plt.plot(CF)
plt.yscale('log')
plt.title('Evolution du coût (log)')
plt.xlabel('Itération')
plt.ylabel('Cost (log-scale)')
plt.show()
```


Coefficients alpha estimés (proximal)
Amplitude
Index
4
3
2
1
0
−1
−2
−3
0
200
400
600
800
1000



Q9. Vérifiez que la fonction coût décroit de façon monotone. Quel est le taux de convergence observé ?

[23]:
```
plt.loglog(CF-np.min(CF))


plt.show()
```



[29]:
```python
def convergence_rate(CF):
    CF = np.asarray(CF)
    rates = []

    for k in range(len(CF)-1):
        e_k = CF[k] - CF[-1]       # erreur à l'itération k
        e_k1 = CF[k+1] - CF[-1]    # erreur à l'itération k+1
        rates.append(e_k1 / e_k)

    return np.array(rates)

rates = convergence_rate(CF)
print("Taux moyen :", np.mean(rates))
```

```
Taux moyen : 0.9887142114841653
```

Votre réponse ici

1.5.2 4.2. Implémentation de l'itération proximale accelérée

On n'a a aucun moment utilisé la convexité de la fonction J pour définir l'algorithme proximal. Celui-ci est de fait sous-optimal et peut être nettement accéléré. Yurii Nesterov a proposé dans les années 1980 plusieurs méthodes permettant l'accélération de la descente de gradient explicite.

L'accélération de la descente de gradient proposée Yurii Nesterov en 1984 et adapatée à FB sous le nom de FISTA (Fast Iterative Soft Shrinckage Algorithm) par Beck et Teboulle en 2009 est d'une mise en oeuvre très simple: considérons à nouveau la fonction composite $F = J + \|\bullet\|_1$ à minimiser. L'algorithme FISTA s'écrit:

$$\begin{aligned} z_k &= \alpha_k + \beta_k(\alpha_k - \alpha_{k-1}) \\ \alpha_{k+1} &= \text{prox}_{s\|\bullet\|_1}(z_k - s\nabla J(z_k)) \end{aligned}$$

avec un pas $s \leq \frac{1}{L}$ et $\beta_k > 0$. On parle de méthode inertielle car cette méthode utilise un terme dit de "mémoire" ou inertiel qui exploite la dernière direction de descente.

Le choix original de Nesterov pour la suite β_k est le suivant :

$$\beta_k = \frac{t_k - 1}{t_{k+1}} \text{ avec } t_1 = 1 \text{ et } t_{k+1} = \frac{1 + \sqrt{1 + t_k^2}}{2} \tag{2}$$

Pour ce choix on a

$$F(\alpha_k) - F(\alpha^*) \leqslant \frac{2\|\alpha_0 - \alpha^*\|^2}{sk^2}$$

On peut prendre plus simplement

$$\beta_k = \frac{k-1}{k+2}$$

et dans ce cas, on a $F(\alpha_k) - F(\alpha^*) = o\left(\frac{1}{k^2}\right)$ et on a convergence de la suite $(\alpha_k)_{k\geqslant 1}$. On peut noter que dans ce cas, la première étape est sans inertie ($\beta_1 = 0$). L'inertie apparait pour le calcul de α_2.

A noter que la suite de terme général $F(\alpha_k) - F(\alpha^*)$ n'est pas nécessairemment décroissante comme dans le cas de FB ou de la descente de gradient. Dans la pratique vous verrez que FISTA est quand même plus rapide que FB.

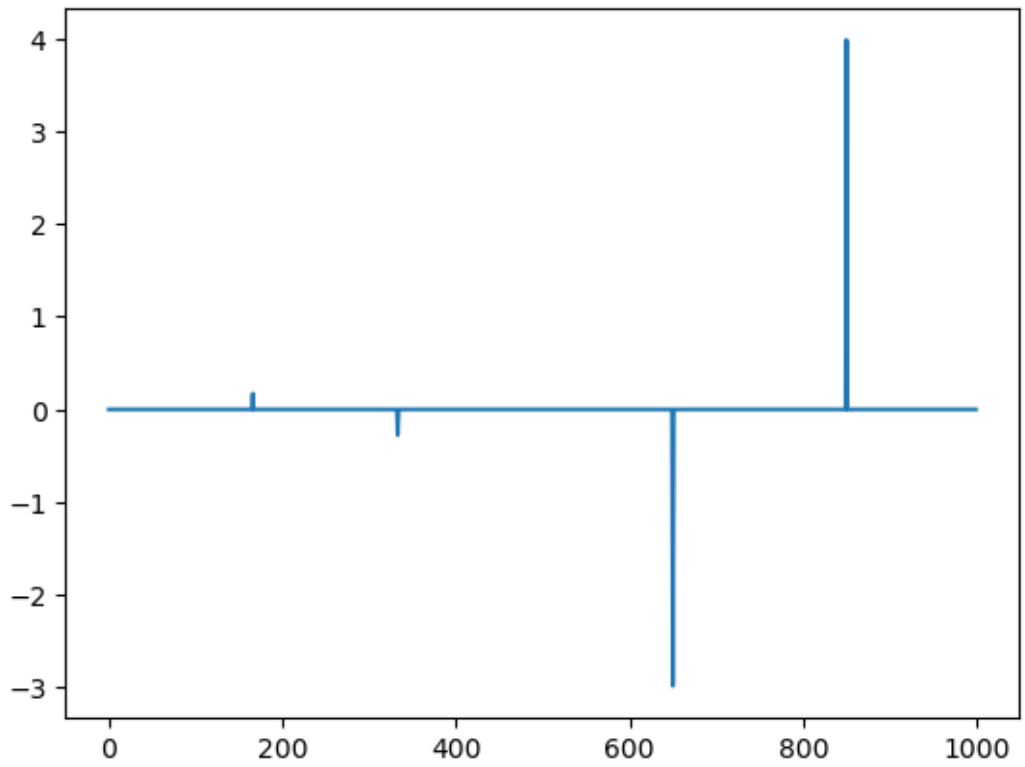
Q10. En vous aidant de ce que vous avez codé dans la partie précédente, implémentez cet algorithme.

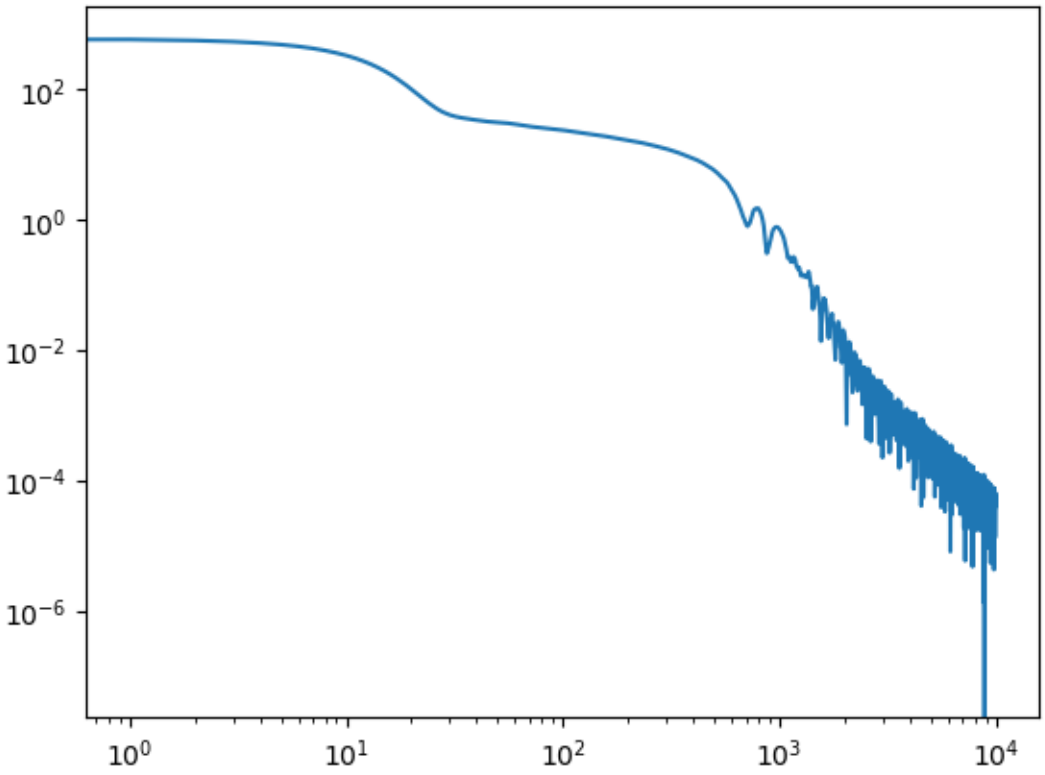
[25]:
```
def Nesterov(A,y,sigma,nit):

    n = A.shape[1]
    alpha = np.zeros(2*n)
    alpham1= np.zeros(2*n)
    CF= []
    k=0
    L= np.sum(A*A)*2*sigma

    while k<nit:

        Bk= (k-1)/(k+2)
        yk = alpha+Bk * (alpha-alpham1)
        alpham1 = alpha
```

```
        r = A@Psi(alpha) - y
        grad = sigma*PsiT(A.T @ r)

        alpha = prox(yk-(1/L)*grad ,1/L)

        yk =Psi(alpha)
        k= k+1
        CF.append(J(alpha, sigma, A, y, Psi)+np.linalg.norm(alpha,1))




    return (alpha,x,np.asarray(CF))
```

Q11. Testez le et comparez la rapidité d'execution de l'algorithme précédent et de celui-ci.

[26]:
```
nit=10000
(alpha,x,CF)=Nesterov(A,y,0.5,nit)
plt.plot(alpha)
plt.show()
plt.loglog(CF-np.min(CF))

plt.show()
```





[27]:
```python
def convergence_rate(CF):
    CF = np.asarray(CF)
    rates = []

    for k in range(len(CF)-1):
        e_k = CF[k] - CF[-1]       # erreur à l'itération k
        e_k1 = CF[k+1] - CF[-1]    # erreur à l'itération k+1
        rates.append(e_k1 / e_k)

    return np.array(rates)

rates = convergence_rate(CF)
print("Taux moyen :", np.mean(rates))
```

```
Taux moyen : 0.9887142114841653
```

Q12. A partir de combien de mesures pouvez-vous reconstruire exactement le signal x ?

Après essaie nous trouvons que p = 10 permet d'obtenir un signal satisfaisant

Q13. Faites un rapide résumé des points qui vous ont semblé les plus importants dans ce TP.

On a vu plein de choses :

Nesterov > BF

Calcul du prox de la norme 1

[ ]: