**CO543: Image Processing**

**Lab 8 – Fourier Transformation**

Ranage R.D.P.R. - E/19/310

The following functions are used throughout the lab to plot each figure to show the results.

import cv2

import numpy as np

import matplotlib.pyplot as plt

def show\_2\_images(image1, image2, title1, title2):

    fig, ax = plt.subplots(1, 2, figsize=(10, 5))

    ax[0].imshow(image1, cmap='gray')

    ax[0].set\_title(title1)

    ax[1].imshow( image2, cmap='gray')

    ax[1].set\_title(title2)

    plt.show()

def show\_3\_images(image1, image2, image3, title1, title2, title3):

    fig, ax = plt.subplots(1, 3, figsize=(15, 5))

    ax[0].imshow(image1, cmap='gray')

    ax[0].set\_title(title1)

    ax[1].imshow( image2, cmap='gray')

    ax[1].set\_title(title2)

    ax[2].imshow( image3, cmap='gray')

    ax[2].set\_title(title3)

    plt.show()

**1. Apply high pass laplacian filter on Car.jpg image.**

# Reading the image

img = plt.imread('car-2.jpg')

gray\_img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

def high\_pass\_laplacian\_filter(img):

    # Convert image to float32 for DFT processing

    img\_float32 = np.float32(img)

    # Compute 2D Fourier transform

    dft = cv2.dft(img\_float32, flags=cv2.DFT\_COMPLEX\_OUTPUT)

    dft\_shift = np.fft.fftshift(dft)

    # Create Laplacian high-pass filter mask

    rows, cols = img.shape

    crow, ccol = rows // 2, cols // 2

    mask = np.ones((rows, cols, 2), np.uint8)

    # Create a Laplacian filter in the frequency domain

    X = np.linspace(-ccol, ccol, cols)

    Y = np.linspace(-crow, crow, rows)

    X, Y = np.meshgrid(X, Y)

    laplacian\_mask = -4 \* (np.pi\*\*2) \* (X\*\*2 + Y\*\*2)

    laplacian\_mask[crow, ccol] = 1  # Avoiding division by zero

    laplacian\_mask = np.dstack((laplacian\_mask, laplacian\_mask))

    # Apply Laplacian high-pass filter

    fshift = dft\_shift \* laplacian\_mask

    # Inverse shift to bring back to original position

    f\_ishift = np.fft.ifftshift(fshift)

    # Inverse DFT to get the filtered image

    img\_back = cv2.idft(f\_ishift)

    img\_back = cv2.magnitude(img\_back[:,:,0], img\_back[:,:,1])

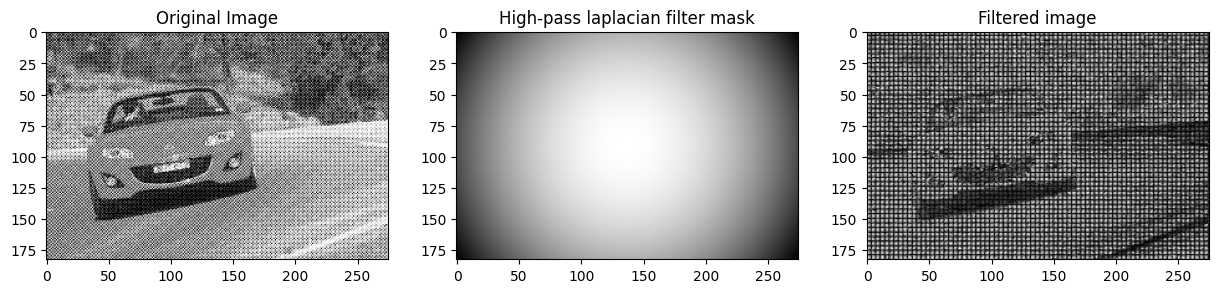
    # Normalize the filtered image to 0-255

    img\_back = cv2.normalize(img\_back, None, 0, 255, cv2.NORM\_MINMAX, dtype=cv2.CV\_8U)

    return img\_back, laplacian\_mask[:,:,0]

filtered\_image, mask = high\_pass\_laplacian\_filter(gray\_img)

show\_3\_images(gray\_img, mask, filtered\_image, 'Original Image', 'High-pass laplacian filter mask', 'Filtered image')

****

**2. Apply ideal high-pass filter on Car.jpg image for D0=100**

def ideal\_high\_pass\_filter(img, d0=100):

    img\_float32 = np.float32(img)

    dft = cv2.dft(img\_float32, flags=cv2.DFT\_COMPLEX\_OUTPUT)

    dft\_shift = np.fft.fftshift(dft)

    # Create ideal high-pass filter mask

    rows, cols = img.shape

    crow, ccol = rows // 2, cols // 2

    mask = np.ones((rows, cols, 2), np.uint8)

    # Create a circular mask with radius d0

    center = (ccol, crow)

    for i in range(rows):

        for j in range(cols):

            if (i - crow) \*\* 2 + (j - ccol) \*\* 2 <= d0 \*\* 2:

                mask[i, j] = 0

    fshift = dft\_shift \* mask

    # Inverse shift to bring back to original position

    f\_ishift = np.fft.ifftshift(fshift)

    # Inverse DFT to get the filtered image

    img\_back = cv2.idft(f\_ishift)

    img\_back = cv2.magnitude(img\_back[:,:,0], img\_back[:,:,1])

    # Normalize the filtered image to 0-255

    img\_back = cv2.normalize(img\_back, None, 0, 255, cv2.NORM\_MINMAX, dtype=cv2.CV\_8U)

    # Convert mask to uint8 for display

    mask\_img = np.uint8(mask[:,:,0] \* 255)

    return img\_back, mask\_img

filtered\_image, mask = ideal\_high\_pass\_filter(gray\_img, 100)

show\_3\_images(gray\_img, mask, filtered\_image, 'Original Image', 'Ideal high-pass filter mask', 'Filtered image')

**A black circle with white text

Description automatically generated**

**3. Apply ideal low-pass filter on Car.jpg image for D0=100**

def ideal\_low\_pass\_filter(img, d0=100):

    # Convert image to float32 for DFT processing

    img\_float32 = np.float32(img)

    # Compute 2D Fourier transform

    dft = cv2.dft(img\_float32, flags=cv2.DFT\_COMPLEX\_OUTPUT)

    dft\_shift = np.fft.fftshift(dft)

    # Create ideal low-pass filter mask

    rows, cols = img.shape

    crow, ccol = rows // 2, cols // 2

    mask = np.zeros((rows, cols, 2), np.uint8)

    # Create a circular mask with radius d0

    center = (ccol, crow)

    for i in range(rows):

        for j in range(cols):

            if (i - crow) \*\* 2 + (j - ccol) \*\* 2 <= d0 \*\* 2:

                mask[i, j] = 1

    # Apply mask in frequency domain

    fshift = dft\_shift \* mask

    # Inverse shift to bring back to original position

    f\_ishift = np.fft.ifftshift(fshift)

    # Inverse DFT to get the filtered image

    img\_back = cv2.idft(f\_ishift)

    img\_back = cv2.magnitude(img\_back[:,:,0], img\_back[:,:,1])

    # Normalize the filtered image to 0-255

    img\_back = cv2.normalize(img\_back, None, 0, 255, cv2.NORM\_MINMAX, dtype=cv2.CV\_8U)

    # Convert mask to uint8 for display

    mask\_img = np.uint8(mask[:,:,0] \* 255)

    return img\_back, mask\_img

filtered\_image, mask = ideal\_low\_pass\_filter(gray\_img, 100)

show\_3\_images(gray\_img, mask, filtered\_image, 'Original Image', 'Ideal low-pass filter mask', 'Filtered image')

**A white circle with black background

Description automatically generated**

**4. Apply FFT2, IFFT2, low-pass Gaussian filter, and high-pass laplacian filter on Car.jpg image.**

def low\_pass\_gaussian\_filter(img, sigma=10):

    # Convert image to float32 for DFT processing

    img\_float32 = np.float32(img)

    # Compute 2D Fourier transform

    dft = np.fft.fft2(img\_float32)

    dft\_shift = np.fft.fftshift(dft)

    # Create Gaussian low-pass filter mask

    rows, cols = img.shape

    crow, ccol = rows // 2, cols // 2

    x = np.linspace(-ccol, ccol, cols)

    y = np.linspace(-crow, crow, rows)

    X, Y = np.meshgrid(x, y)

    gaussian\_mask = np.exp(- (X\*\*2 + Y\*\*2) / (2 \* sigma\*\*2))

    # Apply Gaussian low-pass filter

    low\_pass\_filtered = dft\_shift \* gaussian\_mask

    # Inverse shift to bring back to original position

    f\_ishift = np.fft.ifftshift(low\_pass\_filtered)

    # Inverse DFT to get the filtered image

    img\_back = np.fft.ifft2(f\_ishift)

    img\_back = np.abs(img\_back)

    # Normalize the filtered image to 0-255

    img\_back = cv2.normalize(img\_back, None, 0, 255, cv2.NORM\_MINMAX, dtype=cv2.CV\_8U)

    return img\_back, gaussian\_mask

laplacian\_filtered\_image, mask = high\_pass\_laplacian\_filter(gray\_img)

gaussian\_filtered\_image, mask = low\_pass\_gaussian\_filter(gray\_img, 25)

show\_3\_images(laplacian\_filtered\_image, mask, gaussian\_filtered\_image, 'High-pass laplacian filtered image', 'Gaussian low-pass filter mask', 'Gaussian low-pass filtered image')

**A bright light on a black background

Description automatically generated**

**5. Apply the necessary filter and correct the noise in the image.**

def gaussian\_filter\_in\_frequency\_domain(img, sigma=25):

    img\_float32 = np.float32(img)

    # Compute 2D Fourier transform

    transformed\_img = np.fft.fft2(img\_float32)

    fftshift\_img = np.fft.fftshift(transformed\_img)

    # Get the height and width of the image

    height, width = img.shape

    # Create a Gaussian filter

    x = np.linspace(-width//2, width//2, width)

    y = np.linspace(-height//2, height//2, height)

    X, Y = np.meshgrid(x, y)

    gaussian\_filter = np.exp(-(X\*\*2 + Y\*\*2) / (2 \* sigma\*\*2))

    # Apply Gaussian filter in the frequency domain

    filtered\_fftshift\_img = fftshift\_img \* gaussian\_filter

    # Compute the inverse shift to bring back to original position

    filtered\_img = np.fft.ifftshift(filtered\_fftshift\_img)

    # Compute the inverse DFT to get the filtered image

    filtered\_img = np.fft.ifft2(filtered\_img)

    filtered\_img = np.abs(filtered\_img)

    # Normalize the filtered image to 0-255

    filtered\_img = cv2.normalize(filtered\_img, None, 0, 255, cv2.NORM\_MINMAX, dtype=cv2.CV\_8U)

    return filtered\_img, gaussian\_filter

filtered\_image, mask = gaussian\_filter\_in\_frequency\_domain(gray\_img, 25)

show\_2\_images(img, filtered\_image, 'Original Image', 'Filtered image')

**A close-up of a car

Description automatically generated**

**6. Apply the sobel operator (filter) on Car.jpg in the Fourier domain to detect edges.**

def sobel\_filter(img):

# Convert image to float32 for DFT processing

    img\_float32 = np.float32(img)

    # Compute 2D Fourier transform

    dft = np.fft.fft2(img\_float32)

    dft\_shift = np.fft.fftshift(dft)

    # Get dimensions of the image

    rows, cols = img.shape

    crow, ccol = rows // 2 , cols // 2

    # Create frequency domain grid

    u = np.fft.fftfreq(cols).reshape(1, -1)

    v = np.fft.fftfreq(rows).reshape(-1, 1)

    # Sobel filters in the frequency domain

    sobel\_x\_freq = 1j \* 2 \* np.pi \* u

    sobel\_y\_freq = 1j \* 2 \* np.pi \* v

    # Apply Sobel filters in the frequency domain

    dft\_shift\_sobel\_x = dft\_shift \* sobel\_x\_freq

    dft\_shift\_sobel\_y = dft\_shift \* sobel\_y\_freq

    # Inverse shift to bring back to original position

    sobel\_x\_img = np.fft.ifftshift(dft\_shift\_sobel\_x)

    sobel\_y\_img = np.fft.ifftshift(dft\_shift\_sobel\_y)

    # Inverse Fourier transform to obtain filtered images

    sobel\_x\_filtered = np.fft.ifft2(sobel\_x\_img)

    sobel\_y\_filtered = np.fft.ifft2(sobel\_y\_img)

    # Compute magnitude of gradients

    edge\_detected\_img = np.sqrt(np.abs(sobel\_x\_filtered)\*\*2 + np.abs(sobel\_y\_filtered)\*\*2)

    # Normalize the edge-detected image to 0-255

    edge\_detected\_img = cv2.normalize(np.abs(edge\_detected\_img), None, 0, 255, cv2.NORM\_MINMAX, dtype=cv2.CV\_8U)

    return edge\_detected\_img

sobel\_filtered\_image = sobel\_filter(gray\_img)

show\_2\_images(gray\_img, sobel\_filtered\_image, 'Original Image', 'Sobel Edge Detection')

**A close-up of a car

Description automatically generated**

**7. Discuss applying Butterworth and Chebyshev filters and compare the output image with the**

**Gaussian Filter image**

from scipy import fftpack

def butterworth\_filter(image, cutoff, n):

    fft\_image = fftpack.fft2(image)

    fft\_shifted = fftpack.fftshift(fft\_image)

    rows, cols = image.shape

    center\_row, center\_col = rows // 2, cols // 2

    # Create Butterworth Filter

    butterworth\_filter = np.zeros((rows, cols))

    for i in range(rows):

        for j in range(cols):

            dist = np.sqrt((i - center\_row)\*\*2 + (j - center\_col)\*\*2)

            butterworth\_filter[i, j] = 1 / (1 + (dist / cutoff)\*\*(2\*n))

    # Apply the filter

    filtered = fft\_shifted \* butterworth\_filter

    filtered = fftpack.ifftshift(filtered)

    filtered\_image = fftpack.ifft2(filtered).real

    return np.uint8(filtered\_image)

def chebyshev\_filter(image, cutoff, n, ripple\_db):

    fft\_image = fftpack.fft2(image)

    fft\_shifted = fftpack.fftshift(fft\_image)

    rows, cols = image.shape

    center\_row, center\_col = rows // 2, cols // 2

    # Create Chebyshev Filter

    chebyshev\_filter = np.zeros((rows, cols))

    for i in range(rows):

        for j in range(cols):

            dist = np.sqrt((i - center\_row)\*\*2 + (j - center\_col)\*\*2)

            chebyshev\_filter[i, j] = 1 / np.sqrt(1 + (dist / cutoff)\*\*(2\*n))

    # Apply the filter

    filtered = fft\_shifted \* chebyshev\_filter

    filtered = fftpack.ifftshift(filtered)

    filtered\_image = fftpack.ifft2(filtered).real

    return np.uint8(filtered\_image)

def gaussian\_filter(image, sigma=1.0):

    return cv2.GaussianBlur(image, (0, 0), sigma)

image = cv2.imread('car-2.jpg', cv2.IMREAD\_GRAYSCALE)

butterworth\_filtered = butterworth\_filter(image, 50, 2)

chebyshev\_filtered = chebyshev\_filter(image, 50, 2, 1)

gaussian\_filtered = gaussian\_filter(image, 1.2)

show\_3\_images(butterworth\_filtered, chebyshev\_filtered, gaussian\_filtered, 'Butterworth Filter', 'Chebyshev Filter', 'Gaussian Filter')

**A car on a road

Description automatically generated**

The Butterworth filter provides a trade-off between preserving sharp edges and smoothing, making it useful for applications that require a certain level of detail preservation.  
  
The Chebyshev filter is designed to achieve exact edge enhancement or detection by providing sharp filtering with controlled passband ripple.

The Gaussian Filter is used to generate a blurred or smoothed image, which is particularly useful for reducing noise and achieving a general-purpose smoothing effect.