TP d'image

Guilhem Fouilhé-Lafforgue

Septembre 2022

Introduction

L'objectif de ce TP est de passer en revue les méthodes classiques de restoration d'images (debruitage et réduction du flou dans une image). Dans ce TP, nous considérons que le modèle de formation de l'image est le suivante (ce modèle est utilisé dans beaucoup d'applications) :

$$i(x,y) = h(x,y) * i_{ideal}(x,y) + b(x,y) \tag{1}$$

avec $*$ le produit de convolution, $b(x,y)$ le bruit et $h(x,y)$ le noyeau de convolution, qui ici introduit un flou dans l'image.

1 Réponses aux questions du TP

1.1

Dans le domaine de Fourier, l'équation (1) s'écrit :

$$I(x,y) = H(x,y)I_{ideal}(x,y) + B(x,y) \tag{2}$$

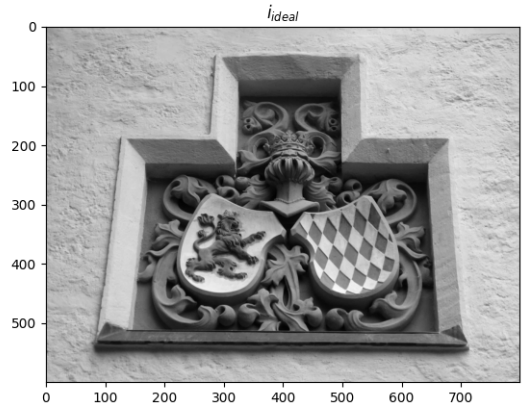
Le produit de convolution est transformé en multiplication. Le filtre agit donc comme un opérateur linéaire dans le domaine de Fourier.

1.2

On importe et affiche l'image en utilisant le code suivant :

```
import numpy as np
import matplotlib.pyplot as plt

i_ideal = plt.imread('coat_of_arms.png')

plt.imshow(i_ideal,cmap='gray')
plt.title('$i_{ideal}$')
plt.show()
```

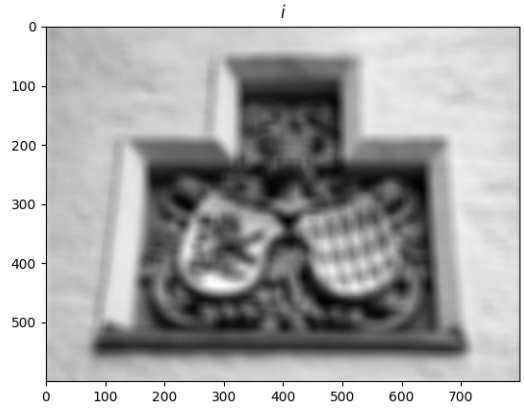
On obtient le résultat :



1.2.1

On génère l'image $i(x, y)$ en utilisant la fonction *scipy.signal.convolve2d* :

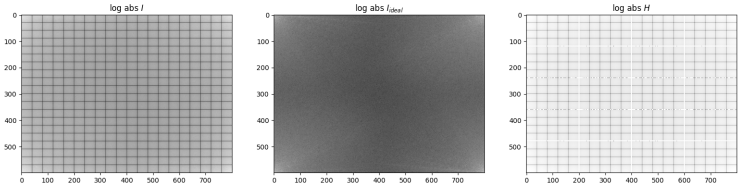
```
1
2  from scipy.signal import convolve2d
3
4  h = 1/(20**2)*np.ones((20,20))
5
6  i = convolve2d(i_ideal, h, mode='same', boundary='wrap')
7
8  plt.imshow(i,cmap='gray')
9  plt.title('$i$')
10 plt.show()
```



1.2.2

On utilise les fonctions *fft2, ifft2* du module *scipy.fft* :

```
from scipy.fft import fft2, ifft2

I = fft2(i)
I_ideal = fft2(i_ideal)
H = fft2(h, s=i.shape)

plt.figure(figsize =(10,30))

plt.subplot(1,3,1)
plt.imshow(np.log(abs(I)),cmap='gray')
plt.title('log abs $I$')
plt.subplot(1,3,2)
plt.imshow(np.log(abs(I_ideal)),cmap='gray')
plt.title('log abs $I_{ideal}$')
plt.subplot(1,3,3)
plt.imshow(np.log(abs(H)),cmap='gray')
plt.title('log abs $H$')

plt.show()
```

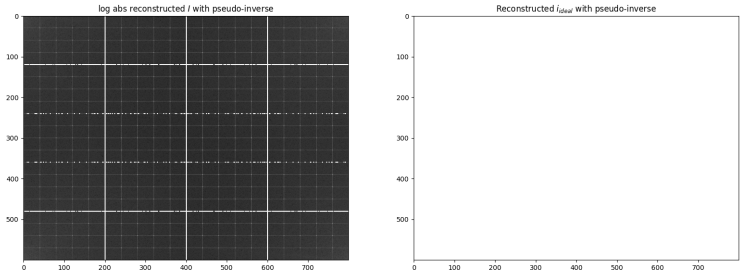


I semble visuellement être une combinaison de $I_{idéal}$ et H. Cette intuition est bonne puisque dans le domaine spectral I correspond à la multiplication de $I_{idéal}$ et H.

1.2.3 Pseudo-inverse sans seuillage

```
pseudo_inv_I = np.zeros_like(I)

for m in range(M):
    for n in range(N):
        pseudo_inv_I[m,n] = I[m,n]/H[m,n]

```

```
plt.figure(figsize =(10,30))

plt.subplot(1,2,1)
plt.imshow(np.log(abs(pseudo_inv_I)),cmap='gray')
plt.title('log abs reconstructed $I$ with pseudo-inverse')

plt.subplot(1,2,2)
plt.imshow(np.real(ifft2(pseudo_inv_I)),cmap='gray')
plt.title('Reconstructed $i_{ideal}$ with pseudo-inverse')

plt.show()
```

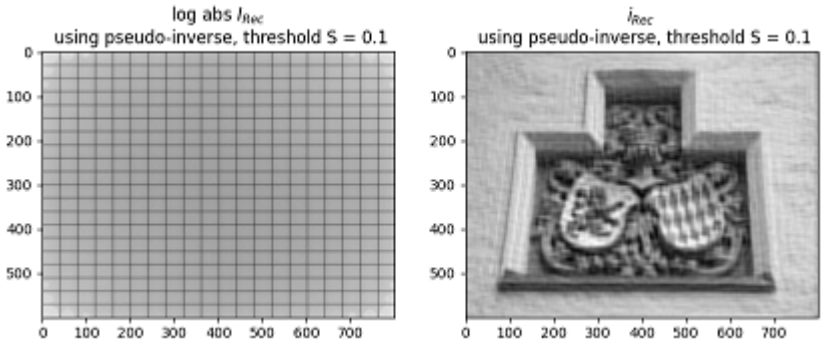


On voit que l'image n'a pas pu être reconstruite. La raison est que l'on divise par 0 pour calculer le pseudo-inverse.

1.2.4 Pseudo-inverse avec seuillage

```
pseudo_inv_I = np.zeros_like(I)

S = 0.1

for m in range(M):
    for n in range(N):
        if abs(H[m,n]) < S:
            pseudo_inv_I[m,n] = I[m,n]
        else:
            pseudo_inv_I[m,n] = I[m,n]/H[m,n]

plt.figure(figsize = (10,30))

plt.subplot(1,2,1)
plt.imshow(np.log(abs(pseudo_inv_I)),cmap='gray')
plt.title('log abs $I_{Rec}$'+f' \n using pseudo-inverse, threshold
     S = {S}')

```

```
plt.subplot(1,2,2)
plt.imshow(np.real(ifft2(pseudo_inv_I)),cmap='gray')
plt.title('$i_{Rec}$' + f' \n using pseudo-inverse, threshold S = {S}')

plt.show()
```



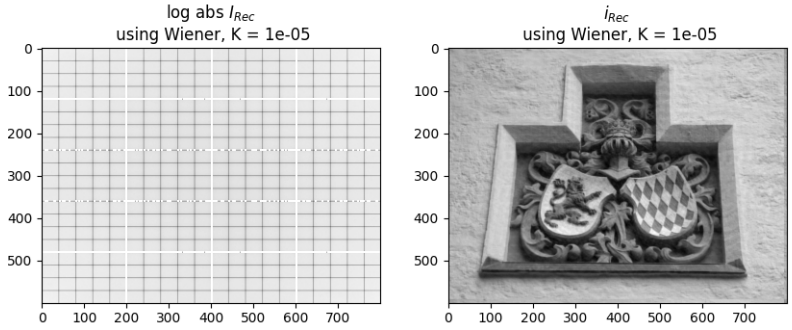
On obtient cette fois un résultat acceptable. Cependant le résultat n'est pas très convaincant : la méthode de seuillage est trop restrictive et ne permet pas de restaurer l'image efficacement.

1.2.5 Filtrage de Wiener

i. Le cas K = 0 correspond au filtrage par pseudo-inverse sans seuillage.

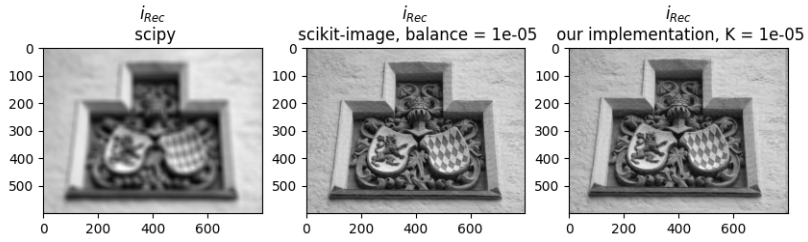
ii. On peut l'implémenter comme celà par soucis de stabilité numérique :

```
wiener_I = np.zeros_like(I)

K = 10**(-5)
for m in range(M):
    for n in range(N):
        Hmn = H[m,n]
        wiener_I[m,n] = I[m,n]*np.conj(Hmn)/(abs(Hmn)**2+K)

plt.figure(figsize =(10,30))

plt.subplot(1,2,1)
plt.imshow(np.log(abs(wiener_I)),cmap='gray')
plt.title('log abs $I_{Rec}$'+f' \n using Wiener, K = {K}')

plt.subplot(1,2,2)
plt.imshow(np.real(ifft2(wiener_I)),cmap='gray')
plt.title('$i_{Rec}$' + f' \n using Wiener, K = {K}')

plt.show()
```



iii. et iv. On va comparer notre implémentation aux fonctions *scipy.signal.wiener* et *skimage.restoration.wiener* :

```
from scipy.signal import wiener
from skimage.restoration import wiener as SKwiener
plt.figure(figsize =(10,30))

plt.subplot(1,3,1)
plt.imshow(wiener(i),cmap='gray')
plt.title('$i_{Rec}$'+f' \n scipy')

plt.subplot(1,3,2)
plt.imshow(SKwiener(i,h,balance=K),cmap='gray')
plt.title('$i_{Rec}$'+f' \n scikit-image, balance = {K}')

plt.subplot(1,3,3)
plt.imshow(np.real(ifft2(wiener_I)),cmap='gray')
plt.title('$i_{Rec}$' + f' \n our implementation, K = {K}')

plt.show()
```

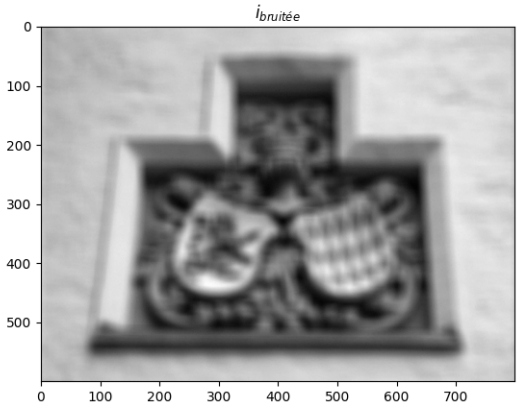


L'implémentation de scipy semble être basée sur une méthode différente, correspondant à *weiner2* sur matlab. Celle de scikit-image, en revanche, est similaire à la notre et donne des résultats qui paraissent identique.

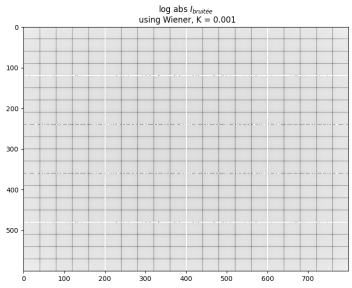
1.3 Image bruitée

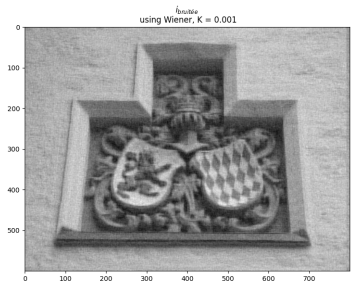
On génère l'image pour un $SNR = 40dB$ en utilisant l'astuce décrite dans l'énoncé :

```
SNR = 40
P_i = sum(sum(i**2))
P_b = P_i/(10**(SNR/10))
b_0 = np.random.normal(size=(M,N))
P_b0 = sum(sum(b_0**2))
b = b_0/np.sqrt(P_b0)*np.sqrt(P_b)

i += b
I = fft2(i)

plt.imshow(i,cmap='gray')
plt.title('$i_{bruit e}$')
plt.show()

wiener_I = np.zeros_like(I)

K = 0.001
for m in range(M):
    for n in range(N):
        Hmn = H[m,n]
        wiener_I[m,n] = I[m,n]*np.conj(Hmn)/(abs(Hmn)**2+K)

plt.figure(figsize =(10,30))

plt.subplot(1,2,1)
plt.imshow(np.log(abs(wiener_I)),cmap='gray')
plt.title('log abs $I_{bruit e}$'+f' \n using Wiener, K = {K}')

plt.subplot(1,2,2)
plt.imshow(np.real(ifft2(wiener_I)),cmap='gray')
plt.title('$i_{bruit e}$' + f' \n using Wiener, K = {K}')

plt.show()
```

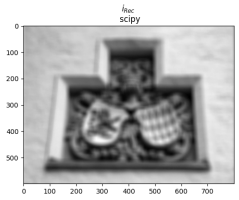


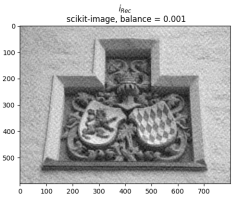
En débruitant, on obtient :

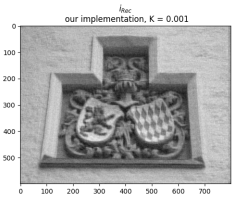




Et on peut à nouveau comparer avec les implémentations existantes :







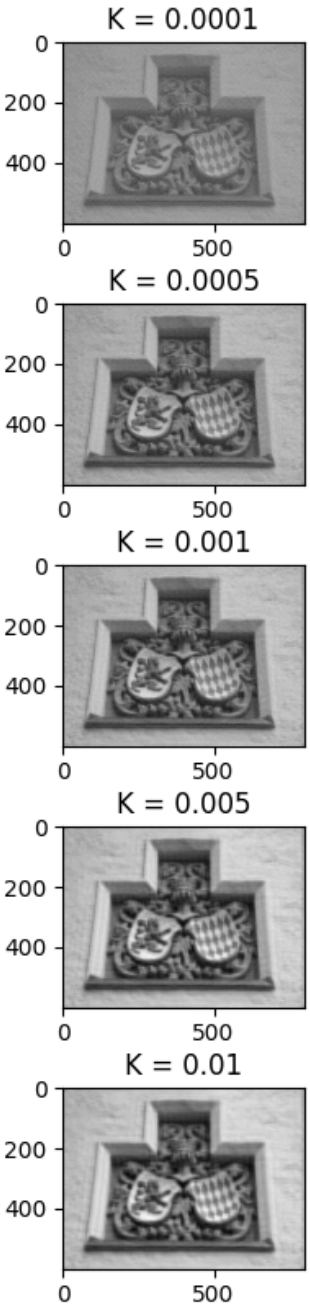
Cette fois, notre implémentation se distingue un peu de celle de scikit-image.

1.4 Optimisation de K

On sait que dans le cas où l'on connait le SNR de l'image, comme ici puisqu'on l'a simulé nous même, le choix de K qui minimise la MSE est $1/SNR$, soit 0.0025.

Cependant dans le cas plus pratique de la restauration d'une image sans connaître le SNR, on doit procéder autrement.

On ne dispose pas de métrique évidente pour optimiser K. On peut toutefois effectuer une recherche linéaire en comparant à vue d'oeil la qualité de restauration de l'image.


K = 0.0001
0
200
400
0
500
K = 0.0005
0
200
400
0
500
K = 0.001
0
200
400
0
500
K = 0.005
0
200
400
0
500
K = 0.01
0
200
400
0
500

Ici, les choix $K \in \{0.001, 0.005\}$ semblent les plus judicieux. Celà correspond bien au résultat théorique.