HW_1

November 12, 2022

1 EE634 HW1

1.0.1 Kutay Ugurlu

[1]:
```
import numpy as np
from scipy.fft import fft, ifft, fft2, ifft2, fftshift
from scipy.signal import convolve2d
from scipy.linalg import toeplitz
from matplotlib import pyplot as plt
from matplotlib import cm
from numpy import pi as pi
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
```

1.1 Q1a

$$x(n_1, n_2) = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$x * h(n_1, n_2) = \begin{bmatrix} 0 & \frac{1}{4} & \frac{1}{4} & 0 \\ \frac{1}{4} & 1 & 1 & \frac{1}{4} \\ \frac{1}{4} & 1 & 1 & \frac{1}{4} \\ 0 & \frac{1}{4} & \frac{1}{4} & 0 \end{bmatrix}$$

Using the linear convolutions dimension expression one can conclude that the filter is 3×3. So let

$$h(-n_1, -n_2) = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Using the corner elements, one can deduce that the corner elements of the filter is 0. With this configuration: * $i = 0$ * $h + i = \frac{1}{4} \implies h = \frac{1}{4}$ * $g + h = \frac{1}{4} \implies g = 0$ * $f + i = \frac{1}{4} \implies f = \frac{1}{4}$ * $e + h + f + i = 1 \implies e = \frac{1}{2}$ * $d + g + e + h = 1 \implies d = \frac{1}{4}$:

By using the symmetry in input and output, one can also conclude that: * $a = c = 0$ * b = $\frac{1}{4}$

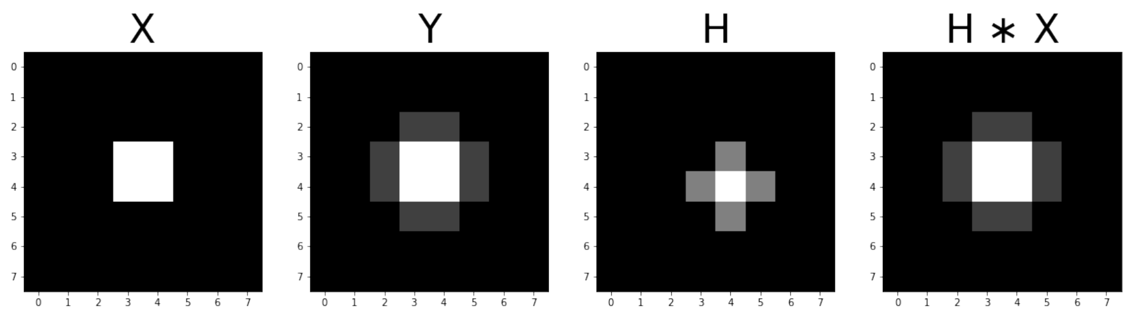
Hence

$$h(n_1, n_2) = \begin{bmatrix} 0 & \frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ 0 & \frac{1}{4} & 0 \end{bmatrix}$$

[2]:

```python
x = np.zeros((8,8))
x[3:5,3:5] = 1
y = np.zeros_like(x)
y[2:6,2:6] = np.array([[0,.25,.25,0],[.25,1,1,.25],[.25,1,1,.25],[0,.25,.25,0]])
h = np.zeros_like(x)
h[3:6,3:6] = np.array([[0,.25,0],[.25,.5,.25],[0,.25,0]])
y_prime = np.abs(fftshift(ifft2(fft2(x)*fft2(h))))
fsize = 40
plt.figure(figsize=(20,80))
plt.subplot(1,4,1)
plt.imshow(np.abs(x))
plt.title('X',fontsize=fsize)
plt.set_cmap(cmap="gray")
plt.subplot(1,4,2)
plt.imshow(np.abs(y))
plt.title('Y',fontsize=fsize)
plt.set_cmap(cmap="gray")
plt.subplot(1,4,3)
plt.imshow(np.abs(h))
plt.title('H',fontsize=fsize)
plt.set_cmap(cmap="gray")
plt.subplot(1,4,4)
plt.imshow(y_prime)
plt.title(r'H $\ast$ X',fontsize=fsize)
plt.set_cmap(cmap="gray")
assert np.isclose(np.sum(y_prime-y),0)
print("Resultant convolution matches the given.")
```

```
Resultant convolution matches the given.
```



2 Q1b

[3]:
```
H = h[3:6,3:6]
H
```

[3]:
```
array([[0.  , 0.25, 0.  ],
       [0.25, 0.5 , 0.25],
       [0.  , 0.25, 0.  ]])
```

[4]:
```
u = np.linalg.svd(H)
singular_values = u[1]
singular_values
```

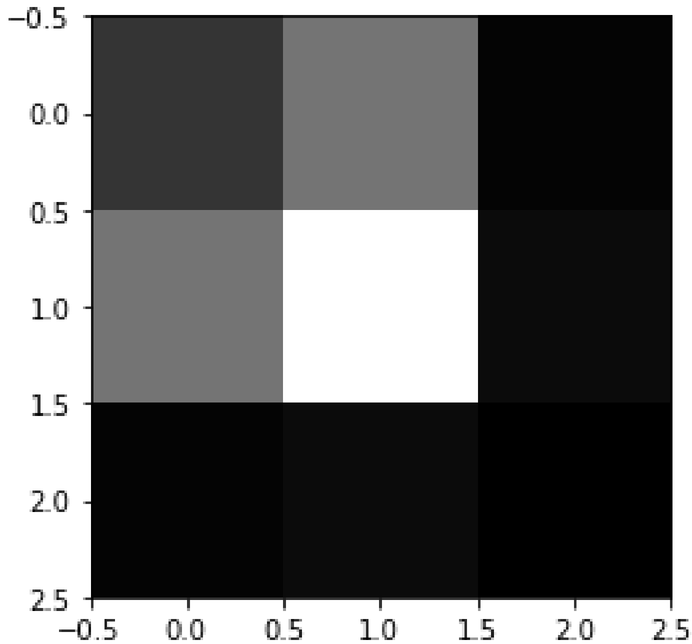
[4]:
```
array([6.83012702e-01, 1.83012702e-01, 8.77708367e-18])
```

2.1 As can be seen above, we have 3 distinct singular values for the convolution kernel. For a kernel to be separable, it should be expressed as one outer product. However, when we use the SVD to decompose the matrix into outer products, we see that it has two nonzero singular values. One separable filter example can be seen below. The kernel is defined as an outer product and it has only one nonzero singular value.

[5]:
```
a = np.random.randint(0,100,(3,1))
b = np.outer(a,a)
u = np.linalg.svd(b)
singular_values = u[1]
print(singular_values)
plt.imshow(b)
```

```
[9.36000000e+03 1.12935218e-12 5.91456839e-16]
```

[5]:
```
<matplotlib.image.AxesImage at 0x1bec5d204f0>
```



3 Q1c

$$\implies |H(w1, w2)| = \frac{1}{2} + cos(\frac{w_1}{2}) + cos(\frac{w_2}{2})$$

This filter acts as low pass filter, since its magnitude have higher values around the origin.

4 Q1c

Since image has n1-n2 symmetry, its DFT has k1-k2 symmetry in the frequency domain, *i.e.* $H(k_1, k_2) = H(k_2, k_1)$

[6]:
```
sum = 0
for n2 in range(3):
    sum += np.exp(-1j*2*pi*n2/3) * (h[0,n2] + h[1,n2] * np.exp(-1j*2*pi/3) +␣
 ↪h[2,n2] * np.exp(-1j*4*pi/3))
print("H(1,1) =",sum)
sum = 0
for n2 in range(3):
    sum += np.exp(-1j*4*pi*n2/3) * (h[0,n2] + h[1,n2] * np.exp(-1j*2*pi/3) +␣
 ↪h[2,n2] * np.exp(-1j*4*pi/3))
print("H(2,2) =",sum)
```

```
H(1,1) = 0j
H(2,2) = 0j
```

4.1 Cross Term calculation example:

[7]:
```
(0.25 + 1*np.exp(-1j*2*pi/3) + 0.25*np.exp(-1j*4*pi/3))
```

[7]:
```
(-0.37499999999999999-0.6495190528383291j)
```

[8]:
```
fft2(H)
```

[8]:
```
array([[ 1.5  -0.j        , -0.375-0.64951905j, -0.375+0.64951905j],
       [-0.375-0.64951905j,  0.   +0.j        ,  0.   -0.j        ],
       [-0.375+0.64951905j,  0.   +0.j        ,  0.   -0.j        ]])
```

4.2 As expected, we again obtained an low pass convolution filter. Higher frequency terms "at the edges" of the filter are zero, whereas center terms have higher magnitude. This is totally expected, since DFT is the sampled version of DTFT where $w = \frac{2\pi k}{N}$.

5 Q1 e

[9]:
```
def linear_conv_mat(h:np.array,output_size):
    L = h.size
    h_ex = np.zeros(output_size)
    h_ex[0:L] = h
    first_row = np.roll(np.flip(h_ex),1)
    return toeplitz(h_ex.T,first_row[0:(output_size-L+1)]) # H + X - 1 =
 ↪output_size

def linear_conv2_mat(h:np.ndarray, image_shape):
    L1,L2 = image_shape
    N1,N2 = h.shape
    H = np.empty(((N1+L1-1)*(N2+L2-1),0))
    image_size = L1*L2
    for i in range(image_size):
        row = np.mod(i,L1)
        col = i//L1
        basis_vec = np.zeros((L1,L2))
        basis_vec[row,col] = 1
        basis_vec_output = convolve2d(h,basis_vec)
        H = np.column_stack((H, basis_vec_output.flatten()))
    return H
```

[10]:
```
for _ in range(50):
    N1 = np.random.randint(0,20)
    N2 = np.random.randint(0,20)
```

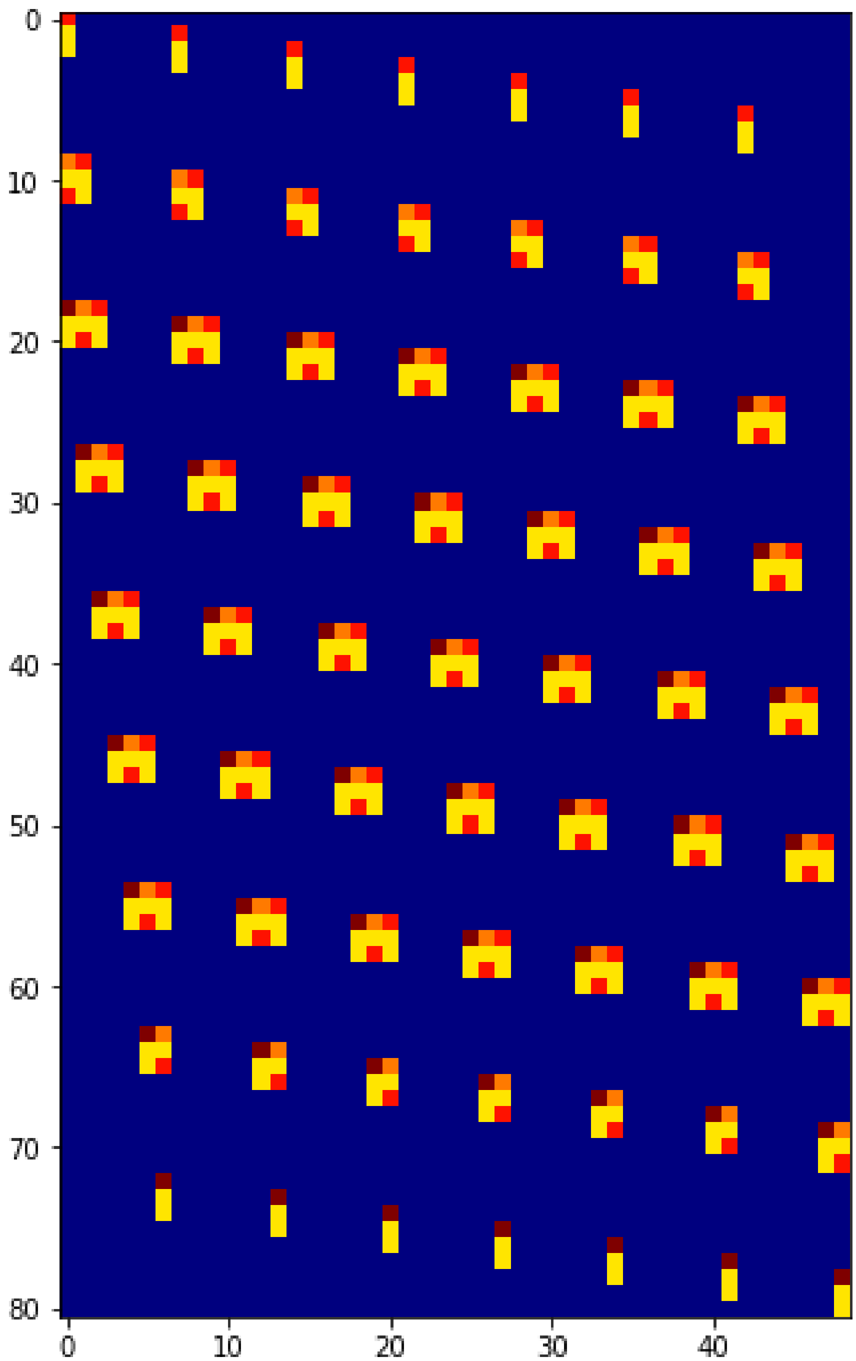
```python
    L1 = np.random.randint(0,20)
    L2 = np.random.randint(0,20)
    H = np.random.randint(0,100,(N1,N2))
    x = np.random.randint(0,100,(L1,L2))
    H_mat = linear_conv2_mat(h=H,image_shape=x.shape)
    y_prime = convolve2d(H,x)
    y_prime_vec = H_mat.dot(x.flatten())
    y_prime_back = np.reshape(y_prime_vec,y_prime.shape,order="F")

    assert np.isclose(np.sum(y_prime-y_prime_back),0) # Check if they are the␣
 ↪same
print("linear_conv2_mat works")
```

```
linear_conv2_mat works
```

[11]:
```python
H = np.random.randint(5,10,(3,3))
x = np.random.randint(5,10,(7,7))
H_mat = linear_conv2_mat(h=H,image_shape=x.shape)
plt.figure(figsize=(18,9))
plt.imshow(H_mat,cmap="jet")
```

[11]:
```
<matplotlib.image.AxesImage at 0x1bec5ce4730>
```


0
10
20
30
40
50
60
70
80
0
10
20
30
40

6 Q1f

[12]:
```
def conv2_by_fft(x:np.ndarray, h:np.ndarray):
    if h.shape[0] > x.shape[0] and h.shape[1] > x.shape[1]:
        x,h = h,x
    L1,L2 = x.shape
    P1,P2 = h.shape
    rows = L1+P1-1
    cols = L2+P2-1
    Y = np.empty((rows,cols))
    for i in range(cols):
        output_col = np.zeros((rows,))
        for p in range(L2): # travel through image columns
            if i-p >= 0 and i-p < P2:
                product = np.multiply(fft(x[:,p],rows),fft(h[:,i-p],rows)) #␣
↪splitted lines for debugging
                output_col += np.real(ifft(product))
                Y[:,i] = output_col
    return Y
```

6.1 Test the function

[13]:
```
for _ in range(1500):
    L1 = np.random.randint(3,15)
    L2 = np.random.randint(3,15)
    N1 = np.random.randint(1,L1-1)
    N2 = np.random.randint(1,L2-1)
    H = np.random.randint(0,100,(N1,N2))
    x = np.random.randint(0,100,(L1,L2))
    y = conv2_by_fft(x,H)
    y_prime = convolve2d(x,H)
    assert np.isclose(np.sum(y_prime-y),0) # Check if they are the same
print("conv2_by_fft works")
```

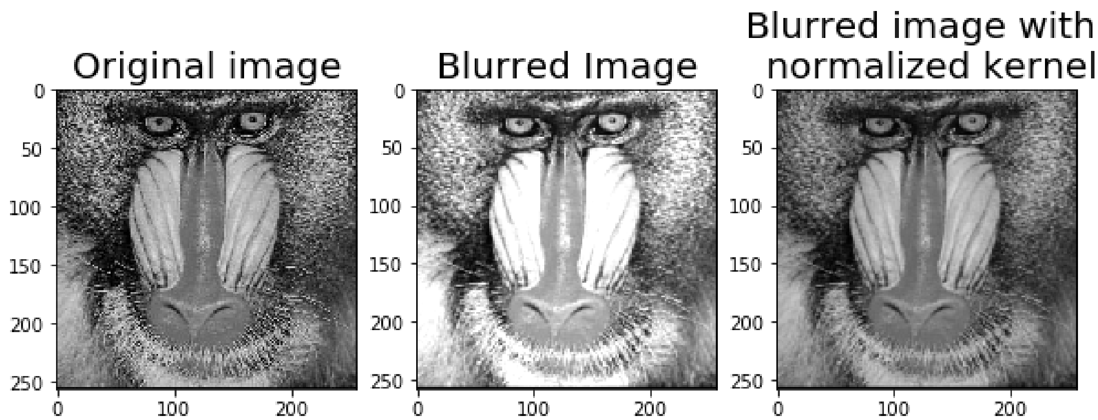
```
conv2_by_fft works
```

7 Q1g

[14]:
```
image = plt.imread("mandrill.bmp")
h = np.array([[0,.25,0],[.25,.5,.25],[0,.25,0]])
blurred_image = conv2_by_fft(h,image)
plt.figure(figsize=(10,40))
plt.subplot(1,3,1)
plt.imshow(image)
plt.title("Original image",fontsize=20)
```

```
plt.subplot(1,3,2)
plt.imshow(blurred_image,vmin=np.min(image), vmax=np.max(image))
blurred_image_normalized = conv2_by_fft(h/np.sum(h),image)
plt.title("Blurred Image",fontsize=20)
plt.subplot(1,3,3)
plt.imshow(blurred_image_normalized,vmin=np.min(image), vmax=np.max(image))
plt.title("Blurred image with \n normalized kernel",fontsize=20)
```

[14]:
```
Text(0.5, 1.0, 'Blurred image with \n normalized kernel')
```



7.1 In the last image, we observe that the hairy part of the cheeks of the mandrill got blurred and is not distinctive as it is in the original image anymore.

8 Q2

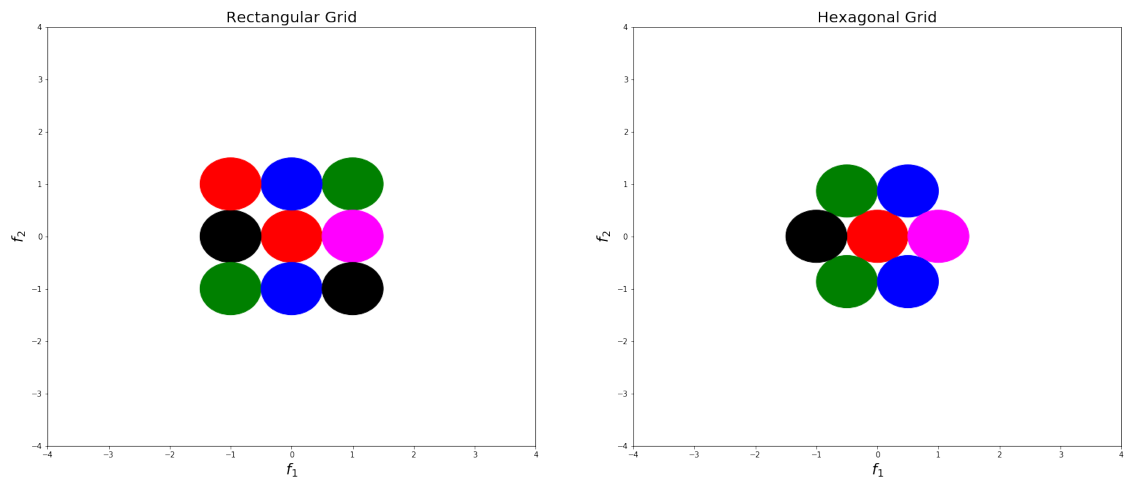
[15]:
```
plt.figure(figsize=(25,10))

plt.subplot(1,2,1)
ax = plt.gca()
ax.cla() # clear things for fresh plot
circle1 = plt.Circle((0, 0), .5, color='r')
circle2 = plt.Circle((1, 1), .5, color='g')
circle3 = plt.Circle((0, 1), .5, color='b')
circle4 = plt.Circle((1, 0), .5, color='magenta')
circle5 = plt.Circle((-1, 0), .5, color='k')
circle6 = plt.Circle((-1, -1), .5, color='g')
circle7 = plt.Circle((0, -1), .5, color='b')
circle8 = plt.Circle((1, -1), .5, color='k')
circle9 = plt.Circle((-1, 1), .5, color='r')
# change default range so that new circles will work
```

```
ax.set_xlim((-4, 4))
ax.set_ylim((-4, 4))
ax.add_patch(circle1)
ax.add_patch(circle2)
ax.add_patch(circle3)
ax.add_patch(circle4)
ax.add_patch(circle5)
ax.add_patch(circle6)
ax.add_patch(circle7)
ax.add_patch(circle8)
ax.add_patch(circle9)
plt.title("Rectangular Grid",fontsize=20)
plt.xlabel(r'$f_1$',fontsize=20)
plt.ylabel(r'$f_2$',fontsize=20)


plt.subplot(1,2,2)
ax = plt.gca()
ax.cla() # clear things for fresh plot
circle1 = plt.Circle((0, 0), .5, color='r')
circle2 = plt.Circle((-0.5, 0.5*np.sqrt(3)), .5, color='g')
circle3 = plt.Circle((0.5, 0.5*np.sqrt(3)), .5, color='b')
circle4 = plt.Circle((1, 0), .5, color='magenta')
circle5 = plt.Circle((-0.5, -0.5*np.sqrt(3)), .5, color='g')
circle6 = plt.Circle((0.5, -0.5*np.sqrt(3)), .5, color='b')
circle7 = plt.Circle((-1, 0), .5, color='k')
# change default range so that new circles will work
ax.set_xlim((-4, 4))
ax.set_ylim((-4, 4))
ax.add_patch(circle1)
ax.add_patch(circle2)
ax.add_patch(circle3)
ax.add_patch(circle4)
ax.add_patch(circle5)
ax.add_patch(circle6)
ax.add_patch(circle7)
plt.title("Hexagonal Grid",fontsize=20)
plt.xlabel(r'$f_1$',fontsize=20)
plt.ylabel(r'$f_2$',fontsize=20)
```

[15]: Text(0, 0.5, 'f_2')

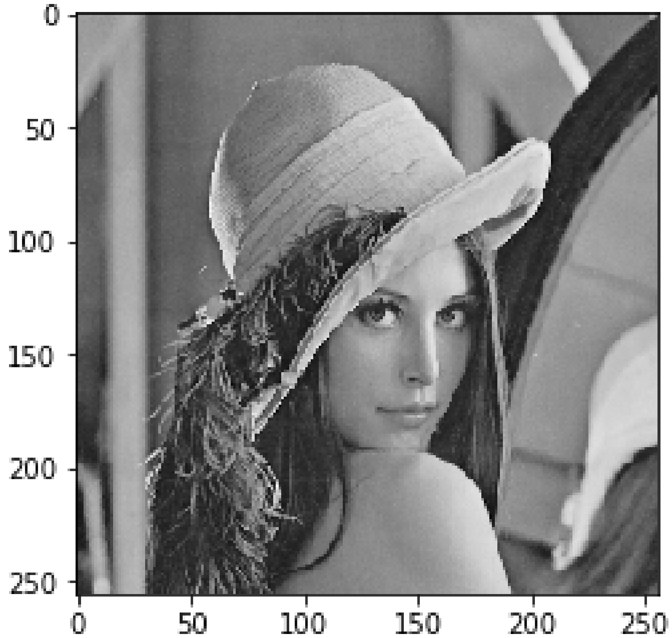


To recover the signal exactly from the frequency spectrum, we should conduct sampling avoiding aliasing. The minimum sampling frequency for this in regular grid turned out to be 1 *cycles*/*meter*, that is maximum 1 meter period.

9 Q3a

[16]:
```
from skimage.color import rgb2gray
image = rgb2gray(plt.imread("256by256grayscaleLena.png"))
minsize = 0
maxsize = image.shape[1]
X = np.arange(minsize, maxsize, 1)
Y = np.arange(minsize, maxsize, 1)
X, Y = np.meshgrid(X, Y)
plt.imshow(image,cmap="gray")
```

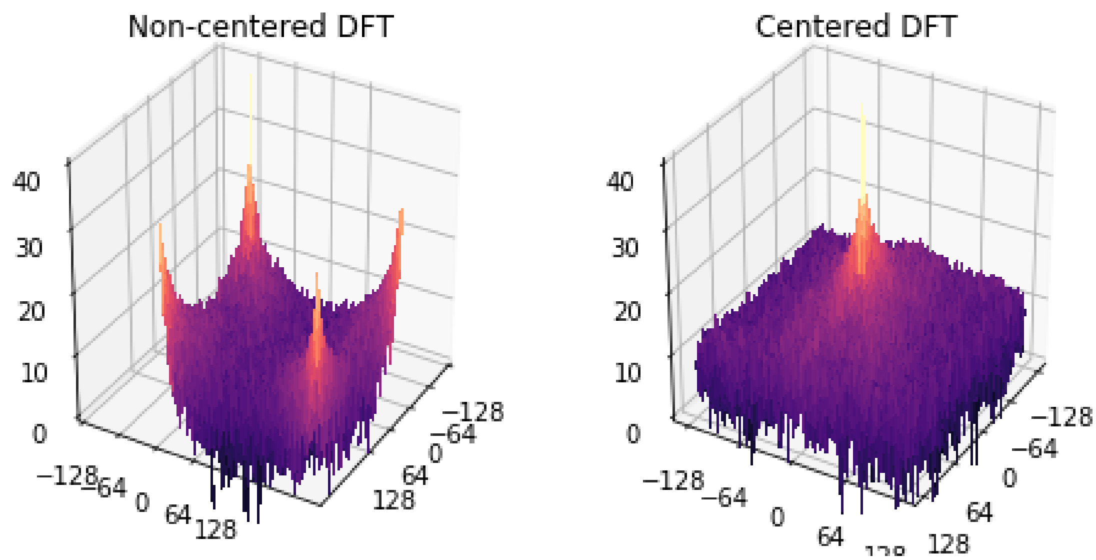
[16]:
```
<matplotlib.image.AxesImage at 0x1bec3fa2400>
```



[17]:
```python
# Plot the surface.
fig = plt.figure(figsize=plt.figaspect(.5))
ax = fig.add_subplot(1, 2, 1, projection='3d')
surf = ax.plot_surface(X, Y, 10*np.log10(np.abs(fft2(image))),rstride=1,
    cstride=1, cmap=cm.magma, linewidth=0, antialiased=False)
ax.set_zlim(0,40)
ax.view_init(30, 30)
ax.set_xticks(np.linspace(-128,128,5))
ax.set_yticks(np.linspace(-128,128,5))
plt.title("Non-centered DFT")

ax = fig.add_subplot(1, 2, 2, projection='3d')
surf = ax.plot_surface(X-128, Y-128, 10*np.log10(np.
 ↪abs(fftshift(fft2(image)))),rstride=1,
    cstride=1, cmap=cm.magma, linewidth=0, antialiased=False)
ax.set_zlim(0,40)
ax.view_init(30, 30)
ax.set_xticks(np.linspace(-128,128,5))
ax.set_yticks(np.linspace(-128,128,5))
plt.title("Centered DFT")
```

[17]:
```
Text(0.5, 0.92, 'Centered DFT')
```

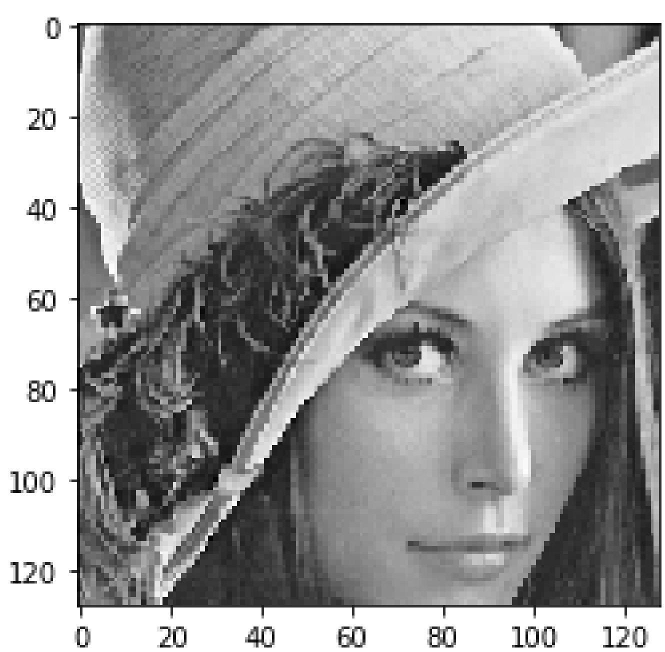


10 Q3b

Take the image and upsample with zeroes in every 2 elements.

[18]:
```
image = rgb2gray(plt.imread("256by256grayscaleLena.png"))
cropped_image = image[60:188,60:188]
plt.imshow(cropped_image)
```

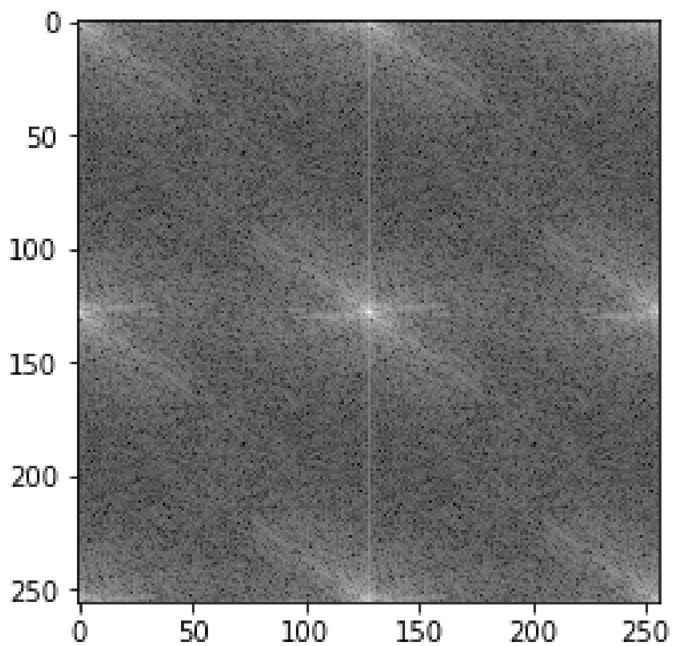
[18]: `<matplotlib.image.AxesImage at 0x1bece034490>`



Calculate its 2D Fourier transform.

[19]:
```
s1,s2 = cropped_image.shape
upsampled_image = np.zeros((2*s1,2*s2))
upsampled_image[::2,::2] = cropped_image
FFT_upsampled = fft2(upsampled_image)
plt.imshow(np.log10(np.abs(FFT_upsampled)))
```

[19]: <matplotlib.image.AxesImage at 0x1bec3b0c160>

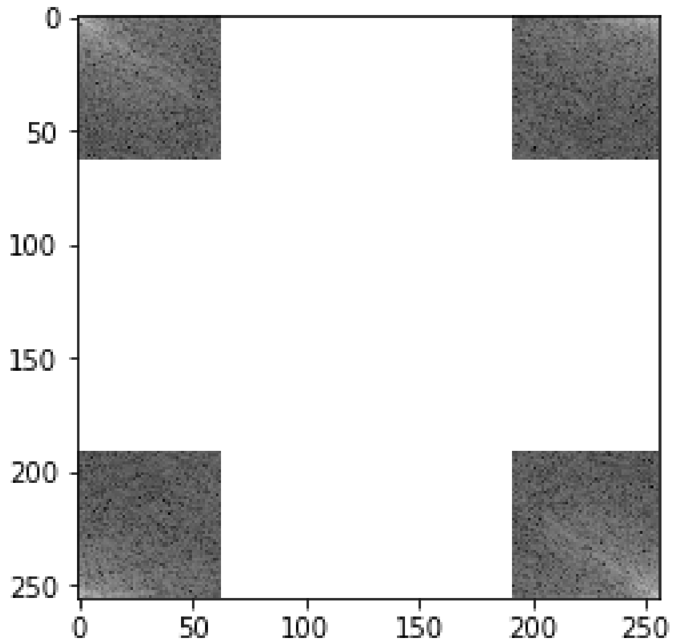


Filter the "central" portion of the frequency spectrum out with an ideal low pass filter.

[20]:
```
## Take the center portion, equate remaining to 0
shifted = fftshift(FFT_upsampled)
Filtered_FFT = np.zeros_like(FFT_upsampled)
Filtered_FFT[64:192,64:192] = FFT_upsampled[64:192,64:192]
Filtered_FFT = fftshift(Filtered_FFT)
plt.imshow(np.log10(np.abs(Filtered_FFT)))
```

```
<ipython-input-20-83fcb38667d6>:6: RuntimeWarning: divide by zero encountered in
log10
  plt.imshow(np.log10(np.abs(Filtered_FFT)))
```

[20]: <matplotlib.image.AxesImage at 0x1becc562220>



Take the inverse 2D FFT to obtain the interpolated image.

[21]:
```
interpolated_image = np.real(ifft2(Filtered_FFT))
plt.imshow(interpolated_image)
```

[21]: <matplotlib.image.AxesImage at 0x1becc6409d0>