ECE_253_hw3_Kalgundi_Srinivas_A59010584

November 22, 2021

1 Srinidhi Bharadwaj Kalgundi Srinivas

2 A59010584

Academic Integrity Policy: Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual to whom it is assigned, without unauthorized aid of any kind. By including this in my report, I agree to abide by the Academic Integrity Policy mentioned above.

2.0.1 Problem 1: Canny Edge Detection

[1]:
```
import numpy as np
import cv2
from matplotlib import pyplot as plt
from math import pi
```

[2]:
```
# Helper function to smoothen the image
def smoothen_image(image):
    local_image = np.copy(image)
    #Kernel is considered
    kernel = np.array([[2, 4, 5, 4, 2],
                       [4, 9, 12, 9, 4],
                       [5, 12, 15, 12, 5],
                       [4, 9, 12, 9, 4],
                       [2, 4, 5, 4, 2]])

    #Flipping the kernel to aid in convolution
    kernel = cv2.flip(kernel, -1)
    kernel = (1/159) * kernel

    #Convolving the image with the kernel to obtain smoothened image
    ret_img = cv2.filter2D(src=local_image, ddepth=-1, kernel=kernel)
    return ret_img


# Helper function to find gradients
```

```python
def find_gradients(image):
    kx = np.array([[-1, 0, 1],
                   [-2, 0, 2],
                   [-1, 0, 1]], np.float32)
    ky = np.array([[-1, -2, -1],
                   [0, 0, 0],
                   [1, 2, 1]], np.float32)

    #Flipping kernel 180 degrees for convolution
    kx = cv2.flip(kx, -1)
    ky = cv2.flip(ky, -1)

    Ix = cv2.filter2D(src=image,ddepth=-1, kernel=kx)
    Iy = cv2.filter2D(src=image, ddepth=-1, kernel=ky)

    #Calculate magnitude and phase of the image
    G = np.hypot(Ix, Iy) #Hypot function does sqrt(x**2 + y**2)
    G = G / G.max() * 255 #Normalizing
    phase = np.arctan2(Iy, Ix)
    return G, phase

#Non maximum suppression
def nms(image, phase):
    [rows, cols] = image.shape
    ret_image = np.zeros((rows, cols), dtype=np.uint8)
    #print(channels)
    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            if (0 <= phase[i,j] < 22.5) or (157.5 <= phase[i,j] <= 180):
                max_u = image[i, j+1]
                max_v = image[i, j-1]
            elif (22.5 <= phase[i,j] < 67.5):
                max_u = image[i+1, j-1]
                max_v = image[i-1, j+1]
            elif (67.5 <= phase[i,j] < 112.5):
                max_u = image[i+1, j]
                max_v = image[i-1, j]
            elif (112.5 <= phase[i,j] < 157.5):
                max_u = image[i-1, j-1]
                max_v = image[i+1, j+1]

            #Condition to check if the given intensity value is greater or␣
 ↪lesser than neighbors
            if (image[i,j] >= max_u) and (image[i,j] >= max_v):
                ret_image[i,j] = image[i,j]
    return ret_image
```

```python
#Helper function to threshold, double thresholding is used for better results

def threshold(image, lower, higher):
    #Calculate lower threshold value and uppper threshold value
    high = image.max() * higher;
    low = high * lower;
    print("Low and high threshold values are {0} and {1}".format(low,high))
    [rows, cols]= image.shape
    ret_image = np.zeros((rows,cols), dtype=np.int32)

    #Lower and upper bound
    weak = np.uint8(25)
    strong = np.uint8(255)

    #Indices of strong and weak intensity values
    strong_i, strong_j = np.where(image >= high)
    weak_i, weak_j = np.where((image < high) & (image >= low))

    ret_image[strong_i, strong_j] = strong
    ret_image[weak_i, weak_j] = weak

    return ret_image

#Main function that performs canny edge detection
def canny_detector(image, te_low, te_high):
    # Image smoothing
    smooth_image = smoothen_image(image)

    #Gradient finding
    mag, phase = find_gradients(smooth_image)
    mag = mag.astype(np.uint8)  # Converting magnitude to an integer matrix
    phase = (phase*180)/(1*pi)  # Converting phase to angle from radians
    phase[phase<0] += 180 # Cycling the negative phases

    #Non-Maximum suppression
    nms_image = nms(mag, phase)

    #Thresholding
    ret_image = threshold(nms_image, te_low, te_high)

    return mag, nms_image, ret_image
```

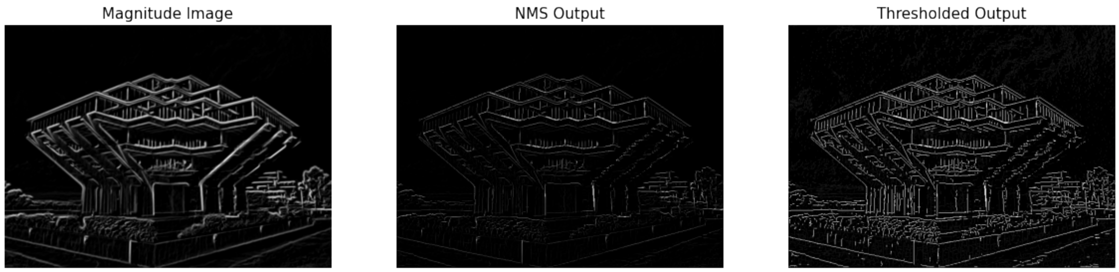
[3]:
```python
geisel = cv2.imread("geisel.jpg")
geisel = cv2.cvtColor(geisel, cv2.COLOR_BGR2GRAY)
plt.imshow(geisel, cmap='gray')
plt.axis("off")
```

[3]: (-0.5, 639.5, 475.5, -0.5)

[4]:
```python
mag, nms_image, ret_image = canny_detector(geisel, 0.05, 0.09)

fig, axs = plt.subplots(1,3, figsize=(20, 5))
ax1= fig.add_subplot(1,3,1)
ax1.title.set_text("Magnitude Image")
ax1.title.set_size(15)
ax1.axis('off')
ax1.imshow(mag, cmap='gray')
axs[0].axis('off')

ax1= fig.add_subplot(1,3,2)
ax1.title.set_text("NMS Output")
ax1.title.set_size(15)
ax1.axis('off')
ax1.imshow(nms_image, cmap='gray')
axs[1].axis('off')

ax1= fig.add_subplot(1,3,3)
ax1.title.set_text("Thresholded Output")
ax1.title.set_size(15)
ax1.axis('off')
ax1.imshow(ret_image, cmap='gray')
axs[2].axis('off')
```

```
Low and high threshold values are 1.1475 and 22.95
```

[4]:
```
(0.0, 1.0, 0.0, 1.0)
```



Solution:

- Canny Edge detector takes in 3 inputs; grayscale image, lower threshold, upper threshold (2 thresholds are used as double thresholding method is employed for better results).
- Gaussain blurring/smoothing is performed using Gaussian kernel by "smoothen_image" function
- Gradients along horizontal and vertical directions are calculated using Sobel filters. Sobel kernels are flipped before performing convolution using cv2.filter2D function. "find_gradients" function returns magnitude and phase images (matrices) and the phase matrix is returned in radians which is then converted to degrees and passed to the non maximum suppression function.
- Non maximum suppression is performed based on the phase values at individual pixel locations by the "nms" function.
- "threshold" function is called on the "nms" image with lower and upper threshold values. Double thresholding is performed where the indices of pixel intensities that are greater than upper threshold values are set to **255** and the values between lower and upper threshold values are set to **25**.

2.0.2 Final output is as show in the above images.

2.1 Threshold values used for double thresholding are 0.05 and 0.09. Values used are 255 and 25. Low and high values calculated from "threshold" function are 1.1475 and 22.95

Note: - Added images from OpenCVs canny output for comparision. - Also attached is canny edge detection function ran on the color image

2.2 Comparing output of implemented canny edge detector with OpenCV edge detector for verification

[5]:
```
canny_out = cv2.Canny(geisel, 25, 255, L2gradient=True)

fig, axs = plt.subplots(1,2, figsize=(20, 5))
ax1= fig.add_subplot(1,2,1)
ax1.title.set_text("Implemented Canny Output")
ax1.title.set_size(15)
ax1.axis('off')
```

```
ax1.imshow(ret_image, cmap='gray')
axs[0].axis('off')

ax1= fig.add_subplot(1,2,2)
ax1.title.set_text("OpenCV Canny Output")
ax1.title.set_size(15)
ax1.axis('off')
ax1.imshow(canny_out, cmap='gray')
axs[1].axis('off')
```

[5]:
```
(0.0, 1.0, 0.0, 1.0)
```

2.2.1 Problem 2: Butterworth Notch Reject Filtering in Frequency Domain

[6]:
```
import numpy as np
import cv2
from matplotlib import pyplot as plt
import math
```

2.2.2 Problem 2: i)

[7]:
```
car = cv2.imread("Car.tif")
car = cv2.cvtColor(car, cv2.COLOR_BGR2GRAY)
```

[8]:
```
car.shape
```

[8]:
```
(246, 168)
```

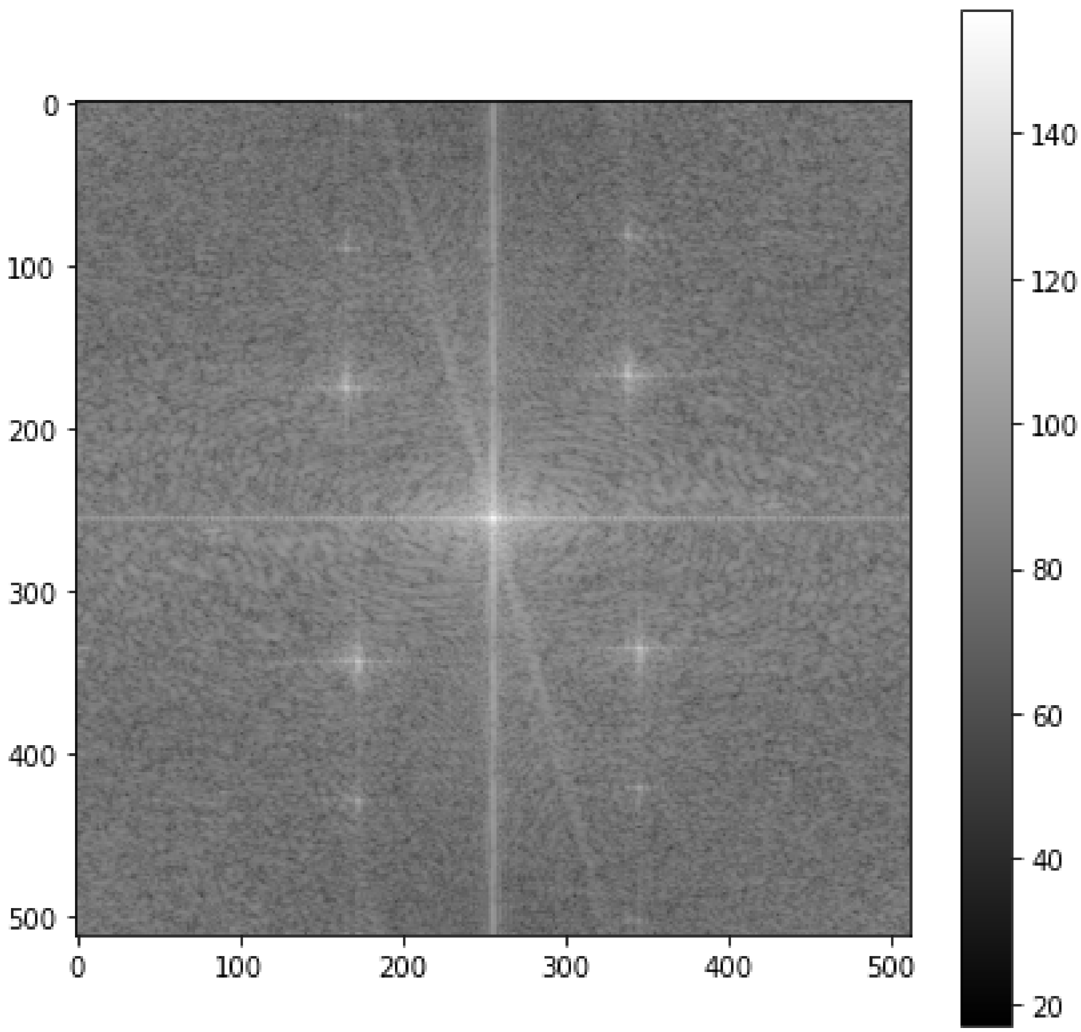
[9]:
```
#Finding the pad size and zero padding the image
x_pad = (512 - car.shape[0]) // 2
y_pad = (512 - car.shape[1]) // 2
padded_car = cv2.copyMakeBorder(car, x_pad, x_pad, y_pad, y_pad, cv2.
↪BORDER_CONSTANT, (0,0,0))
```

[10]:
```
#FFT calculation using numpy's fft function
#fft is calculated, shifted and scaled with as 10l*og(F(u,v))
```

```
car_fft = np.fft.fft2(padded_car)
car_shift = np.fft.fftshift(car_fft)
magnitude_spectrum = 10*np.log(np.abs(car_shift))
magnitude_spectrum = magnitude_spectrum.astype(np.uint8)
```

[11]:
```
plt.figure(figsize=(7, 7))
ax = plt.imshow(magnitude_spectrum, cmap='gray')
plt.colorbar(ax)
```

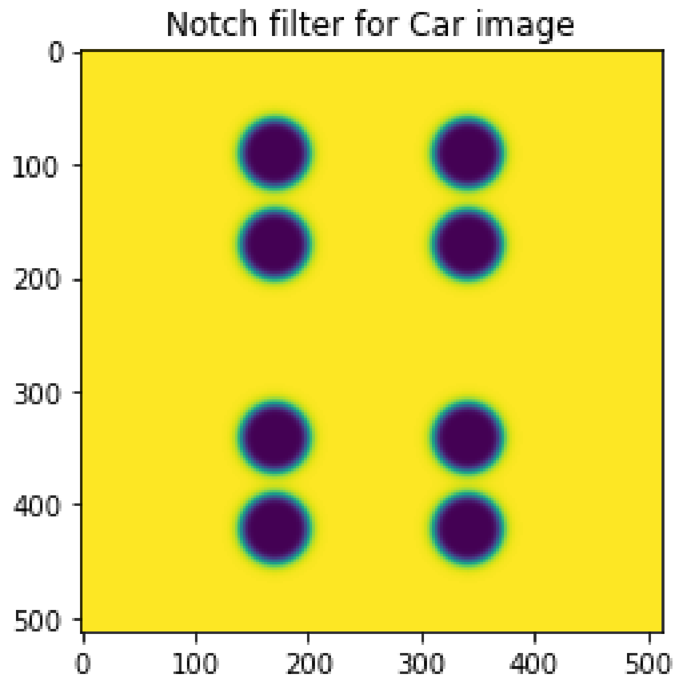
[11]: <matplotlib.colorbar.Colorbar at 0x7fd32009a430>



[12]:
```
n = float(6)  #Filter order
D_0 = float(30) #Radius (Empirically found)

notch_filter = np.zeros((512, 512), dtype=np.float64)
```

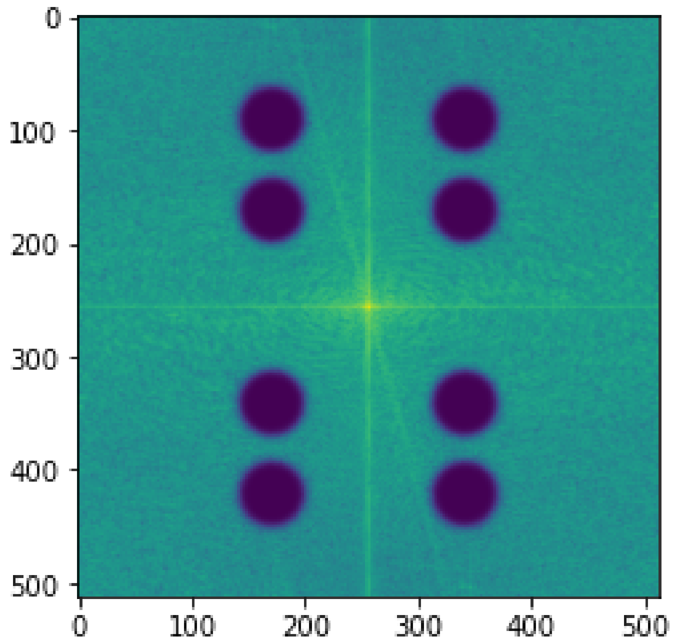
```
#Calculating the values in the notch_filter according to the equation

x_axis = np.linspace(-256,255,512)
y_axis = np.linspace(-256,255,512)
[u,v] = np.meshgrid(x_axis,y_axis)
eps = 10**-5
filt_order = 2*n
for i in range(512):
    for j in range(512):
        D1 = math.sqrt((u[i, j]-85)**2 + (v[i, j]+165)**2) + eps
        D1_k = math.sqrt((u[i, j]+85)**2 + (v[i, j]-165)**2)+ eps

        D2 = math.sqrt((u[i, j]-85)**2 + (v[i, j]+85)**2)+ eps
        D2_k = math.sqrt((u[i, j]+85)**2 + (v[i, j]-85)**2)+ eps

        D3 = math.sqrt((u[i, j]-85)**2 + (v[i, j]-85)**2)+ eps
        D3_k = math.sqrt((u[i, j]+85)**2 + (v[i, j]+85)**2)+ eps

        D4 = math.sqrt((u[i, j]-85)**2 + (v[i, j]-165)**2)+ eps
        D4_k = math.sqrt((u[i, j]+85)**2 + (v[i, j]+165)**2)+ eps

        val1 = (1/(1+(D_0/D1)**filt_order)) * (1/(1+(D_0/D1_k)**filt_order))
        val2 = (1/(1+(D_0/D2)**filt_order)) * (1/(1+(D_0/D2_k)**filt_order))
        val3 = (1/(1+(D_0/D3)**filt_order)) * (1/(1+(D_0/D3_k)**filt_order))
        val4 = (1/(1+(D_0/D4)**filt_order)) * (1/(1+(D_0/D4_k)**filt_order))
        #print(val1, val2, val3, val4)
        notch_filter[i, j] = val1*val2*val3*val4

plt.imshow(notch_filter)
plt.title("Notch filter for Car image")
```

[12]:
```
Text(0.5, 1.0, 'Notch filter for Car image')
```



[13]:
```
# Filter the magnitude spectrum using the calculated notch filter and plot the␣
↪resulting output
#This is for visualization purposes
filtered_output = magnitude_spectrum * notch_filter
plt.imshow(filtered_output)
```

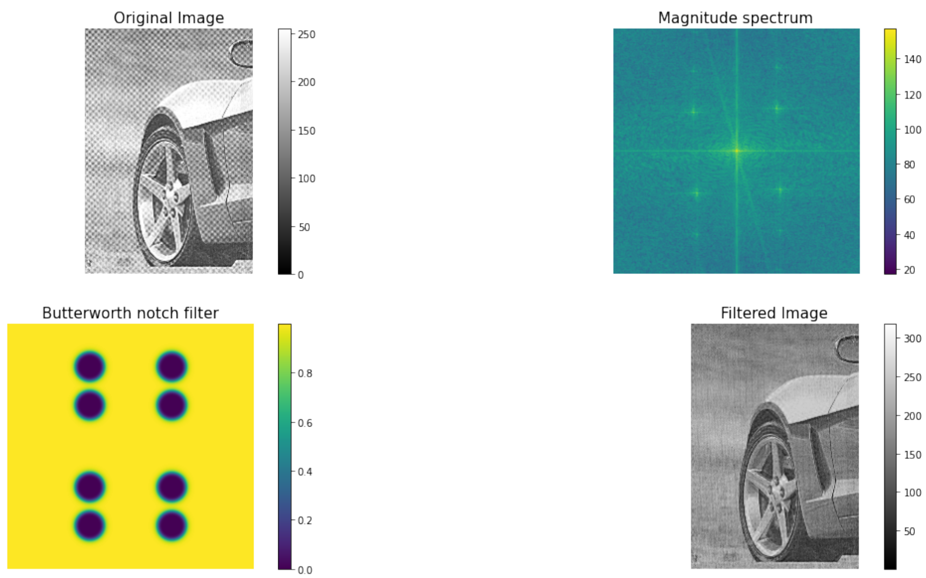
[13]:
```
<matplotlib.image.AxesImage at 0x7fd3503ed070>
```



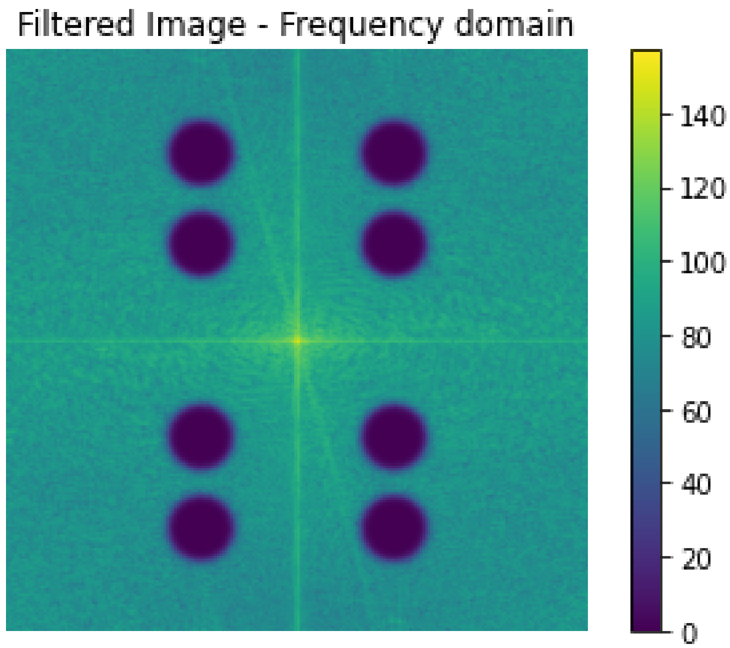
[14]:
```
#Filter the given input image that is in the frequency domain
#Shift the origin to its original position (to counteract the shifting done␣
↪from spatial->frequency domain)
#Calculate the inverse fft to convert from frequency->spatial domain
filtered_out = car_shift * notch_filter
shifted = np.fft.ifftshift((filtered_out))
spatial = np.fft.ifft2(shifted)

#Remove padding
spatial_out = np.abs(spatial)[x_pad:512-x_pad, y_pad:512-y_pad]
```

[15]:
```
fig, axs = plt.subplots(2,2, figsize=(20, 10))
ax1= fig.add_subplot(2,2,1)
ax1.title.set_text("Original Image")
ax1.title.set_size(15)
ax1.axis('off')
ax = ax1.imshow(car, cmap='gray')
axs[0, 0].axis('off')
plt.colorbar(ax)

ax1= fig.add_subplot(2,2,2)
ax1.title.set_text("Magnitude spectrum")
ax1.title.set_size(15)
ax1.axis('off')
ax = ax1.imshow(magnitude_spectrum)
axs[0, 1].axis('off')
```

```
plt.colorbar(ax)

ax1= fig.add_subplot(2,2,3)
ax1.title.set_text("Butterworth notch filter")
ax1.title.set_size(15)
ax1.axis('off')
ax = ax1.imshow(notch_filter)
axs[1, 0].axis('off')
plt.colorbar(ax)

ax1= fig.add_subplot(2,2,4)
ax1.title.set_text("Filtered Image")
ax1.title.set_size(15)
ax1.axis('off')
ax = ax1.imshow(spatial_out, cmap='gray')
axs[1, 1].axis('off')
plt.colorbar(ax)

plt.show()

ax =  plt.imshow(filtered_output)
plt.title("Filtered Image - Frequency domain")
plt.axis("off")
plt.colorbar(ax)
```



[15]: `<matplotlib.colorbar.Colorbar at 0x7fd330be24c0>`



2.2.3 Parameters used for filtering the moire effect from the given image are:

- Order of the filter - n = 6
- D0 = 30 (Experimented with multiple values and found that the best result is obatained for D0 = 30)
- u1, v1 = (85, -165)
- u2, v2 = (85, -85)
- u3, v3 = (85, 85)
- u4, v4 = (85, 165)

Note: (uk, vk) values are caluclated by visual inspection.

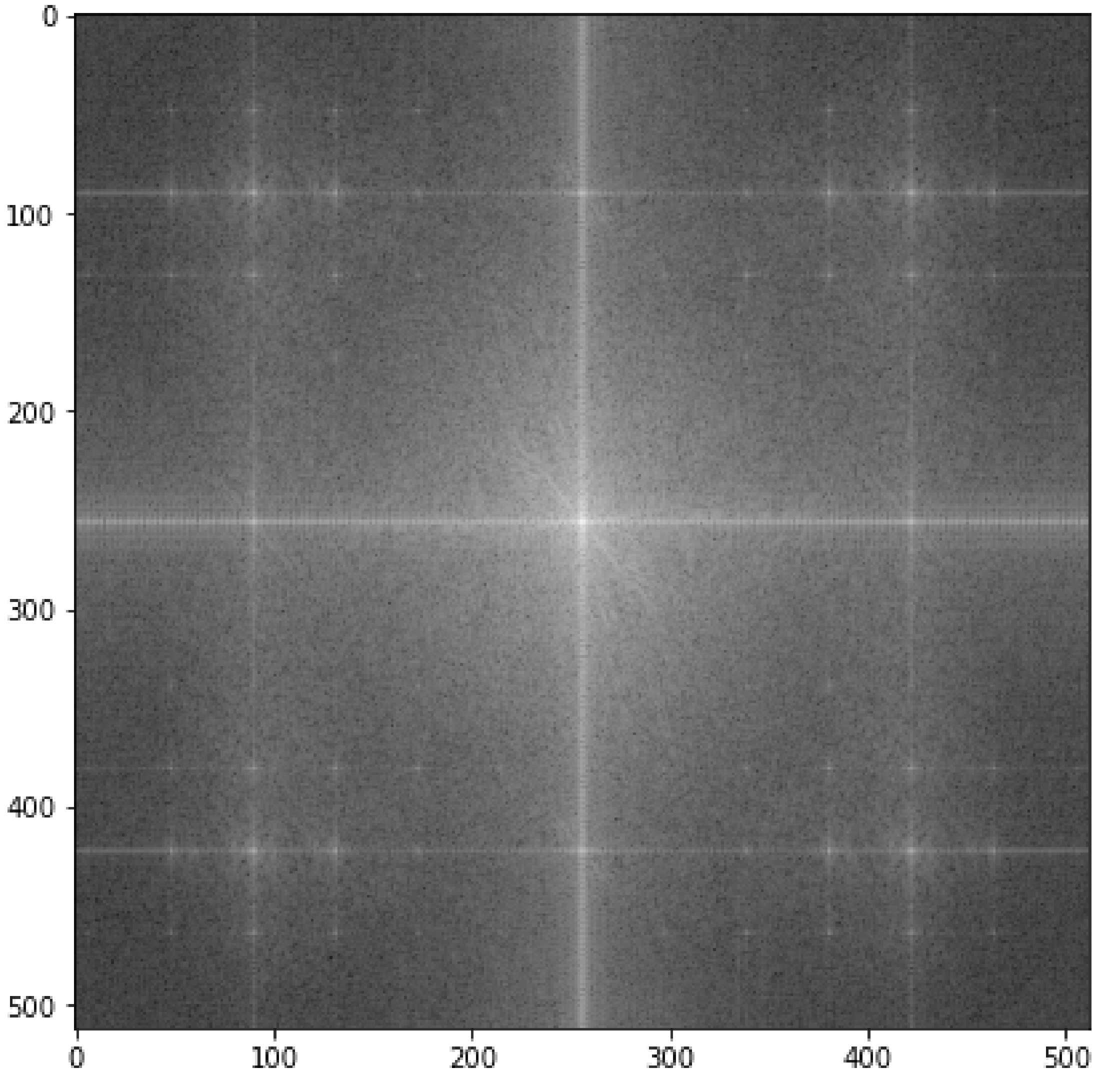
2.2.4 Problem 2: ii)

[16]:
```
street = cv2.imread("Street.png")
street = cv2.cvtColor(street, cv2.COLOR_BGR2GRAY)
```

[17]:
```
#Finding the pad size and zero padding the image
x_pad = (512 - street.shape[0]) // 2
y_pad = (512 - street.shape[1]) // 2
padded_street = cv2.copyMakeBorder(street, x_pad, x_pad, y_pad, y_pad+1, cv2.
 ↪BORDER_CONSTANT, (0,0,0))

#FFT calculation using numpy's fft function
#fft is calculated, shifted and scaled with as 10l*og(F(u,v))
street_fft = np.fft.fft2(padded_street)
```

```python
street_shift = np.fft.fftshift(street_fft)
magnitude_spectrum = 10*np.log(np.abs(street_shift))
magnitude_spectrum = magnitude_spectrum.astype(np.uint8)

plt.figure(figsize=(7, 7))
plt.imshow(magnitude_spectrum, cmap='gray')
```

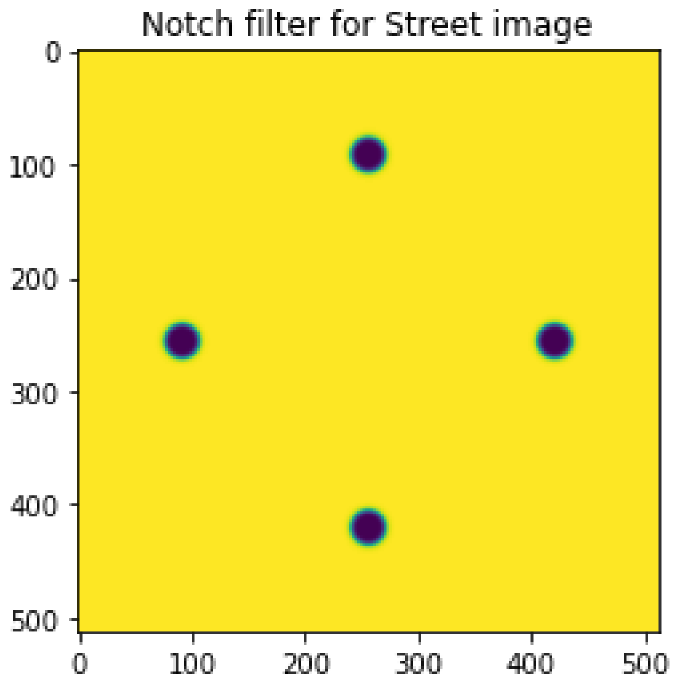
[17]: <matplotlib.image.AxesImage at 0x7fd342332a30>



[18]:
```python
n = float(6) # Filter order
D_0 = float(15) #Radius

notch_filter = np.zeros((512, 512), dtype=np.float64)
```

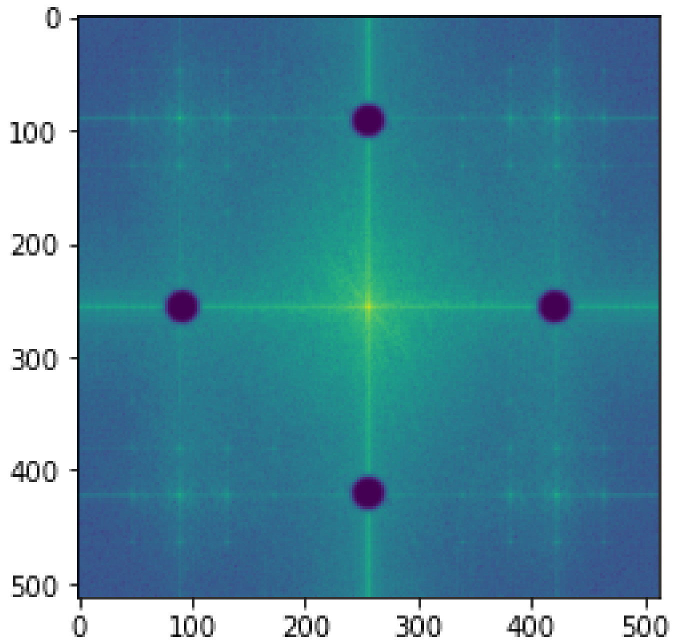
```python
#Calculating the values in the notch_filter according to the equation
#Has 4 nothces as the symmetry is broken when origin was moved to (0,0) instead␣
↪of (P/2, Q/2)
#If the origin was at (P/2, Q/2), 2 points would have been sufficient
# (u, v) values are found by visual inspection
x_axis = np.linspace(-256,255,512)
y_axis = np.linspace(-256,255,512)
[u,v] = np.meshgrid(x_axis,y_axis)
eps = 10**-5
filt_order = 2*n
for i in range(512):
    for j in range(512):
        D1 = math.sqrt((u[i, j]-0)**2 + (v[i, j]-164)**2) + eps
        D1_k = math.sqrt((u[i, j]+0)**2 + (v[i, j]+164)**2)+ eps

        D2 = math.sqrt((u[i, j]-164)**2 + (v[i, j]-0)**2)+ eps
        D2_k = math.sqrt((u[i, j]+164)**2 + (v[i, j]+0)**2)+ eps


        val1 = (1/(1+(D_0/D1)**filt_order)) * (1/(1+(D_0/D1_k)**filt_order))
        val2 = (1/(1+(D_0/D2)**filt_order)) * (1/(1+(D_0/D2_k)**filt_order))

        notch_filter[i, j] = val1*val2*val3*val4

plt.imshow(notch_filter)
plt.title("Notch filter for Street image")
```

[18]: Text(0.5, 1.0, 'Notch filter for Street image')



[19]:
```
# Filter the magnitude spectrum using the calculated notch filter and plot the
 ↪resulting output
#This is for visualization purposes
filtered_output = magnitude_spectrum * notch_filter
plt.imshow(filtered_output)
```

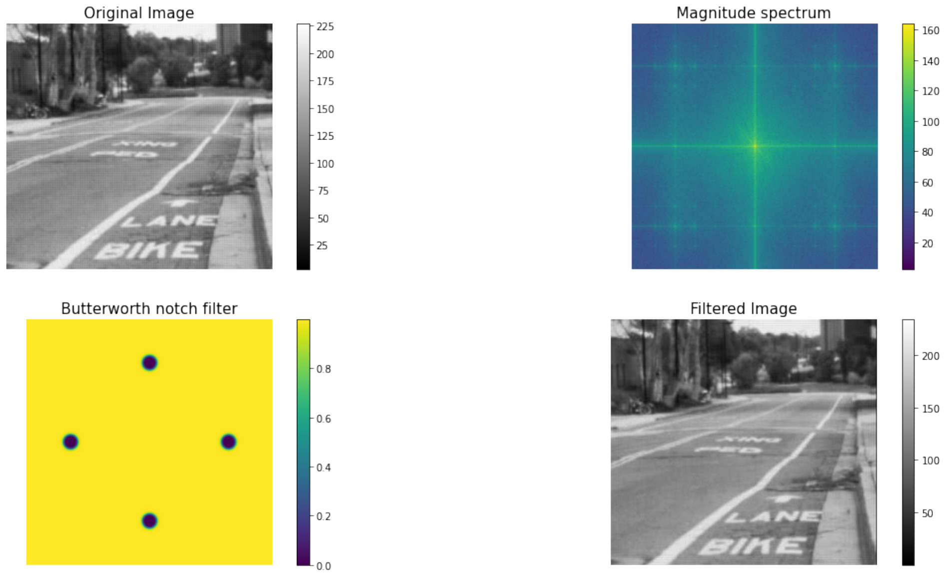
[19]: <matplotlib.image.AxesImage at 0x7fd2f0095310>



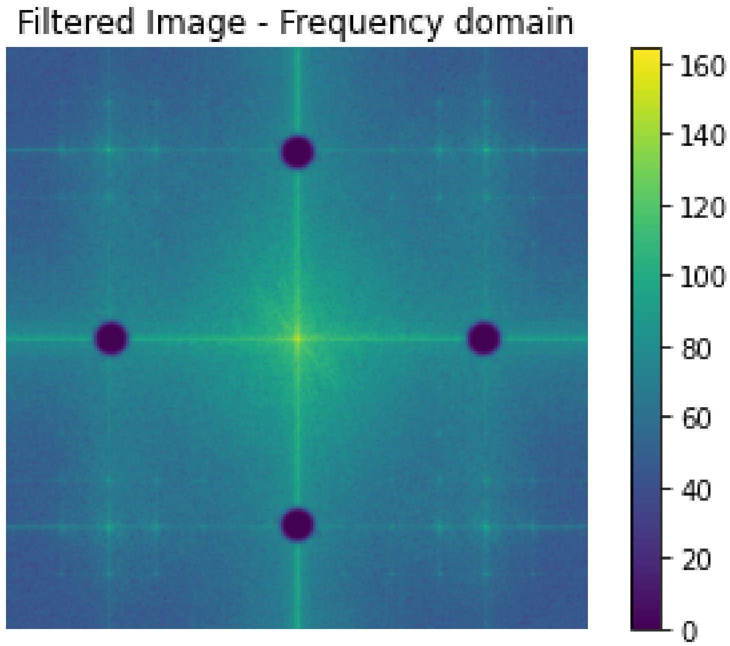
[20]:
```
#Filter the given input image that is in the frequency domain
#Shift the origin to its original position (to counteract the shifting done␣
↪from spatial->frequency domain)
#Calculate the inverse fft to convert from frequency->spatial domain
filtered_out = street_shift * notch_filter
shifted = np.fft.ifftshift((filtered_out))
spatial = np.fft.ifft2(shifted)

#Remove padding
spatial_out = np.abs(spatial)[x_pad:512-x_pad, y_pad:512-y_pad]
```

[21]:
```
fig, axs = plt.subplots(2,2, figsize=(20, 10))
ax1= fig.add_subplot(2,2,1)
ax1.title.set_text("Original Image")
ax1.title.set_size(15)
ax1.axis('off')
ax = ax1.imshow(street, cmap='gray')
axs[0, 0].axis('off')
plt.colorbar(ax)

ax1= fig.add_subplot(2,2,2)
ax1.title.set_text("Magnitude spectrum")
ax1.title.set_size(15)
ax1.axis('off')
ax = ax1.imshow(magnitude_spectrum)
axs[0, 1].axis('off')
```

```
plt.colorbar(ax)

ax1= fig.add_subplot(2,2,3)
ax1.title.set_text("Butterworth notch filter")
ax1.title.set_size(15)
ax1.axis('off')
ax = ax1.imshow(notch_filter)
axs[1, 0].axis('off')
plt.colorbar(ax)

ax1= fig.add_subplot(2,2,4)
ax1.title.set_text("Filtered Image")
ax1.title.set_size(15)
ax1.axis('off')
ax = ax1.imshow(spatial_out, cmap='gray')
axs[1, 1].axis('off')
plt.colorbar(ax)

plt.show()

ax = plt.imshow(filtered_output)
plt.title("Filtered Image - Frequency domain")
plt.axis("off")
plt.colorbar(ax)
```



[21]: <matplotlib.colorbar.Colorbar at 0x7fd330cb1fa0>



2.2.5 Parameters used for filtering the horizontal and vertical lines from the given image are:

- Filter order - n = 6
- D0 = 15 (Experimented with multiple values and found that the best result is obatained for D0 = 15)
- u1, v1 = (0, 164)
- u2, v2 = (164, 0)

Note: - (uk, vk) values are calculated empirically - Difference between the original image and the filtered image can be observed when zoomed in

2.2.6 Problem - 3

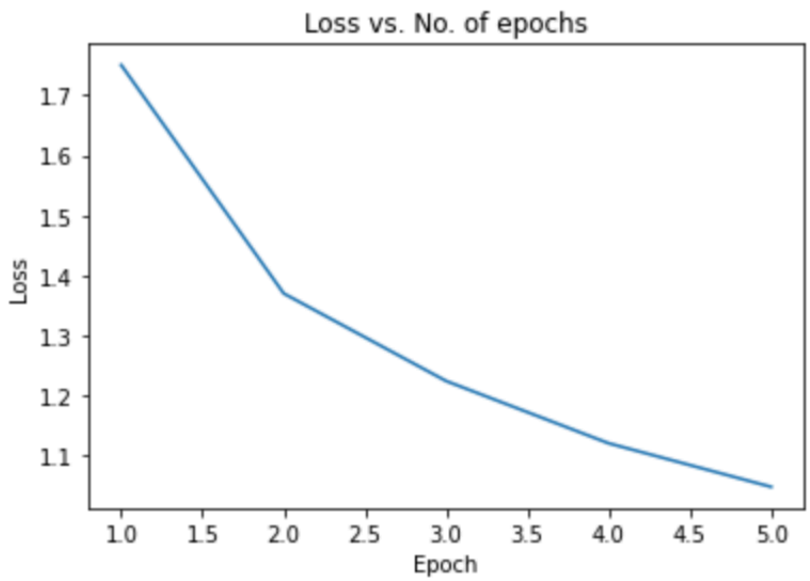
ii) How many images and batches are used to train the network?

Ans: **There are 50,000 images in the training set of size 32x32x3. Size of the training set is found using "trainset.data.shape" whose output is (50000, 32, 32, 3). Batch size of 4 is used for training purposes.**

iii) Do we normalize the images? What do we do in the example?

Ans: **Yes, the images are being normalized. The output of torchvision datasets are in the range [0, 1], which is transformed to tensors of normalized range [-1, 1]. Mean and Standard Deviation used are 0.5**

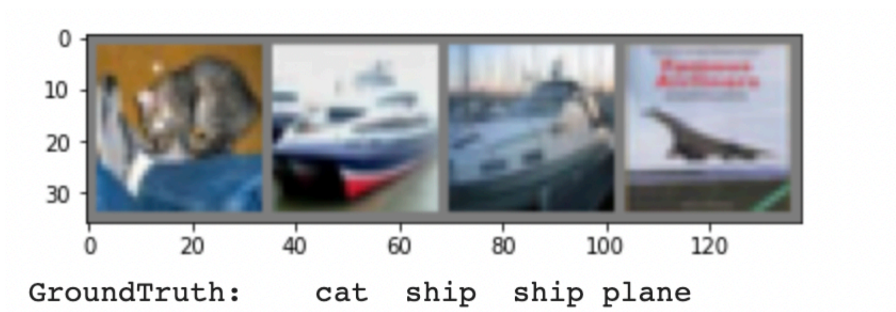
iv) The losses are dropping! Can you plot out the training loss?



Loss Versus Epoch

v) Now the network is done training. Can you check some successful cases and some failure cases (show some images classified by the network)?

Test images and classes



Test images and classes

Predicted

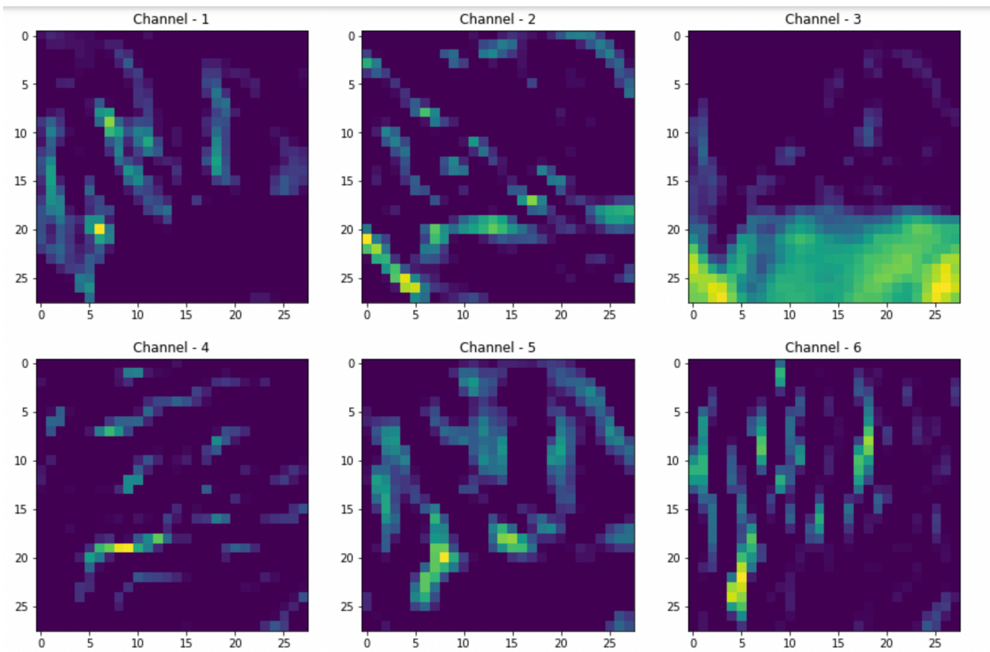
Predictions

vi) Can you visualize the output of the 1st layer of CNN using one image from the training set?

Ans: **Ouput of the first layer of CNN is as shown below. Output consists of 6 channels and each of the channels are of size 28x28. Each channels are plotted separately. "Net" class' conv1 attribute is utilized for plotting.**

```
outputs = net(images)
conv1_output = F.relu(net.conv1(images[0].unsqueeze(0)))
```

Code for generating the convolution output



Convolution and ReLu output