Compte rendu du TP 2, réalisé par Haithem Daghmoura

Section I : Imports

In [1]:
```
import numpy as np
import platform
import tempfile
import os
import matplotlib.pyplot as plt
from scipy import ndimage as ndi
# necessite scikit-image
from skimage import io as skio

# POUR LA MORPHO
from skimage.segmentation import watershed # watershed was moved to the skimage.
from skimage.feature import peak_local_max

from usful_functions import *
```

Section II

2- Transformation géométrique

Exploration de la fonction rotation

In [22]:
```
im=skio.imread("images/tmp.tif")
viewimage(im)
```

In [3]:
```
rotated_im= rotation(im,45,ech=0)
viewimage(rotated_im)
```

Au début on commence avec la methode à plus proche voisin

Original Image

Image with 90° rotation

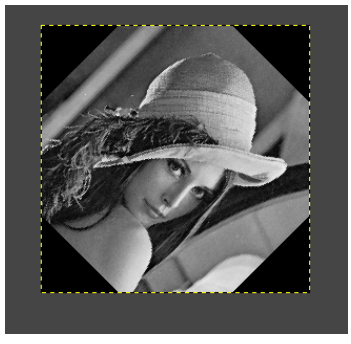


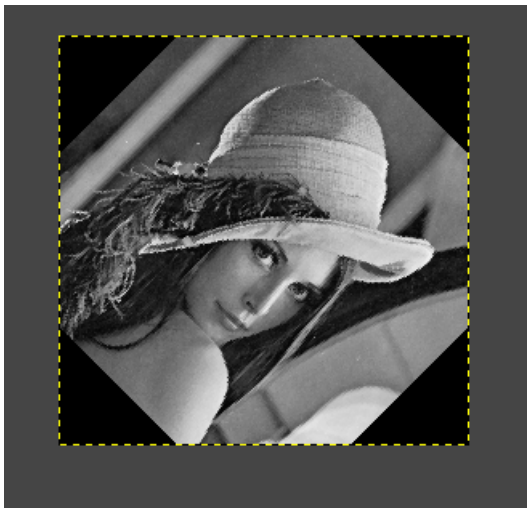
Image with 45° rotation

Dans la troisième image on a fait de maniere que l'image avec une rotation non congru à 0 modulo 90 a des contour noir pour convserver la forme carré

In [18]:
```
rotated_im= rotation(im,45,ech=0,clip= False)
viewimage(rotated_im)
```

In [20]:
```
rotated_im.shape
```

Out[20]: (364, 364)



Clip=True

Clip=False

On remarque que si on met clip à False on aura une image de plus grande taille (364x364) à peu pres égale à (256x256) multiplié par racine de 2 apres une seule rotation et qui conserve tous les detail de l'image original

Difference entre la rotation utilisant le méthode à plus proche voisin et la méthode bilinéaire

In [29]:
```
rotated_im= rotation(im,45,ech=0)
viewimage(rotated_im)
rotated_im=rotation(im,45,ech=1)
viewimage(rotated_im)
```

méthode de plus proche voisin

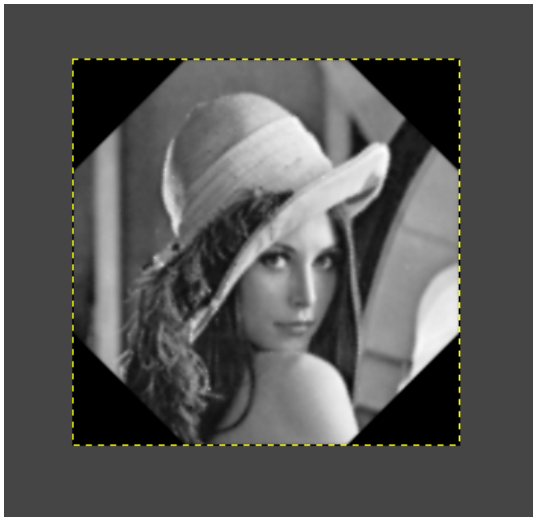
méthode de bilinéaire

On remarque que pour la premiere méthode (plus proches) l'images est constante par morceau : chaque pavé d'une certaine longeur à une valeur constante ce qui explique une certain forme carré dans la premiere image. Pour la deuxieme image on constate que l'interpolation est de degré 1

In [34]:
```
# image 8 rotation pour la plus proche voisin
rotated_im = im
for i in range (8):
    rotated_im=rotation(rotated_im,45,ech=0,clip=True)
viewimage (rotated_im)

# image 8 rotation pour le bilatérale
rotated_im = im
for i in range (8):
    rotated_im=rotation(rotated_im,45,ech=1,clip=True)
viewimage (rotated_im)
```

8 rotation avec le plus proche voisin



8 rotation avec le bilinéaire

on constate qu'on introduit une espèce de flou du au erreurs commulé dans chaque rotation

In [43]:
```
rotated_im=rotation(im,45,0.5,ech=1,clip=True)
viewimage(rotated_im)
rotated_im=rotation(im,-45,0.5,ech=1,clip=True)
```

rotation 45 + dezoom

on peut voire qu'il y'a des pixel carré pour la methode de bilinéaire qui n'était pas présent sans le dézoom on aurait du faire un filtrage passe bas pour annuler les hautes fréquence pour annuler cet effet

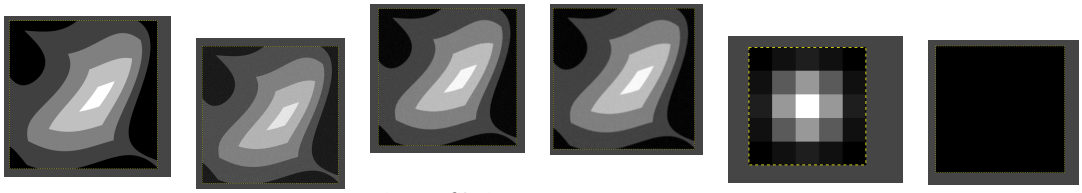
3-Filtrage linéaire et médian

Le parametre passé à ger_gau_ker est l'ecart type de la gaussienne qu'on veut filtrer avec. Dans l'implimentation de la fonction ce parametre influence la taille du filtre carré de coté noté ss dans l'implementaiton. Donc ss = int(max(3,2*round(2.5s)+1))

In [24]:
```
def noise(im,br):
    imt=np.float32(im.copy())
    sh=imt.shape
    bruit=br*np.random.randn(*sh)
    imt=imt+bruit
    return imt
```

In [25]:
```
im1=skio.imread("images/pyramide.tif")
viewimage(im1)
noise_im1=noise(im1,6)
viewimage(noise_im1)
gau_ker=get_gau_ker(1)
cst_ker=get_cst_ker(5)
filtred_gau_im1=filtre_lineaire(noise_im1,gau_ker)
filtred_cst_im1=filtre_lineaire(noise_im1,cst_ker)
viewimage(filtred_gau_im1)
viewimage(filtred_cst_im1)
```

```
viewimage(get_cst_ker(5))
viewimage(get_gau_ker(1))
```



Original image | Noisy image | Linear filtring using gaus filter | Linear filtring using cst filter | gaussien kernal | cte kernal

pour evaluer le débruitage d'un bruit dans ce type d'image on peut regarder les zones constante dans l'image original. Ces zones doivent avoir une variance nulle car ils sont constante. donc on evalue la variance de ces zones la dans l'image apres débruitage. Et donc plus la variance est petit plus le débruitage est bon. Dans la section suivant j'ai essayer d'implementer une telle function

In [26]:
```
#calcule de la vaiance dans chaque zone et puis on somme
def compute_var_metric (original_im,filtred_im):
    valeurs_cte = np.unique(original_im)
    zones=[]
    for cte in valeurs_cte :
        zones.append( (im1 == cte))
    var=filtred_im[zones[0]].var()
    for zone in zones:
        var=min(filtred_im[zone].var(),var)
    return var
print("metric for evaluating the original image",compute_var_metric(im1,im1))
```

```
metric for evaluating the original image 0.0
```

In [27]:
```
print("metric for evaluating the cte_kernal_denoising : ",compute_var_metric(im1
```

```
metric for evaluating the cte_kernal_denoising :  0.22127737780861886
```

In [28]:
```
print("metric for evaluating the cte_kernal_denoising ",compute_var_metric(im1,f
```

```
metric for evaluating the cte_kernal_denoising  0.2871575283799921
```

Ici, j'ai decomposer les zones consante de l'image simple mais j'ai remarquer que pour l'image de pyramide j'ai 190 niveau de gris bien plus que les 5 zones. celui est du au fait que au bord on a un changement des pixel et donc la fonction compute metric ne sera pas performante dans ce cas pour evaluer le bruit résiduel. Donc on utilisera la methode decrit dans le tp et on construira une fentre de taille 20x20 et on calculera la variane dans l'image. puisqu'on a plusieurs zones constante on remarquera par example dans la zone [0,50]x[0,50] on a une zone constante.

In [29]:
```
np.unique(im1[0:50,0:50])
```

Out[29]:
```
array([32], dtype=uint8)
```

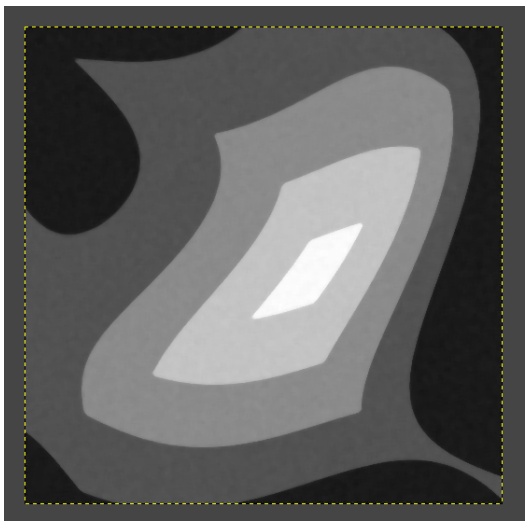
In [31]:
```
def compute_var_tp(filtred_im):
    return var_image(filtred_im,0,0,50,50)
```

```
compute_var_tp(filtred_gau_im1)
```

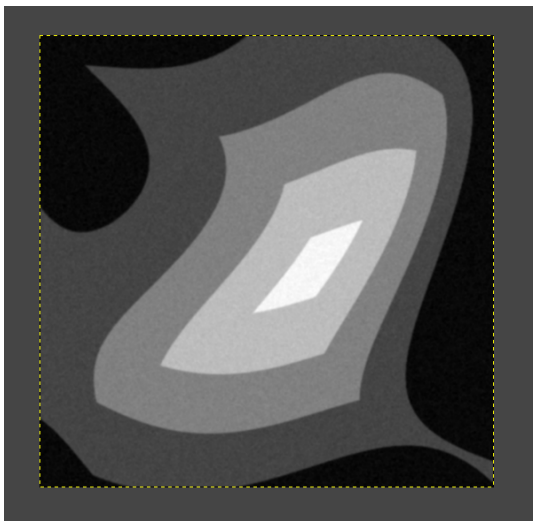
Out[31]: 3.2728293641541053

In [126…
```
filtre_med_im1=median_filter(noise_im1,r=3)
viewimage(filtre_med_im1)
```

In [125…
```
viewimage(filtred_gau_im1)
```



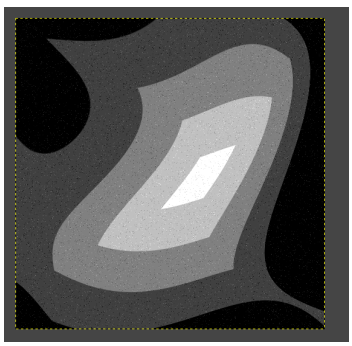
Linear filter



medain filter

on constate que l'image avec le median a une plus forte distinction entre les zones

In [10]:
```
im2=skio.imread("images/pyra-impulse.tif")
viewimage(im2)
cst_ker=get_cst_ker(5)
filtred_cst_im2=filtre_lineaire(im2,cst_ker)
viewimage(filtred_cst_im2)
filtre_med_im2=median_filter(im2,r=3)
viewimage(filtre_med_im2)
```



impluse noise image

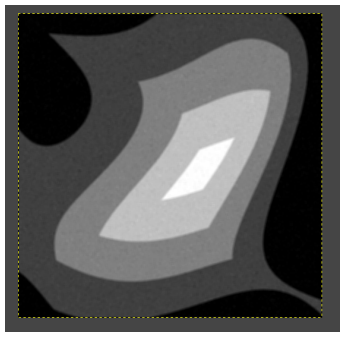


image with linear filter

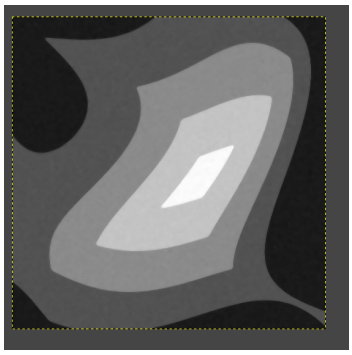


image with median filter

face au bruit impulsionnel le filtre median est plus performant que le filtre linear

In [13]:
```
im3=skio.imread("images/carre_orig.tif")
viewimage(im3)
cst_ker=get_cst_ker(5)
filtred_cst_im3=filtre_lineaire(im3,cst_ker)
viewimage(filtred_cst_im3)
filtre_med_im3=median_filter(im3,r=3)
viewimage(filtre_med_im3)
```

In [138…
```
filtre_med_im3=median_filter(im3,r=10)
viewimage(filtre_med_im3)
```

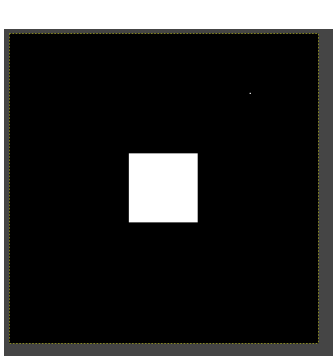


image originale

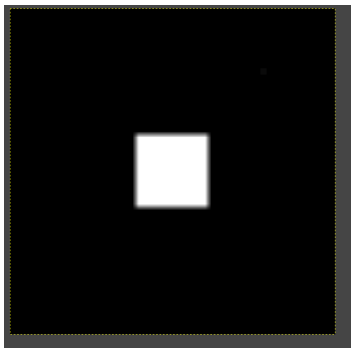


image apres application du filtre lineare

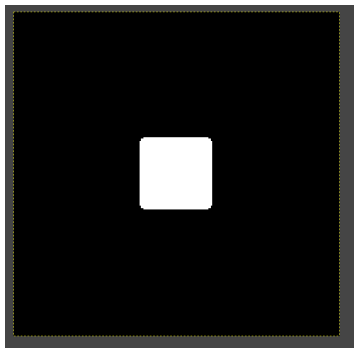
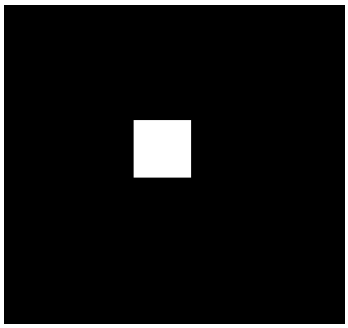


image apres application du filtre median

on remarque que dans l'image original il y'a un point lumineux en haut et à droite. Le filtrage de ce point là donnera la reponse indicielle du filtre étudier. Voici les images en faisant un meme zoom dans la zone contenat ce point.



zoom sur l'image 1

zoom sur l'image 2

zoom sur l'image 3

4. Restoration

In [3]:
```
im=skio.imread("images/tmp.tif")
mask=(1/13)*np.array([[0,1,1,1],
                      [1,1,1,1],
                      [1,0,1,1],
                      [1,1,1,0]])
mask
```

Out[3]:
```
array([[0.        , 0.07692308, 0.07692308, 0.07692308],
       [0.07692308, 0.07692308, 0.07692308, 0.07692308],
       [0.07692308, 0.        , 0.07692308, 0.07692308],
       [0.07692308, 0.07692308, 0.07692308, 0.        ]])
```

In [4]:
```
im_filtre =filtre_lineaire(im,mask)
#viewimage(im_filtre)
```

In [5]:
```
im_inverse_filtre = filtre_inverse(im_filtre,mask)
#viewimage(im_inverse_filtre)
```

imge originale

image filtré

image avec filtre inverse

Ici on constate qu'on a pu restauré parfaitemment l'image originale avec l'application du filtre inverse. Voyons ce que se passe lorsque on ajoute un peu de bruit

In [12]:
```
im_filtre_noisy = noise(im_filtre,3)
viewimage(im_filtre_noisy)
im_inverse_filtre_noisy = filtre_inverse(im_filtre_noisy,mask)
viewimage(im_inverse_filtre_noisy)
```

Image Original

Image filtré avec bruit 1

Image après l'application du filtre inverse

on constate ici que pour un bruit gaussien de variance 3 qui est negligeable devant les valeurs de donnée (entre 0 et 255) l'application du filtre inverse fait explosé le bruit et l'image restauré est dégradé

In [14]:
```
im2=skio.imread("images/carre_flou.tif")
viewimage(im2)
```

pour determiner le filtre qu'a subit l'image de carré en regarde la zone ou on a eu un point lumineux au debut et on regarde sa transformation. Ceci nous donnera le kernel de la convolution appliqué a li'mage.

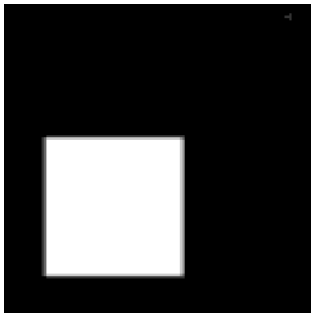
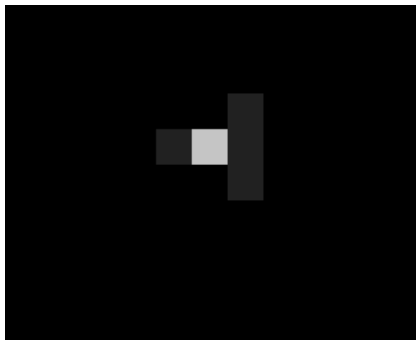


Image flouté

zoom dans la partie haute à droite



on superpose l'image original et l'image flouté dans gimp pour savoir le centre de kernel

Ici pour connaitre exactement le centre de kernel et sa taille on doit supperposer les deux images et la position du point lumineu d'origine sera le centre du kernel dans notre cas le kernel est proportionnel au suivant au mask suivant : [[0,0,1] [1,1,1] [0,0,1]]

In [36]:
```
im3[47:52,197:203]
```

Out[36]:
```
array([[  0,   0,   0,   0,   0,   0],
       [  0,   0,   0,   0,   0,   0],
       [  0,   0, 255,   0,   0,   0],
       [  0,   0,   0,   0,   0,   0],
       [  0,   0,   0,   0,   0,   0]], dtype=uint8)
```

In [38]:
```
im2[47:52,197:203]
```

Out[38]:
```
array([[ 0,  0,  0,  0,  0,  0],
       [ 0,  0,  0, 51,  0,  0],
       [ 0, 51, 51, 51,  0,  0],
       [ 0,  0,  0, 51,  0,  0],
       [ 0,  0,  0,  0,  0,  0]], dtype=uint8)
```

In [49]:
```
# ainsi le mask est le suivant :
mask = 1/5 *np.array([[0,0,1],[1,1,1],[0,0,1]])
mask
```

Out[49]:
```
array([[0. , 0. , 0.2],
       [0.2, 0.2, 0.2],
       [0. , 0. , 0.2]])
```

In [47]:
```
#restauration de l'image :
im_inverse_filtre2 = filtre_inverse(im2,mask)
viewimage(im_inverse_filtre2)
(abs(im_inverse_filtre2-im3)<1e-5).all()
```

Out[47]:
```
True
```

Ici on remarque que l'image apres l'application du filtre inverse est egale à l'image original (au sens que en chaque pixel la difference est inferieur à 1e-5) et donc le mask deduit est bien celui de la transformation

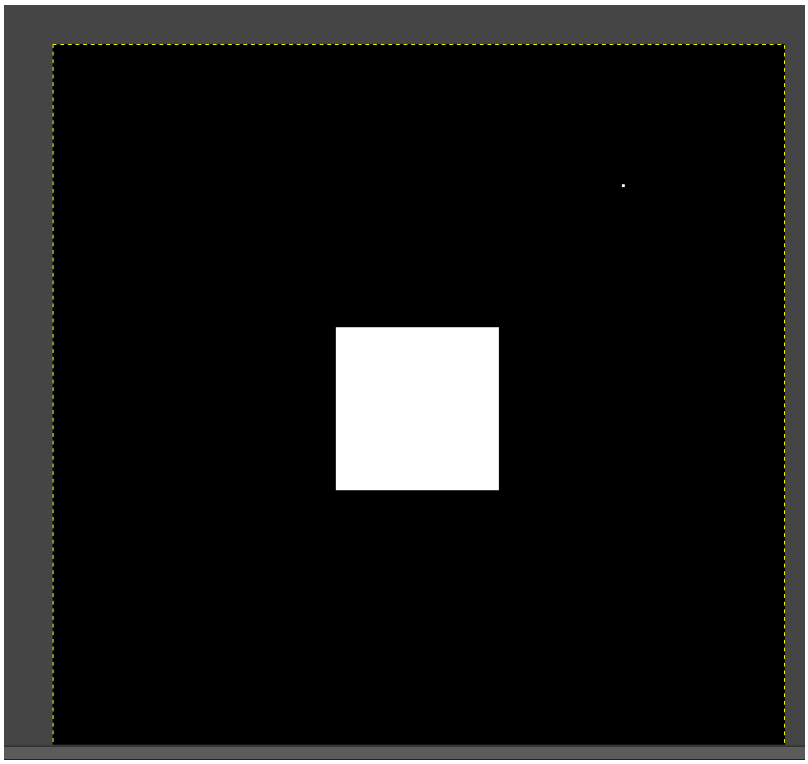


Image restauré

In [50]:
```
noisy_square = noise(im2,5)
viewimage(noisy_square)
lamdas=[0,1,2,5,10,100,1000,10000]
results=[]
for lamda in lamdas :
    result= wiener(noisy_square,mask,lamda)
    results.append(result)
for result in results :
    viewimage(result)
```

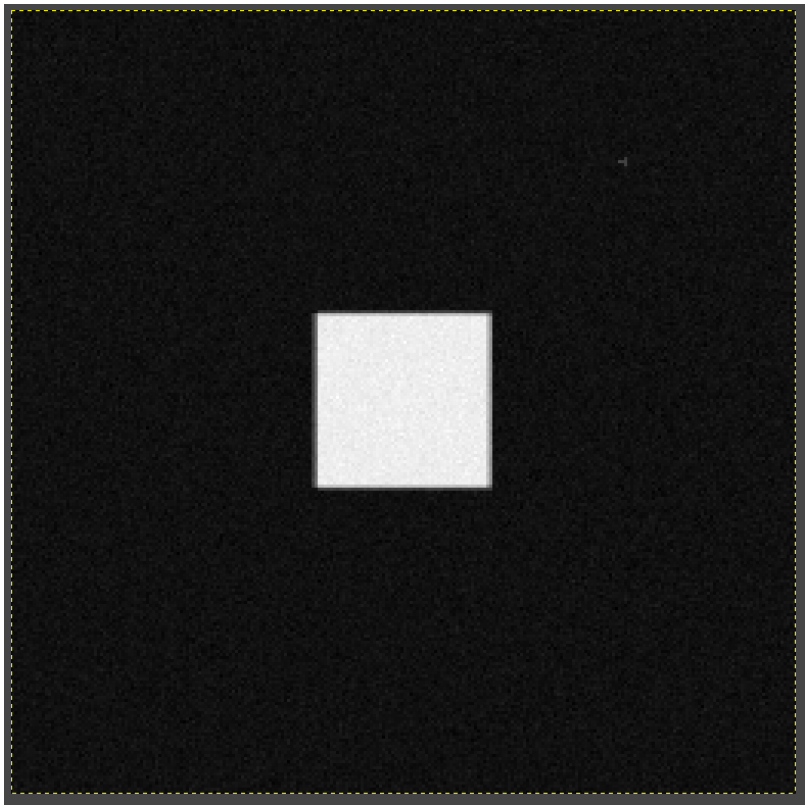
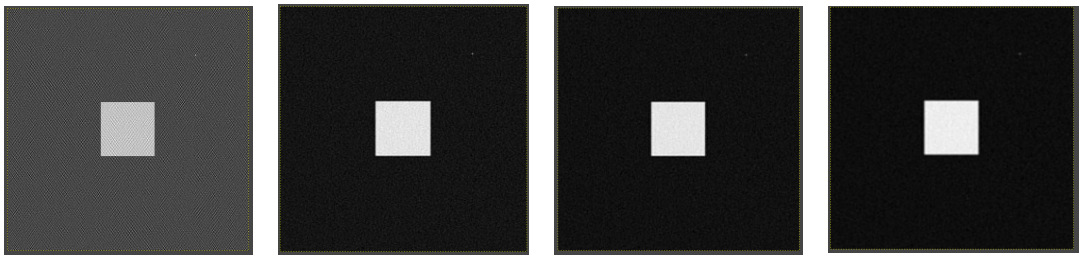
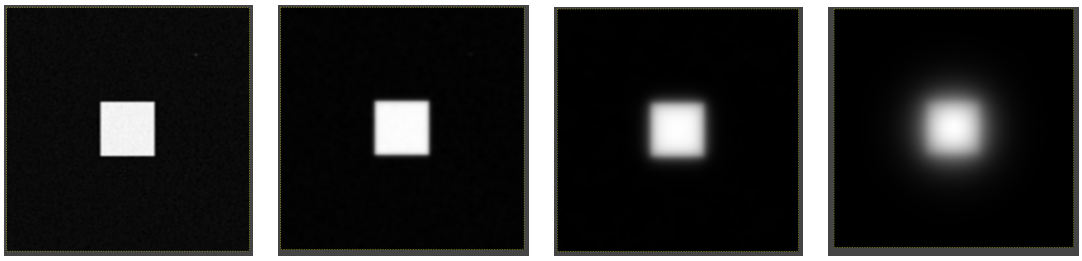


image à filtrer



lamda = 0 lamda = 1 lamda = 2 lamda = 5



lamda = 10 lamda = 100 lamda = 1000 lamda = 10000

On remarque que pour lamda = 0 on aura le meme resultat que pour la fonction inverse filtre qui explose le bruit. En augmentant lamda on tient compte de plus en plus de l'importance du bruit et particulirerment pour lambda egale à la variance de bruit on a le meilleur resultat ici (lamda = 5). En faisant augmenter lamda encore on considere que le bruit est tres eleve et que le rapport signale à bruit est faible donc le meilleur choix c'est

de rendre tous rendre null dans le domaine frequencielle ce qui explique les resultat obtenu pour lamda =1000 et 10000

5. Application

5.1 Comparaison filtrage linéaire et médian

In [33]:
```
im=skio.imread("images\carre_orig.tif")
noisy_im = noise(im,5)
viewimage(noisy_im)
filtred_image=median_filter(noisy_im,typ=2,r=4)
compute_var_tp(filtred_image)
```

Out[33]:
```
4.088321800008753
```

In [43]:
```
kernel = get_cst_ker(2)
filtred_image_cst1=filtre_lineaire(noisy_im,kernel)
compute_var_tp(filtred_image_cst1)
```

Out[43]:
```
6.08635641885736
```

In [44]:
```
kernel = get_cst_ker(3)
filtred_image_cst1=filtre_lineaire(noisy_im,kernel)
compute_var_tp(filtred_image_cst1)
```

Out[44]:
```
2.662461759316227
```

Pour un noyau constant on peut montrer que pour cette image de carre que plus le noyeau est grand plus le bruit residuelle sera faible car cette image et partiquement constante dans des region tres large. Donc pour avoir un meme resultat que le filtre median il suffit de comparer la valeur residuelle obtenu pour differentes valeurs de filtre constant jusqu'à arriver a celui qui a moin de variance de bruit residuelle. anisi ici un filtre constant de taille 3 suffira pour avoir des resultat comparable a celle de filtrage par le filtre median

5.2 Calcul théorique du paramètre de restauration

In [61]:
```
def wiener2(im,K,var=1):
    """effectue un filtrage de wiener de l'image im par le filtre K.
       lamb=0 donne le filtre inverse
       on rappelle que le filtre de Wiener est une tentaive d'inversion du noyau
       avec une regularisation qui permet de ne pas trop augmenter le bruit.
       """
    fft2=np.fft.fft2
    ifft2=np.fft.ifft2
    (ty,tx)=im.shape
    (yK,xK)=K.shape
    KK=np.zeros((ty,tx))
    KK[:yK,:xK]=K
    x2=tx/2
    y2=ty/2
```

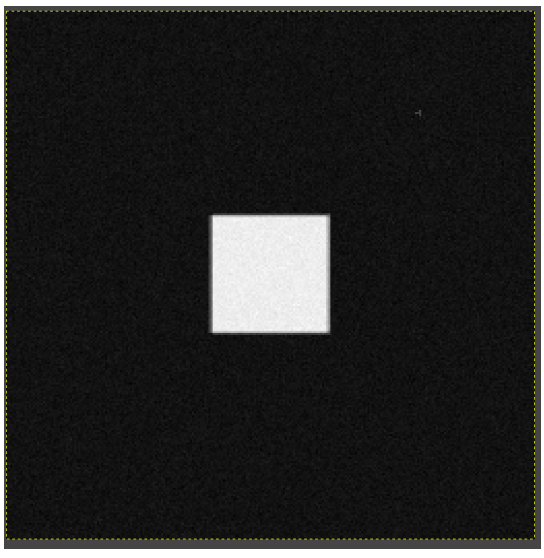
```
    fX=np.concatenate((np.arange(0,x2+0.99),np.arange(-x2+1,-0.1)))
    fY=np.concatenate((np.arange(0,y2+0.99),np.arange(-y2+1,-0.1)))
    fX=np.ones((ty,1))@fX.reshape((1,-1))
    fY=fY.reshape((-1,1))@np.ones((1,tx))
    fX=fX/tx
    fY=fY/ty

    w2=fX**2+fY**2
    w=w2**0.5

    #tranformee de Fourier de l'image degradeee
    g=fft2(im)
    #transformee de Fourier du noyau
    k=fft2(KK)
    #nouveau quotien introduit
    quotient = ( ty*tx * var) /(g**2)
    #fonction de mutilplication
    mul=np.conj(k)/(abs(k)**2+quotient)
    #filtrage de wiener
    fout=g*mul

    # on effectue une translation pour une raison technique
    mm=np.zeros((ty,tx))
    y2=int(np.round(yK/2-0.5))
    x2=int(np.round(xK/2-0.5))
    mm[y2,x2]=1
    out=np.real(ifft2(fout*(fft2(mm))))
    return out
```

In [65]:
```
im2=skio.imread("images/carre_flou.tif")
noisy_square = noise(im2,5)
result= wiener2(noisy_square,mask,5)
viewimage(result)
```



noisy_suqare

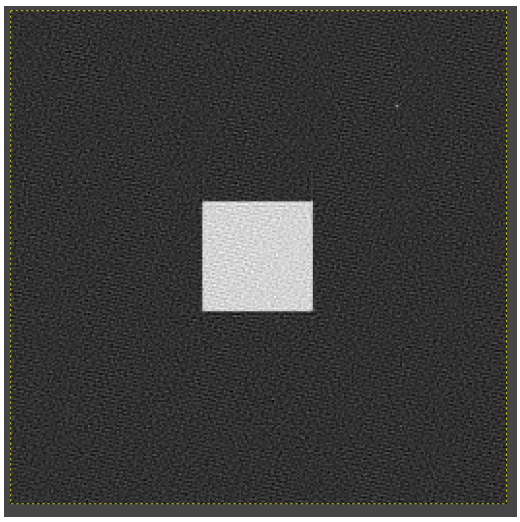


image après filtrage