

# DenkowskiStanislawlab2

October 20, 2022

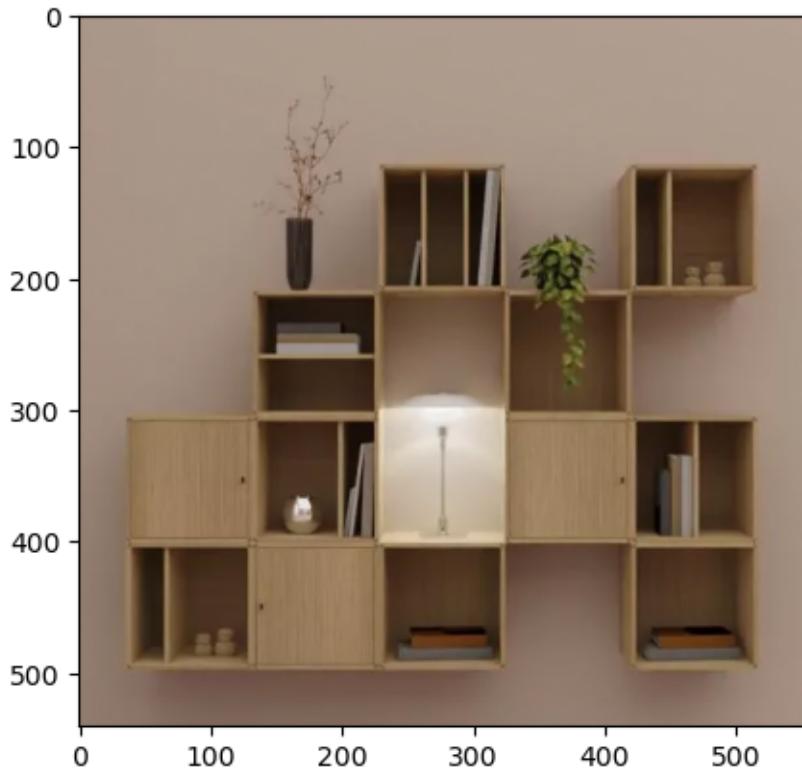
```
[ ]: import torch
import torch.nn.functional as F
import pytorch_lightning as pl
import numpy as np
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
from imageio.v2 import imread
```

torch.nn.functional.conv2d  
input = (minibatch,in\_channels,iH,iW)  
weight = (out\_channelsin\_channels/groups,kH,kW)

Wczytujemy obrazek i od razu normalizujemy.

```
[ ]: ori_img = torch.tensor(imread('furniture.png'))/255
print(ori_img.shape)
plt.imshow(ori_img)
plt.show()
```

torch.Size([541, 555, 3])



```
[ ]: def to_conv(a):
    a = torch.unsqueeze(a, dim=-1).T.float()
    return a if len(a.shape)==4 else torch.unsqueeze(a, dim=0)

def from_conv(a):
    return torch.squeeze(a).T
print(f'{to_conv(ori_img).shape} - {ori_img.shape}')
print(f'{from_conv(to_conv(ori_img)).shape} - {ori_img.shape}')
```

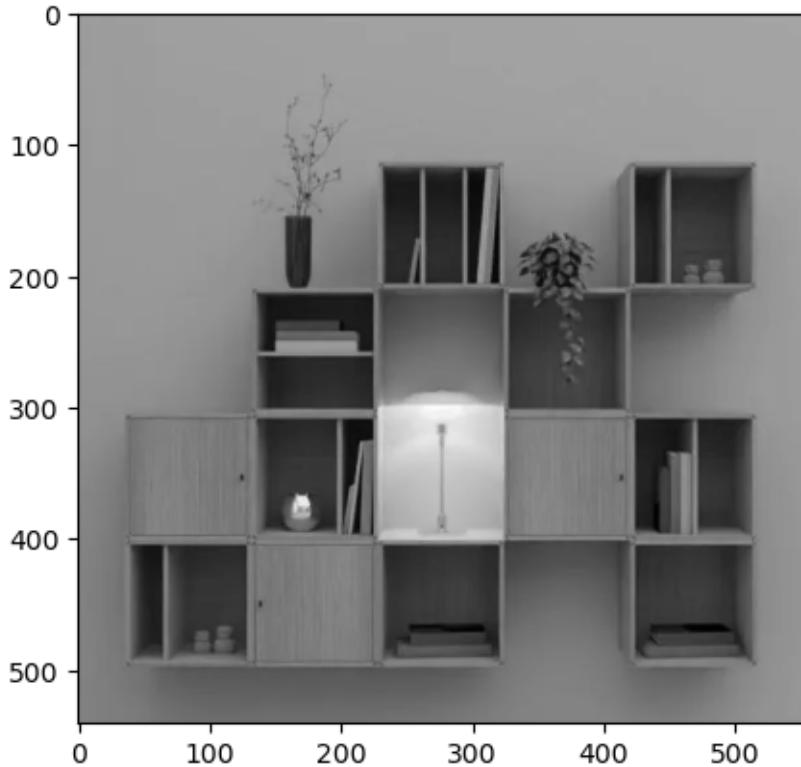
```
torch.Size([1, 3, 555, 541]) - torch.Size([541, 555, 3])
torch.Size([541, 555, 3]) - torch.Size([541, 555, 3])
```

```
[ ]: wei = torch.tensor([1/3]*3).float().reshape((1,3,1,1))
wei
```

```
[ ]: tensor([[[[0.3333]],
[[0.3333]],
[[0.3333]]]])
```

```
[ ]: img = from_conv(F.conv2d(to_conv(ori_img), wei, padding='same', stride=1))
```

```
[ ]: def print_gray(img):
    plt.imshow(img, cmap='gray')
    plt.show()
print_gray(img)
```



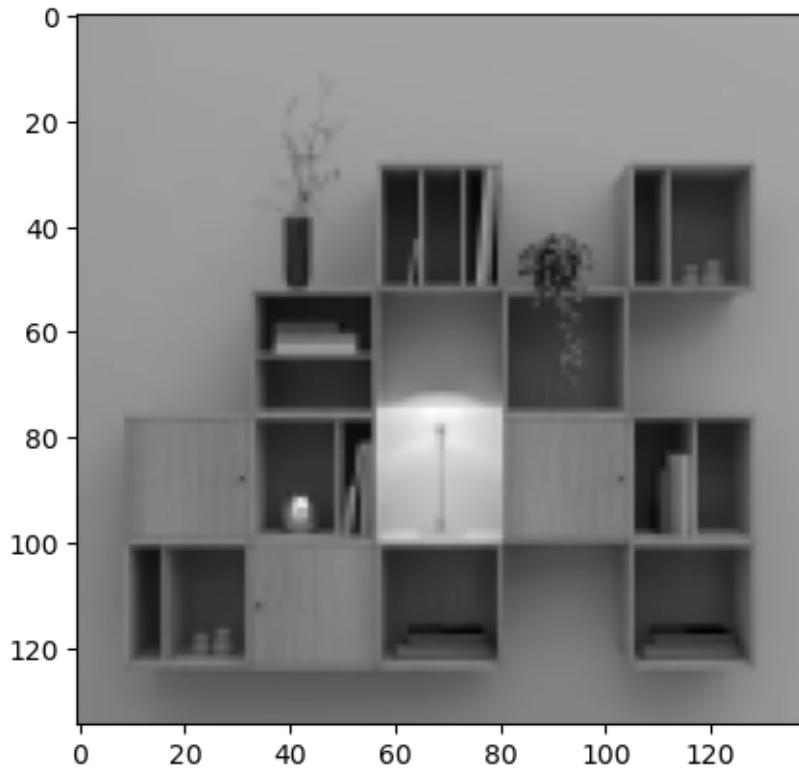
Wykorzystałem funkcję `torch.nn.functional.conv2d`, która wykonuje konwolucję 2d, ale dodatkowo ma podejście funkcyjne, a nie obiektowe, w przeciwieństwie do `torch.nn.conv2d`.

Upewniłem się, że będziemy mieć padding, który nie zmieni nam rozmiaru obrazu wyjściowego, oraz stride wynosi 1.

Aby móc załadować obraz i filtr do funkcji `conv2d`, musiałem je odpowiednio przerobić - dodać odpowiednie wymiary oraz transponować macierze, żeby wymiary się zgadzały (szerokość i wysokość obrazu mogą być stosowane zamiennie, byle zawsze tak samo).

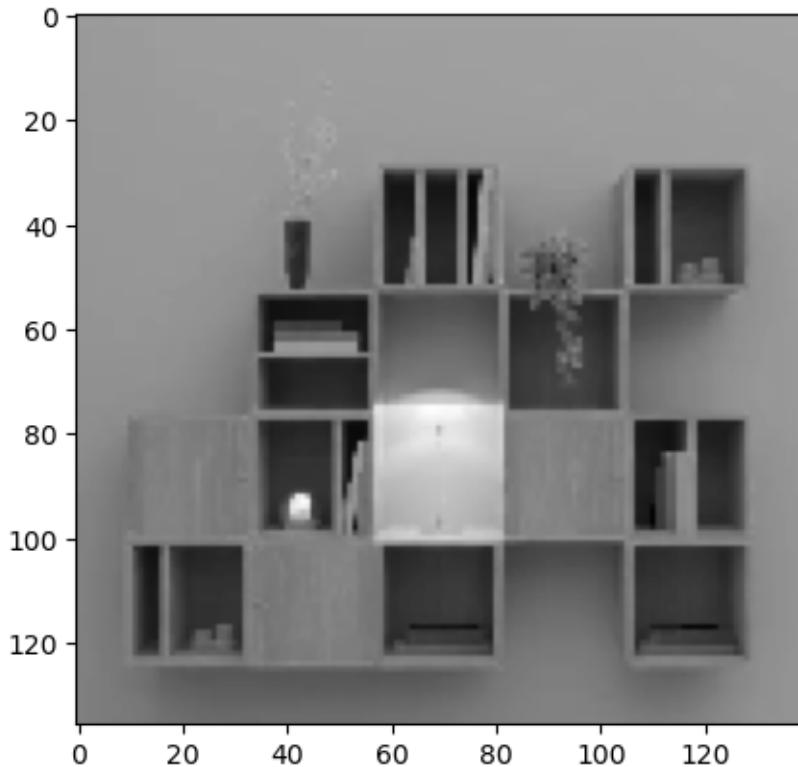
Na sam koniec musiałem usunąć zbędne wymiary z macierzy wynikowej oraz transponować, aby przywrócić obraz do poprzedniej orientacji.

```
[ ]: img1 = torch.clone(img)
wei = torch.tensor([1/16]*16).float().reshape((1,1,4,4))
img1 = from_conv(F.conv2d(to_conv(img1), wei, padding='valid', stride=4))
print_gray(img1)
img1.shape
```



[ ]: torch.Size([135, 138])

```
[ ]: img2 = torch.clone(img)
img2 = from_conv(F.max_pool2d(to_conv(img2), (4,4), padding=(2,2)))
print_gray(img2)
img2.shape
```



```
[ ]: torch.Size([136, 139])
```

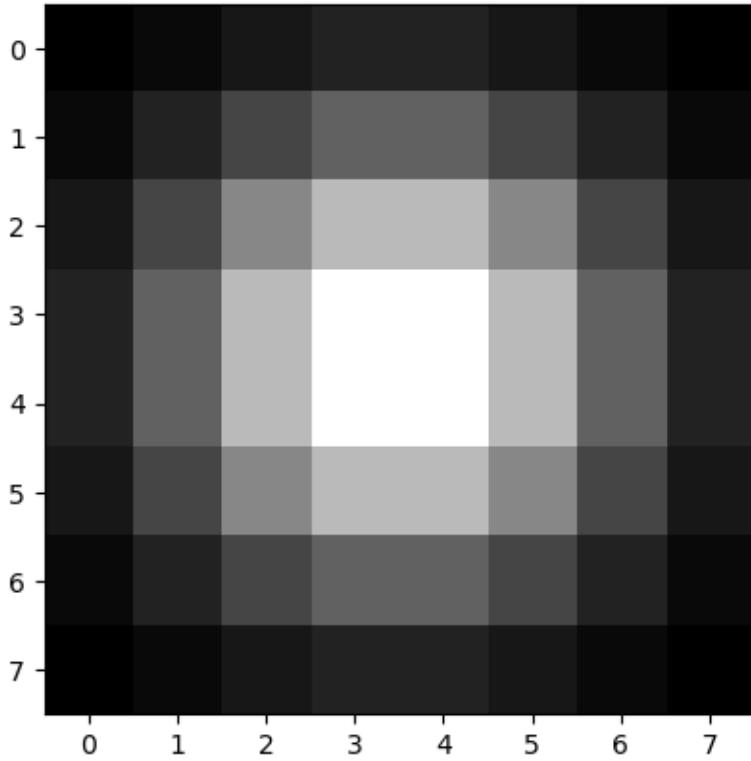
Pooling wykonany przy pomocy konwolucji, biorący jako wartość średnia oraz pooling wykonany przy pomocy funkcji `torch.nn.functional.max_pool2d`(przyjmuje bardzo podobne argumenty do konwolucji). Dodatkowo przy pierwszej metodzie nie użyłem paddingu, więc obraz jest trochę mniejszy, niż przy drugiej gdzie użyłem paddingu będącego połową filtru.

Wybieram opcję drugą - czyli poolingu wykorzystującego max. Bardziej podoba mi się obraz, oraz krawędzi wydają się być mocniejsze, co liczę, że poprawi wyniki.

```
[ ]: img = img2
```

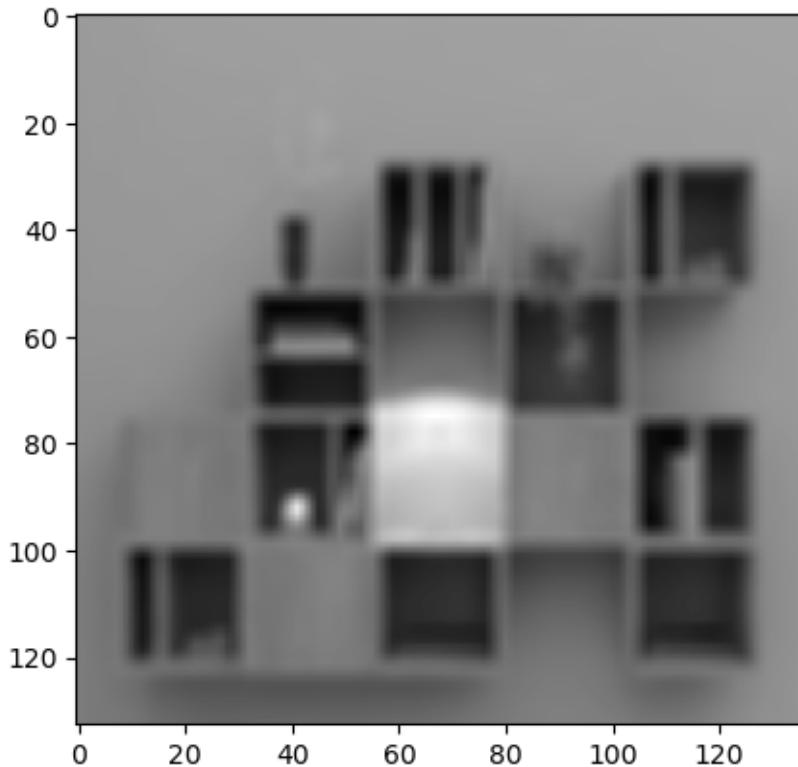
```
[ ]: def prep_gauss(n, std=torch.tensor(1.8)):
    ax = torch.linspace(-(n-1)/2, (n-1)/2, n)
    gauss = torch.exp(-0.5*torch.square(ax)/torch.square(std))
    kernel = torch.outer(gauss, gauss)
    return kernel/torch.sum(kernel)

print_gray(prep_gauss(8))
```



```
[ ]: def try_gauss(img, n, std=torch.tensor(1.8)):
    gauss = prep_gauss(n, std)
    new_img = from_conv(F.conv2d(to_conv(img), to_conv(gauss), padding='valid', stride=1))
    print_gray(new_img)
    return new_img

img = try_gauss(img, 4)
img.shape
```



[ ]: torch.Size([133, 136])

Wybieram rozmiar 4, ponieważ wydaje mi się, że traci on najmniej względem oryginału, ale równocześnie pozbywa się różnych zbędnych artefaktów.

Korzystam głównie z funkcji matematycznych do wyliczenia filtru gaussa, który wykorzystuję do rozmycia obrazu.

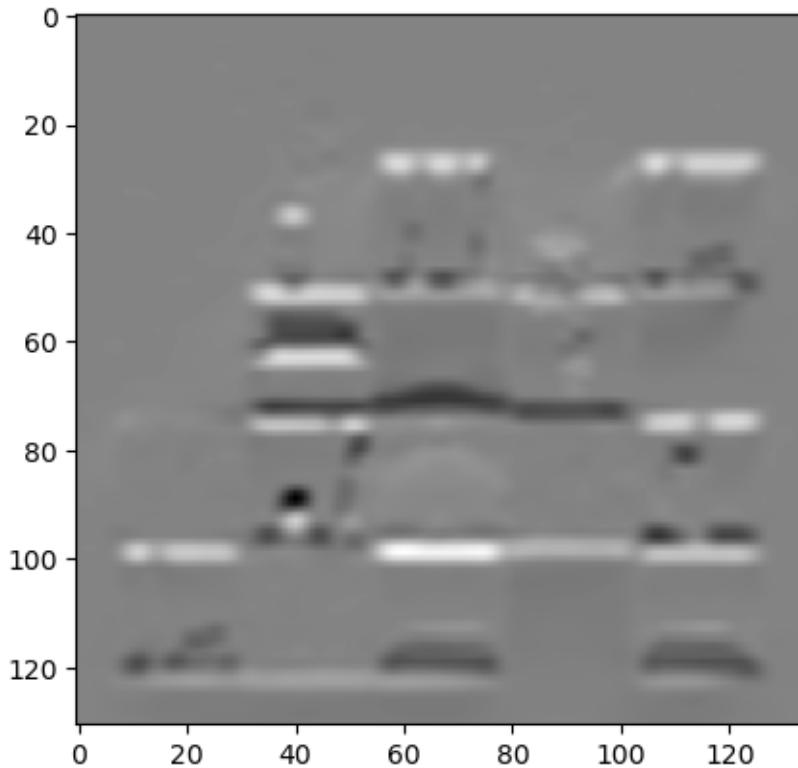
```
[ ]: grad1 = torch.tensor([[1,2,1],[0,0,0],[-1,-2,-1]])
grad2 = grad1.T
grad1 = torch.unsqueeze(grad1, dim=-1).T.float()
grad2 = torch.unsqueeze(grad2, dim=-1).T.float()
grad = torch.stack((grad1,grad2), dim=0)
print(grad.shape)
grad
```

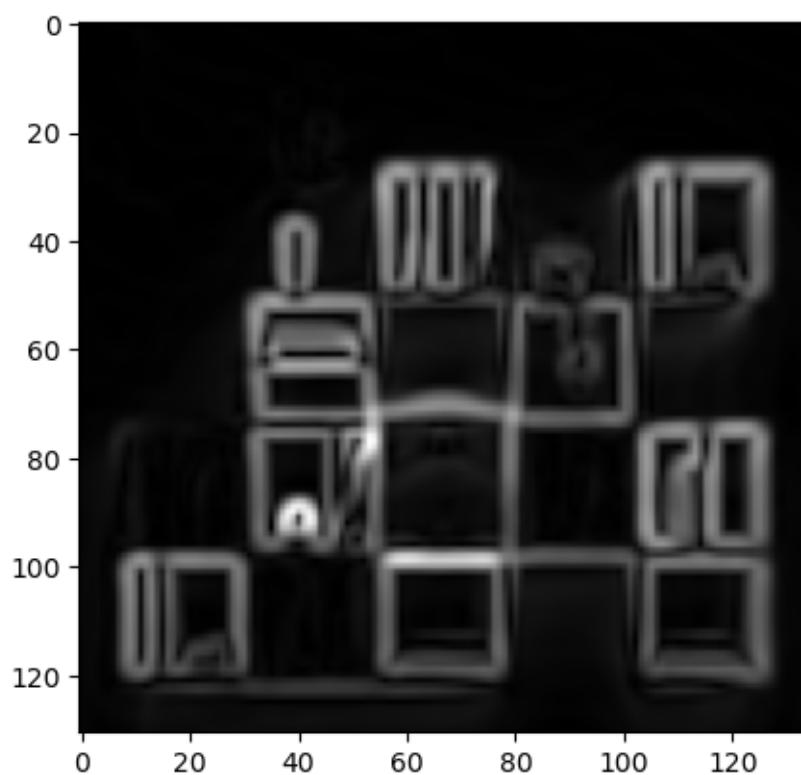
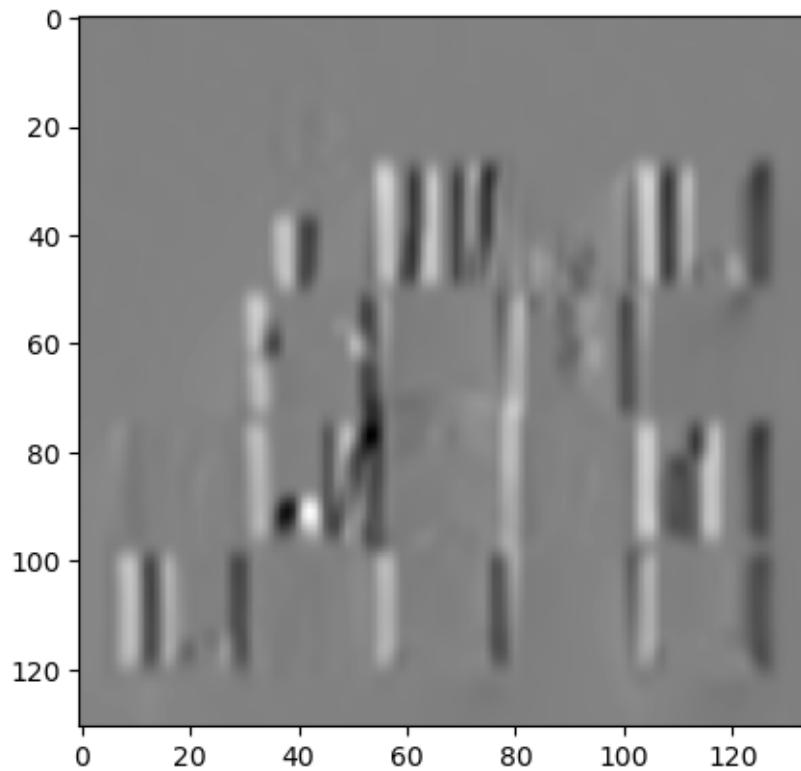
torch.Size([2, 1, 3, 3])

```
[ ]: tensor([[[[ 1.,  0., -1.],
              [ 2.,  0., -2.],
              [ 1.,  0., -1.]]],
```

[[[ 1., 2., 1.],

```
[ 0.,  0.,  0.],  
[-1., -2., -1.]]])  
  
[ ]: sobol_img = from_conv(F.conv2d(to_conv(img), grad, stride=1, padding='valid'))  
print_gray(sobel_img[:, :, 0])  
print_gray(sobel_img[:, :, 1])  
sobel_img = torch.sqrt(torch.square(sobel_img[:, :, 0]) + torch.square(sobel_img[:  
˓→, :, 1]))  
print_gray(sobel_img)
```



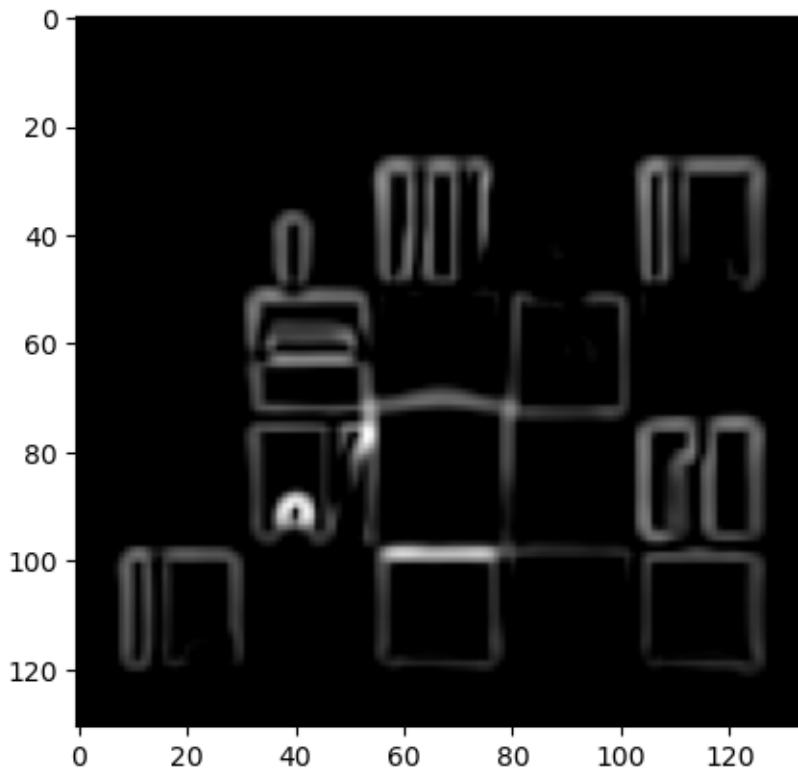


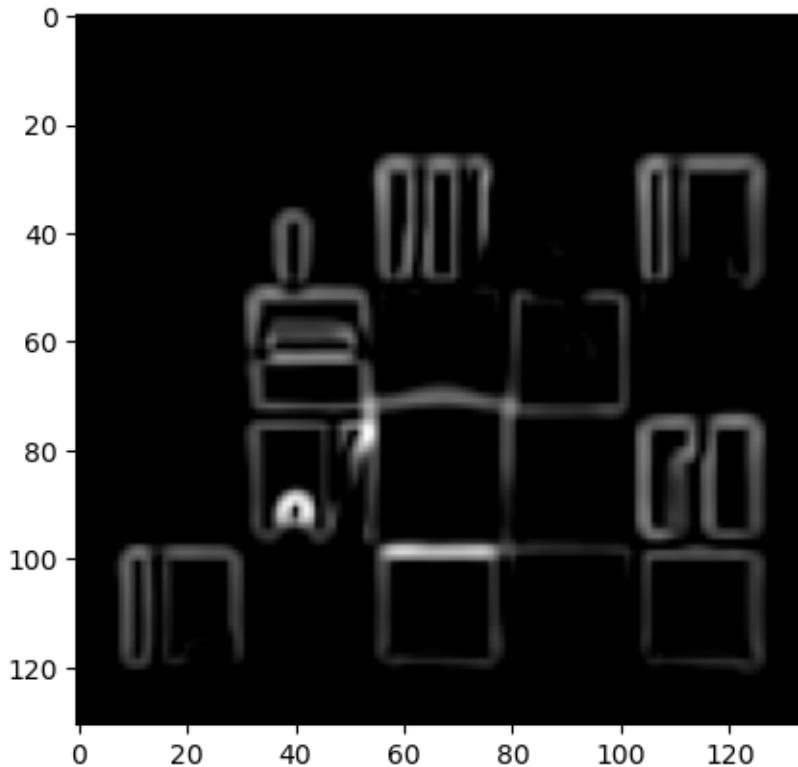
Użyłem filtra o 2 kanałach wyjściowych, na każdą składową gradientu, więc otrzymałem 1 tensor z oboma gradientami.

Aby uzyskać obrazek o tylko jednym kanale, podniosłem tensor do kwadratu i dodałem oba kanały.

```
[ ]: def try_relu(img, val, draw=True):
    new_img = F.relu(img-val)
    if draw:
        print_gray(new_img)
    return new_img

sobel_img=try_relu(sobel_img, 1/3)
sobel_img=sobel_img/sobel_img.max()
print_gray(sobel_img)
(sobel_img>1).any() or (sobel_img<0).any()
```





```
[ ]: tensor(False)
```

```
[ ]: img=sobol_img#[2:-2,2:-2]
```

Można użyć funkcji konwolucji, z wielowymiarowym filtrem, gdzie każdy kanał filtru odpowiada za inny rozmiar kwadratu.

Zdecydowałem jednak, że nie wiem jak zoptymalizowane są mnożenia macierzy, a potencjalnie w takiej sytuacji moglibyśmy wykonywać mnóstwo pustych mnożeń raza 0, zużywających jedynie moc obliczeniową. (np. podczas liczenia kwadratu 3x3, ale mając filtr rozmiaru 100x100)

Dlatego będę liczył iteracyjnie, z filtrami o tylko jednym kanale.

```
[ ]: def sq(n):
    sq = torch.ones((n,n))
    sq[1:-1,1:-1] = 0
    return sq

sq(10)
```

```
[ ]: tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
           [1., 0., 0., 0., 0., 0., 0., 0., 1.],
           [1., 0., 0., 0., 0., 0., 0., 0., 1.],
           [1., 0., 0., 0., 0., 0., 0., 0., 1.],
```

```

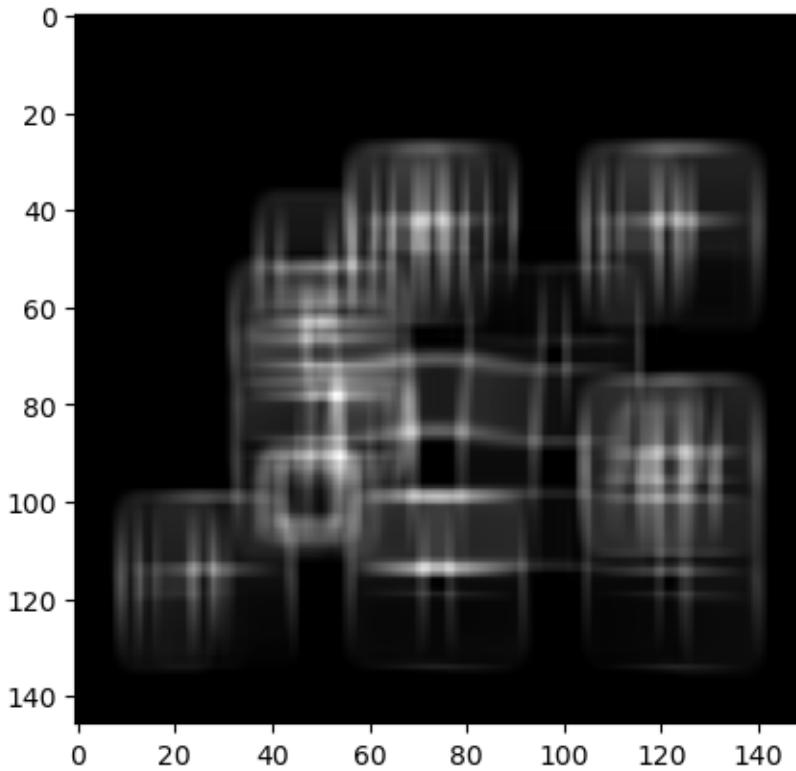
[1., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
[1., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
[1., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
[1., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
[1., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])

[ ]: def try_hough(img, n):
    wei = to_conv(sq(n))
    res = from_conv(F.conv_transpose2d(to_conv(img), wei, stride=1, ↴
    output_padding=(0,0)))
    print_gray(res)
    return res

imgs = []
for i in range(16,32):
    print(i)
    imgs = imgs + [try_hough(img,i)]

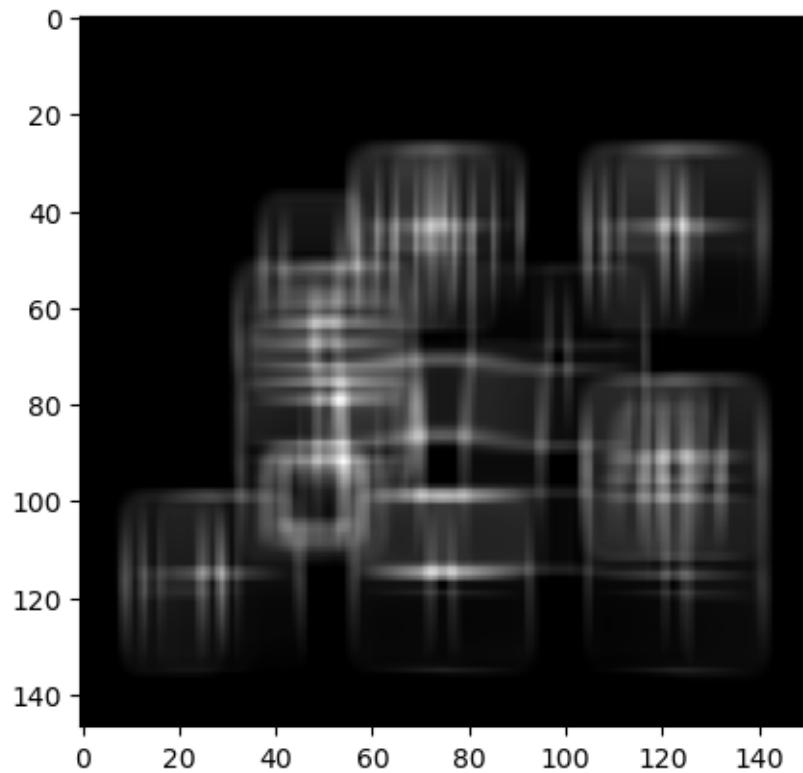
```

16



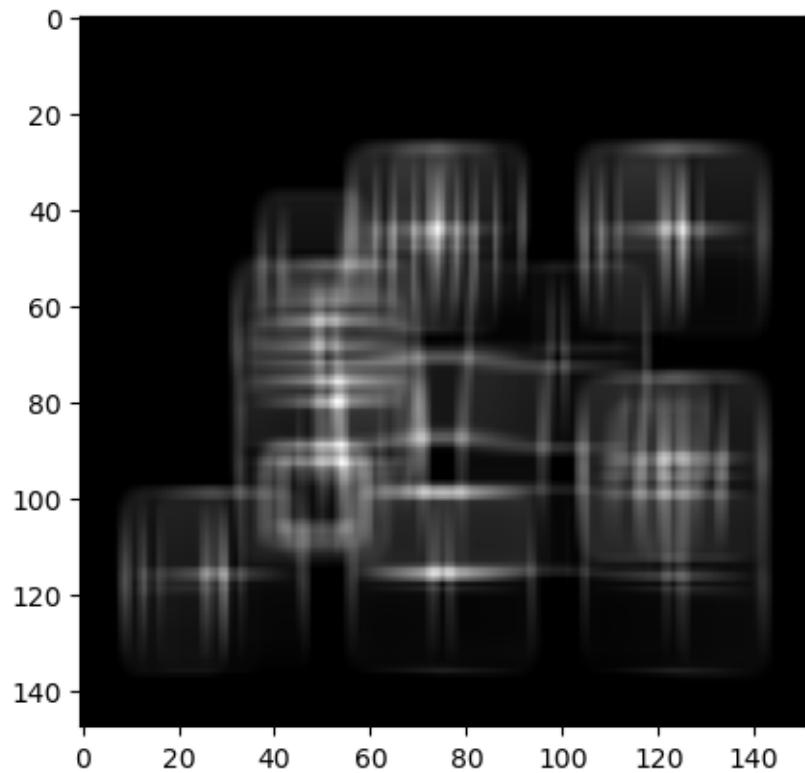
17

12



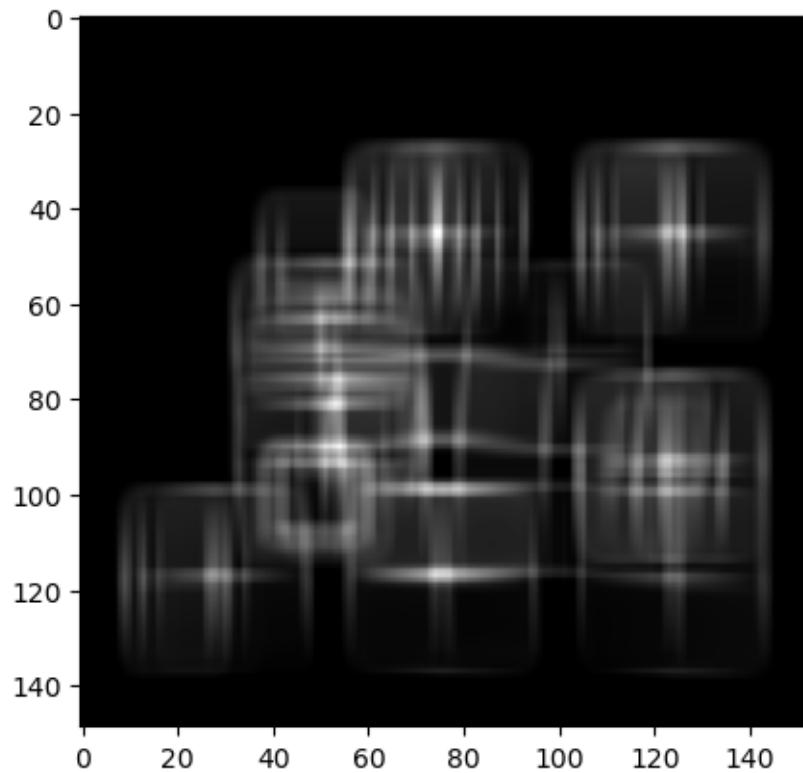
18

13

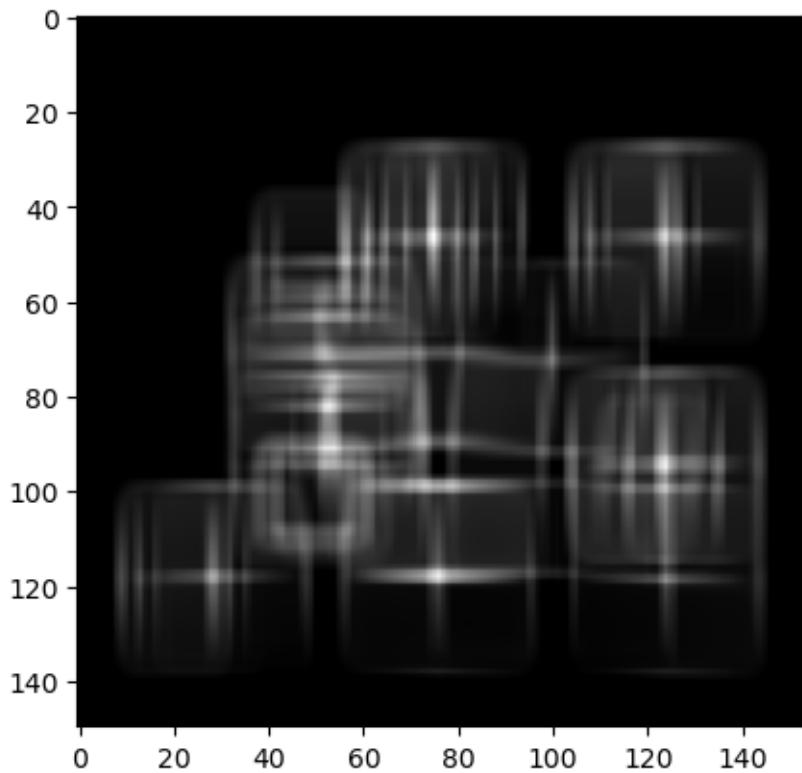


19

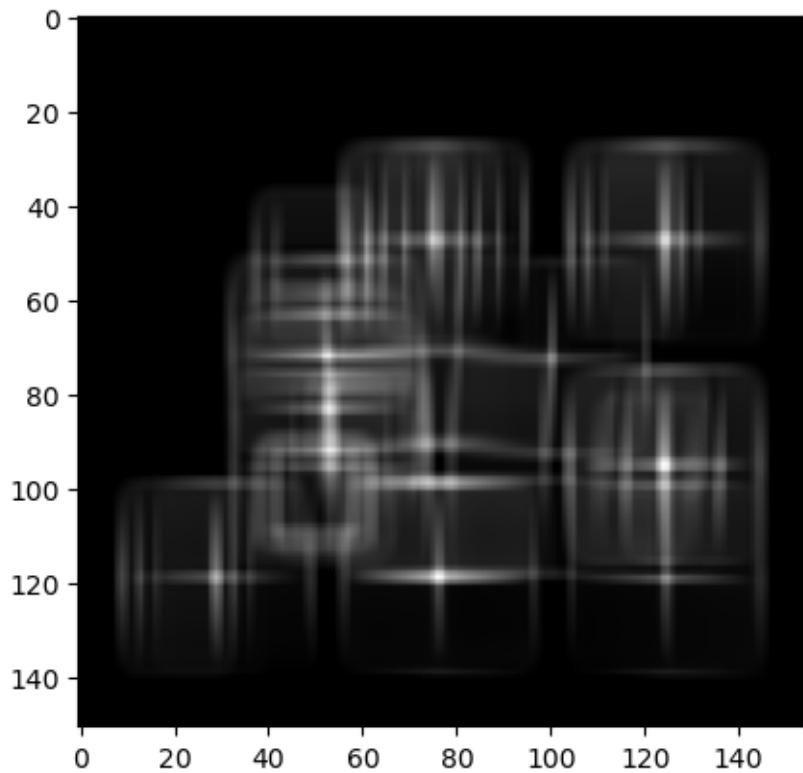
14



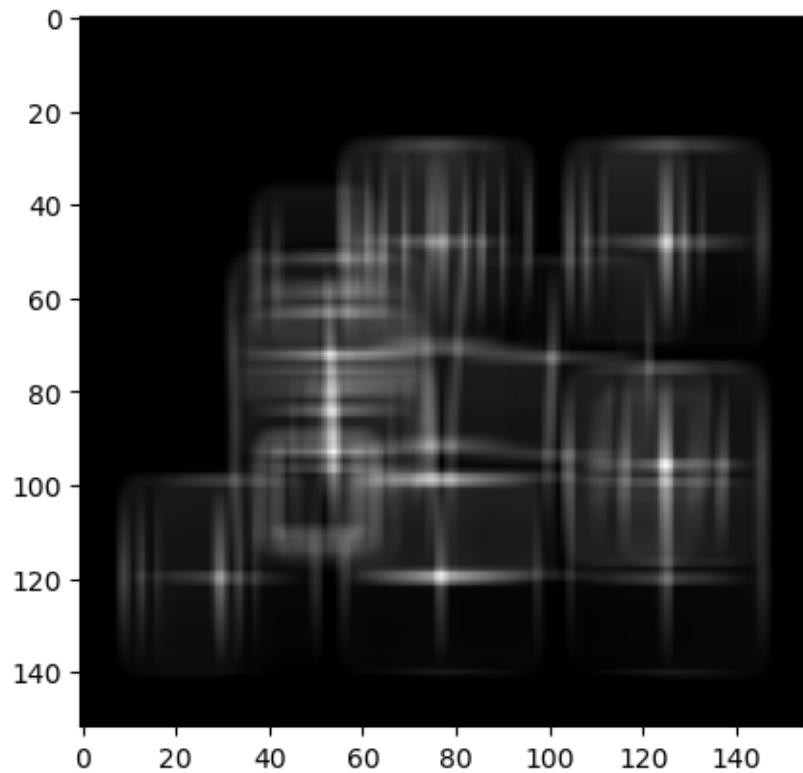
20



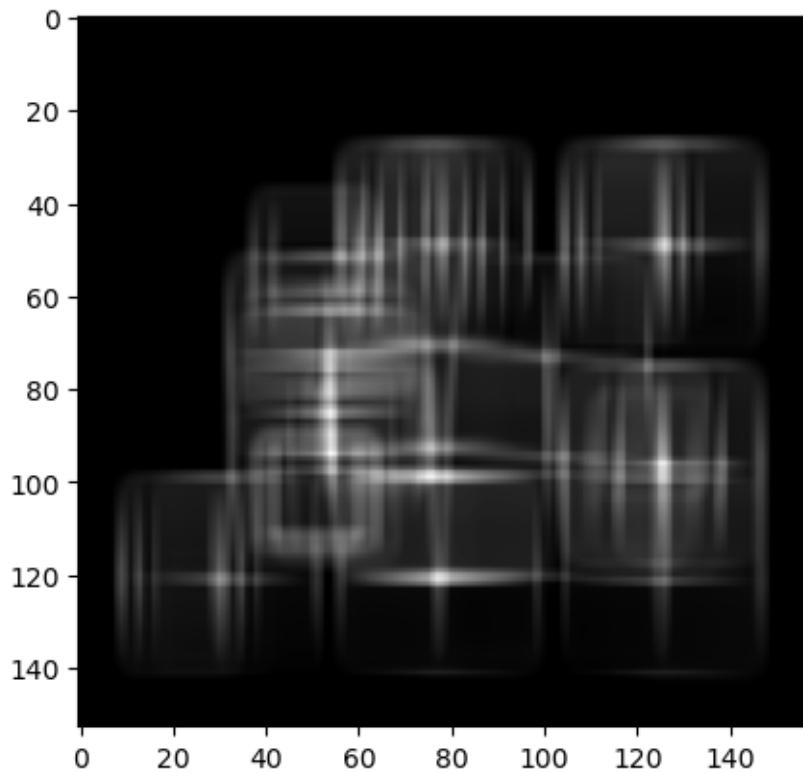
21



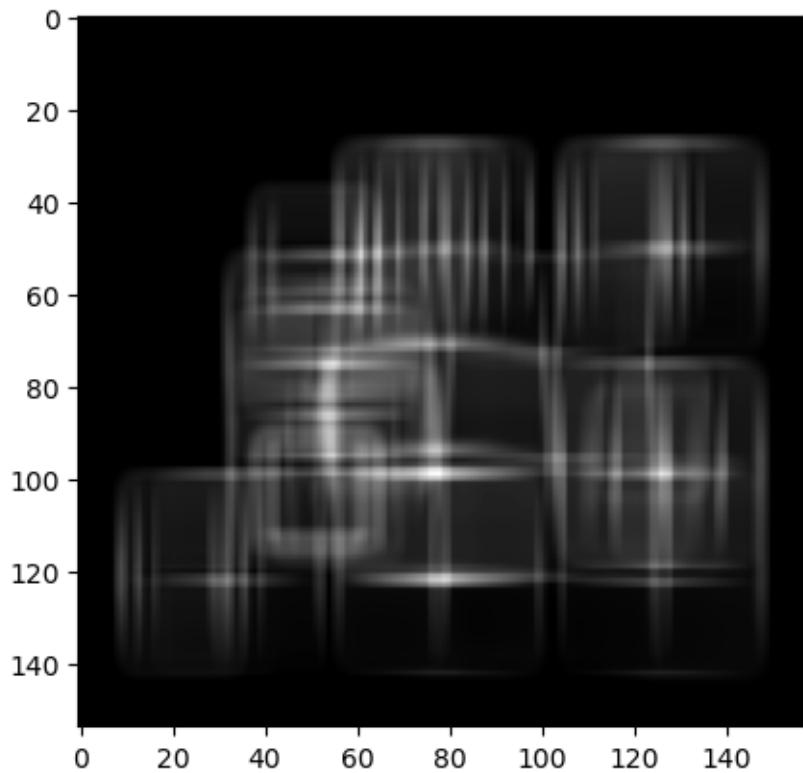
22



23

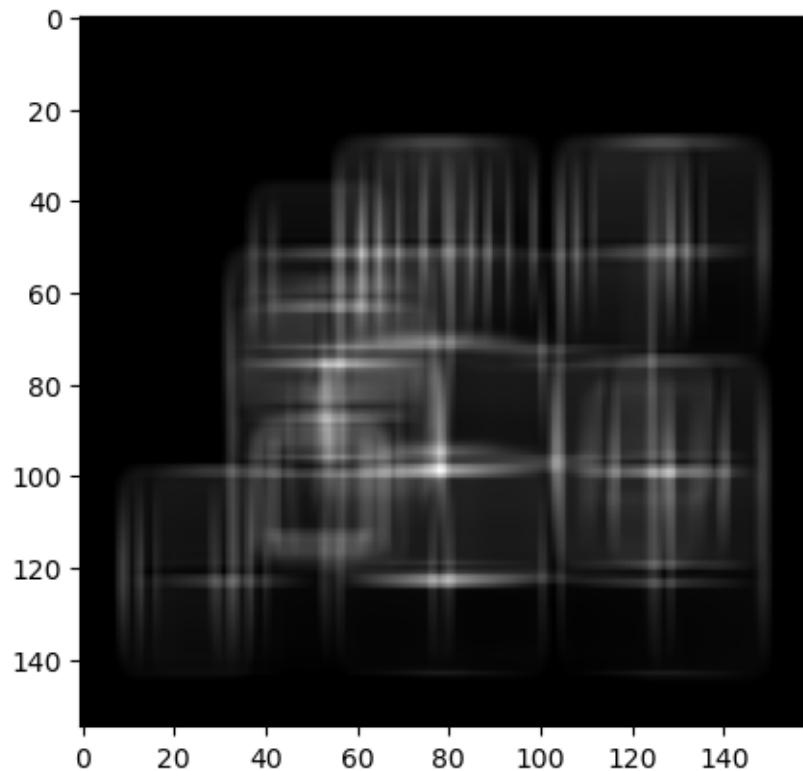


24



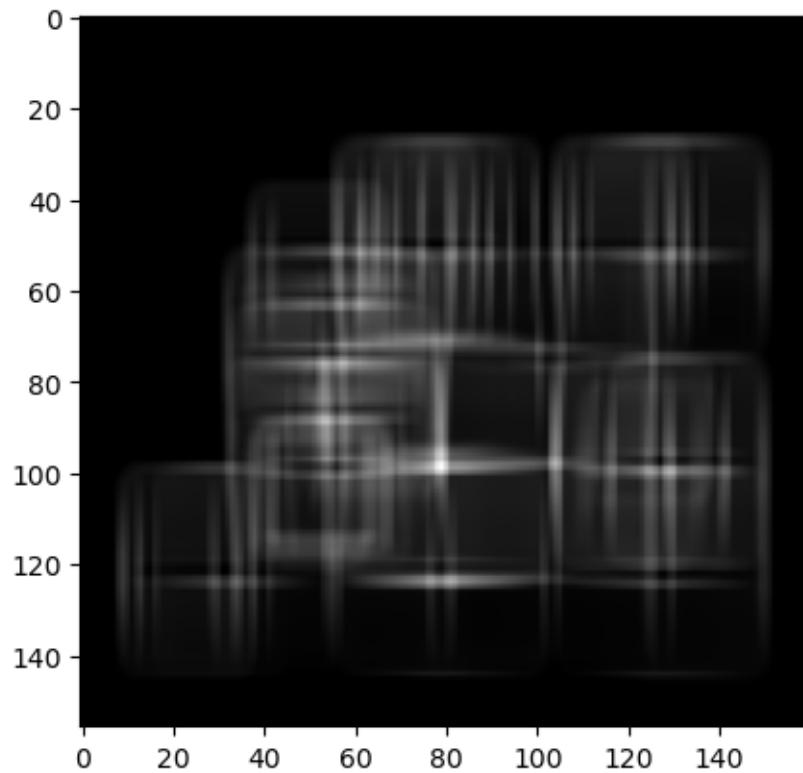
25

20



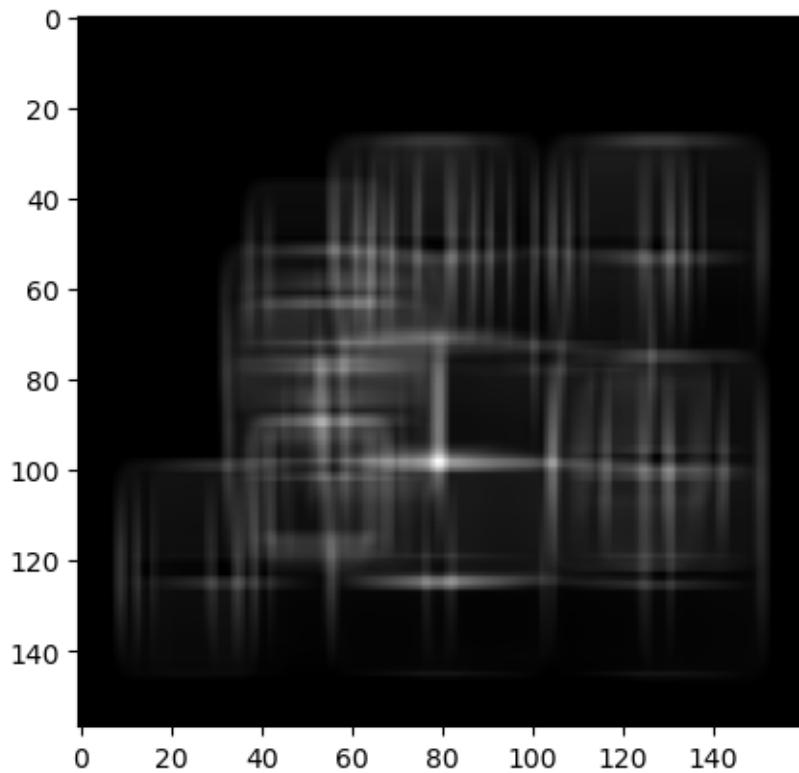
26

21



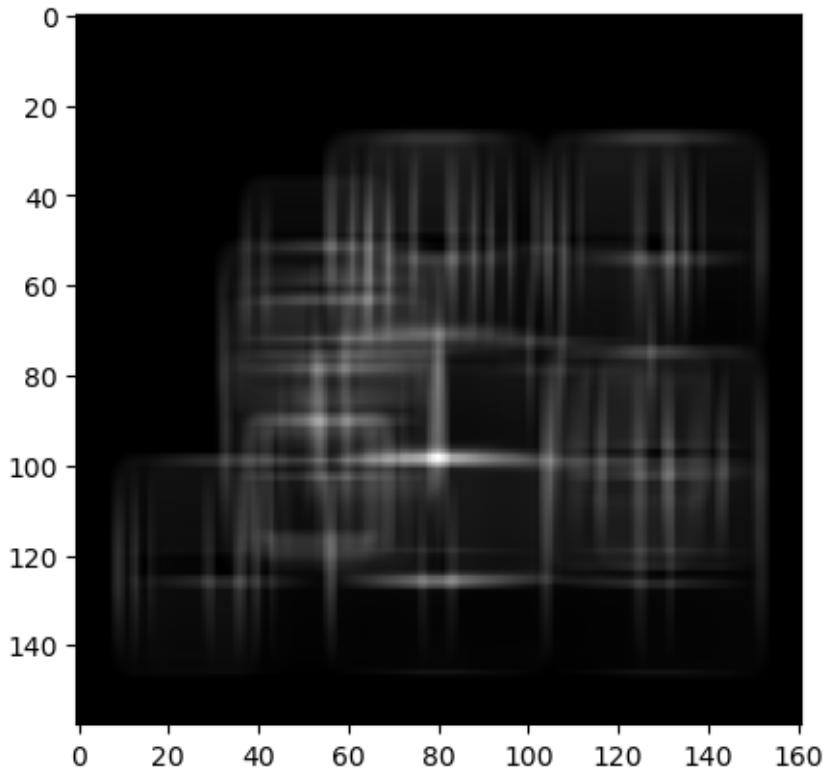
27

22



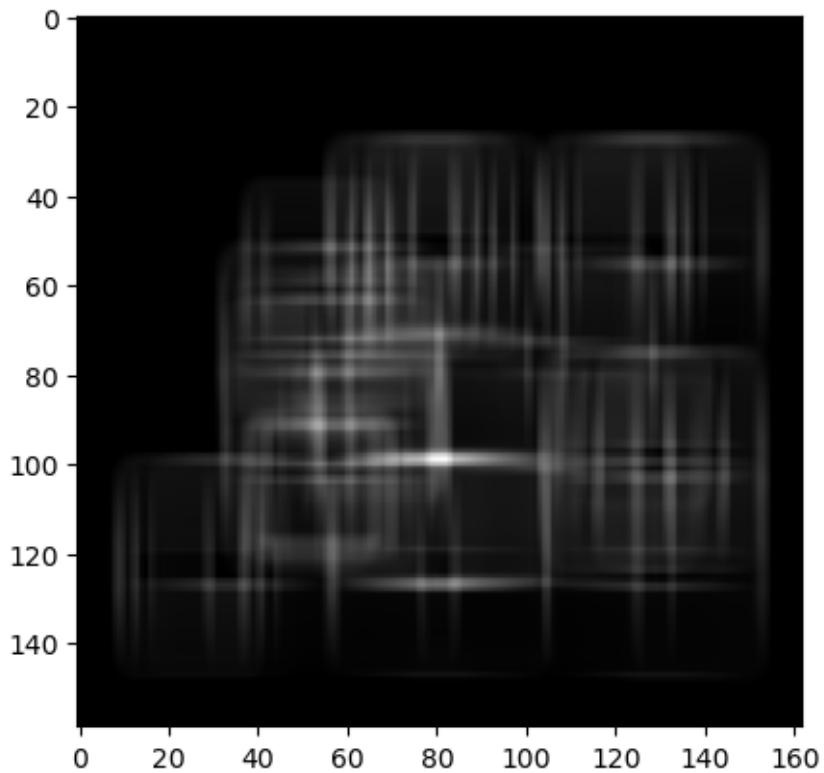
28

23

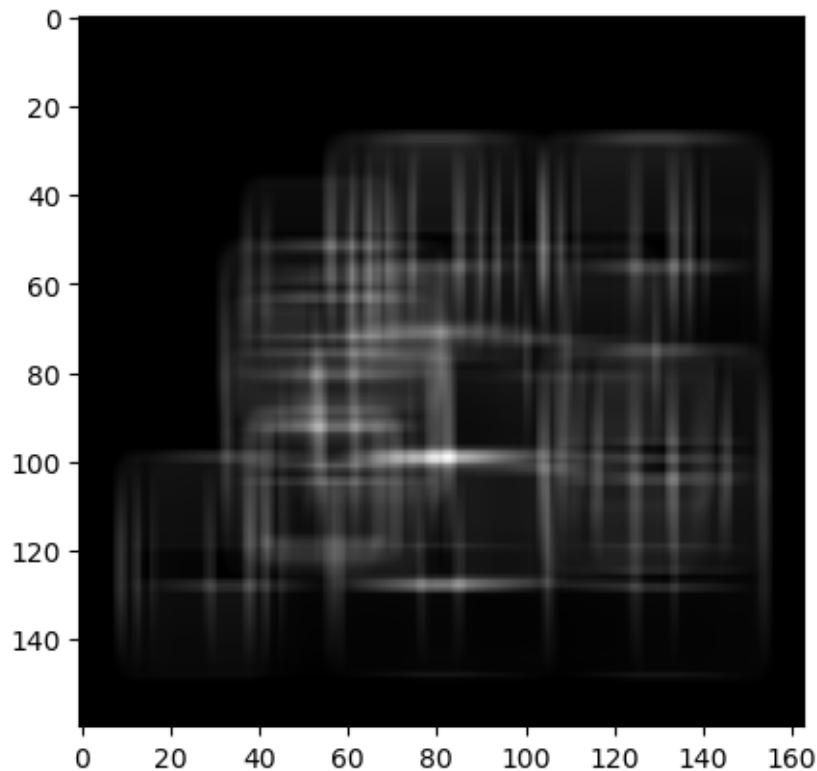


29

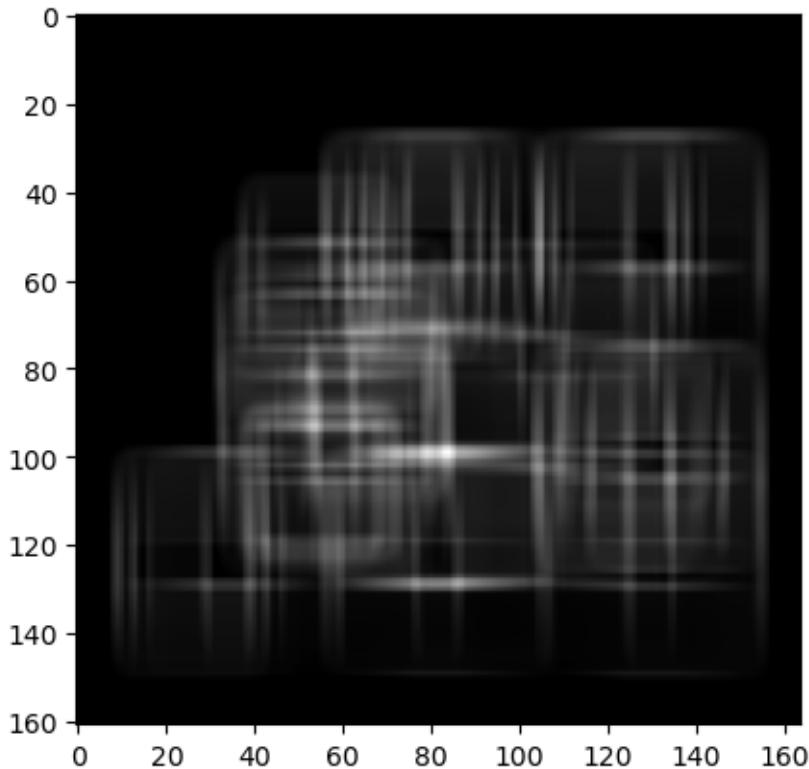
24



30

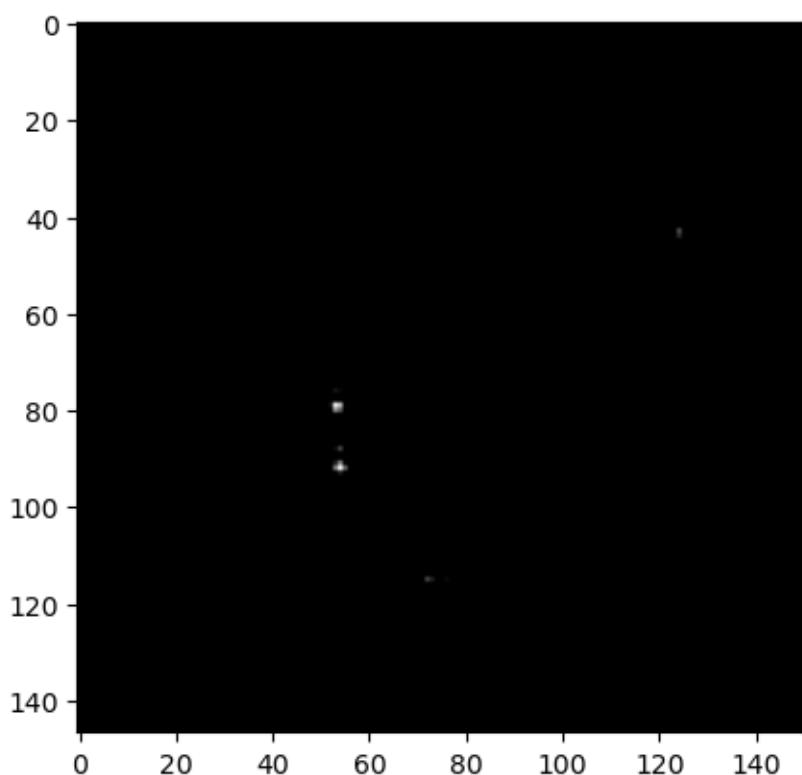
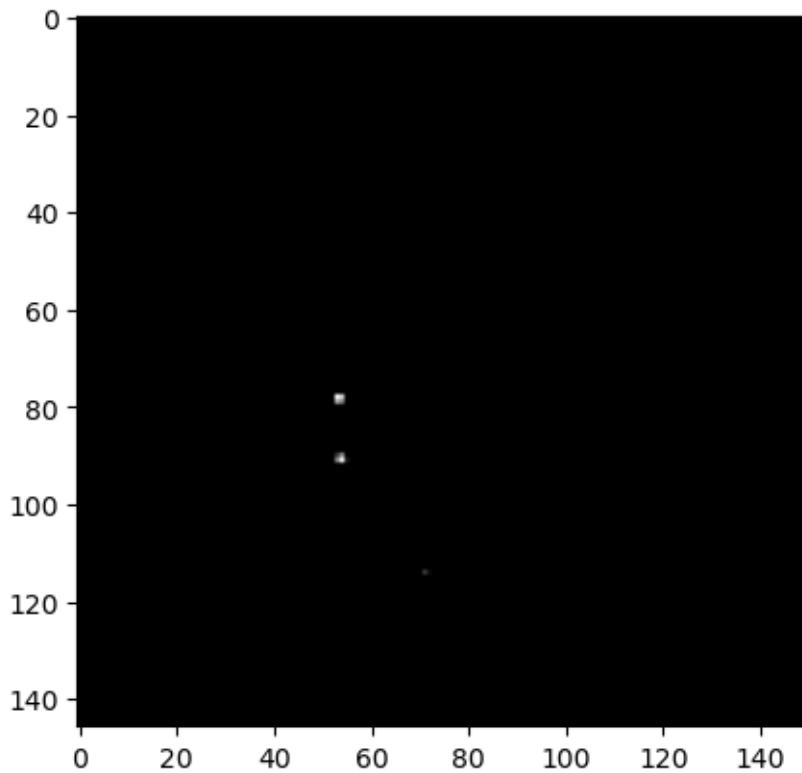


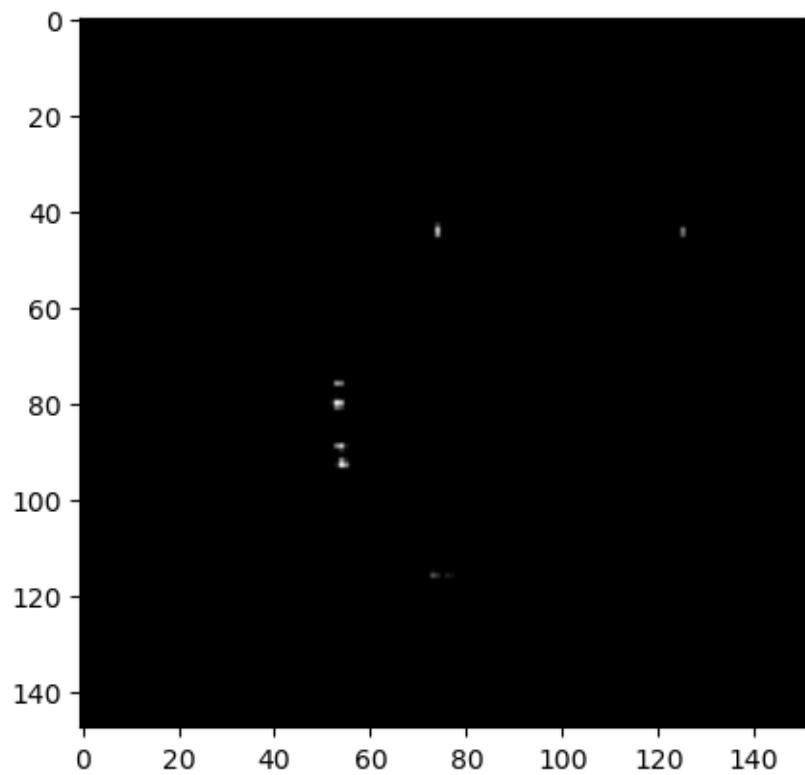
31

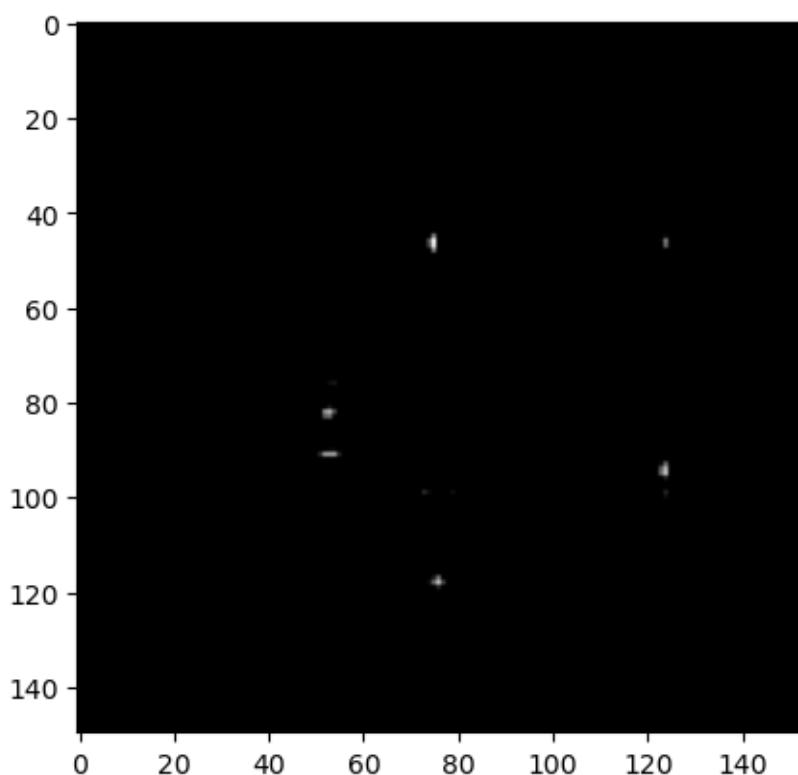
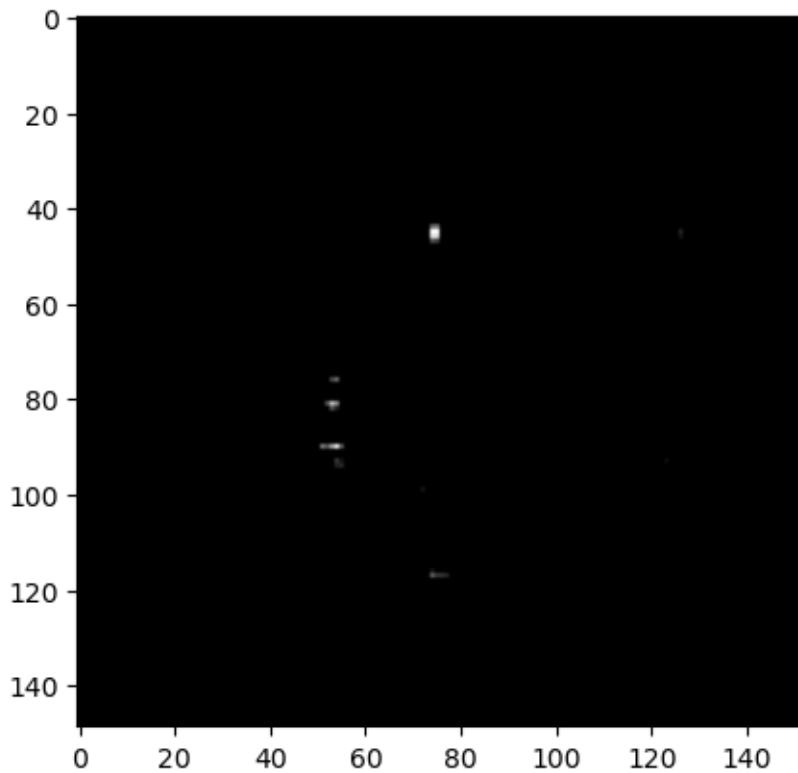


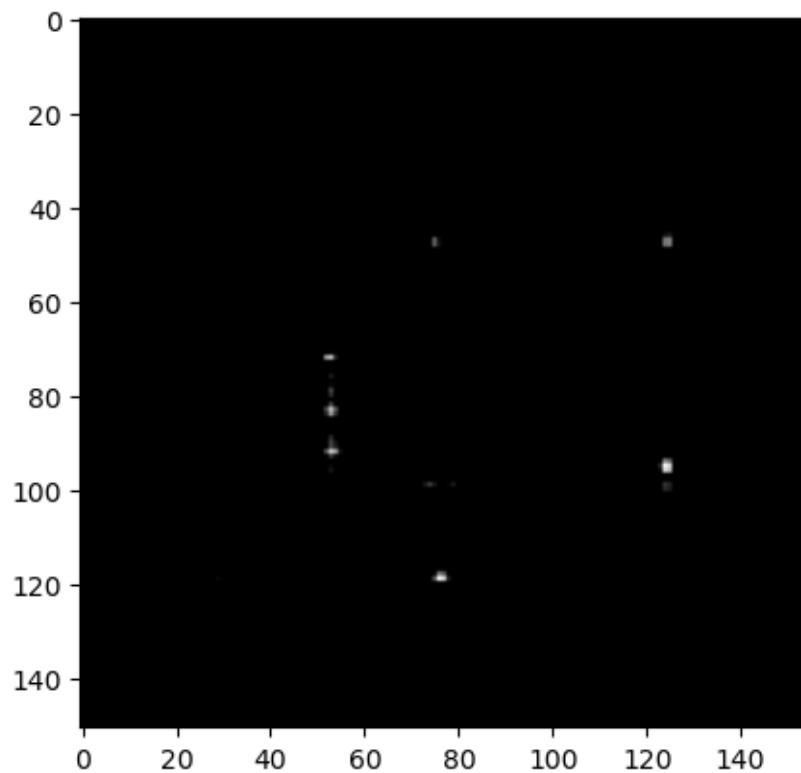
Przestrzenie głosowania wyglądają dosyć ciekawie, powinniśmy je jeszcze przepuścić przez relu. Dla każdego kwadratu. otrzymujemy zasadniczo krzyż z najjaśniejszym punktem w środku. Dzieje się tak ponieważ w środku potencjalnie lapiemy wszystkie punkty na obwodzie, a przesuwając się pionowo lub poziomo, wciąż dostajemy głosy od 2 równoległych krawędzi, ale tracimy głosy od 2 pozostałych krawędzi (również równoległych, ale prostopadłych do przemieszczenia). Do liczenia głosów wykorzystałem conv2d\_transpose, które działa bardzo podobnie do conv2d, ale dodatkowo dokonuje transpozycji macierzy wejściowej.

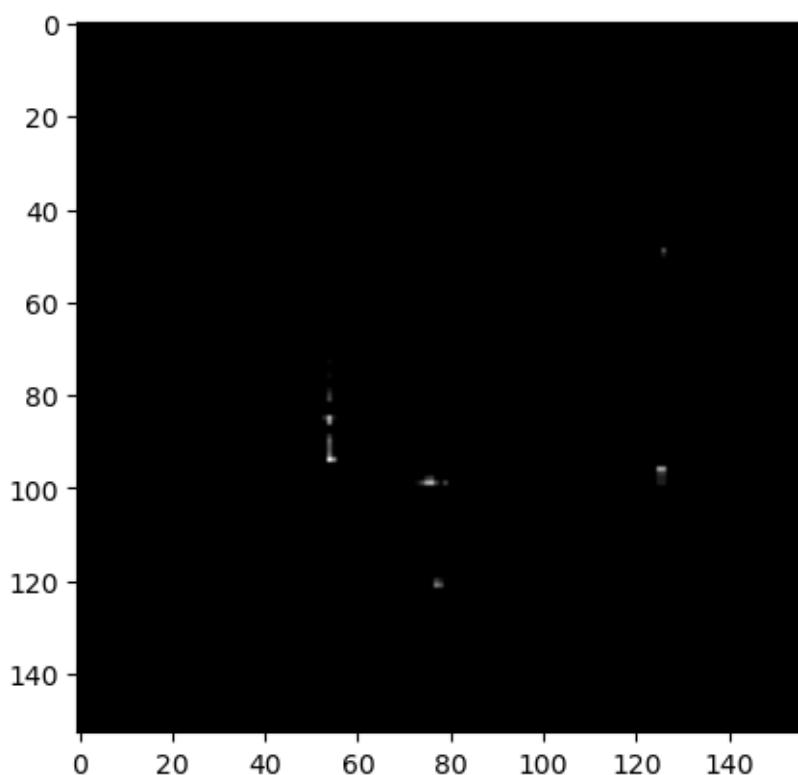
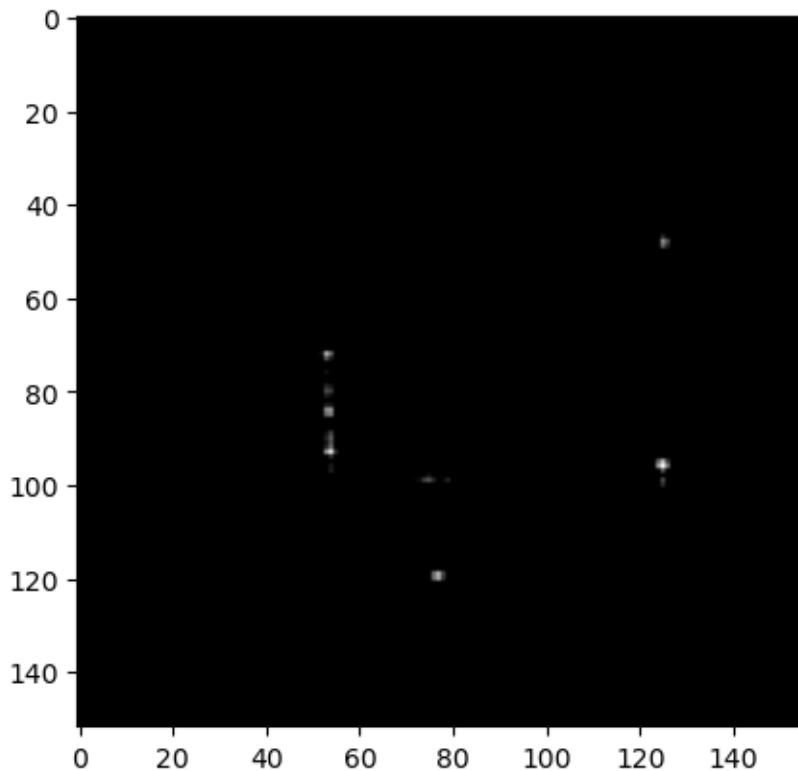
```
[ ]: for ix, img in enumerate(imgs):
    imgs[ix]=try_relu(img, 2*ix/3+16)
```

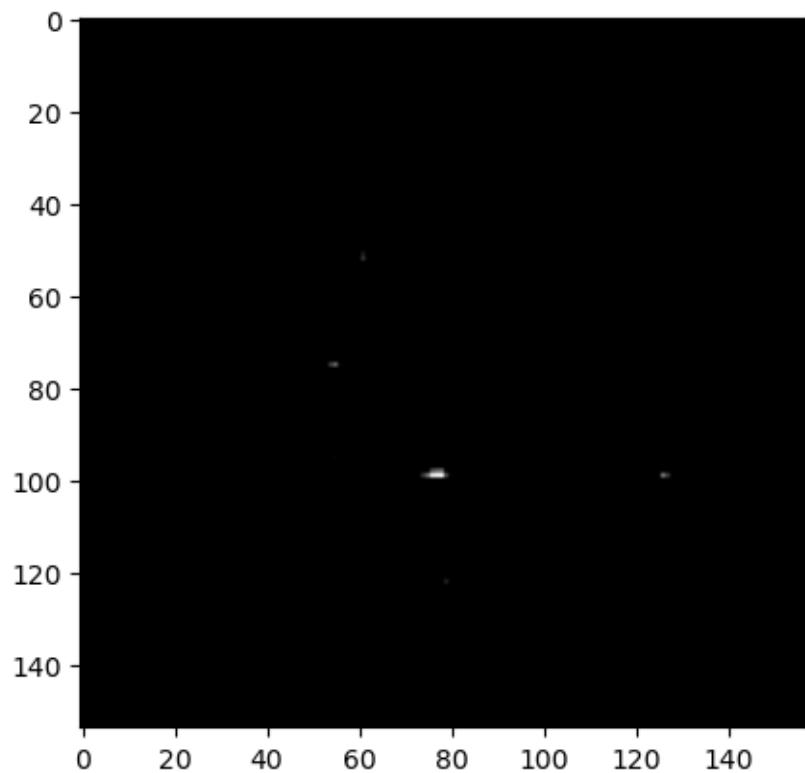


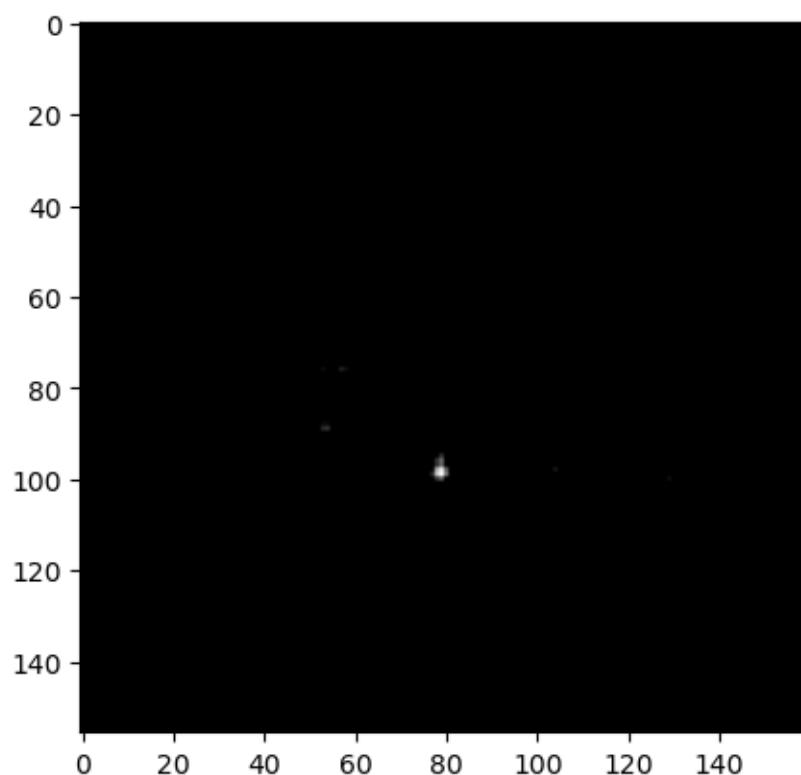
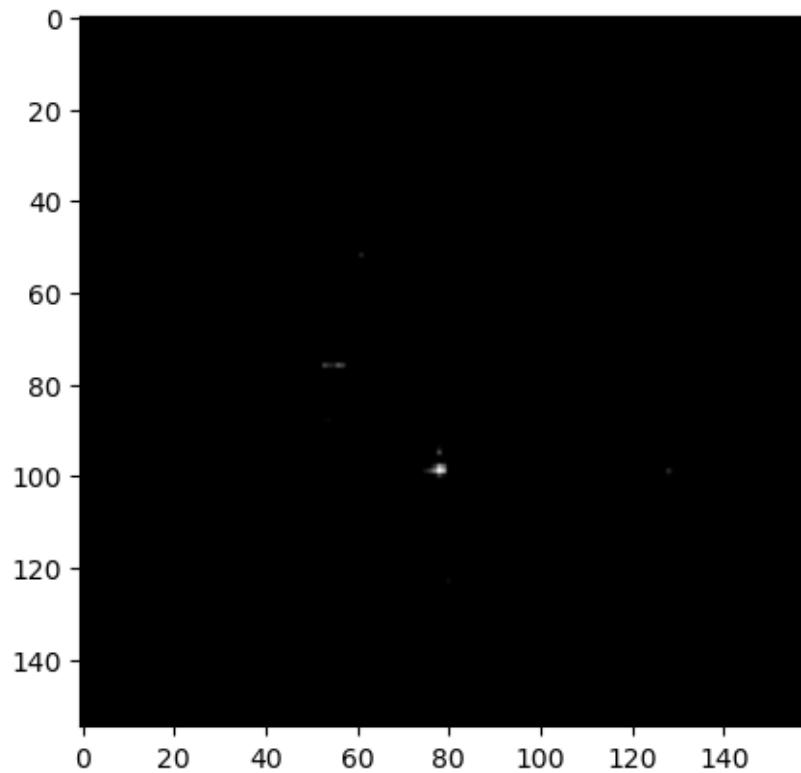


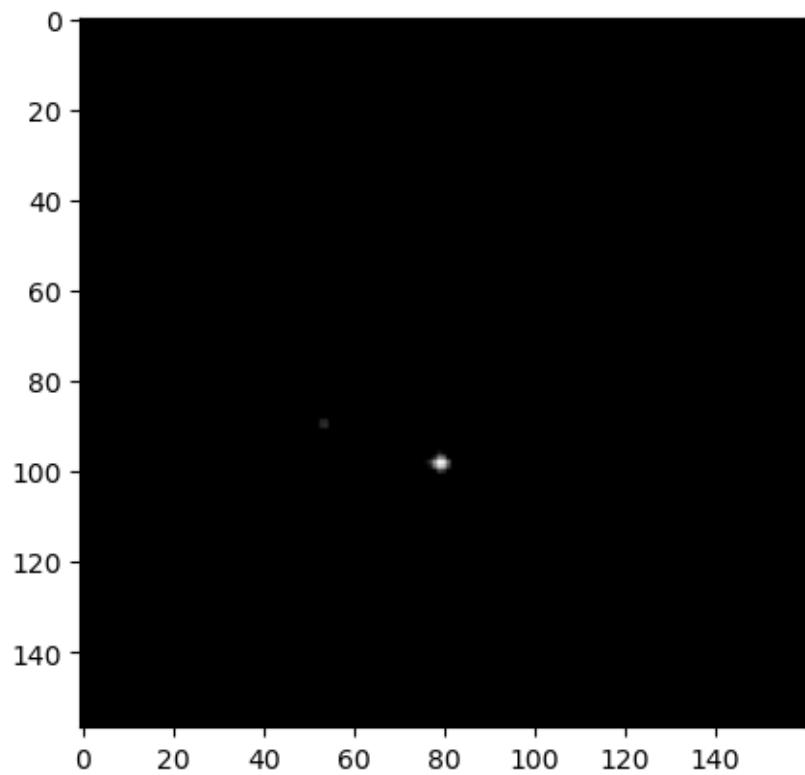


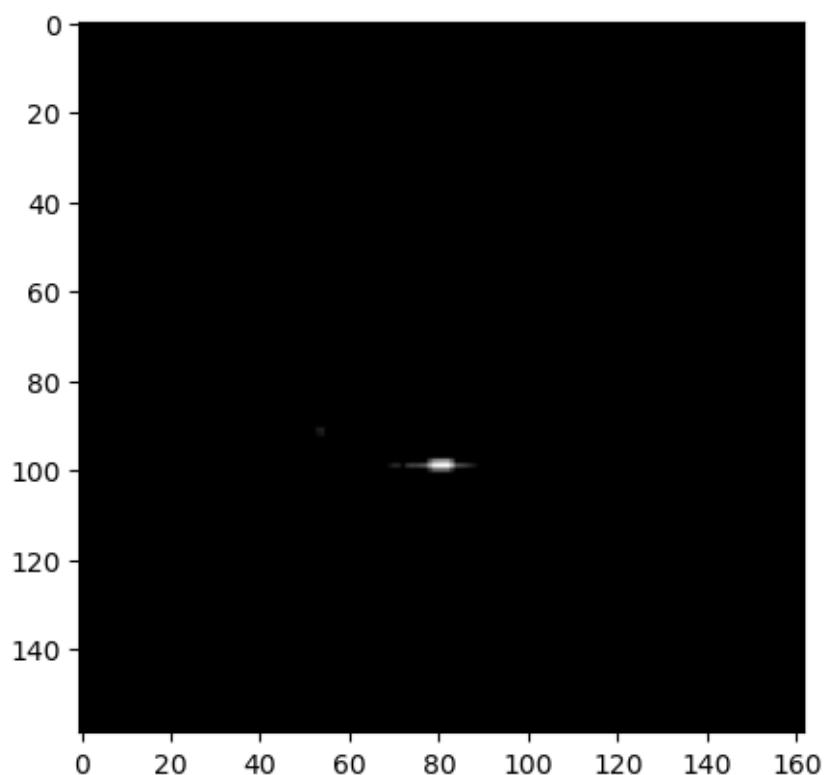
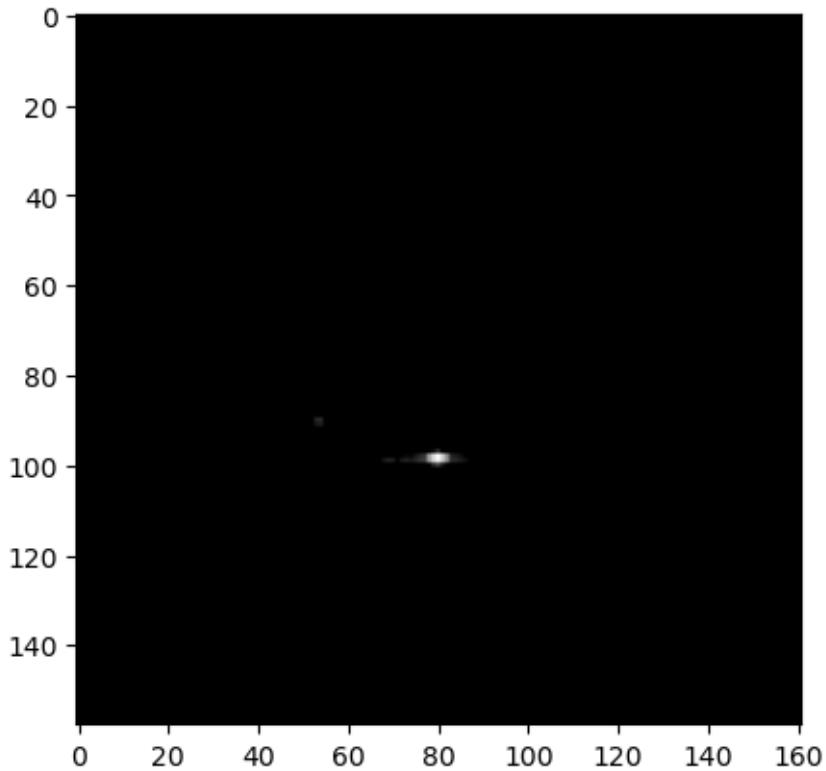


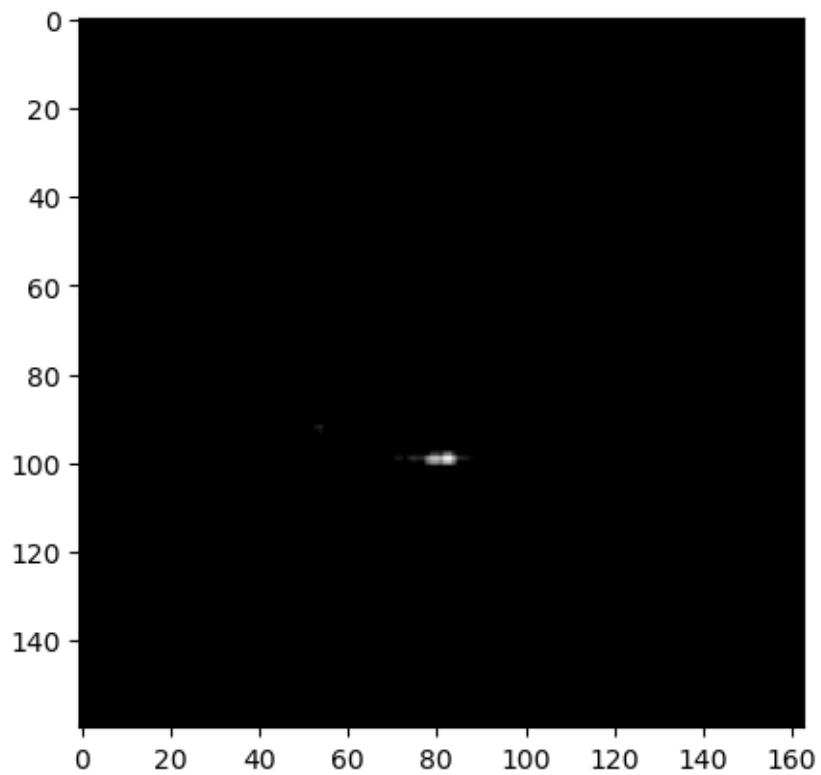


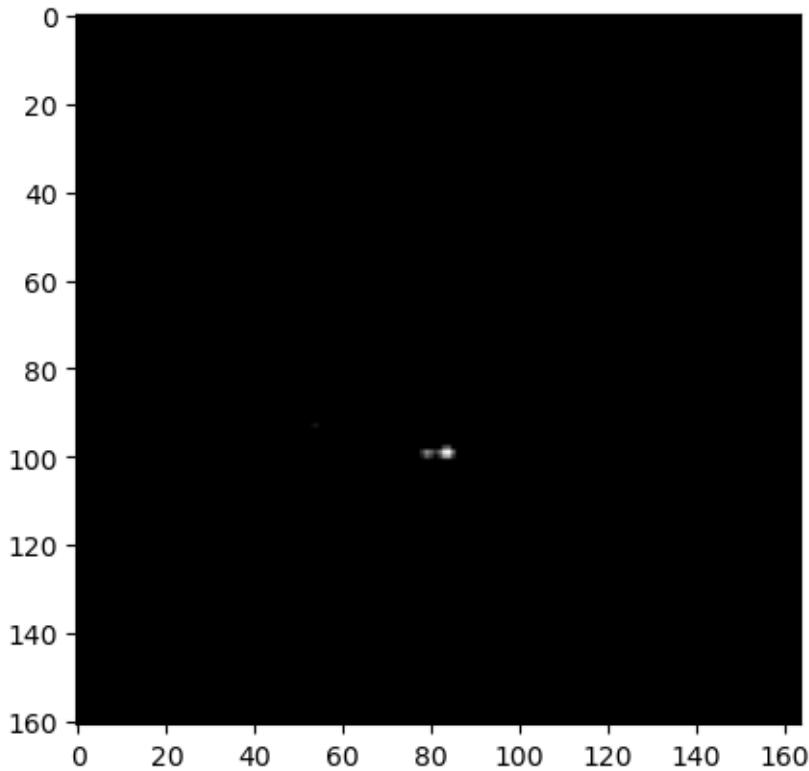






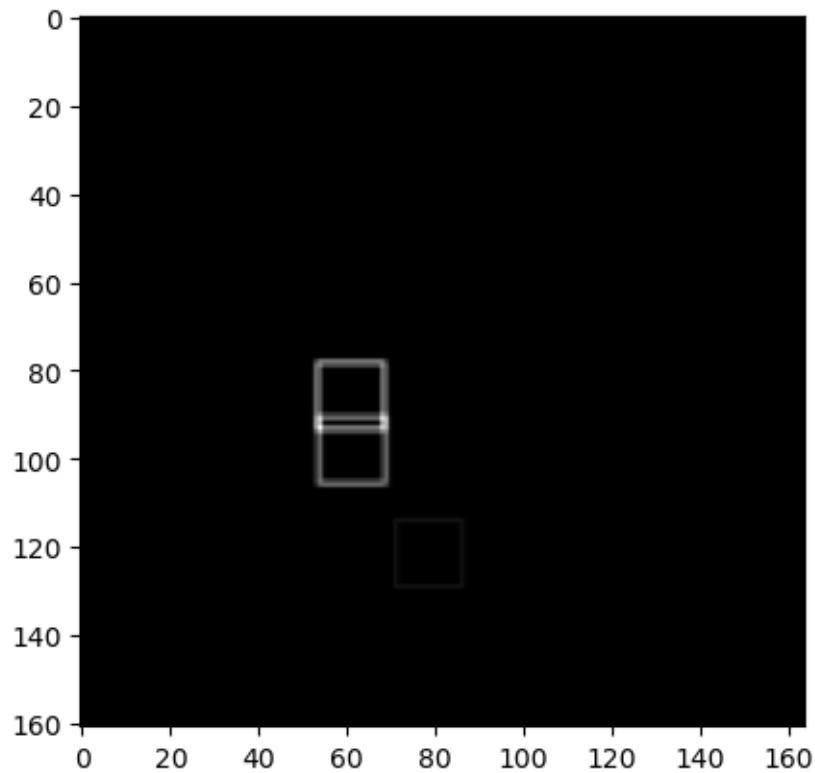




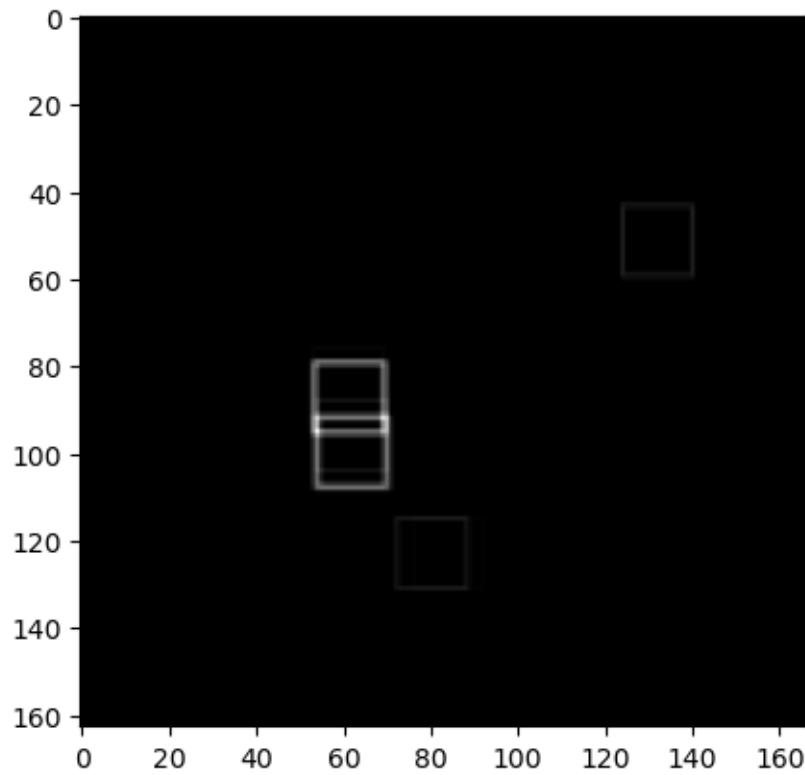


```
[ ]: res = torch.zeros(img.shape)
for ix, imgx in enumerate(imgs):
    print(ix+16)
    tmp=try_hough(imgx, ix+16)[0:img.shape[0],0:img.shape[1]]
    res+=(tmp/tmp.max())#*(16-ix+2))
```

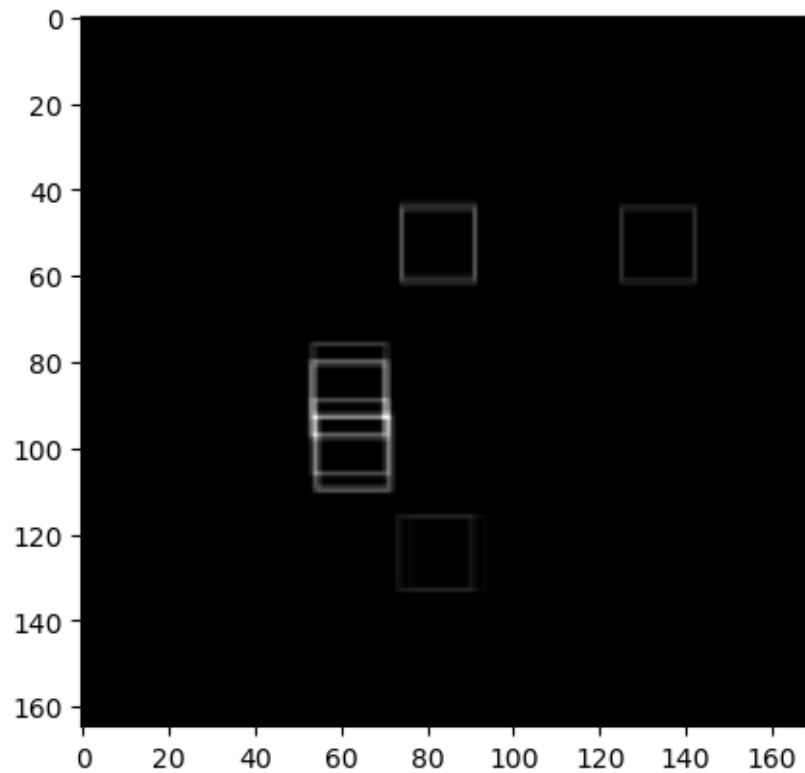
16



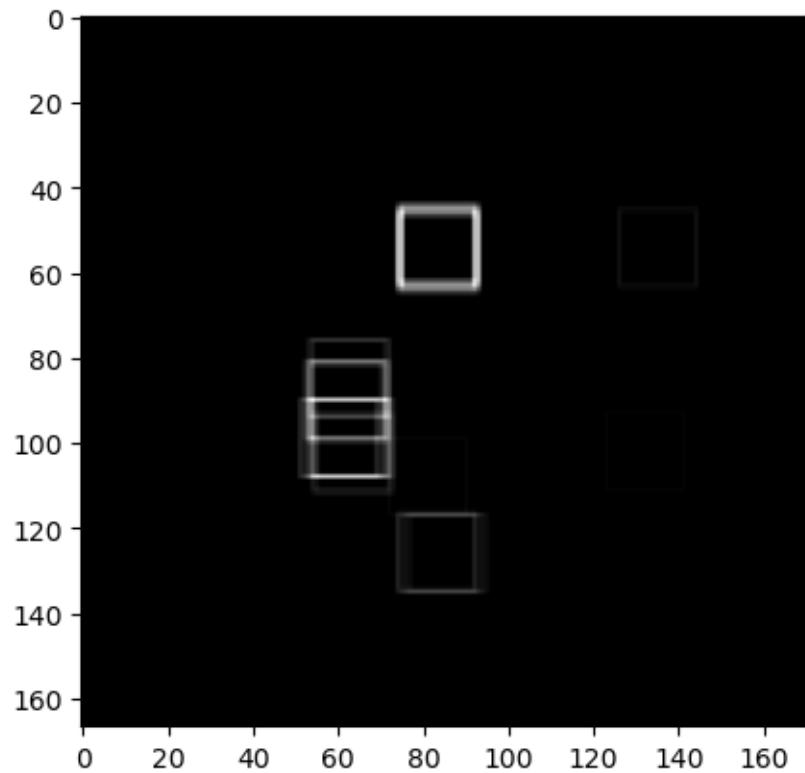
17



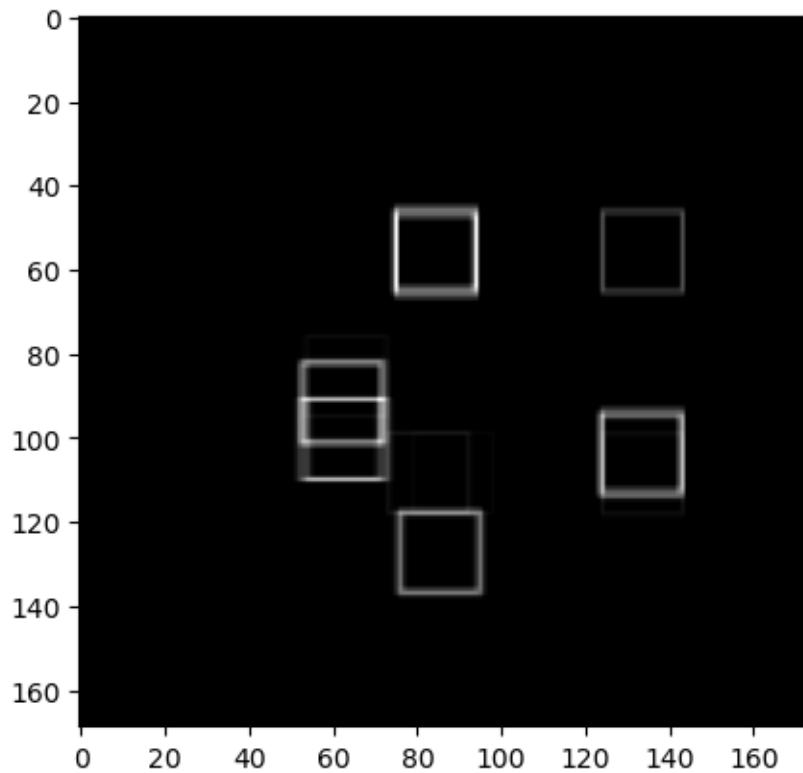
18



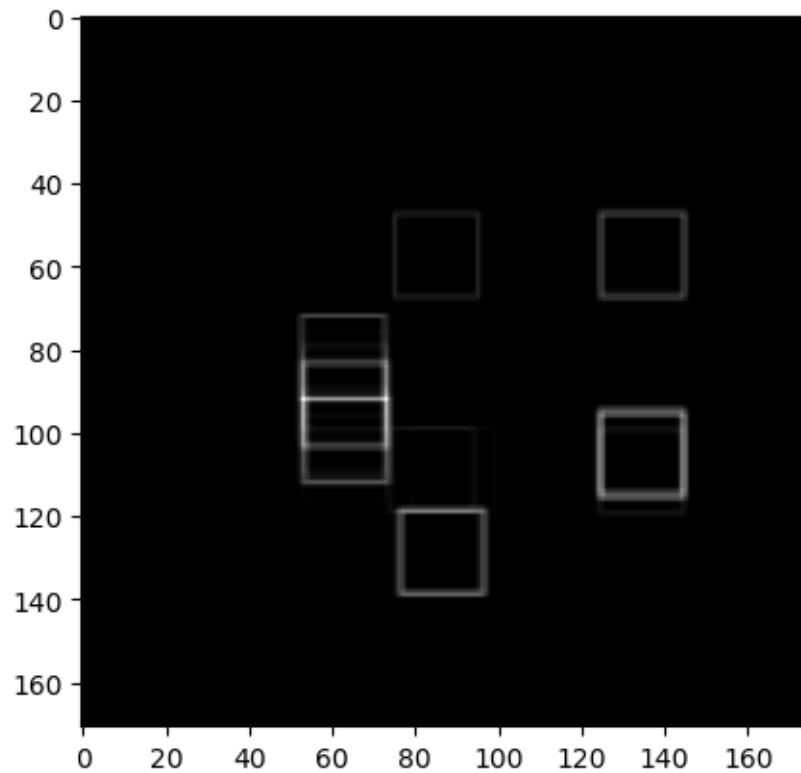
19



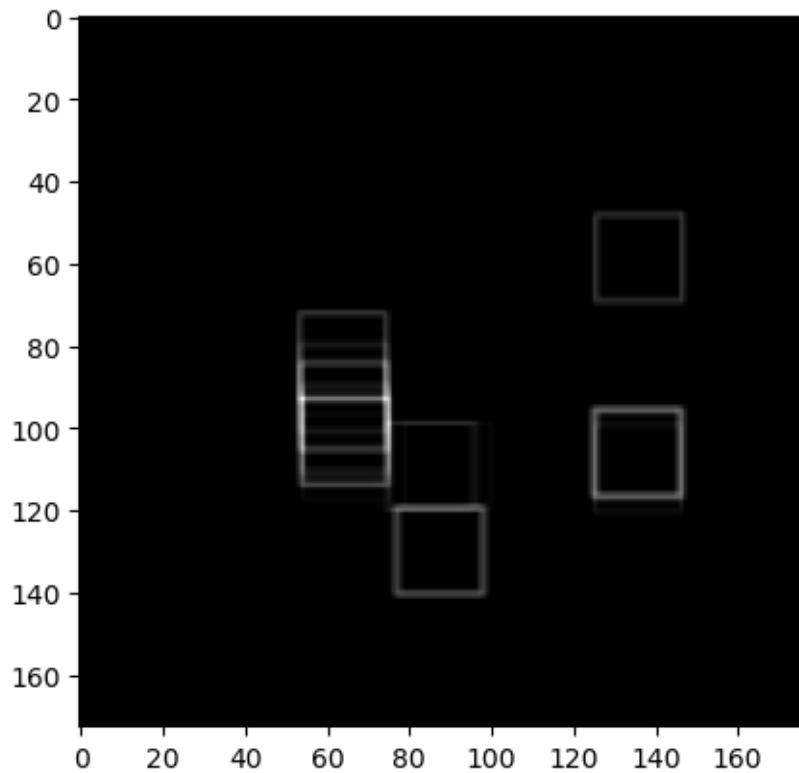
20



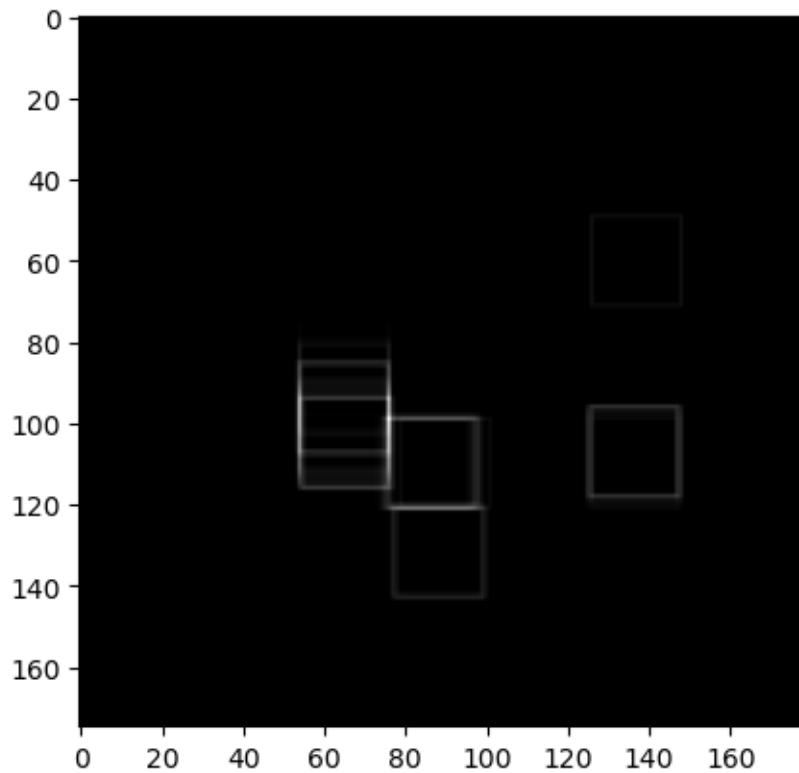
21



22

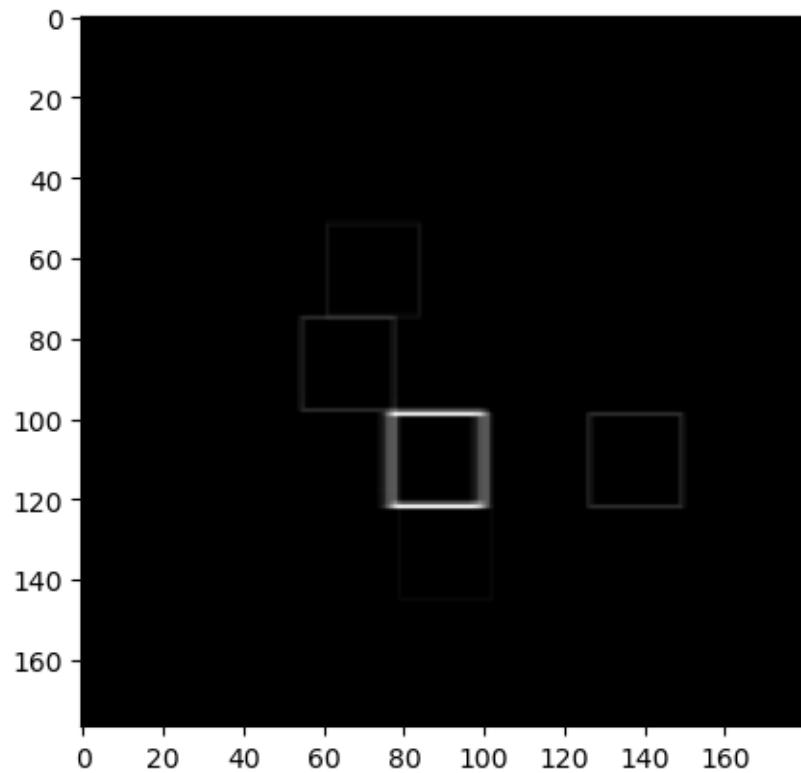


23

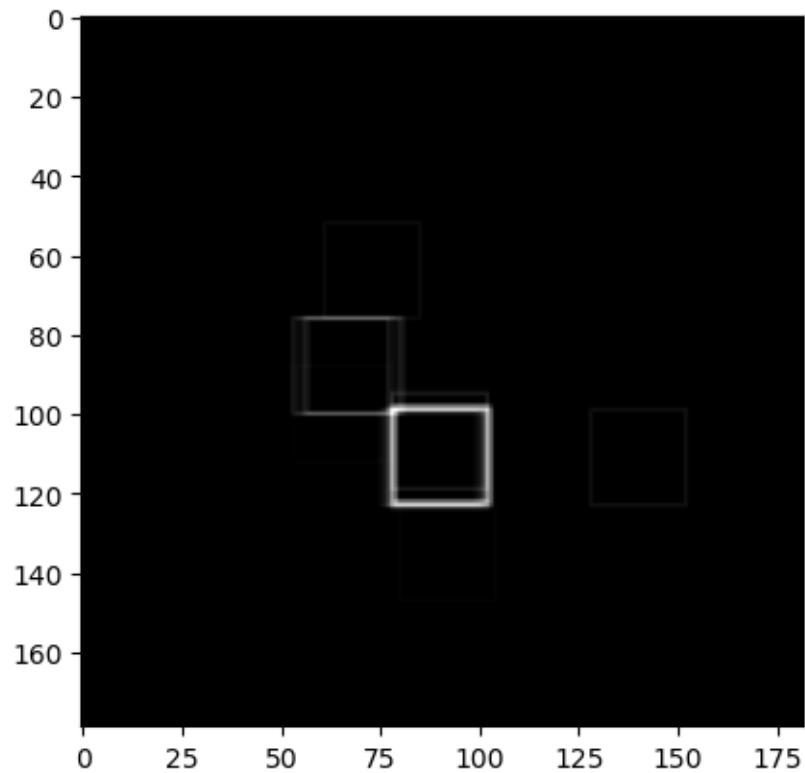


24

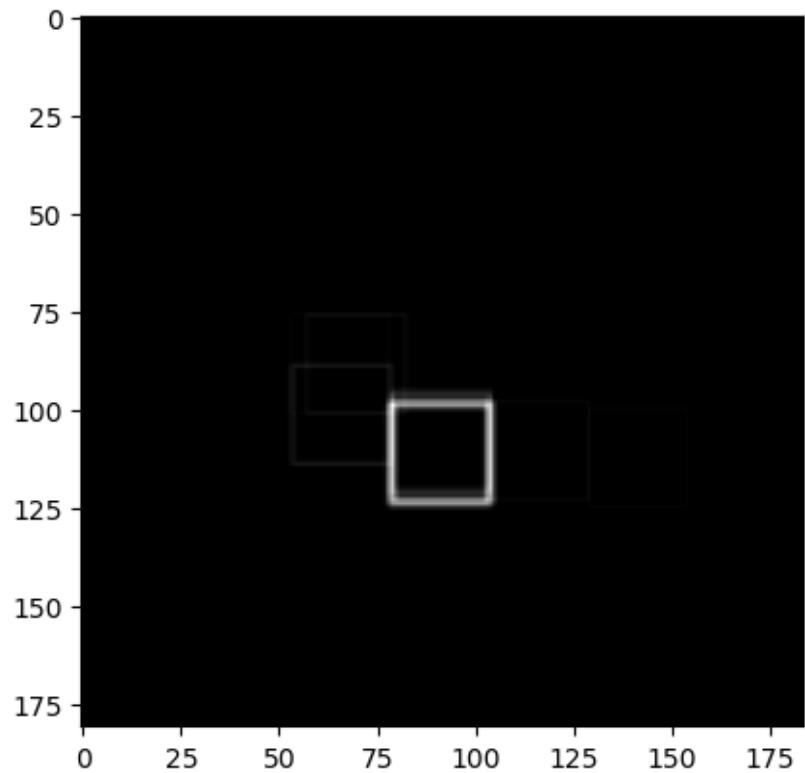
46



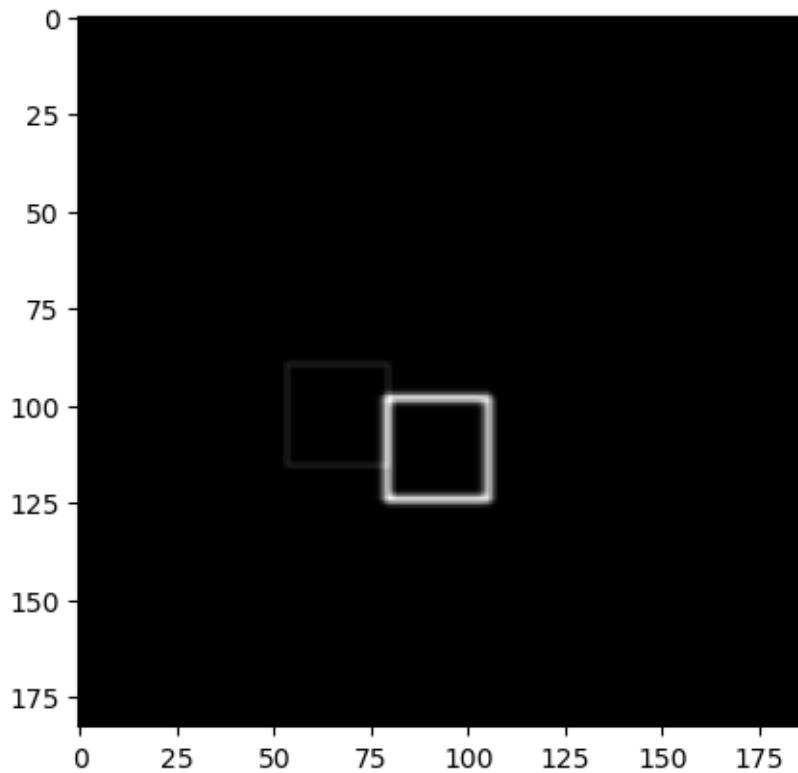
25



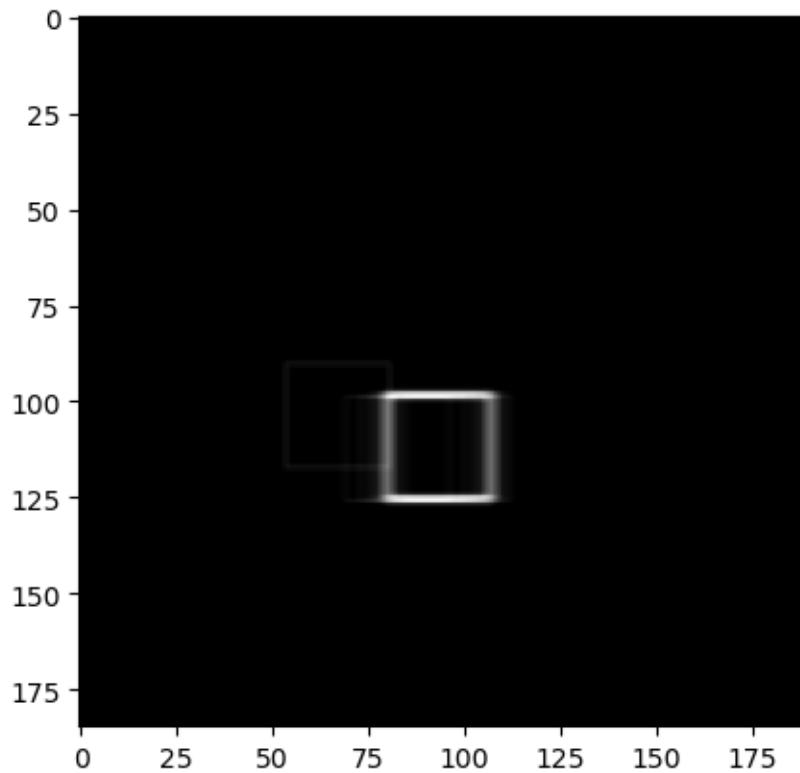
26



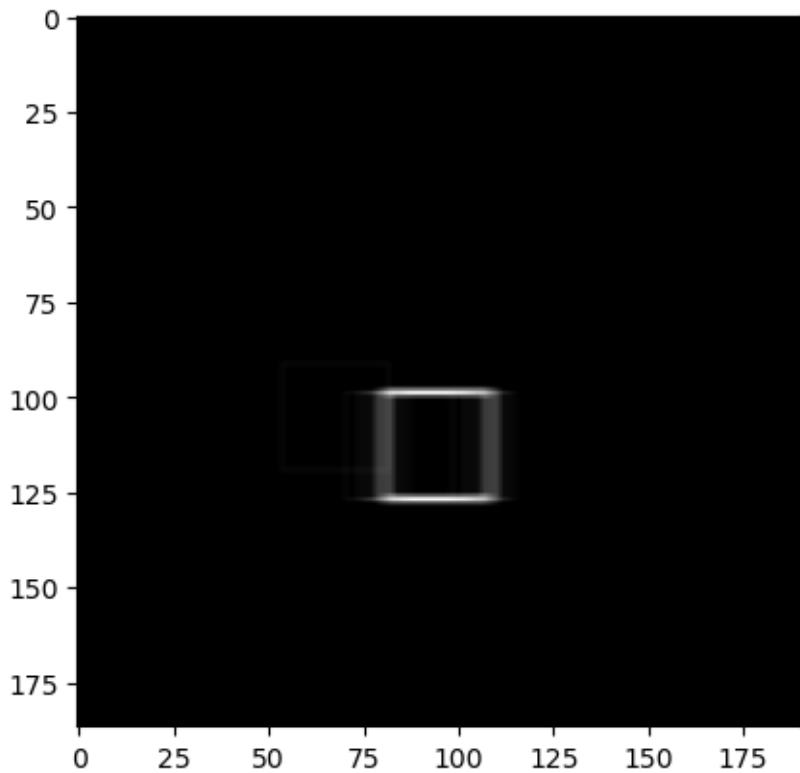
27



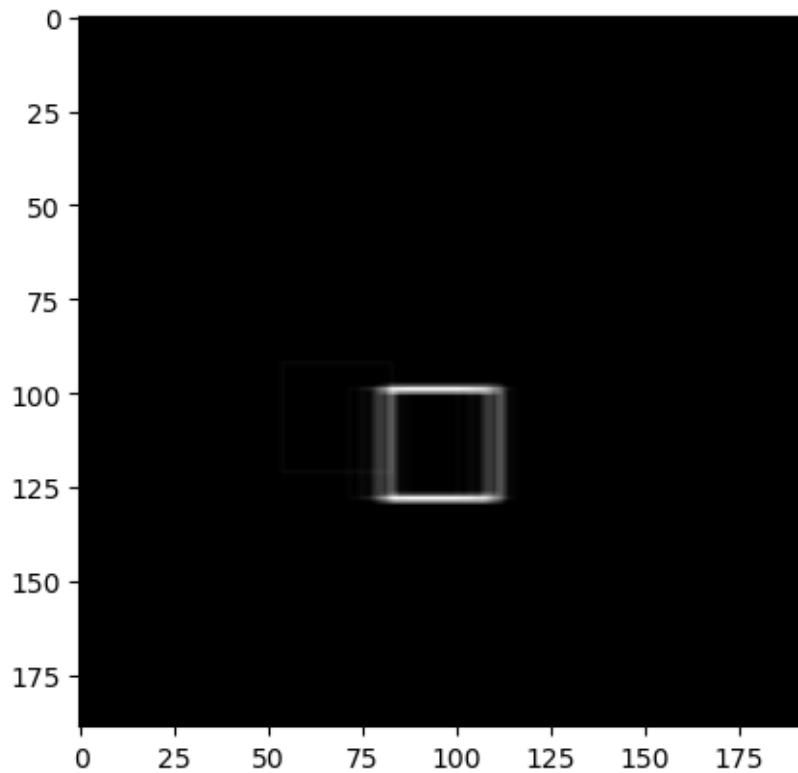
28



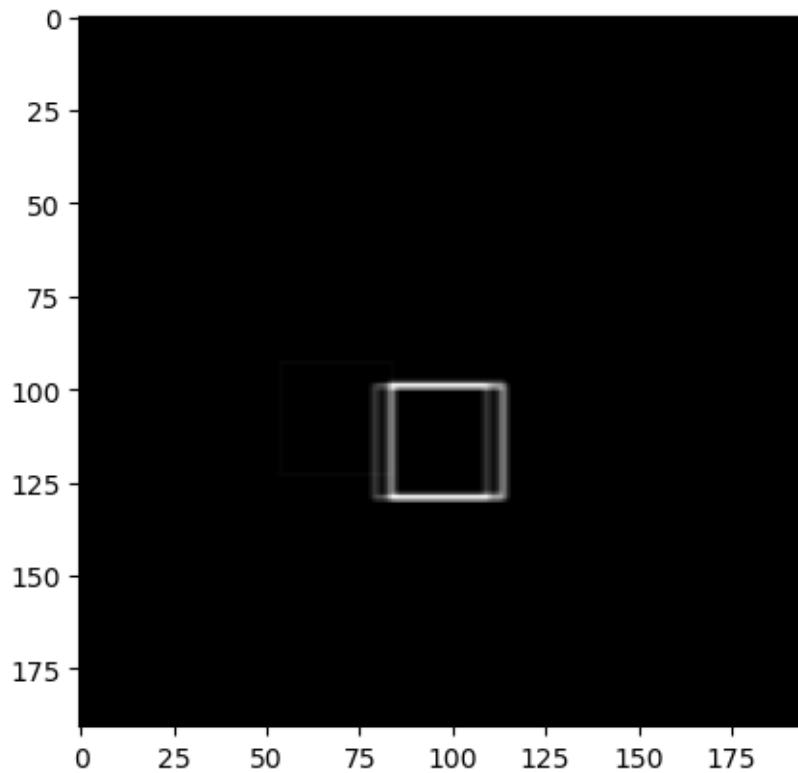
29



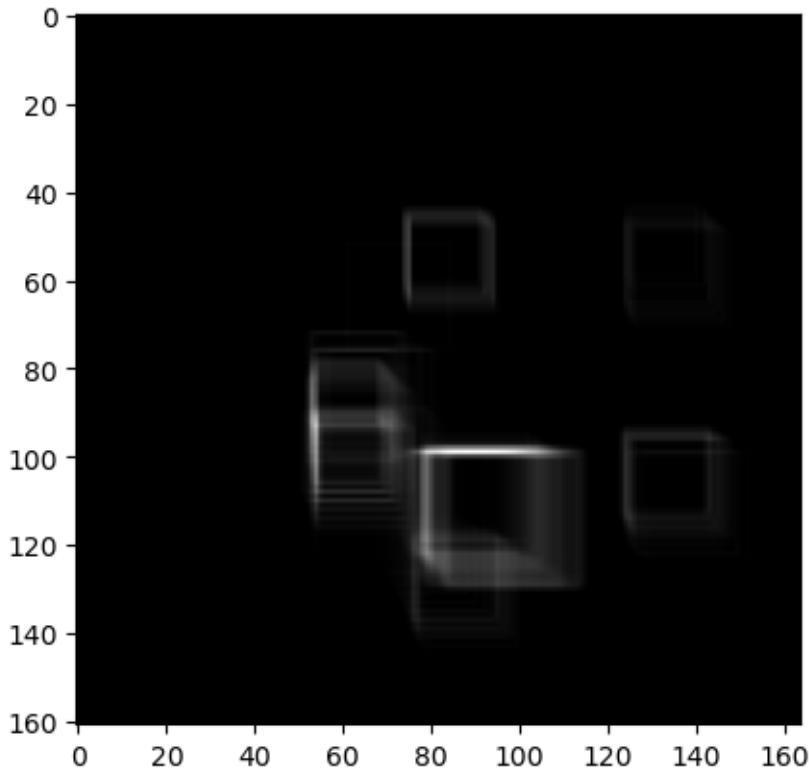
30



31



```
[ ]: print_gray(res)
```



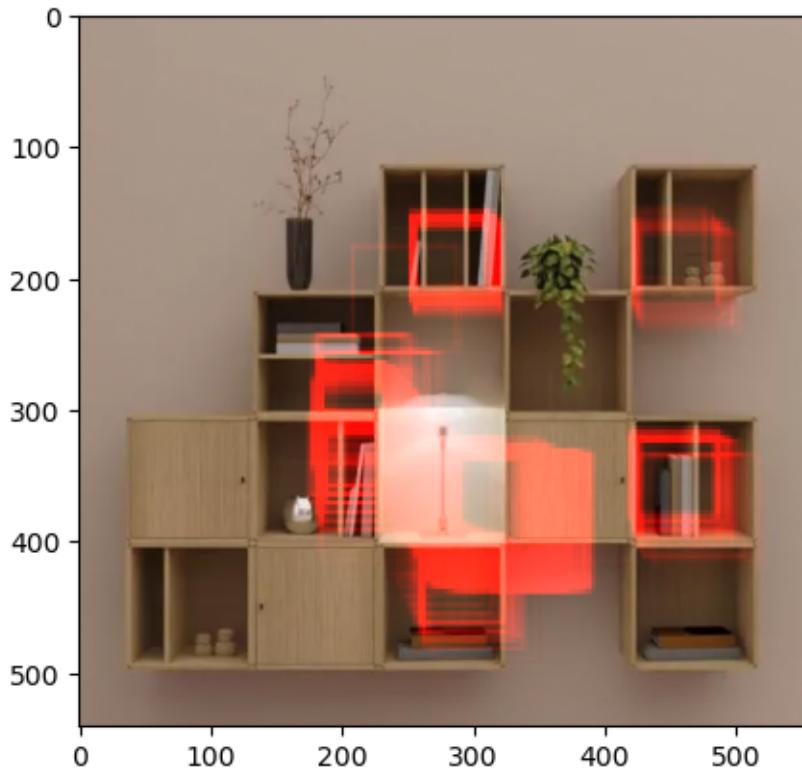
```
[ ]: row_f = ori_img.shape[0]//res.shape[0]
      col_f = ori_img.shape[1]//res.shape[1]

[ ]: upscaled_sq = torch.squeeze(F.interpolate(torch.unsqueeze(torch.
      ↵unsqueeze(res, dim=0), dim=0), size=ori_img[:, :, 0].shape))

[ ]: img_with_sq = ori_img
      img_with_sq[:, :, 0] += upscaled_sq

[ ]: plt.imshow(img_with_sq)
      plt.show()
```

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



Niestety wynik nie jest wybitny, ale moim zdaniem ma trochę sensu - kwadraty znajdują się mniej więcej w dobrych miejscach.

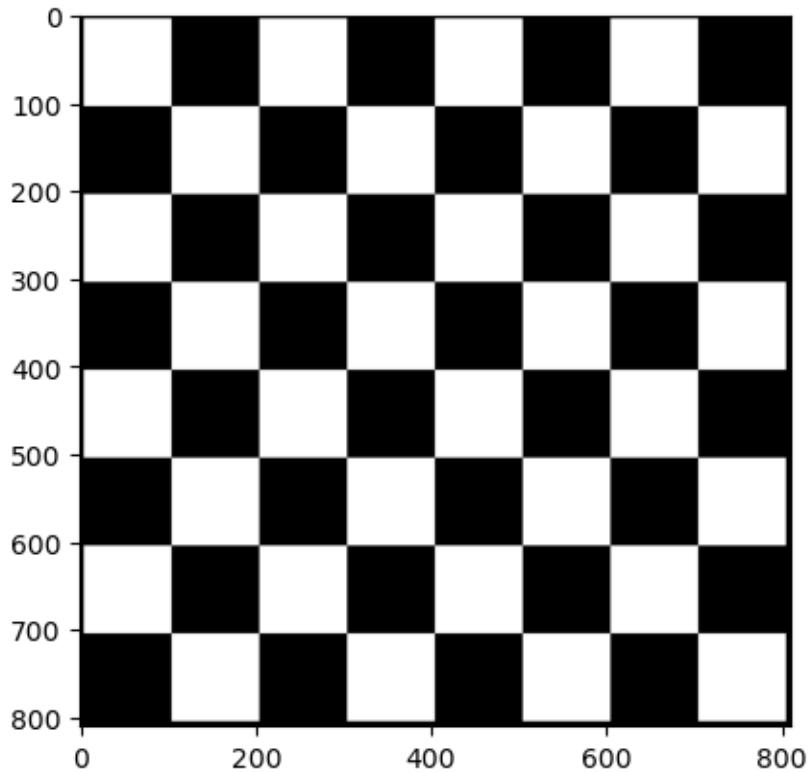
Niestety nie mam wytarczająco czasu, ani na ten moment pomysłu jak poprawić moją implementację, ale wydaje mi się, że największe problemy wynikają z:

- Upscaling - szczególnie że przy różnych konwolucjach - szczególnie transponowanych, obraz zwiększał rozmiar a nie powinien (nie umiałem temu zapobiec...), przez co kwadraty zostały powiększone mniej niż powinny - downscaling - później przy upscalingu wynikają problemy, dodatkowo wybrałem pooling max, co sprawiało, że krawędzie były grubsze i intensywniejsze niż powinny, przez co wiele kwadratów jest bardzo rozmażanych - conv\_transpose2d - niestety nie wiem jak zapobiec rośnieciu obrazka, może powiniensem samemu na koniec obcinać obrazek do odpowiedniego rozmiaru.
- Nie dokończyliśmy wykrywania krawędzi - Ciężko jest dobrać sensowne wartości przy filtrowaniu dobrych kwadratów, aby nie usunąć za dużo oraz nie mieć za dużych zakłóceń - Do tego wszystkiego, sam obrazek jest dosyć trudny Moim zdaniem z tych powodów miejsca gdzie kwadraty się znajdują nie zgadzają się bardzo dokładnie oraz rozmiary kwadratów pozostawiają trochę do życzenia.

Poniżej powtarzam wszystko dla nowego obrazka.

```
[ ]: ori_img = torch.tensor(imread('Chess_Board.png'))/255
print(ori_img.shape)
print_gray(ori_img)

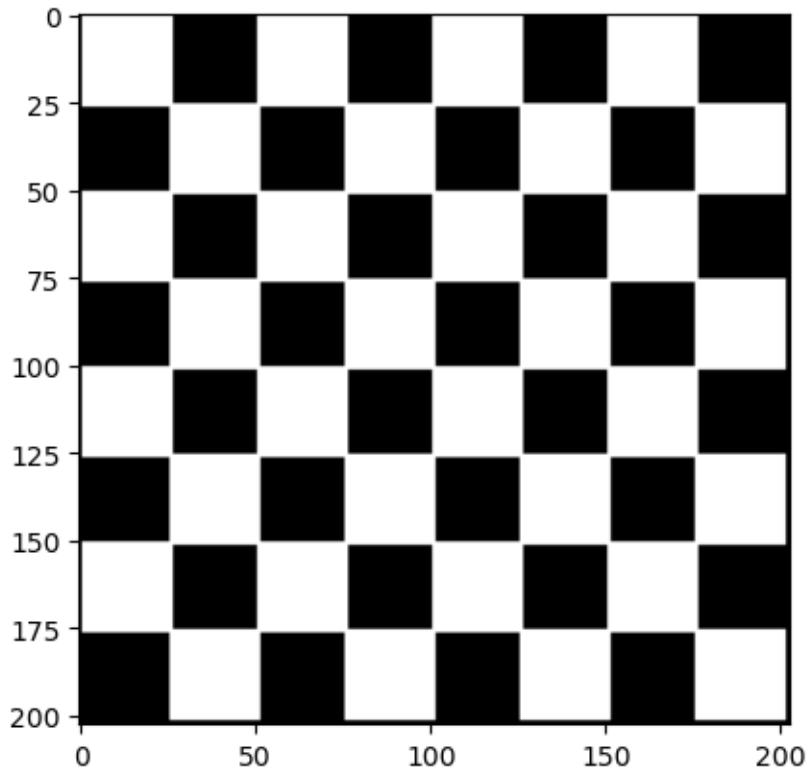
torch.Size([810, 810])
```



```
[ ]: print(f'{to_conv(ori_img).shape} - {ori_img.shape}')
print(f'{from_conv(to_conv(ori_img)).shape} - {ori_img.shape}')
```

```
torch.Size([1, 1, 810, 810]) - torch.Size([810, 810])
torch.Size([810, 810]) - torch.Size([810, 810])
```

```
[ ]: img2 = torch.clone(ori_img)
img2 = from_conv(F.max_pool2d(to_conv(img2), (4,4), padding=(2,2)))
print_gray(img2)
img2.shape
```



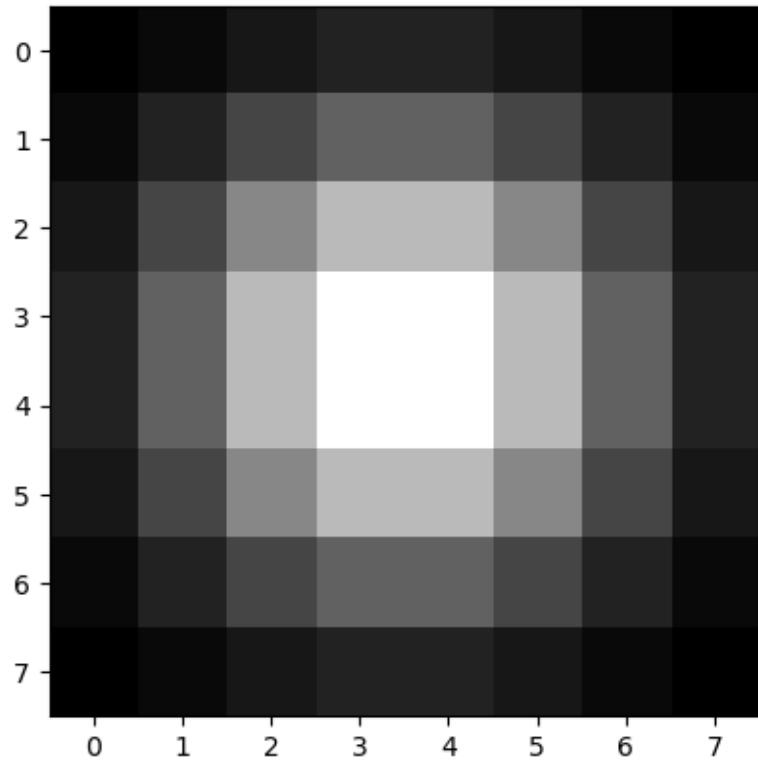
```
[ ]: torch.Size([203, 203])
```

Pooling wykonany przy pomocy konwolucji, biorący jako wartość średnia oraz pooling wykonany przy pomocy funkcji `torch.nn.functional.max_pool2d`(przyjmuje bardzo podobne argumenty do konwolucji). Dodatkowo przy pierwszej metodzie nie użyłem paddingu, więc obraz jest trochę mniejszy, niż przy drugiej gdzie użyłem paddingu będącego połową filtru.

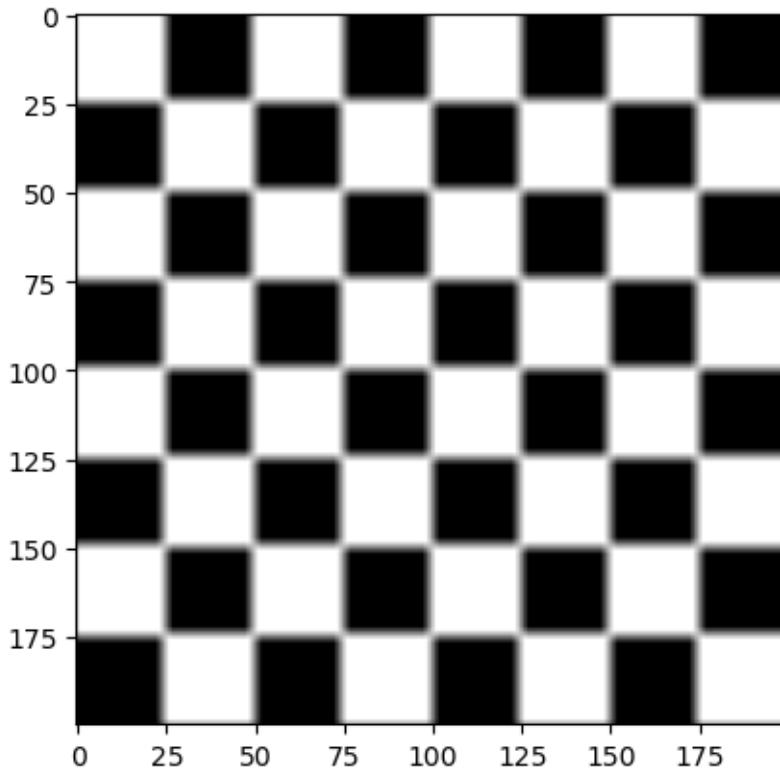
Wybieram opcję drugą - czyli poolingu wykorzystującego max. Bardziej podoba mi się obraz, oraz krawędzi wydają się być mocniejsze, co liczę, że poprawi wyniki.

```
[ ]: img = img2
```

```
[ ]: print_gray(prep_gauss(8))
```



```
[ ]: img = try_gauss(img, 4)
img.shape
```



```
[ ]: torch.Size([200, 200])
```

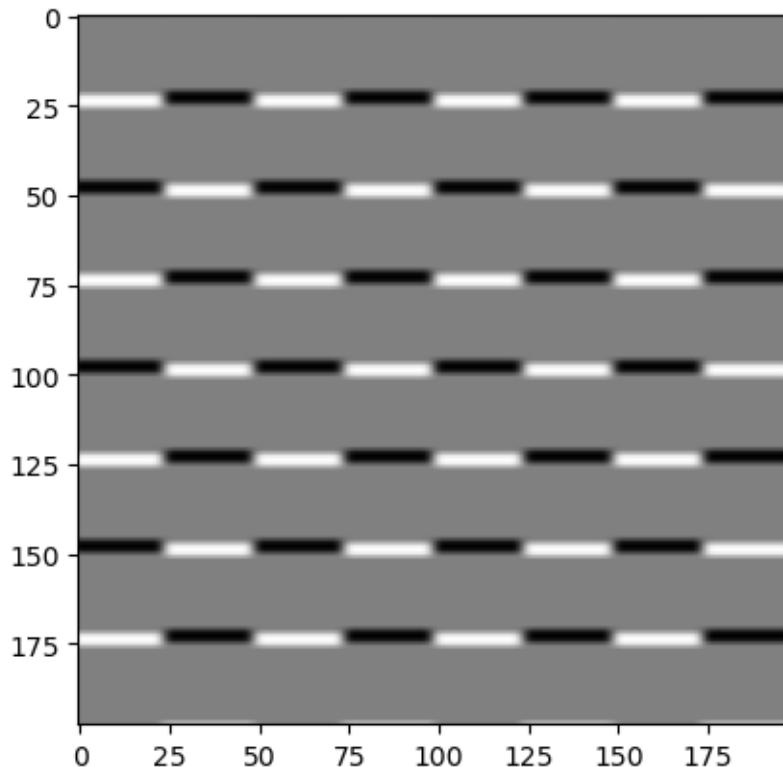
```
[ ]: grad1 = torch.tensor([[1,2,1],[0,0,0],[-1,-2,-1]])
grad2 = grad1.T
grad1 = torch.unsqueeze(grad1, dim=-1).T.float()
grad2 = torch.unsqueeze(grad2, dim=-1).T.float()
grad = torch.stack((grad1,grad2), dim=0)
print(grad.shape)
grad
```

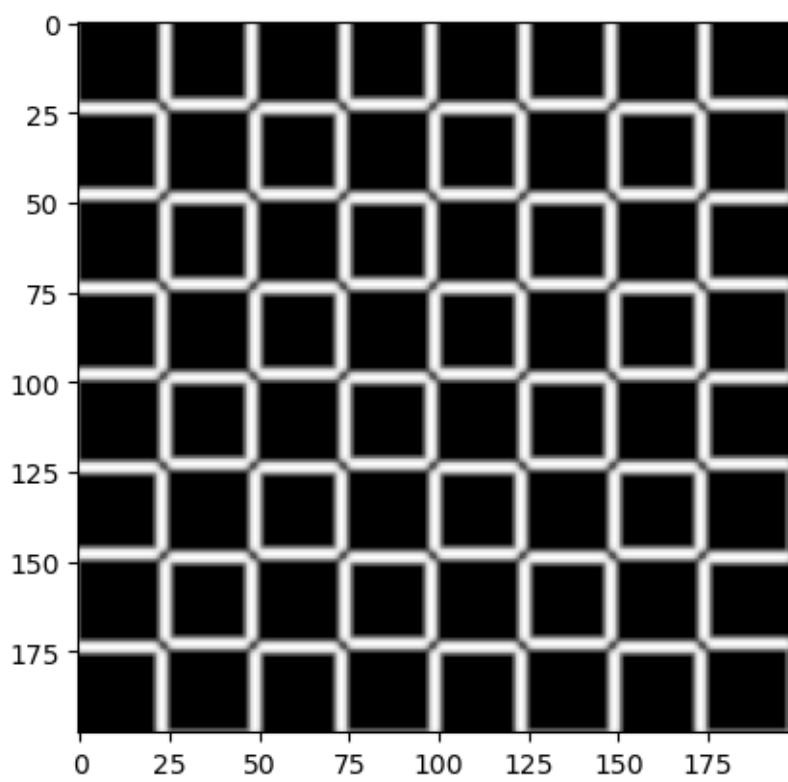
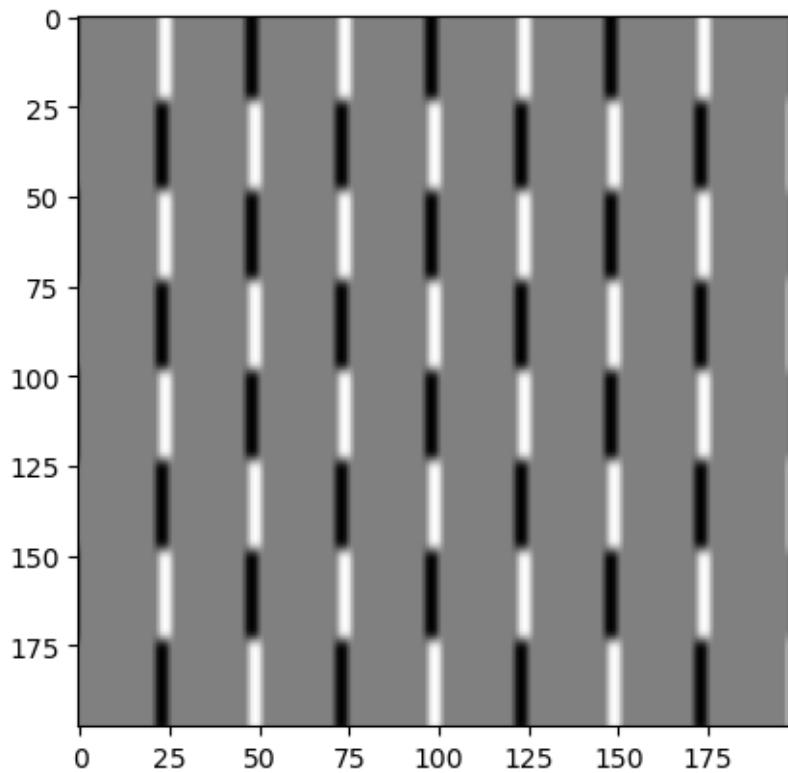
```
torch.Size([2, 1, 3, 3])
```

```
[ ]: tensor([[[[ 1.,  0., -1.],
              [ 2.,  0., -2.],
              [ 1.,  0., -1.]]],
```

```
[[[ 1.,  2.,  1.],
  [ 0.,  0.,  0.],
  [-1., -2., -1.]]]])
```

```
[ ]: sobol_img = from_conv(F.conv2d(to_conv(img), grad, stride=1, padding='valid'))
print_gray(sobol_img[:, :, 0])
print_gray(sobol_img[:, :, 1])
sobel_img = torch.sqrt(torch.square(sobol_img[:, :, 0]) + torch.square(sobol_img[:, :, 1]))
print_gray(sobol_img)
```



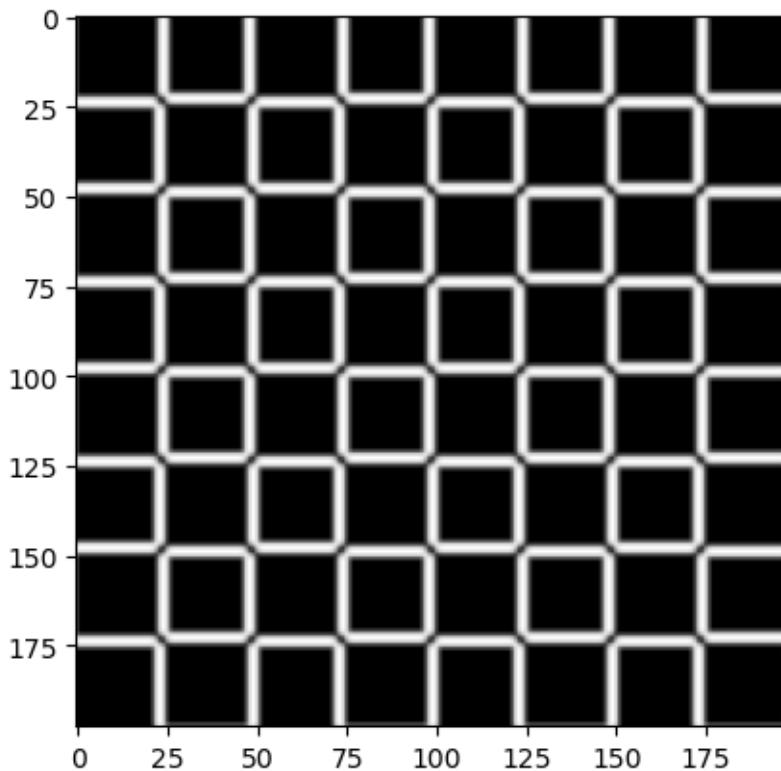


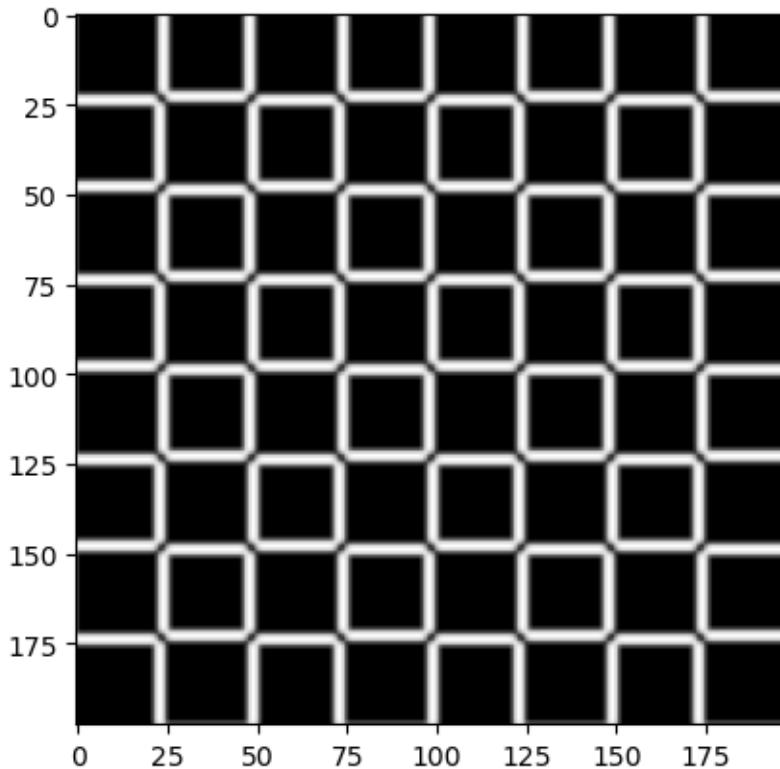
Użyłem filtra o 2 kanałach wyjściowych, na każdą składową gradientu, więc otrzymałem 1 tensor z oboma gradientami.

Aby uzyskać obrazek o tylko jednym kanale, podniósłem tensor do kwadratu i dodałem oba kanały.

```
[ ]: def try_relu(img, val, draw=True):
    new_img = F.relu(img-val)
    if draw:
        print_gray(new_img)
    return new_img

sobel_img=try_relu(sobel_img, 1/3)
sobel_img=sobel_img/sobel_img.max()
print_gray(sobel_img)
(sobel_img>1).any() or (sobel_img<0).any()
```





```
[ ]: tensor(False)
```

```
[ ]: img=sobol_img#[2:-2,2:-2]
```

Można użyć funkcji konwolucji, z wielowymiarowym filtrem, gdzie każdy kanał filtru odpowiada za inny rozmiar kwadratu.

Zdecydowałem jednak, że nie wiem jak zoptymalizowane są mnożenia macierzy, a potencjalnie w takiej sytuacji moglibyśmy wykonywać mnóstwo pustych mnożeń raza 0, zużywających jedynie moc obliczeniową. (np. podczas liczenia kwadratu 3x3, ale mając filtr rozmiaru 100x100)

Dlatego będę liczył iteracyjnie, z filtrami o tylko jednym kanale.

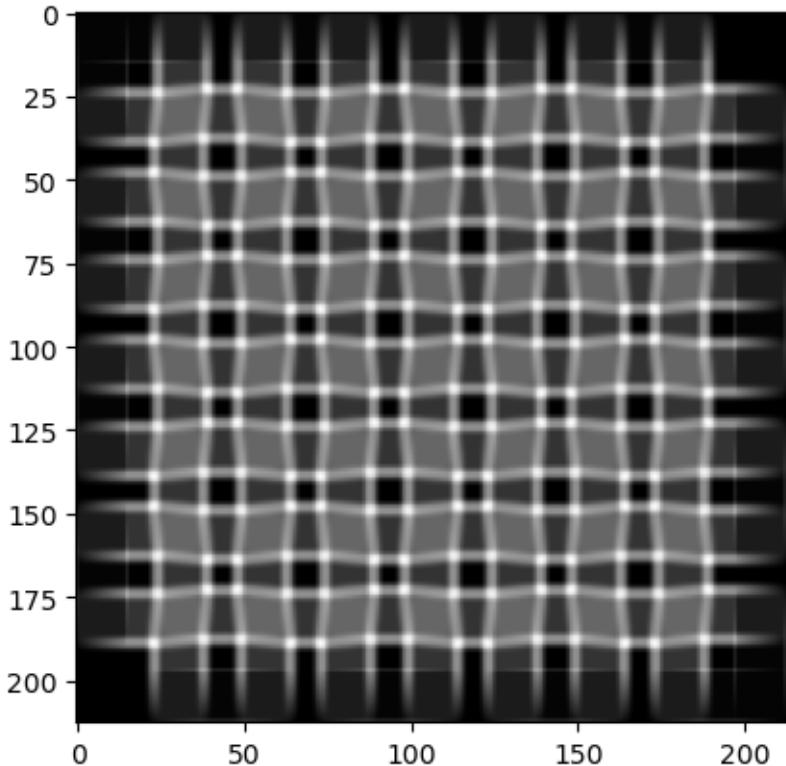
```
[ ]: sq(10)
```

```
[ ]: tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1.],
           [1., 0., 0., 0., 0., 0., 0., 0., 1.],
           [1., 0., 0., 0., 0., 0., 0., 0., 1.],
           [1., 0., 0., 0., 0., 0., 0., 0., 1.],
           [1., 0., 0., 0., 0., 0., 0., 0., 1.],
           [1., 0., 0., 0., 0., 0., 0., 0., 1.],
           [1., 0., 0., 0., 0., 0., 0., 0., 1.],
           [1., 0., 0., 0., 0., 0., 0., 0., 1.],
           [1., 0., 0., 0., 0., 0., 0., 0., 1.],
```

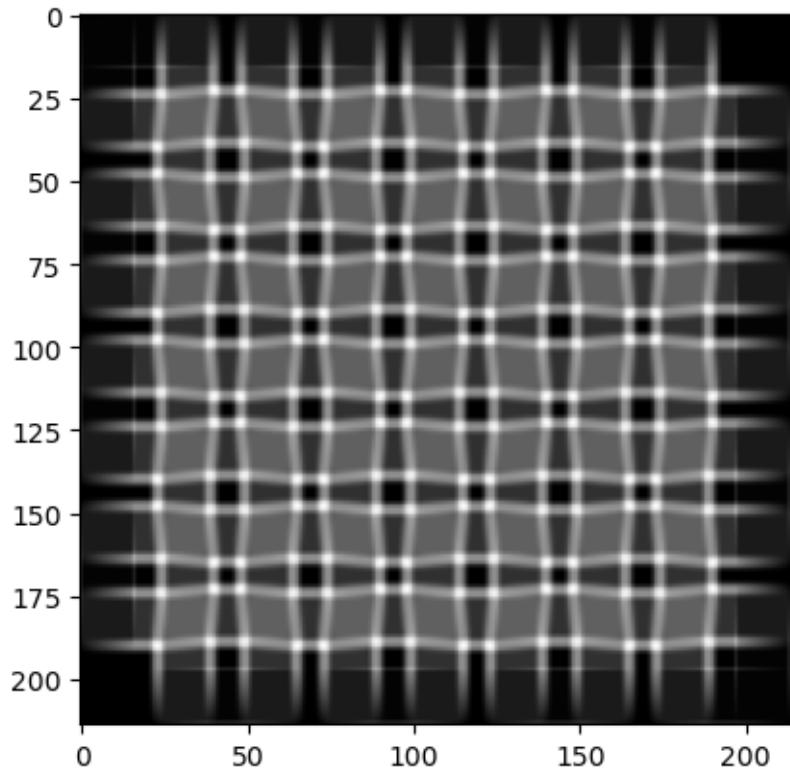
```
[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

```
[ ]: imgs = []
for i in range(16,32):
    print(i)
    imgs = imgs + [try_hough(img,i)]
```

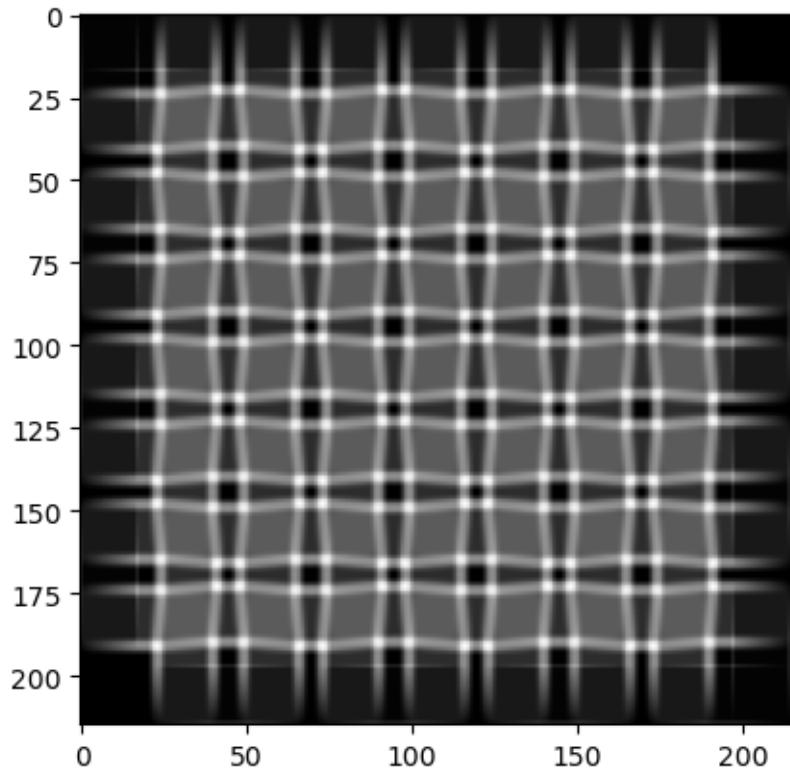
16



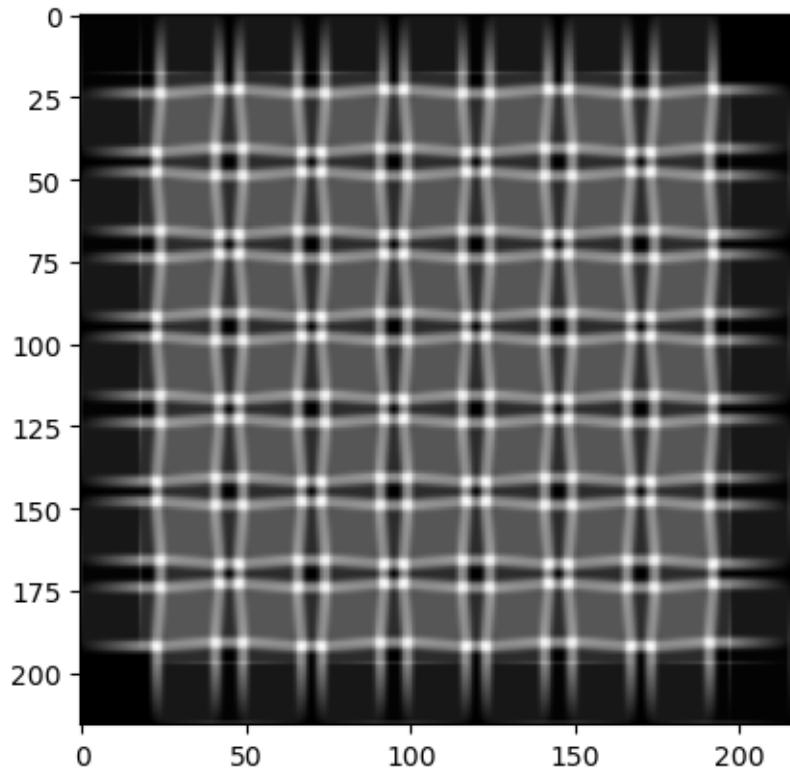
17



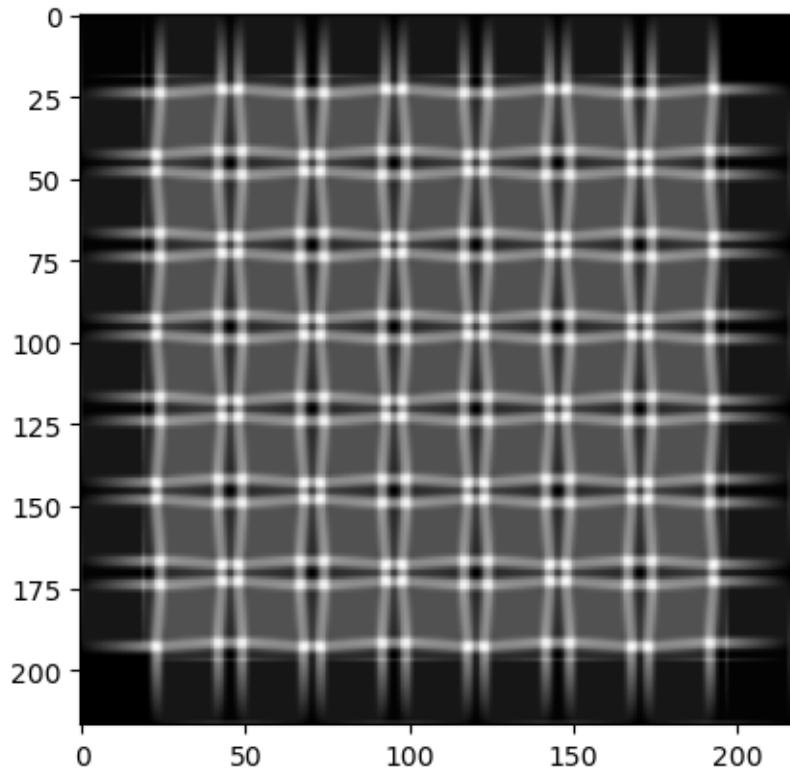
18



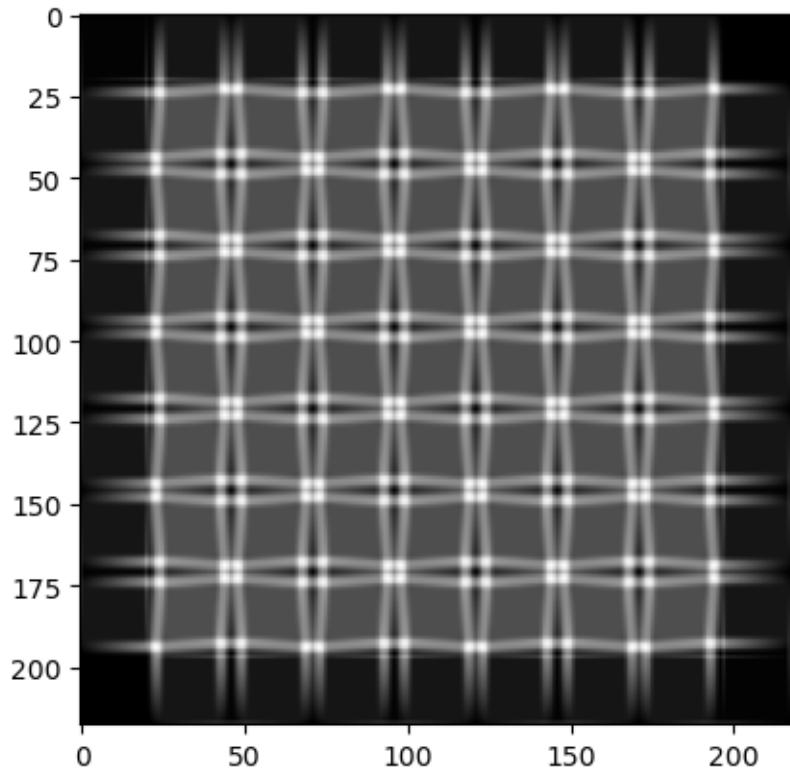
19



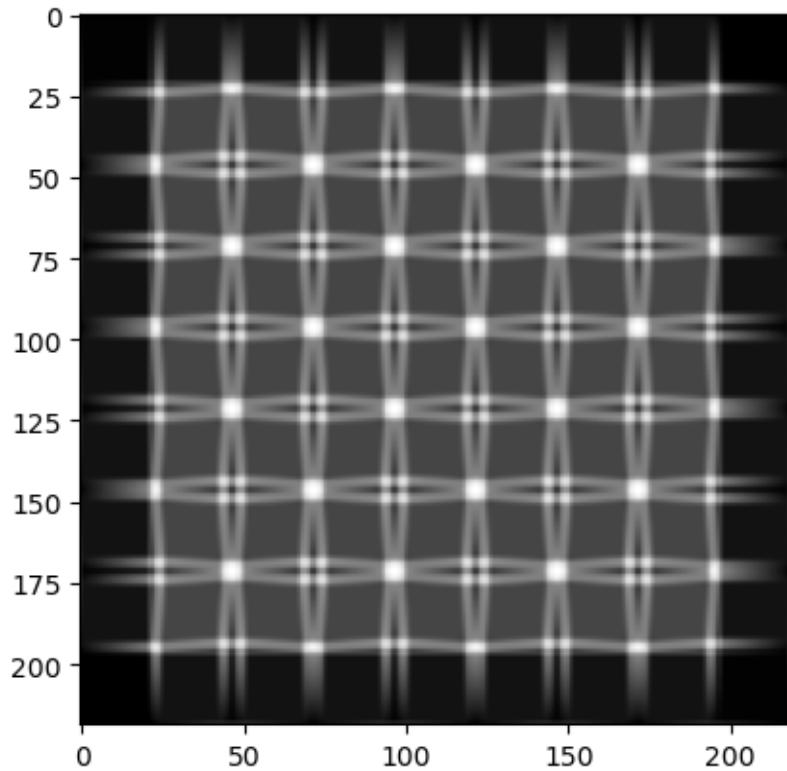
20



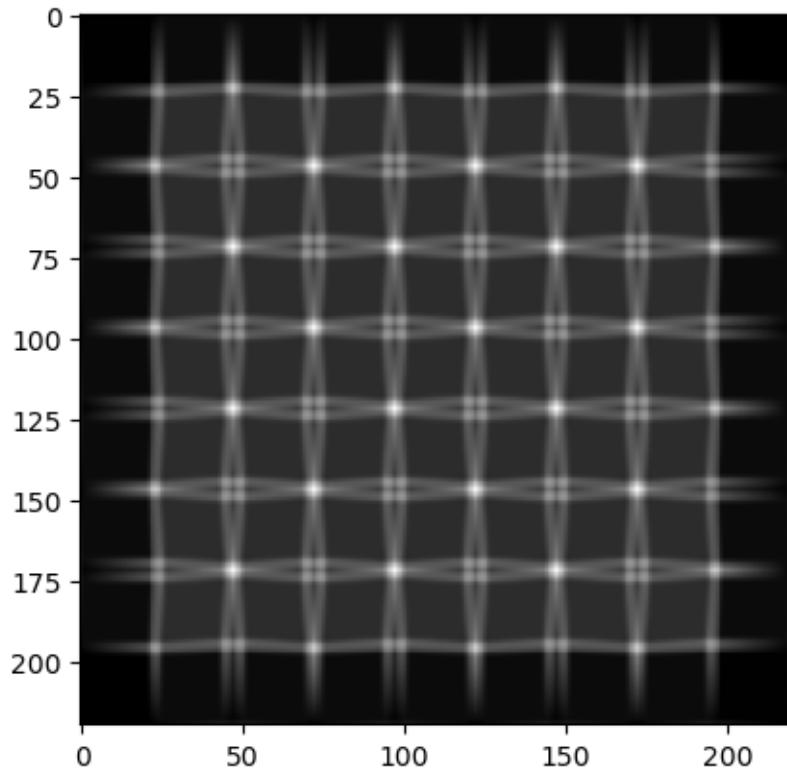
21



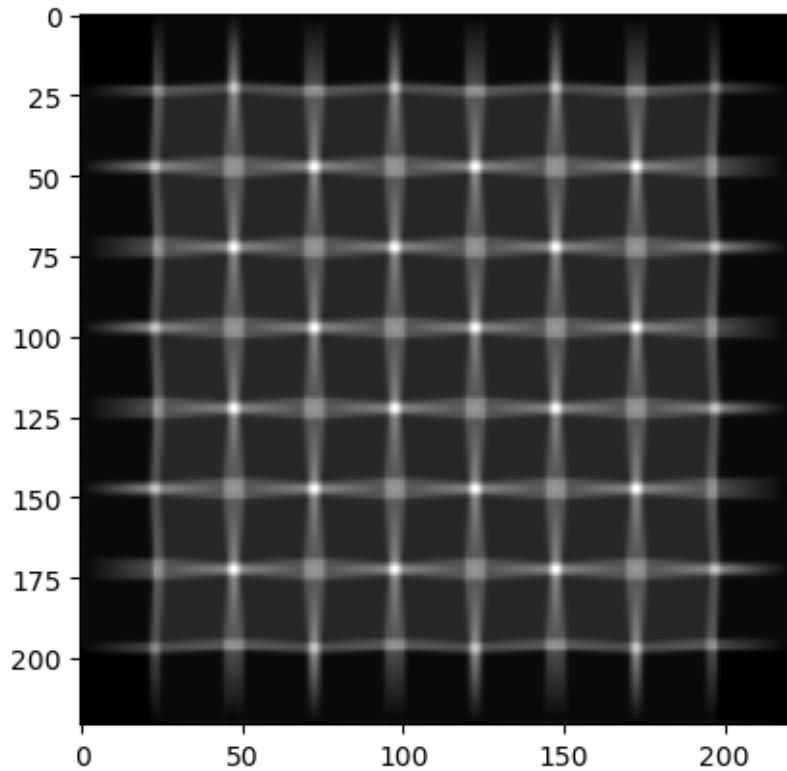
22



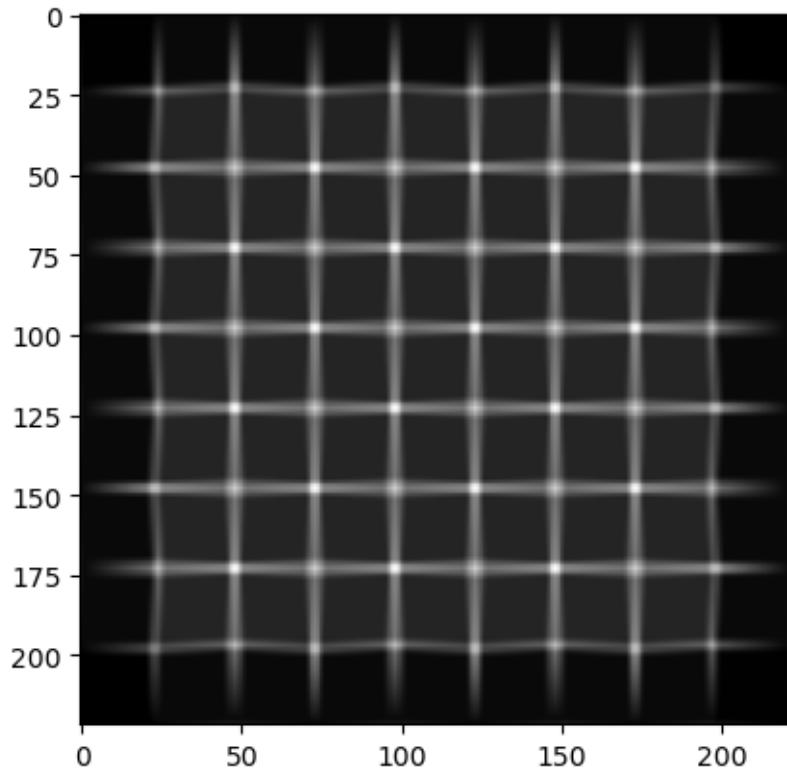
23



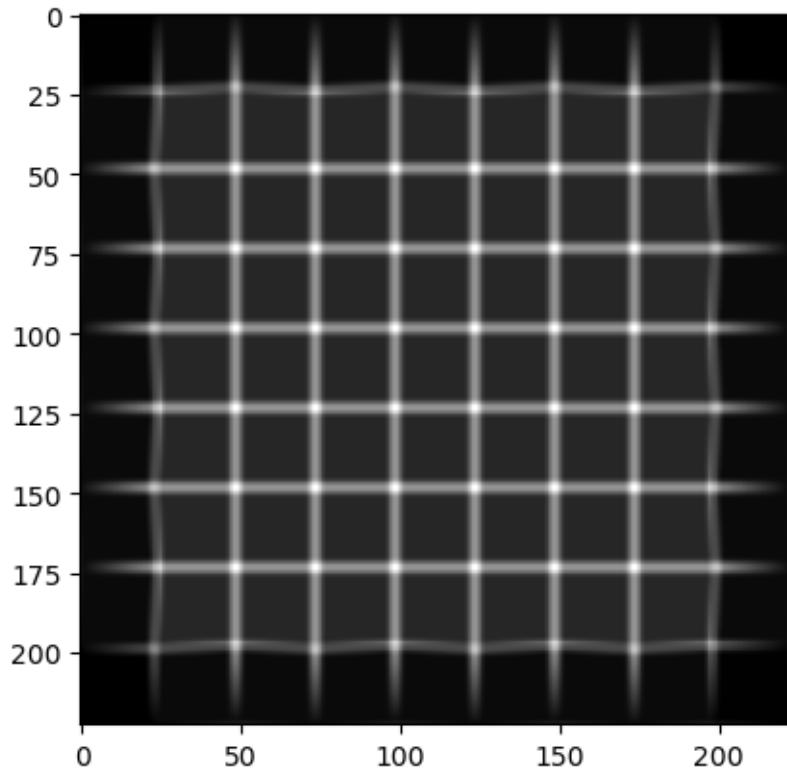
24



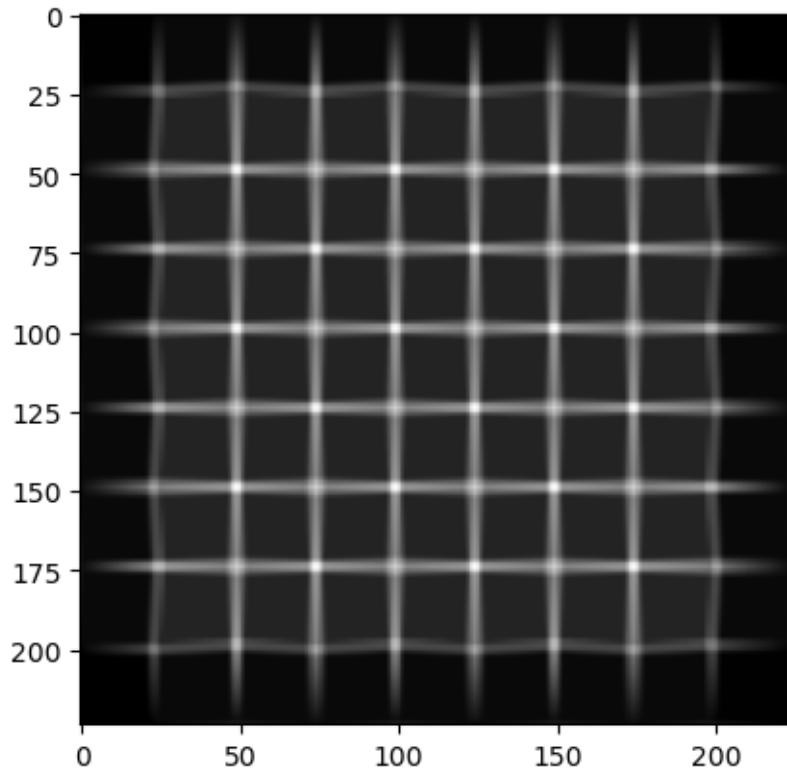
25



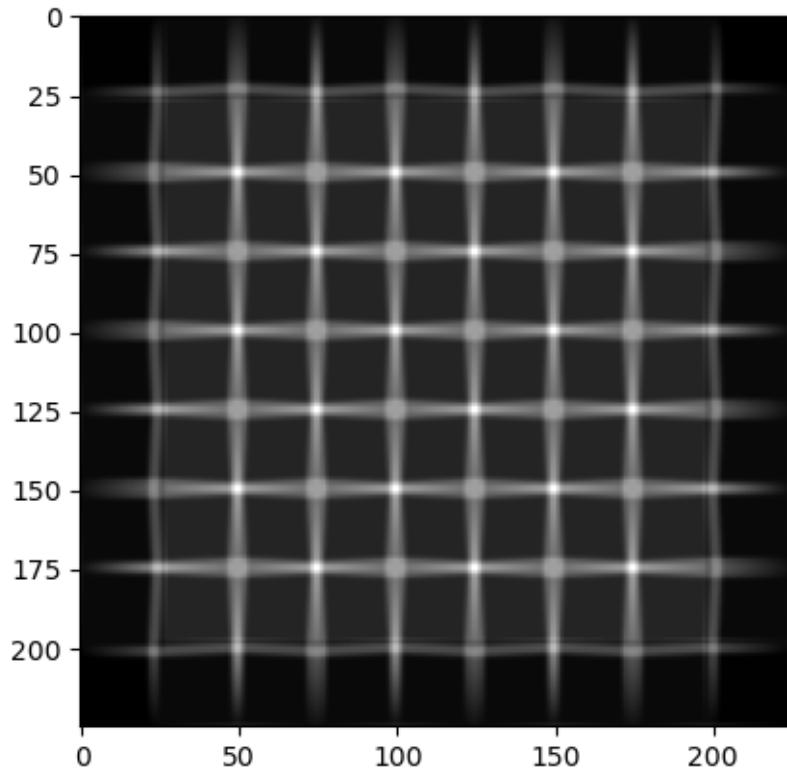
26



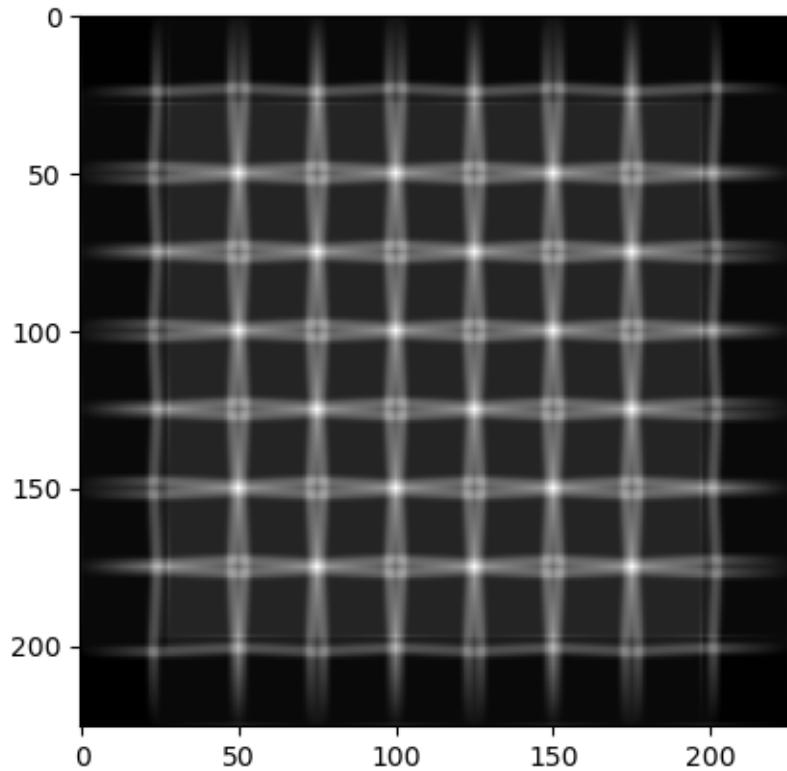
27



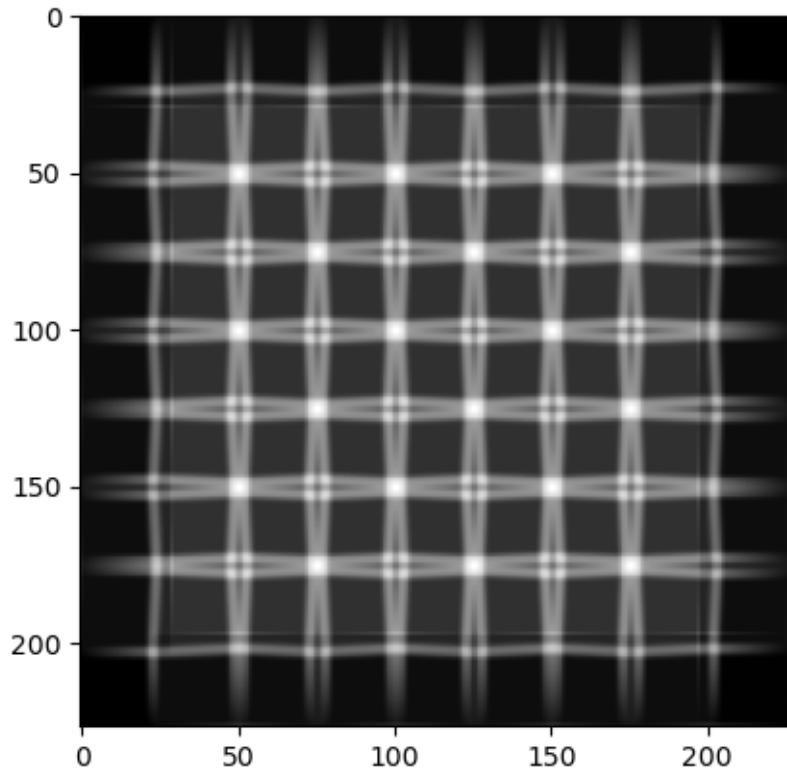
28



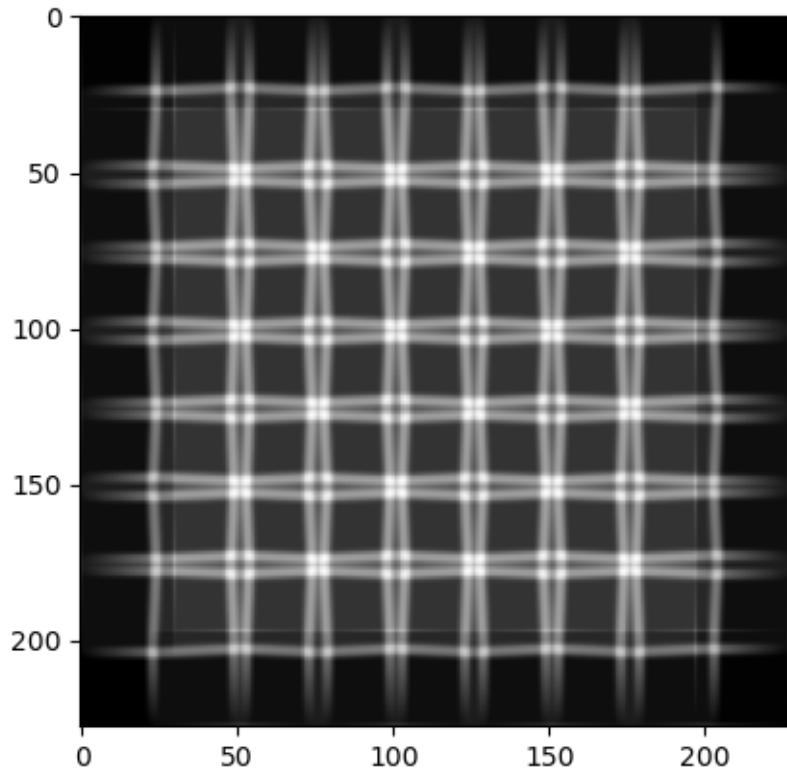
29



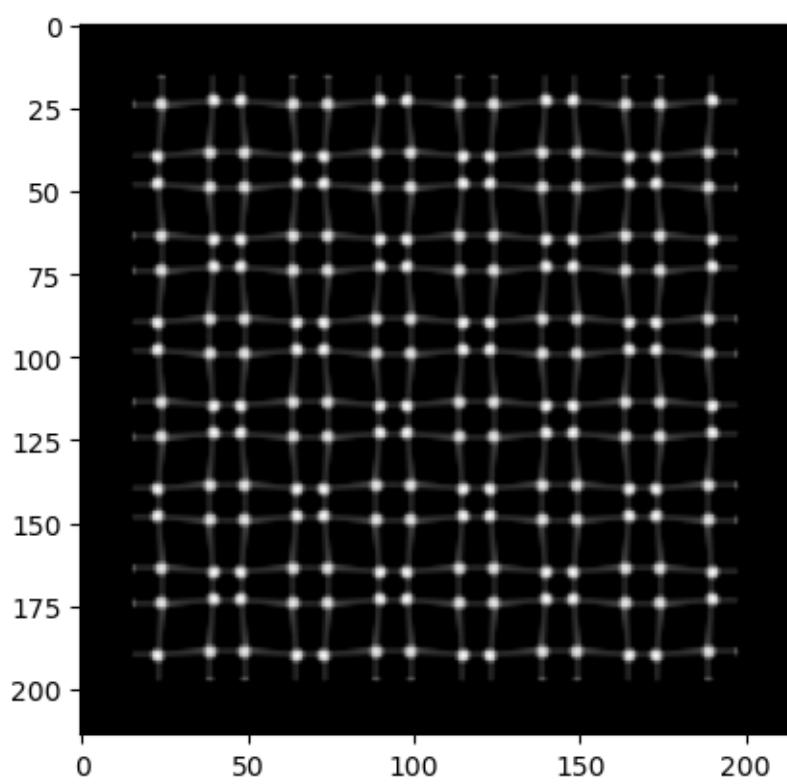
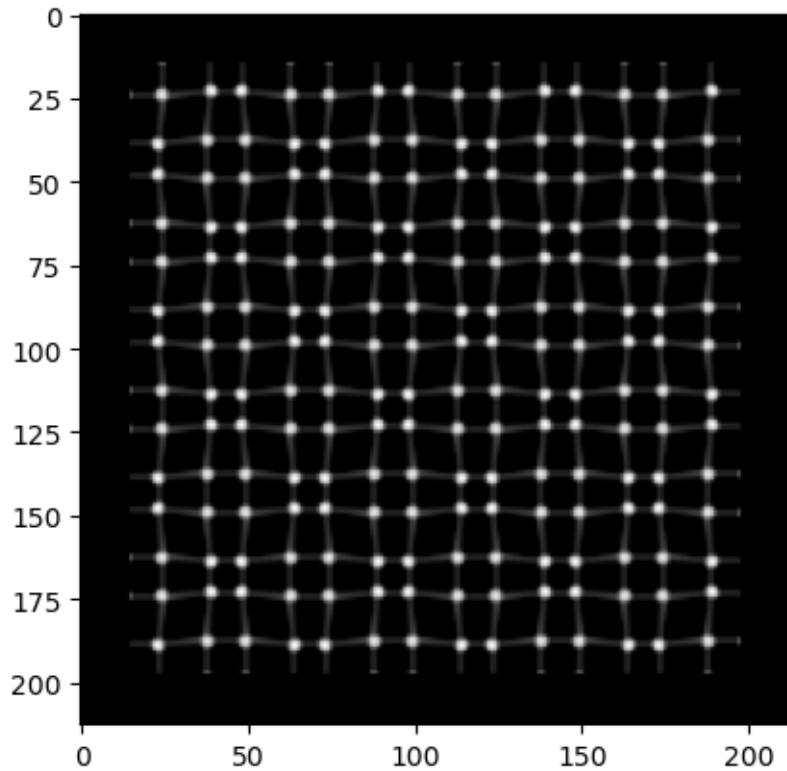
30

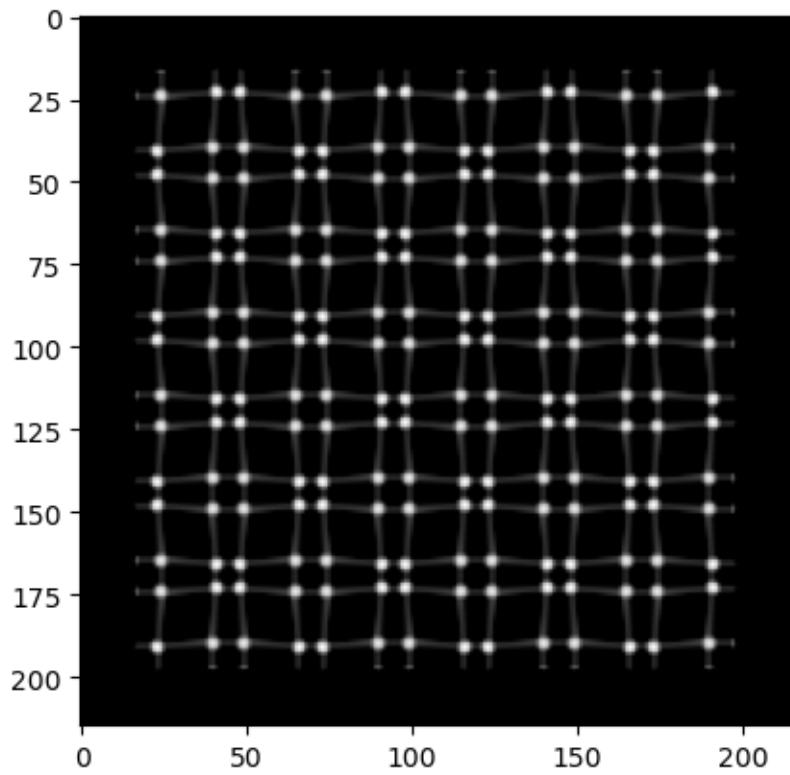


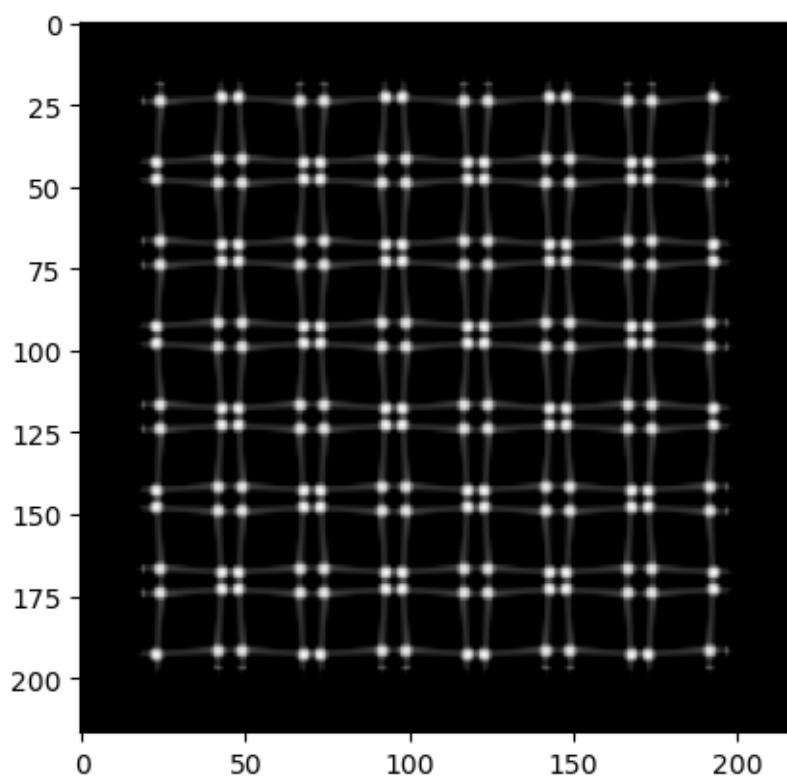
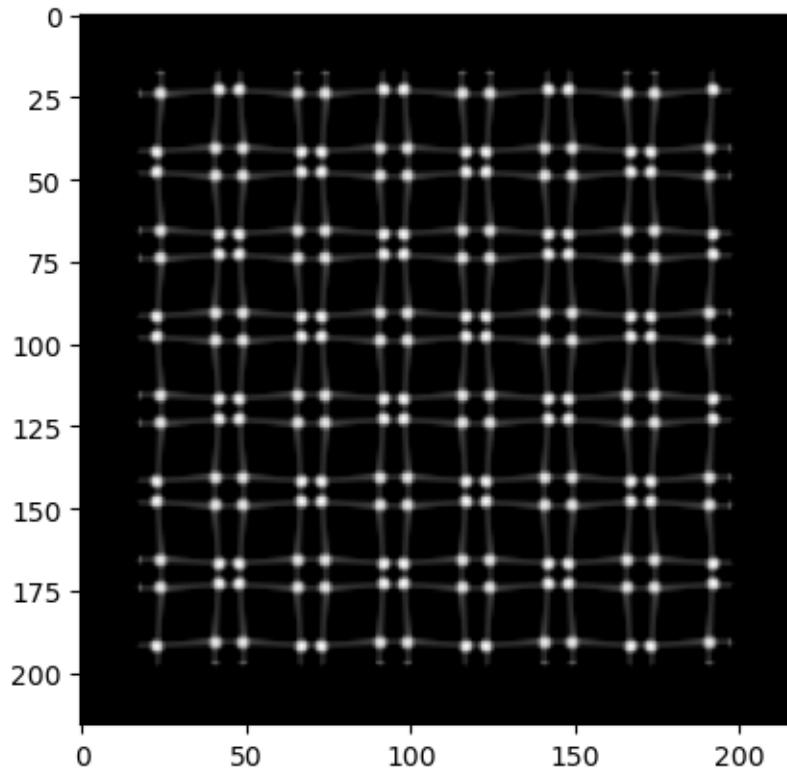
31

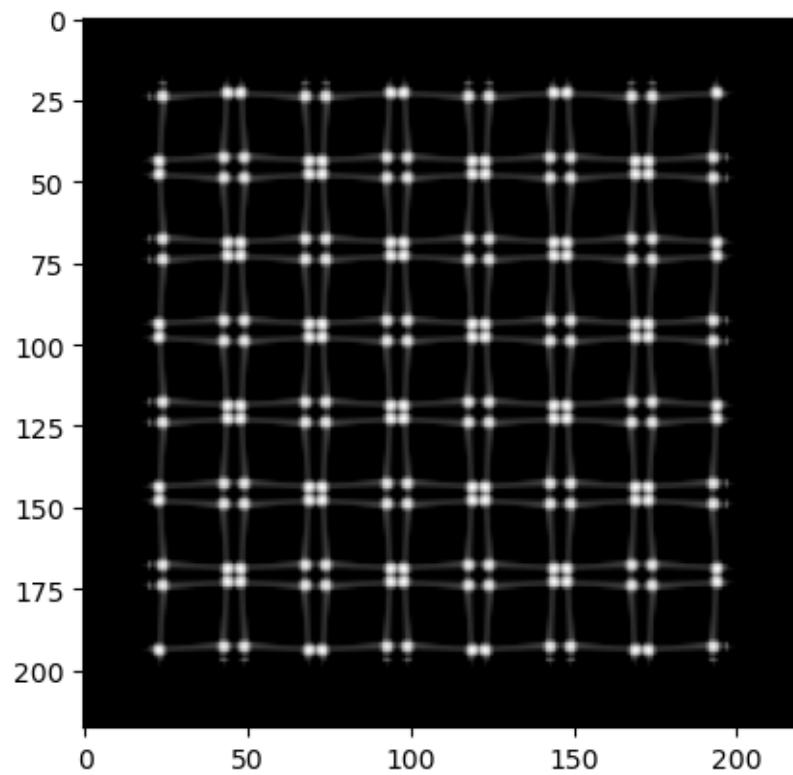


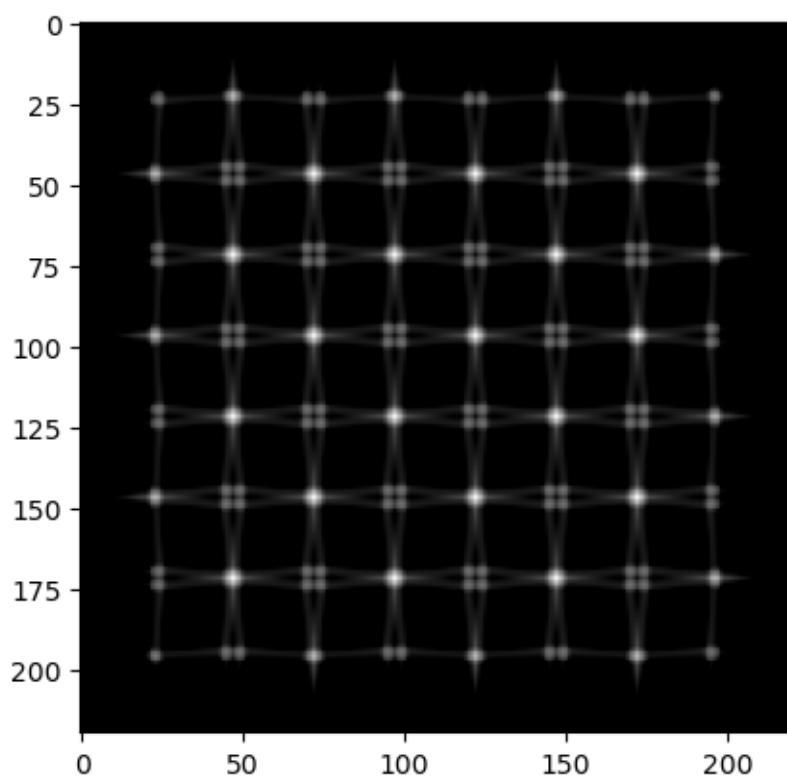
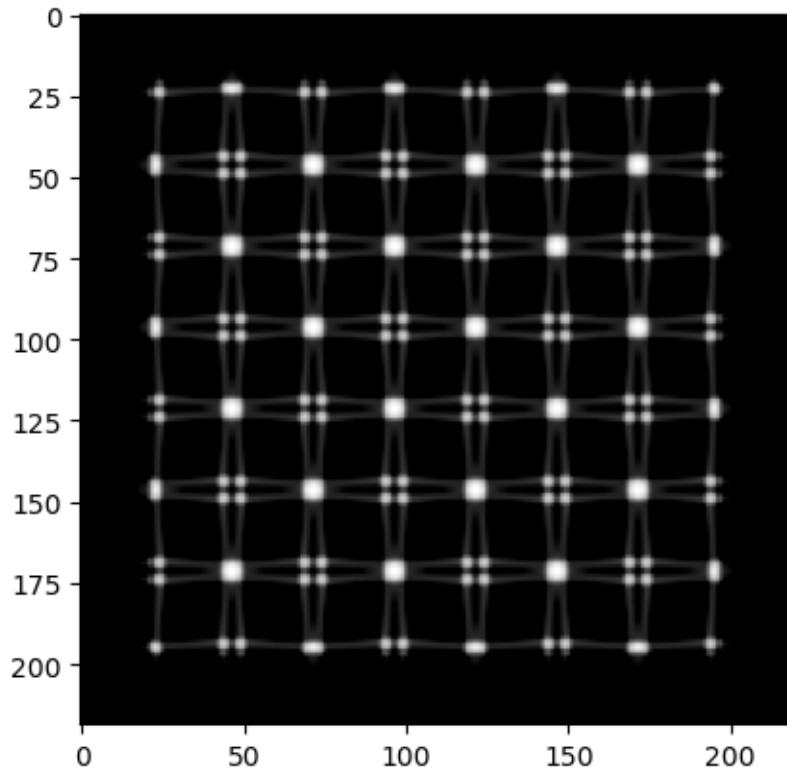
```
[ ]: for ix, img in enumerate(imgs):
    imgs[ix]=try_relu(img, 2*ix/3+16)
```

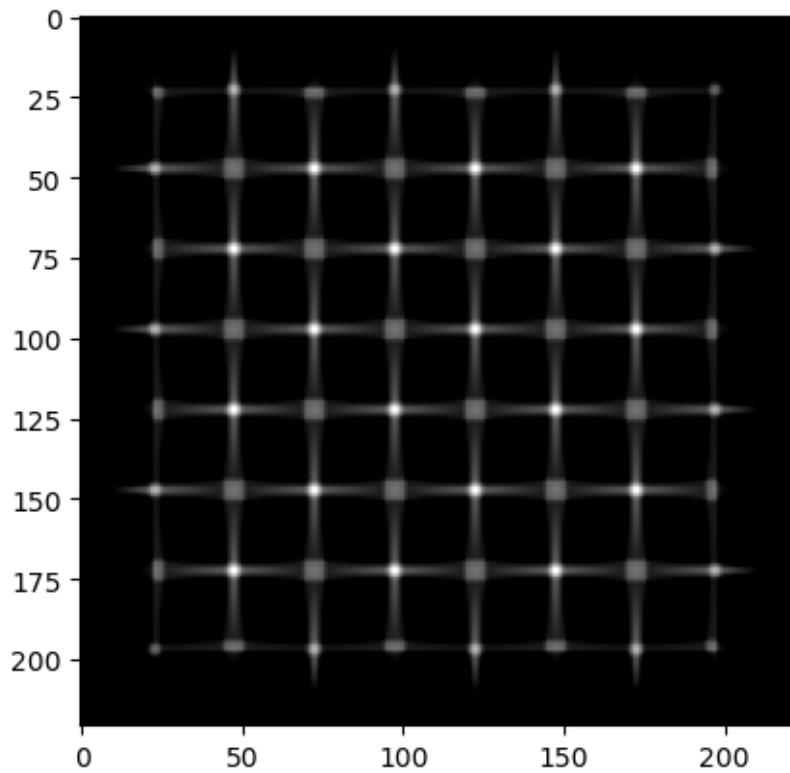


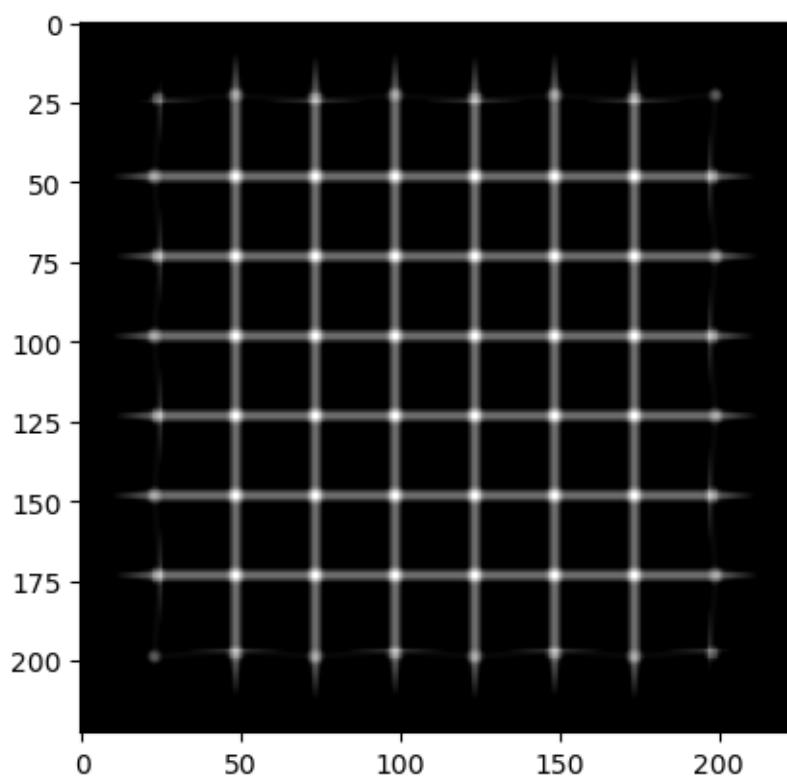
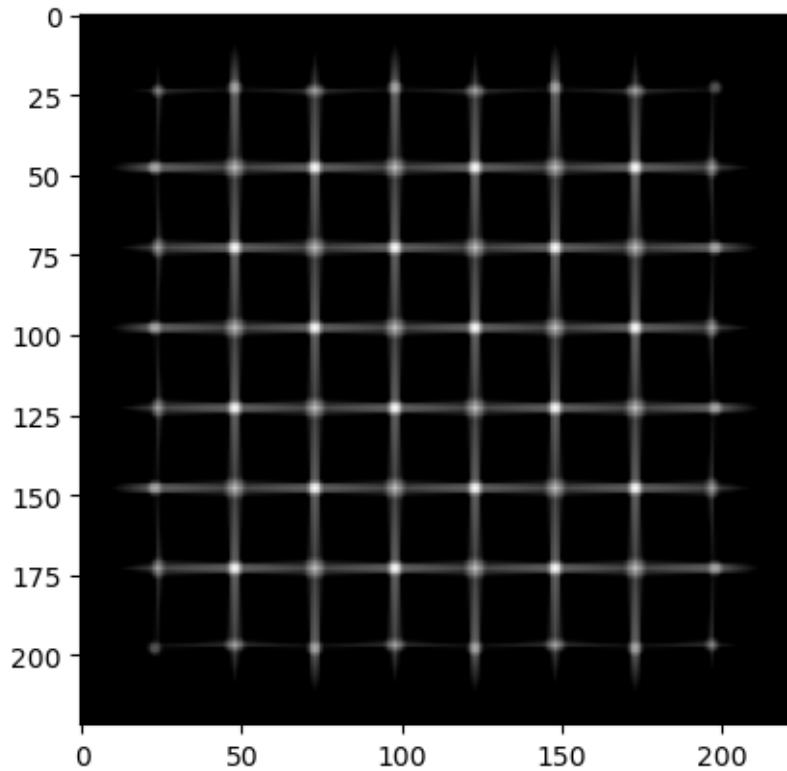


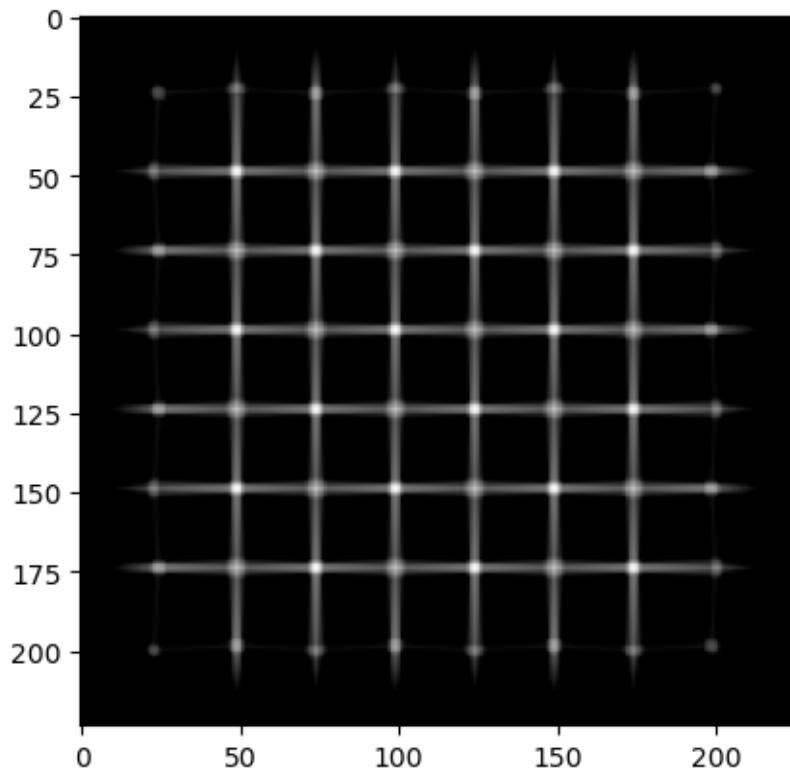


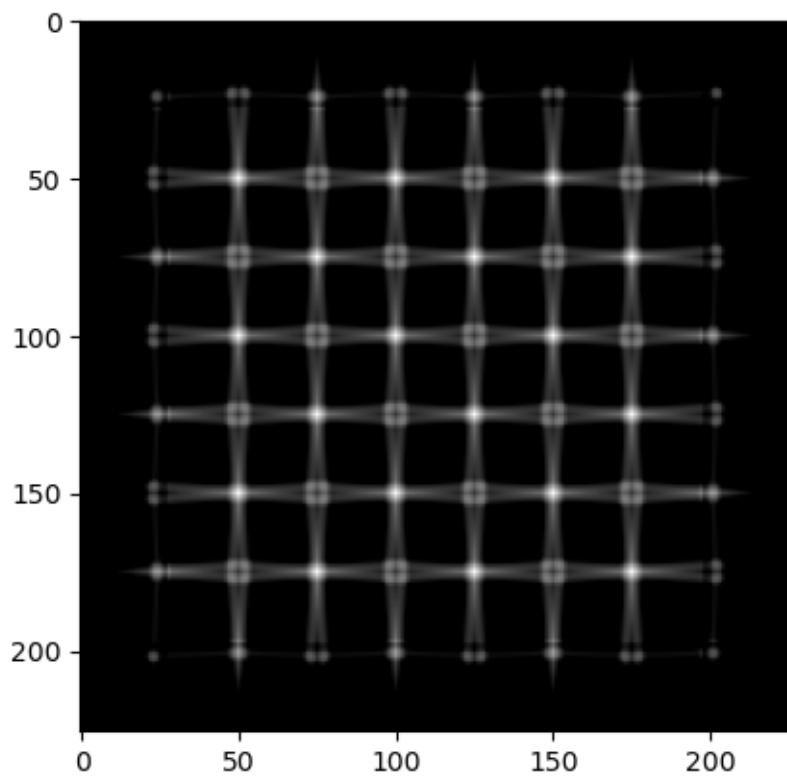
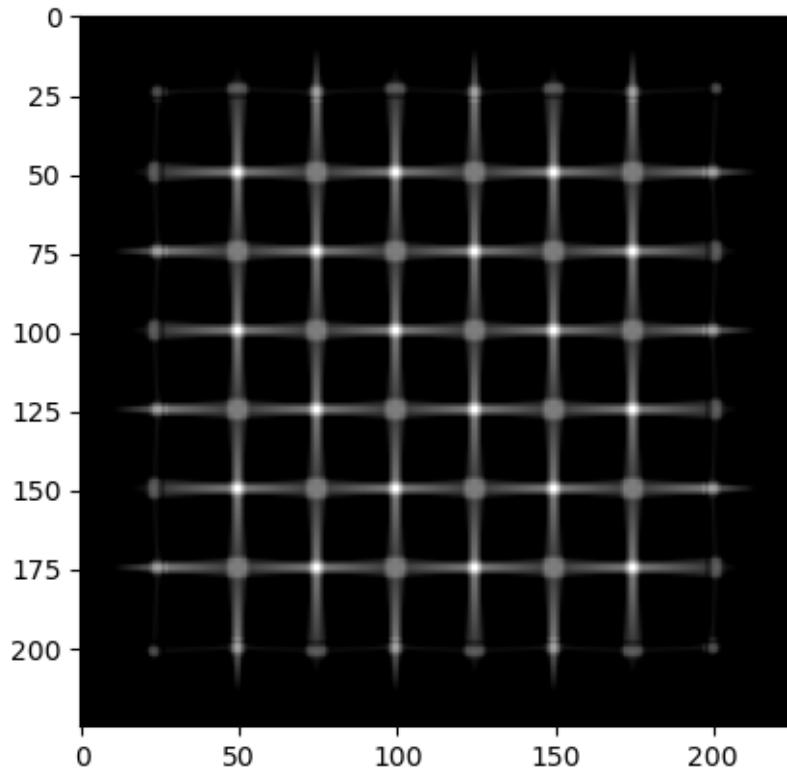


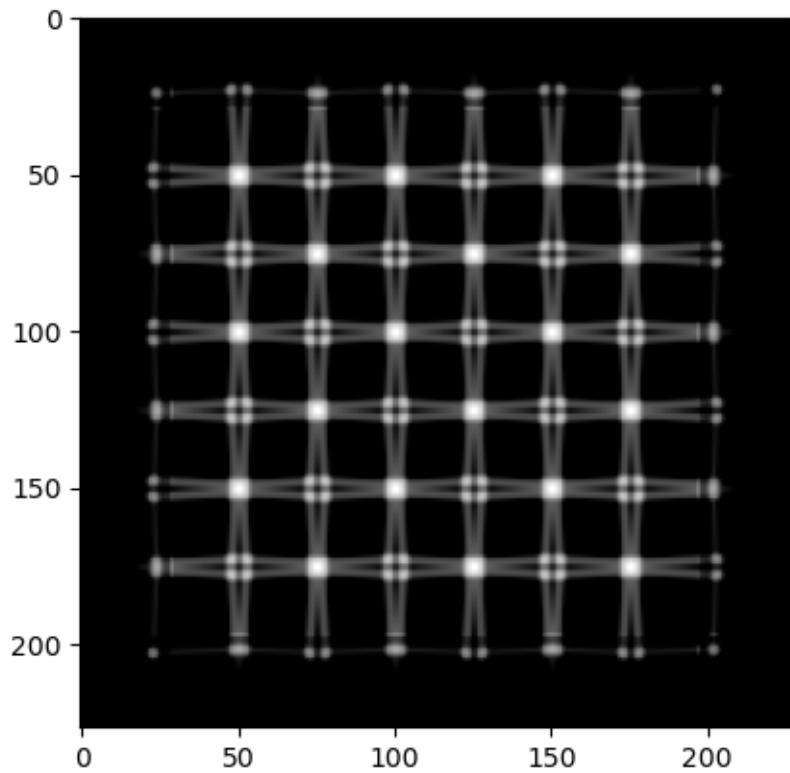


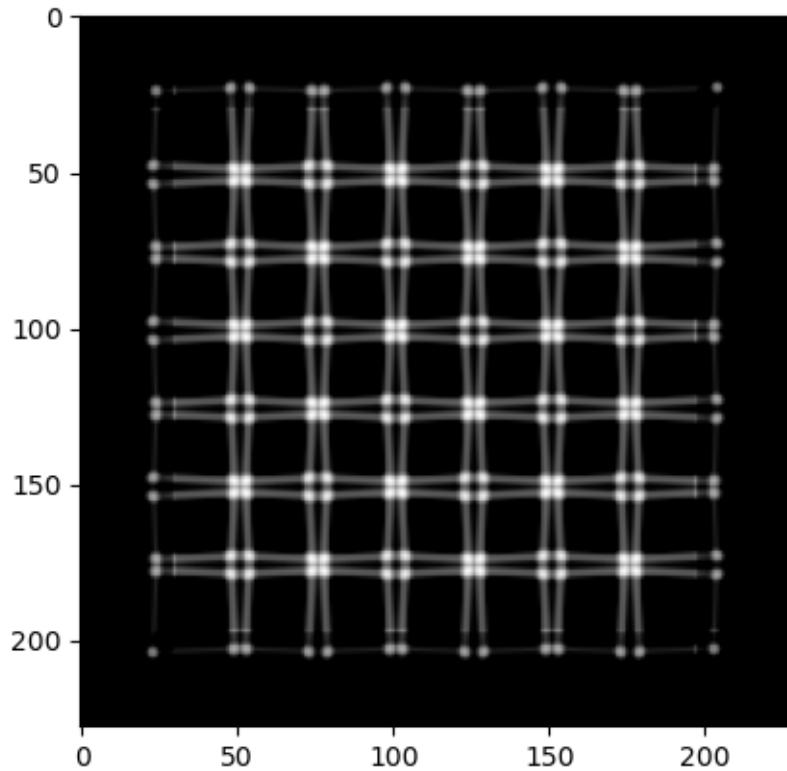






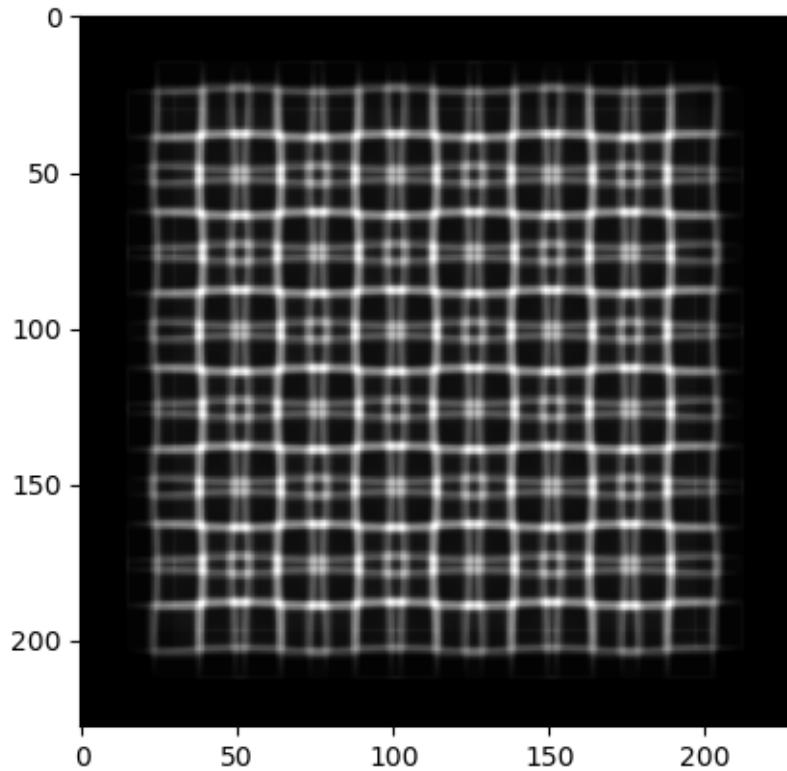




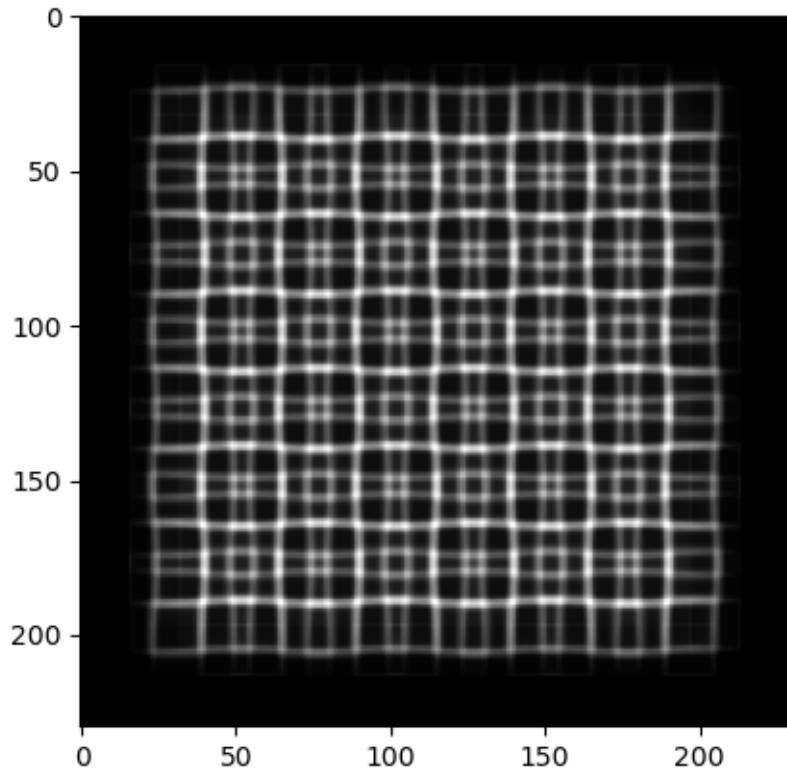


```
[ ]: res = torch.zeros(img.shape)
for ix, imgx in enumerate(imgs):
    print(ix+16)
    tmp=try_hough(imgx, ix+16)[0:img.shape[0],0:img.shape[1]]
    res+=(tmp/tmp.max())#*(16-ix+2))
```

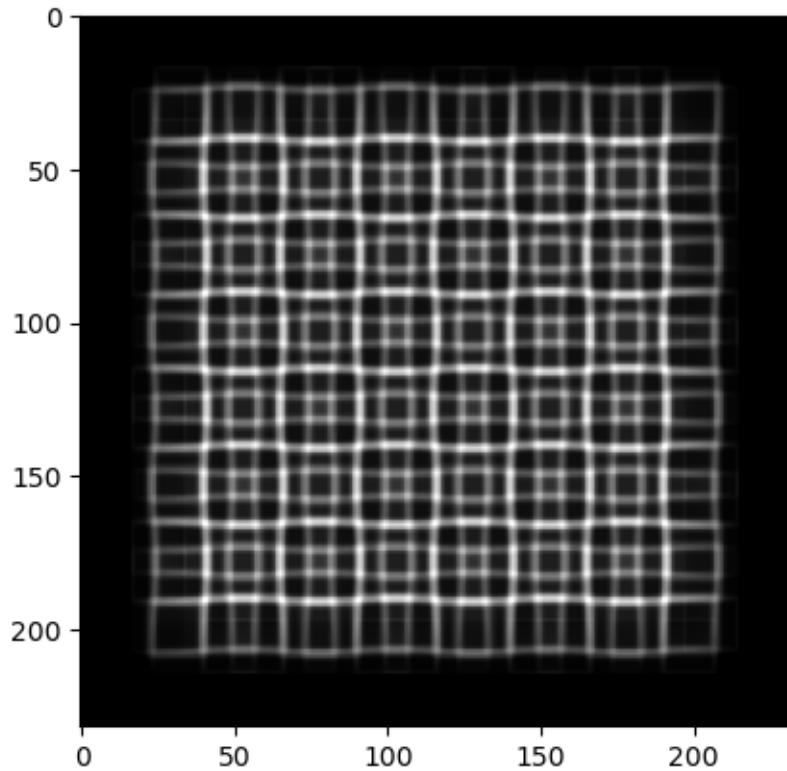
16



17

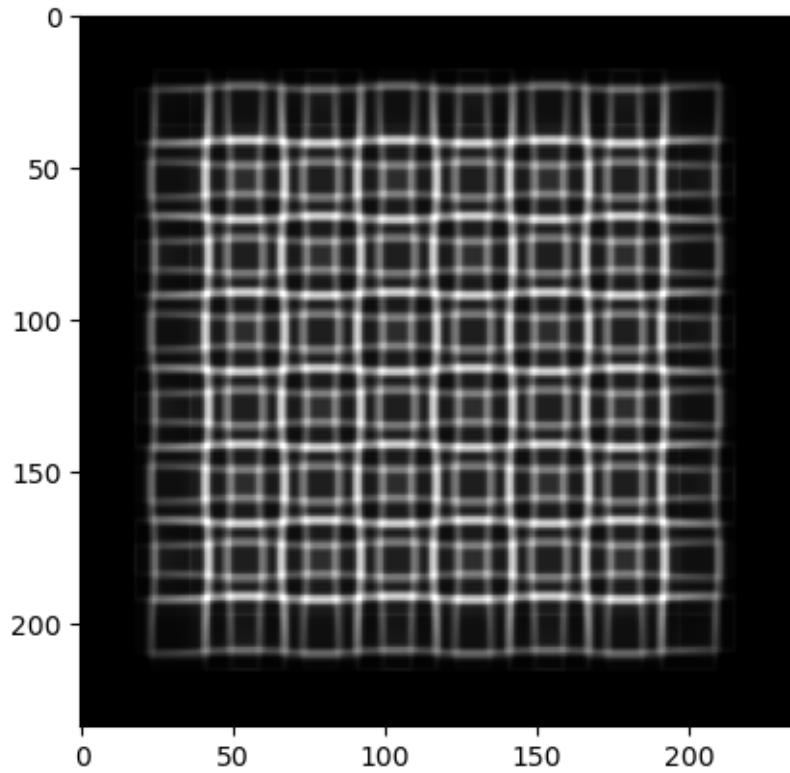


18

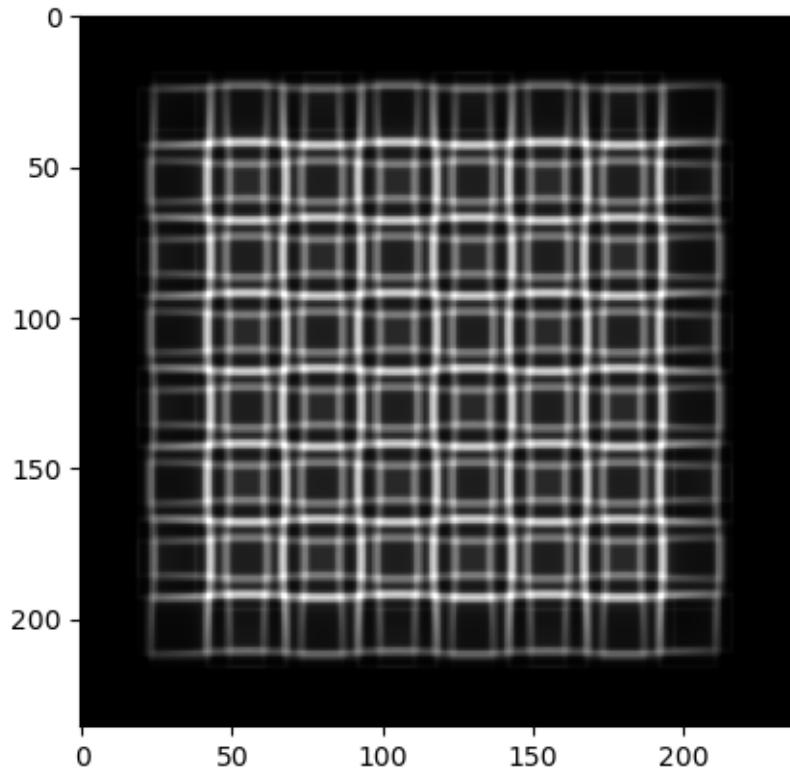


19

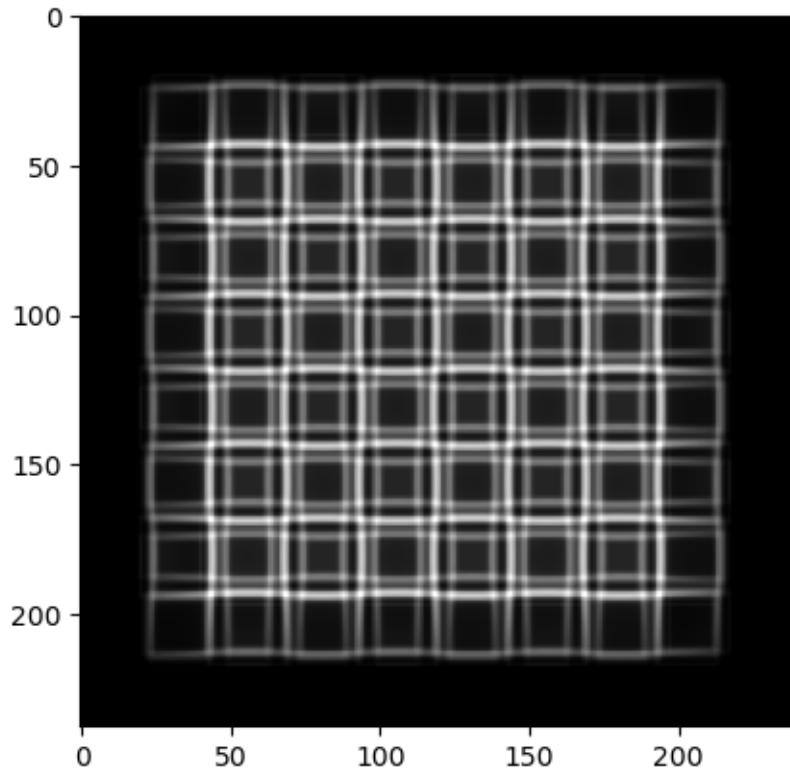
94



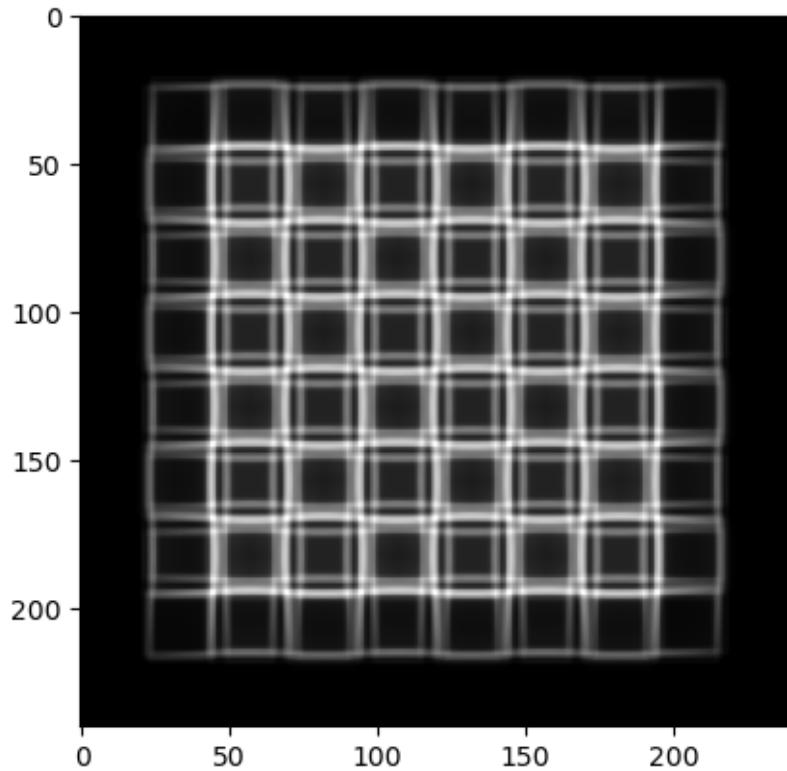
20



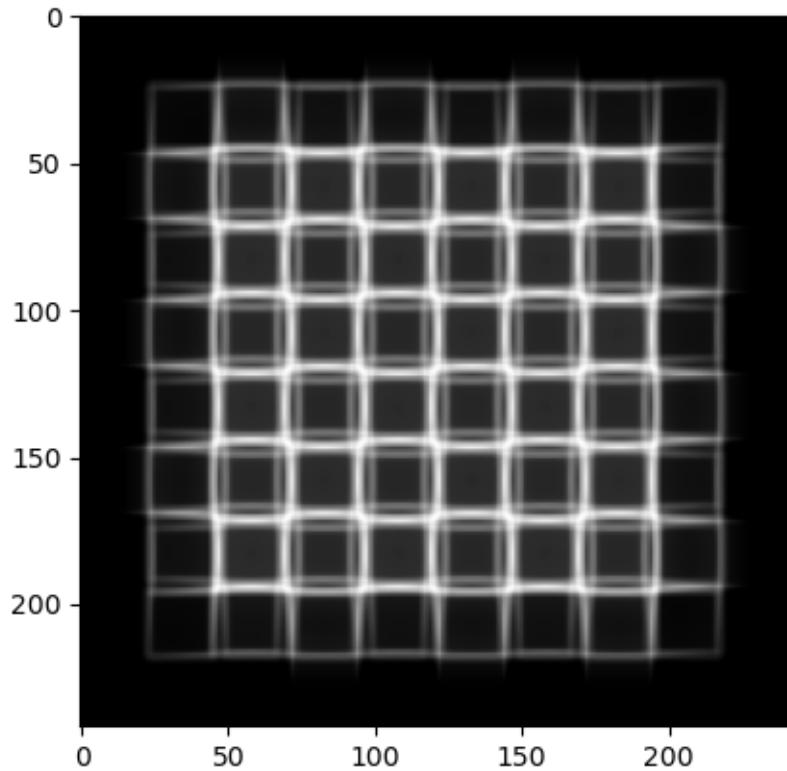
21



22

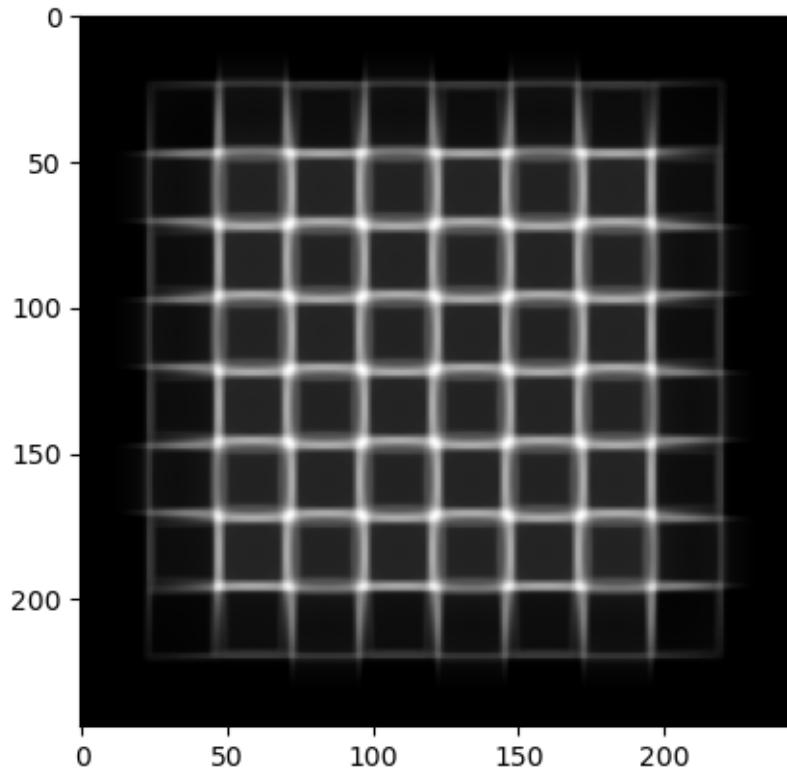


23



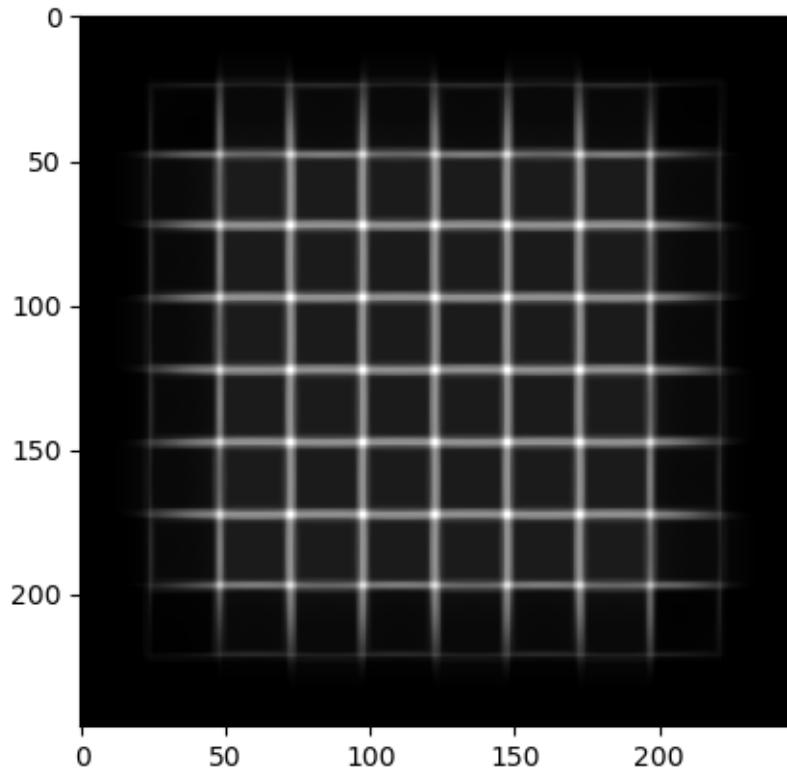
24

99

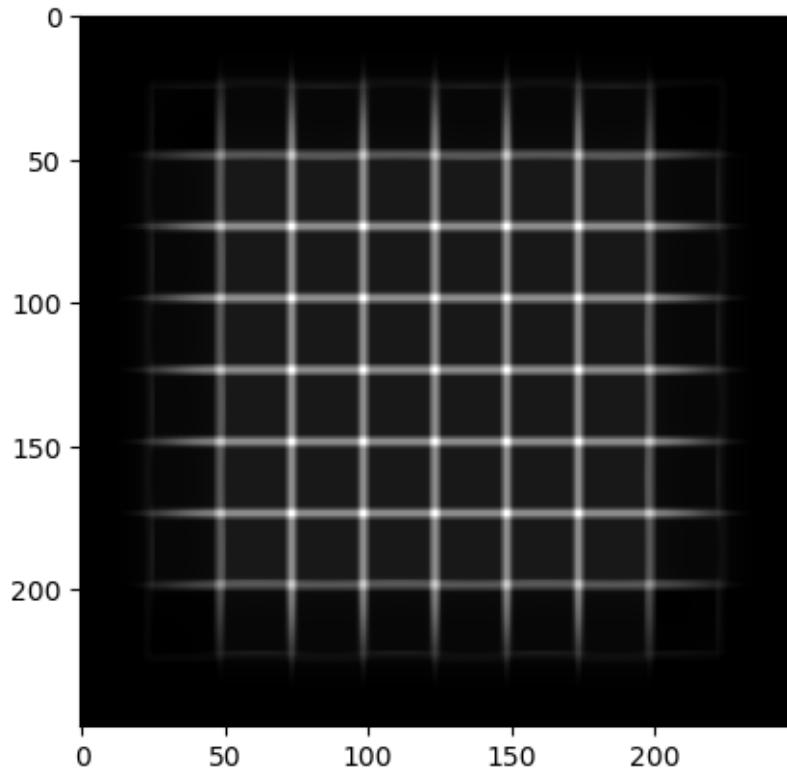


25

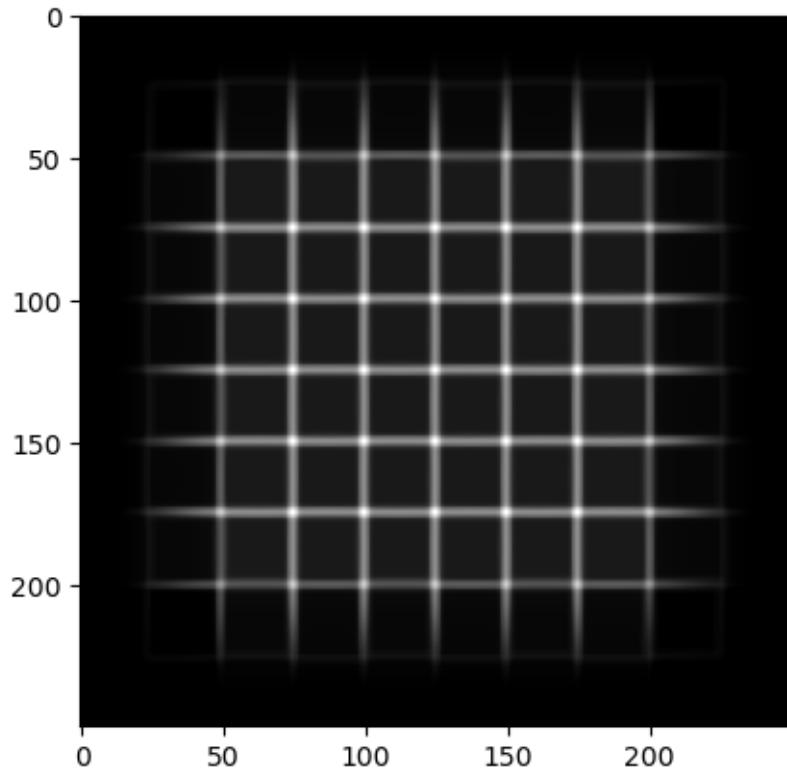
100



26

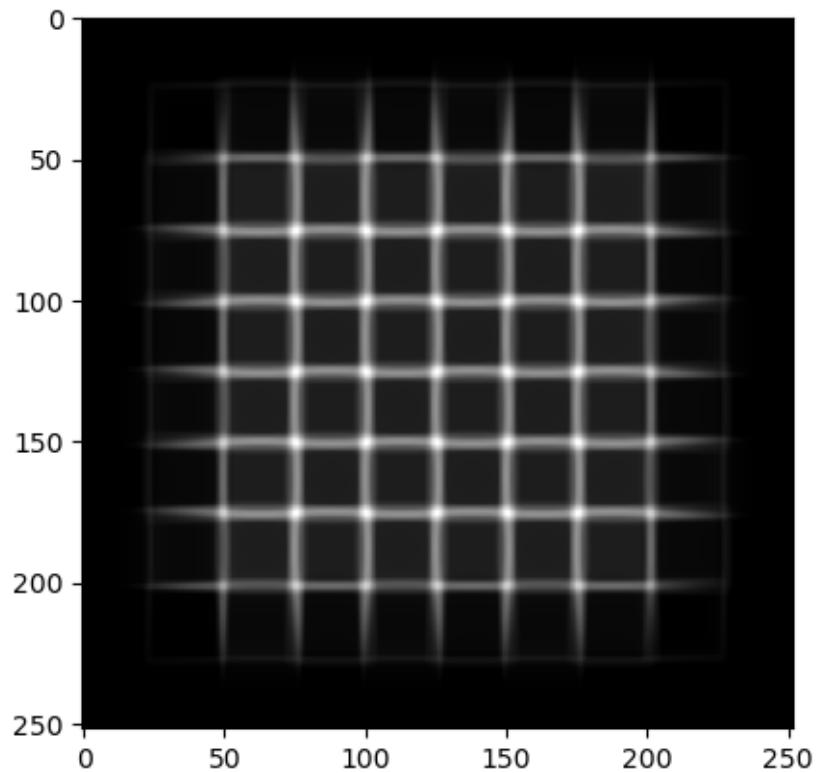


27



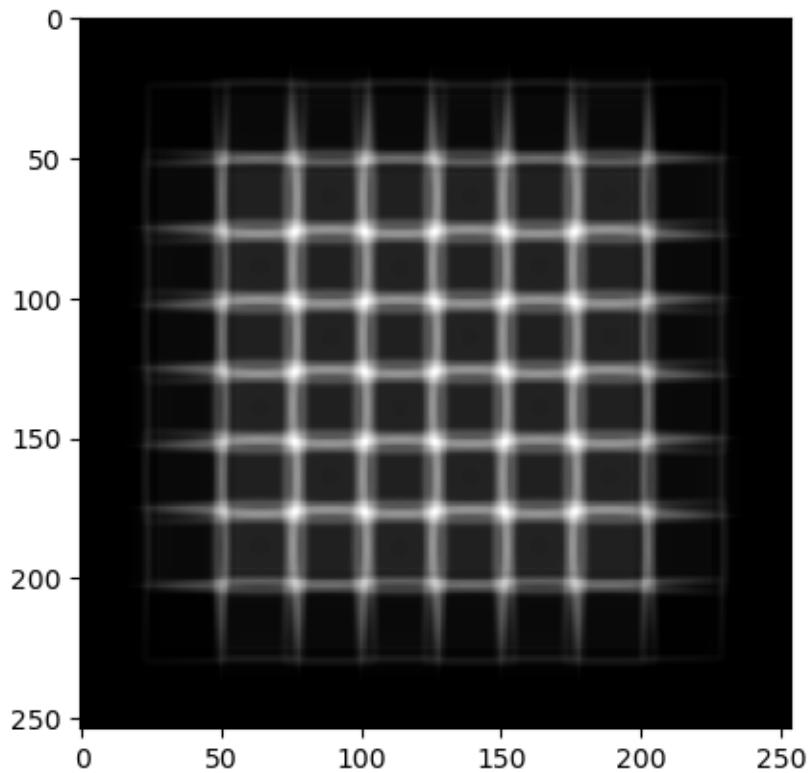
28

103



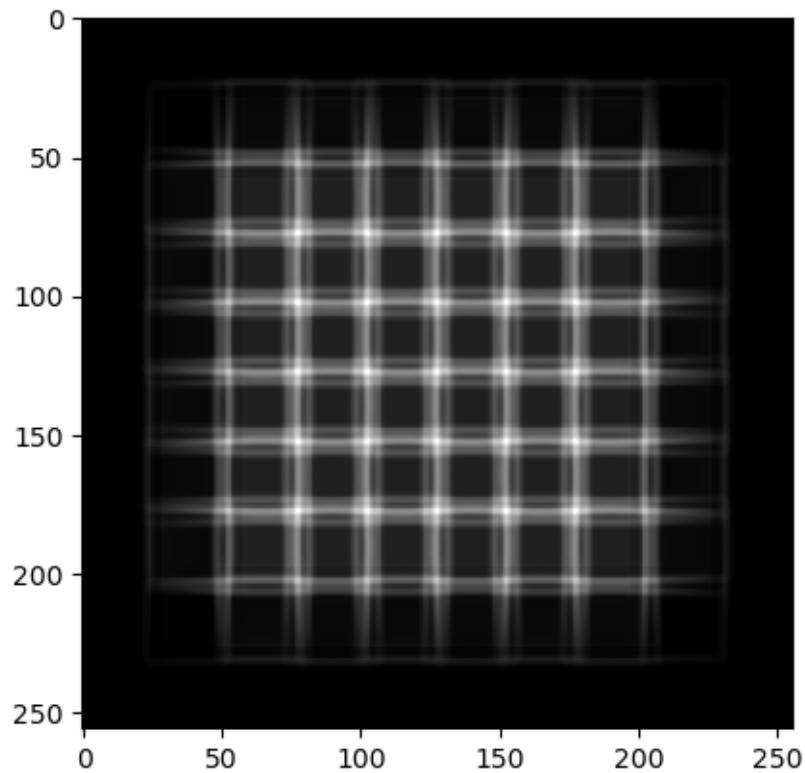
29

104



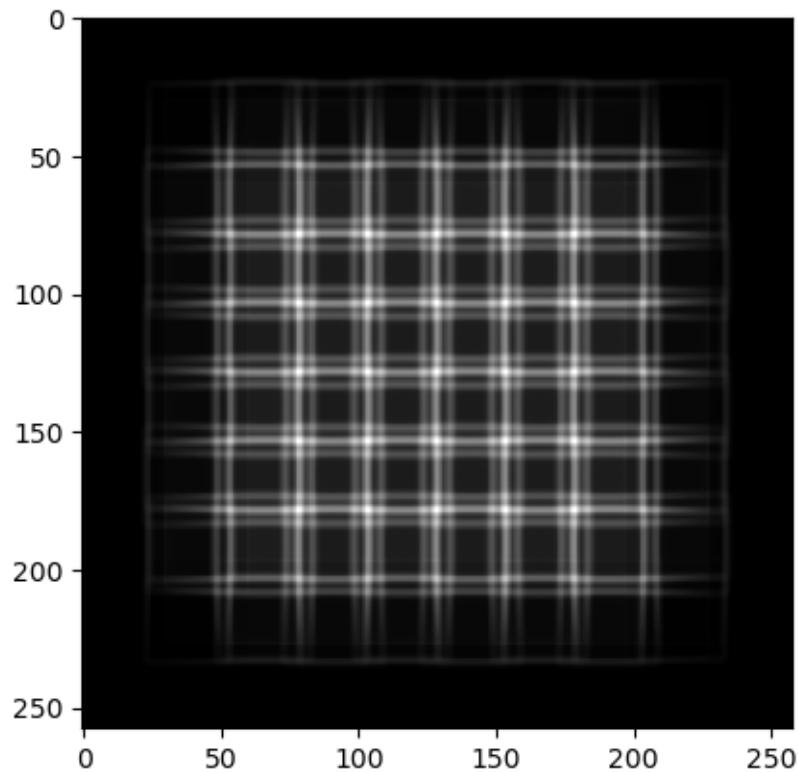
30

105

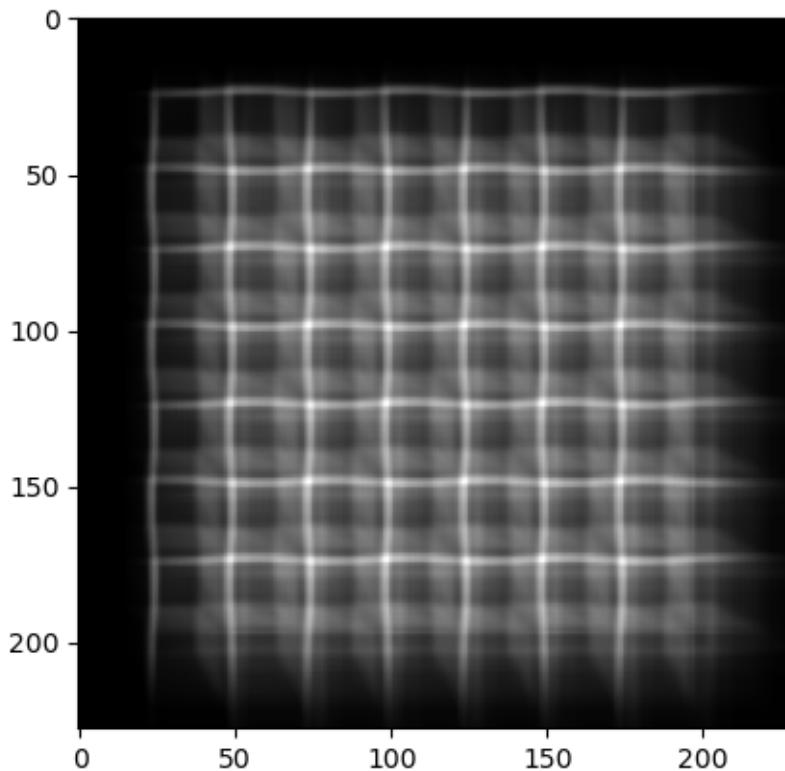


31

106



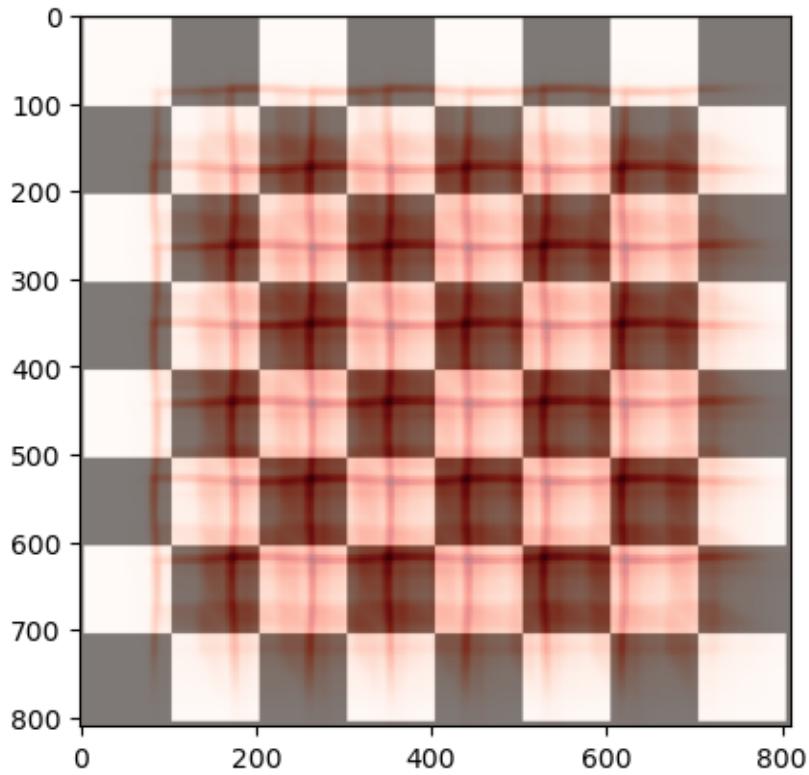
```
[ ]: print_gray(res)
```



```
[ ]: row_f = ori_img.shape[0]//res.shape[0] + 1
      col_f = ori_img.shape[1]//res.shape[1] + 1

[ ]: upscaled_sq = torch.squeeze(F.interpolate(torch.unsqueeze(torch.
      unsqueeze(res, dim=0), dim=0), size=ori_img.shape))

[ ]: plt.imshow(ori_img, cmap='gray')
      plt.imshow(upscaled_sq, cmap='Reds', alpha=0.5)
      plt.show()
```



Moim zdaniem jest to wynik zadawalający, trochę się psuje przez krawędzie (najbardziej zewnętrzne kwadraty mają za mało głosów by być brane pod uwagę).

Dodatkowo rozmiary trochę się psują przez przeskalowania, ale ogólnie wydaje się być dobrze, moim zdaniem. Dodatkowo ja liczę teraz różne rozmiary, a mógłbym wziąć np. tylko 26, co dałoby dużo lepszy wynik.