

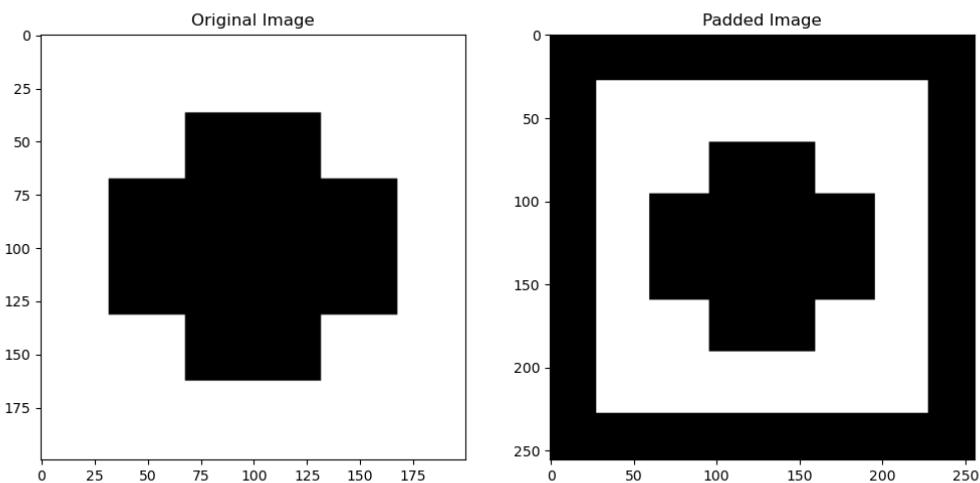
Digital Image Processing (261453) and Digital Image Analysis (261752)

Computer Assignment 2

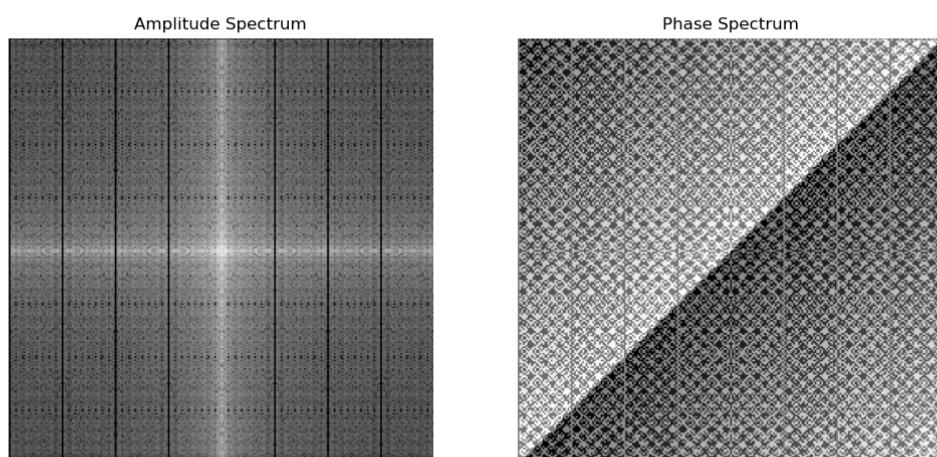
1. Properties of the Fourier Transform

1.1) ทำการแปลง Fourier Transform ของภาพ “Cross.pgm” (200×200) เนื่องจากในการใช้ FFT จำเป็นต้องให้ภาพมีขนาดที่อยู่ในรูปของ 2^n ดังนั้น อาจต้องทำการ Pad ก่อน และให้แสดงภาพผลลัพธ์ในรูปของ amplitude และ phase spectra

หลังจากการ Pad รูปจะได้ผลลัพธ์ดังนี้

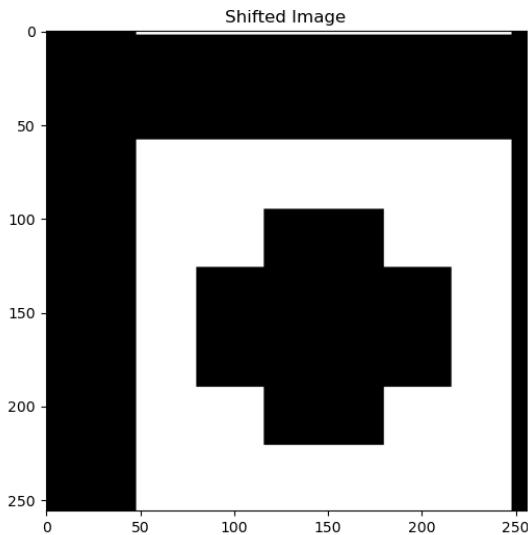


เราต้องทำการ Padding เพื่อเพิ่มประสิทธิภาพของการทำ Fourier Transform และจะได้ผลลัพธ์ในรูปของ amplitude และ phase spectra ดังนี้



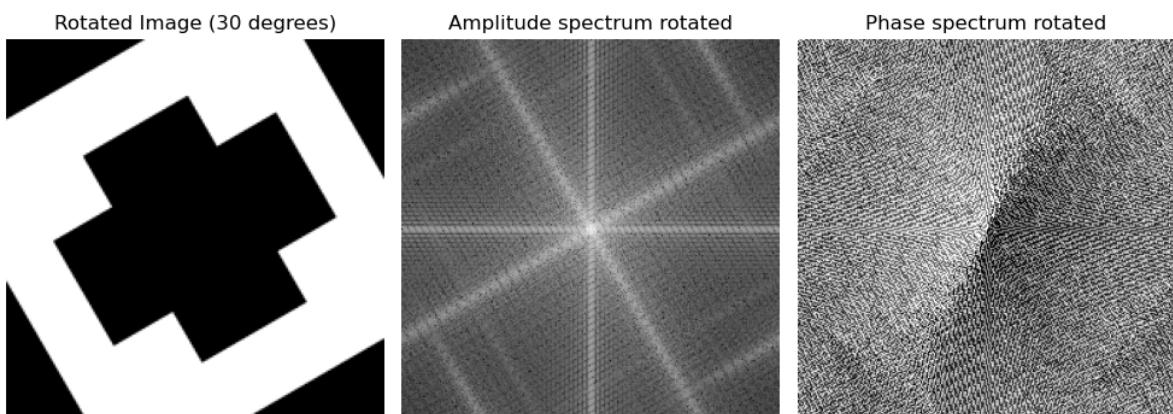
ชิ้น Amplitude Spectrum แสดงถึงขนาดของสัญญาณภาพในแต่ละความถี่ และ Phase Spectrum บ่งบอกถึงข้อมูลเฟสหรือมุมของสัญญาณในแต่ละความถี่

1.2) ให้ทำการคูณ phase spectrum ที่ได้ในข้อ 1.1 ด้วย complex number ค่าหนึ่ง เพื่อที่ว่าเมื่อทำการ inverse Fourier transform แล้ว ภาพผลลัพธ์ที่ได้ ย้ายด้วยจำนวนในแกน x และ y เป็น (20,30)



ทำการคูณ Phase Spectrum ที่ได้จากข้อ 1.1 ด้วยค่า complex number เพื่อสร้างการเลื่อนภาพ ในโดยเด่นความถี่ เมื่อทำการ inverse Fourier transform ภาพผลลัพธ์จะถูกย้ายไปตามแกน x และ y ตามที่กำหนด (20, 30) ซึ่งการเลื่อนนี้จะไม่เปลี่ยนแปลง Amplitude แต่เปลี่ยนแปลง Phase ของสัญญาณ

1.3) ทำการหมุนภาพ “Cross.pgm” ไป 30 องศา และแสดงผลของการแปลง Fourier ให้ทำการวิเคราะห์ว่า เกิดอะไรขึ้น

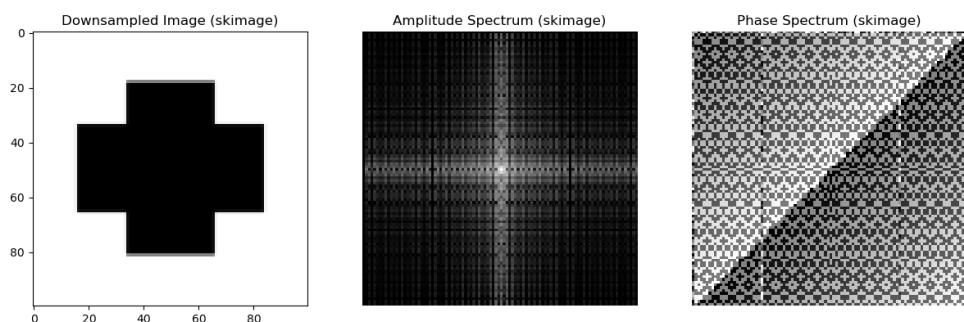


ภาพถูกหมุนไป 30 องศา และทำการแปลง Fourier ใหม่ การหมุนภาพเป็นการเปลี่ยนแปลงที่ส่งผลต่อทั้ง Amplitude และ Phase Spectrum การวิเคราะห์ผลการแปลง Fourier หลังจากหมุนภาพ ทำให้เห็นถึงการเปลี่ยนแปลงในโครงสร้างของภาพในโดยเด่นความถี่

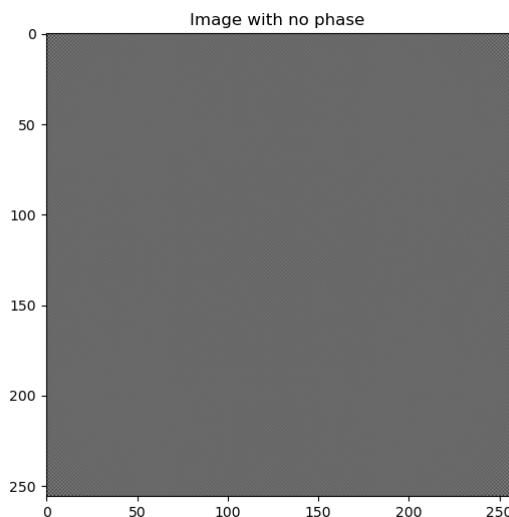
โดยที่ Amplitude Spectrum อาจไม่มีการเปลี่ยนแปลงที่เห็นได้ชัดเจน เนื่องจากมันไม่ได้ขึ้นอยู่กับทิศทางของสัญญาณ แต่ Phase Spectrum แสดงการเปลี่ยนแปลงอย่างชัดเจนเนื่องจากการหมุนภาพเปลี่ยนแปลงเฟสของสัญญาณ

1.4) ทำการ Down-sample “Cross.pgm” เพื่อให้รูปมีขนาด 100×100 หลังจากนั้นทำการแปลง Fourier Transform ให้แสดงภาพผลลัพธ์ในรูปของ amplitude และ phase spectra ให้ทำการวิเคราะห์ว่าเกิดอะไรขึ้น

การลดขนาดภาพ จะลดความละเอียดทางพื้นที่ของภาพ ซึ่งนำไปสู่การสูญเสียข้อมูลความถี่สูงเนื่องจากมีพิกเซลน้อยลง โดย Amplitude Spectrum จะแสดงช่วงความถี่ที่แอบลงเมื่อเทียบกับภาพต้นฉบับ เพราะมีการลดความละเอียดลง ส่วน Phase spectrum ยังคงความจัดเรียงทางพื้นที่ของเนื้อหาภาพที่ยังดูออกอยู่ และคล้ายกับต้นฉบับ แต่มีความซับซ้อนน้อยลงเนื่องจากขนาดภาพที่เล็กลง

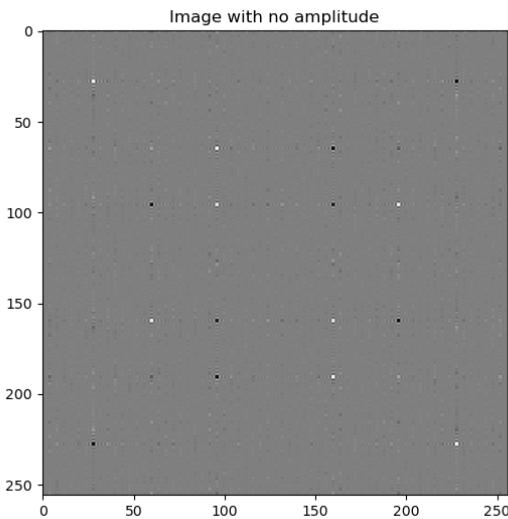


1.5) ใช้การแปลง inverse Fourier Transform ของผลลัพธ์ที่ได้ในข้อ 1.1 โดยที่
1.5.1) ไม่ใช้ข้อมูล phase



การลบข้อมูล Phase และใช้เพียง Amplitude Spectrum เพื่อทำการแปลงฟูเรียร์ย้อนกลับ จะเห็นว่าภาพนั้นสูญเสียโครงสร้างภาพต้นฉบับและดูไม่อกรกว่าภาพต้นฉบับนั้นเป็นอย่างไร

1.5.2) ไม่ใช้ข้อมูล amplitude

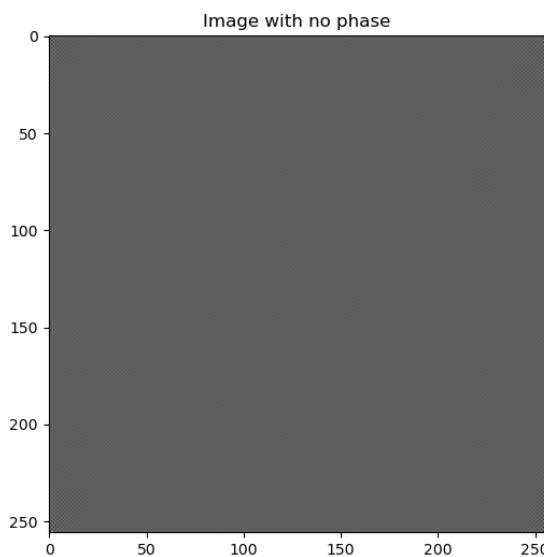


การไม่ใช้ข้อมูล Amplitude ในขณะที่เก็บ Phase ไว้ จะสามารถเห็นโครงสร้างของภาพได้บ้าง

ดังนั้นจะสรุปได้ว่า Phase มีความสำคัญต่อการรักษาโครงสร้างทางพื้นที่และคุณลักษณะบางอย่างไว้ในขณะที่ Amplitude มีผลต่อรายละเอียดความสว่างและความคมชัดของภาพ

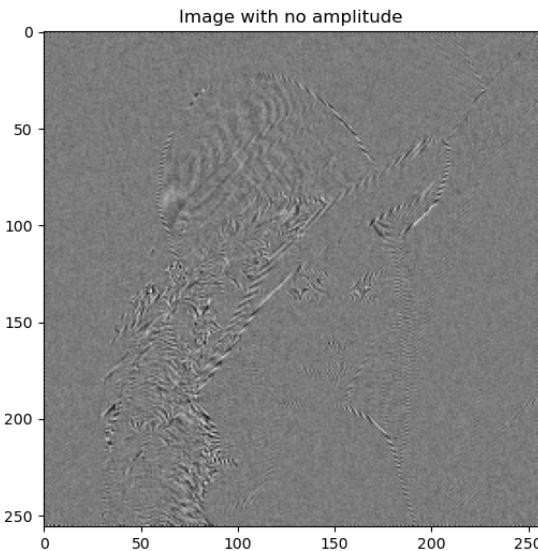
1.6) ให้ทำการทดลองเช่นเดียวกับข้อ 1.5 ด้วยภาพ “Lenna.pgm” (256x256)

1.6.1) ไม่ใช้ข้อมูล phase



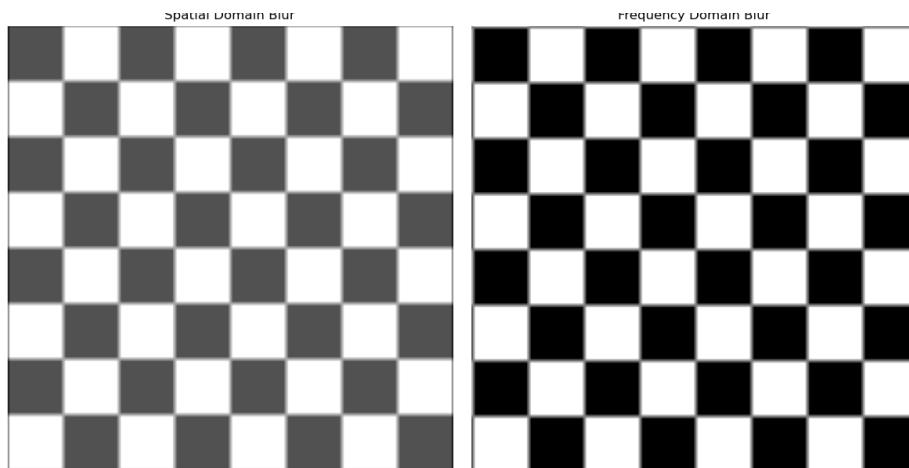
จะเห็นผลลัพธ์คล้ายกับรูป cross ก็คือหากไม่มีข้อมูล Phase ก็จะไม่สามารถรู้ได้เลยว่าภาพต้นฉบับนั้นเป็นอย่างไร

1.6.2) ไม่ใช้ข้อมูล amplitude



และหากไม่ใช้ข้อมูล Amplitude ก็ยังจะเห็นโครงสร้างบางอย่างของภาพต้นฉบับอยู่

- 1.7) (a) ทำ convolution ภาพ “Chess.pgm” (256x256) ด้วย mask หรือ kernel ขนาดเล็กอันหนึ่ง เพื่อทำการ Blur ภาพ และ (b) ให้ทำการ filter ใน frequency domain ด้วย Fourier transform ของ Kernel ที่ใช้ในข้อ (a) เพื่อทำการ Blur ภาพด้วย เปรียบเทียบผลที่ได้ทั้งสองแบบ ว่ามีความแตกต่างหรือหรือเหมือนกันอย่างไร



(a) Convolution ใน Spatial Domain

ในโดเมนเวลา การทำ Convolution ของภาพด้วย kernel หรือ mask ขนาดเล็กสำหรับการทำ Blur เป็นกระบวนการที่ส่งผลให้ภาพเบลอโดยการ "ผสม" ค่าพิกเซลในพื้นที่ใกล้เคียง Kernel ที่ใช้ในการทดลองนี้มีขนาด 3×3 พิกเซลและเป็นของเฉลี่ย (averaging kernel) ซึ่งแต่ละส่วนประกอบมีค่าเท่ากัน และมีผลรวมเป็น 1 เพื่อรักษาความสมดุลของความสว่างในภาพ และจะอธิบายวิธีการทำงานดังนี้

1. การพลิกและหมุน Kernel ก่อนทำการ Convolution
2. การเติม Padding ให้กับภาพด้วยค่าเป็นศูนย์เพื่อรักษาขนาดภาพเดิม

3. การคำนวณการ Convolution โดยการคูณ kernel กับพื้นที่ใกล้เคียงที่เกี่ยวข้องในภาพและรวมค่าเหล่านั้นเข้าด้วยกัน

(b) Filtering ใน Frequency Domain

ในโดเมนความถี่ การทำ Blur ถูกทำโดยการคูณ Fourier Transform ของภาพกับ Fourier Transform ของ kernel การนี้สามารถทำการ Convolution ได้โดยอัตโนมัติในโดเมนความถี่ เนื่องจาก การคูณในโดเมนความถี่คือการ Convolution ในโดเมนเวลา และจะอธิบายวิธีการทำงานดังนี้

1. การปรับขนาด (Pad) Kernel ให้มีขนาดเท่ากับภาพและการเลื่อนให้เป็นจุดศูนย์กลาง
2. การคำนวณ FFT ของภาพและของ kernel ที่ถูก Pad แล้ว
3. การคูณ FFT ของภาพกับ FFT ของ kernel และทำ inverse FFT เพื่อได้ภาพผลลัพธ์

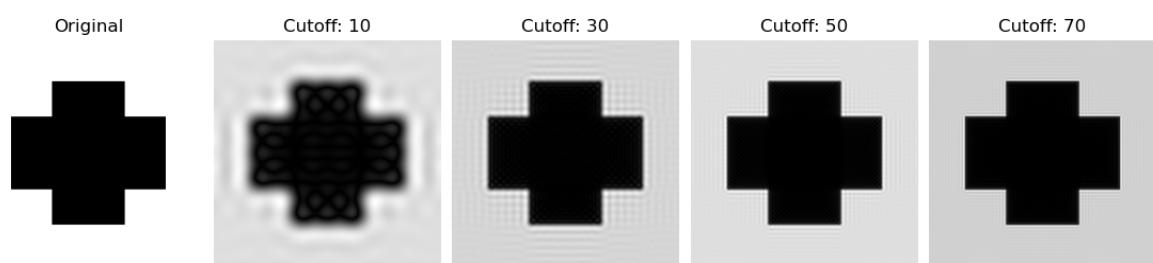
จากการทำ (a) Convolution ใน Spatial Domain และ (b) Filtering ใน Frequency Domain ในด้านการเบลอภาพ ซึ่งลดความคมชัดของรายละเอียดในภาพ ผลลัพธ์จะคล้ายคลึงกัน แต่ในด้านประสิทธิภาพการคำนวณนั้น การ Convolution ใน Frequency Domain มีประสิทธิภาพสูงกว่าเมื่อเทียบกับ Spatial Domain สำหรับ kernel ขนาดใหญ่หรือภาพขนาดใหญ่ เนื่องจาก FFT สามารถลดเวลาการคำนวณโดยรวม

2. Filter Design

2.1) ให้ใช้ ideal low-pass filter กับภาพ “Cross.pgm” โดยที่เปลี่ยน cutoff frequency และให้ศึกษา ringing effect ที่เกิดขึ้น หลังจากนั้นให้ทำการทดลองซ้ำด้วย Non-ideal filter อื่น

การทำ Ideal Low-Pass Filter และ Non-Ideal Filter (ในที่นี้คือ Gaussian Low-Pass Filter) ได้ผลลัพธ์ดังนี้

Ideal Low-Pass Filter



Ideal Low-Pass Filter จะผ่อนความถี่ต่ำไปยังผลลัพธ์และตัดความถี่สูงออก ซึ่งสามารถทำให้ภาพมีความเบลอได้ การทดลองนี้ใช้ cutoff frequencies ต่างๆ (10, 30, 50, 70) เพื่อสังเกตการเปลี่ยนแปลงของภาพหลังจากการ filter

ผลลัพธ์ที่สังเกตได้คือเมื่อ cutoff frequency ต่ำ ภาพที่ได้จะเบลอมากขึ้นเนื่องจากความถี่สูงถูกตัดออกมากกว่า และการเพิ่ม cutoff frequency ทำให้รายละเอียดของภาพคงที่มากขึ้น เนื่องจากความถี่สูงขึ้นจะยังสามารถผ่านไปได้ และจะเห็นว่ามี Ringing Effect ที่เกิดจากการ Filter ในบริเวณใกล้กับขอบของวัตถุในภาพ

Gaussian Low-Pass Filter



Gaussian Low-Pass Filter เป็นตัวอย่างของ Non-Ideal Filter ที่ใช้ฟังก์ชัน Gaussian ในการกำหนดค่าความถี่ที่จะผ่านหรือถูกตัดออก โดยการทดลองใช้ Sigma values ต่างๆ (10, 30, 50, 70)

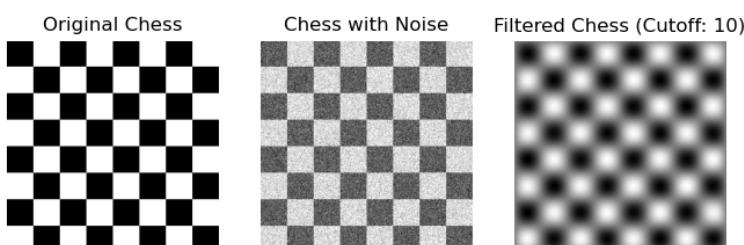
ผลลัพธ์ที่สังเกตได้คือ Gaussian Filter ให้ผลลัพธ์ที่มีความเรียบเนียนมากขึ้นเมื่อเทียบกับ Ideal Low-Pass Filter ที่เห็นได้ชัดคือการลดลงของ Ringing Effect โดยที่การเพิ่ม Sigma value นั้นจะทำให้ภาพเบลอมากขึ้น ด้วยการกระจายตัวของฟังก์ชัน Gaussian ทำให้การเปลี่ยนแปลงนั้นดูเป็นธรรมชาติมากกว่าการ cut off ของ Ideal Filter

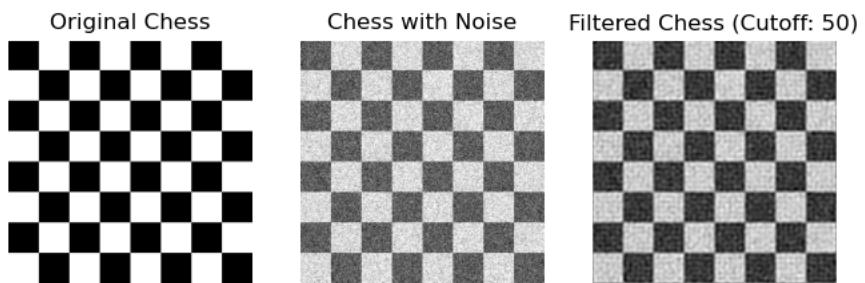
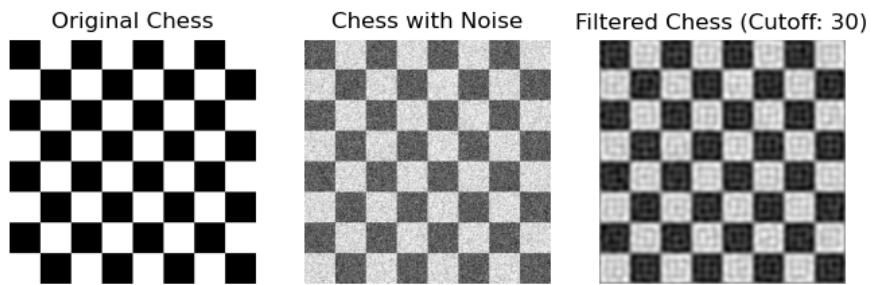
ดังนั้นจะสรุปได้ว่า การเลือกใช้ Filter แบบต่างๆ มีผลต่อผลลัพธ์ที่ได้จากการประมวลผลภาพ เพราะอาจได้ผลลัพธ์หลายแบบ เช่นมีการเกิด Ringing Effect ซึ่งภาพที่ได้ผ่าน Gaussian Filter จะช่วยลดเอฟเฟกต์ไม่พึงประสงค์เช่น Ringing Effect ได้ แสดงให้เห็นถึงความสำคัญในการเลือก Filter ที่เหมาะสมสำหรับงานประมวลผลภาพต่างๆ

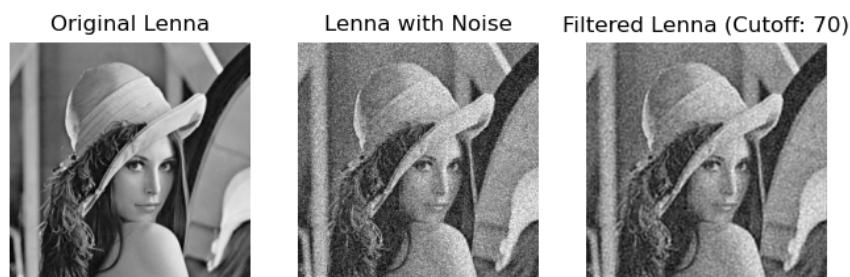
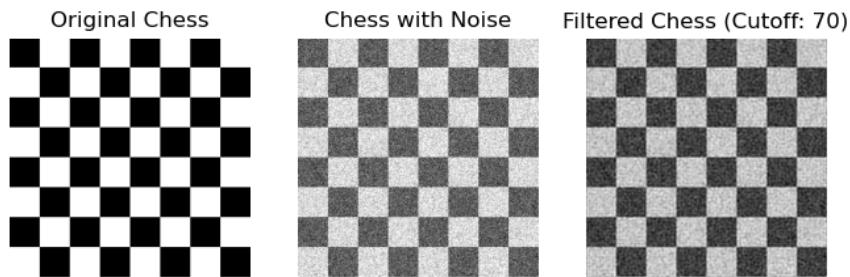
2.2) ให้ใช้ low-pass filter หลายแบบ โดยที่เปลี่ยนค่าตัวแปรต่างๆ รวมถึงขนาดของ filter ด้วย กับภาพ “Chess noise.pgm” และ “Lenna noise.pgm” เพื่อลด noise และให้เปรียบเทียบผลกับ Median filter ที่มีการเปลี่ยนขนาด และให้ใช้ RMS ในการคำนวณหาความแตกต่างระหว่างผลลัพธ์ที่ได้ กับภาพที่ไม่มี Noise “Chess.pgm” and “Lenna.pgm”

Low-Pass Filter

Low-Pass Filter ทำหน้าที่ลด Noise โดยการลดความถี่สูงออกจากภาพ







Chess RMS Error for cutoff 10:

35.125449184398995

Chess RMS Error for cutoff 30:

19.444904020996653

Chess RMS Error for cutoff 50:

16.850375683629924

Chess RMS Error for cutoff 70:

16.152676626941112

Lenna RMS Error for cutoff 10:

24.249084193698923

Lenna RMS Error for cutoff 30:

14.198145045587577

Lenna RMS Error for cutoff 50:

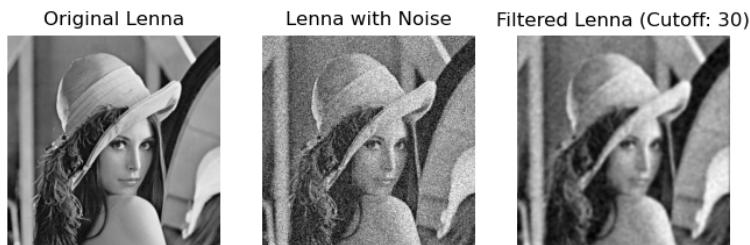
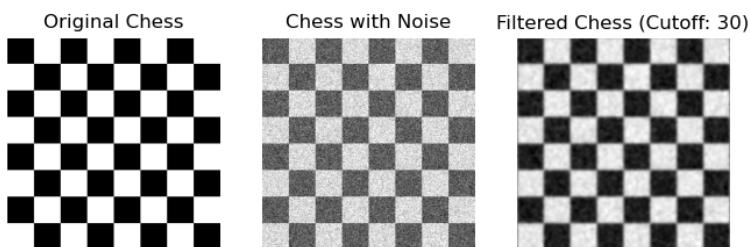
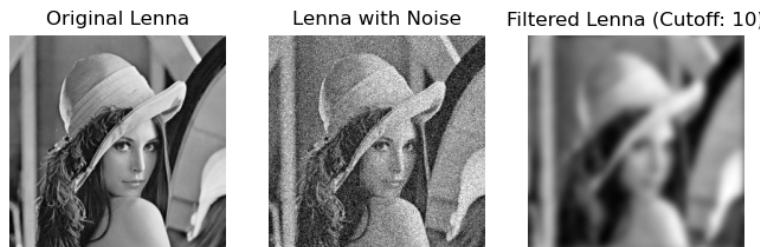
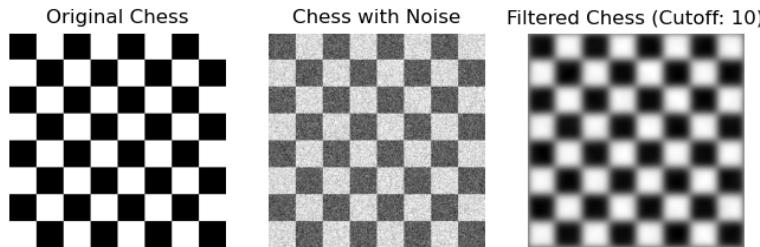
12.682265431776656

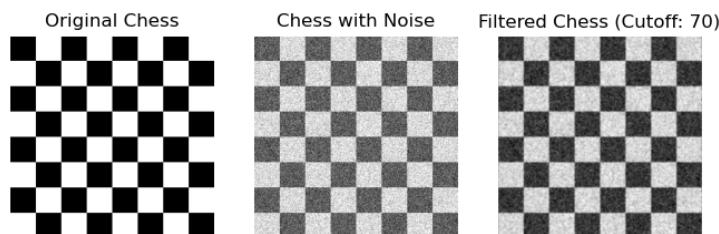
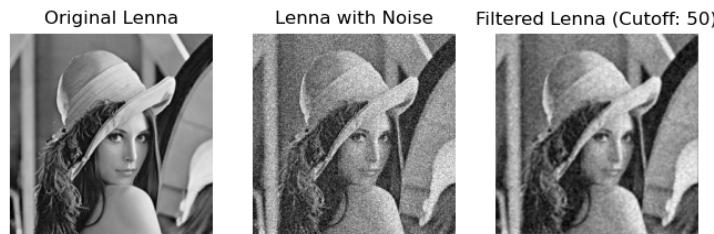
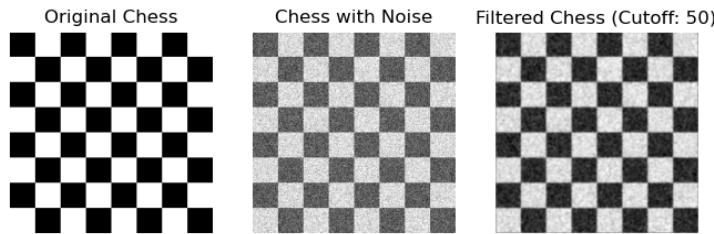
Lenna RMS Error for cutoff 70:

13.960062797669773

Gaussian Low-Pass Filter

Gaussian Filter ใช้การกระจายตัว Gaussian ในการตัดความถี่สูง โดยมีคุณสมบัติในการลด Noise และรักษาขอบของวัตถุในภาพได้ดี และจะเห็นว่า ค่า RMS Error จากการใช้ Gaussian Filter จะต่ำที่สุด เมื่อเทียบกับ Filters อื่นๆ





Chess RMS Error for Gaussian Filter (Cutoff: 10): 31.8577

Chess RMS Error for Gaussian Filter (Cutoff: 30): 18.3915

Chess RMS Error for Gaussian Filter (Cutoff: 50): 14.7348

Chess RMS Error for Gaussian Filter (Cutoff: 70): 13.8633

Lenna RMS Error for Gaussian Filter (Cutoff: 10): 22.4513

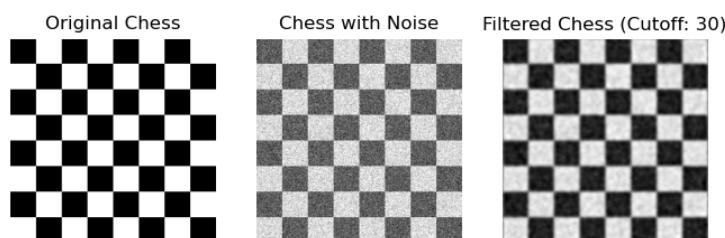
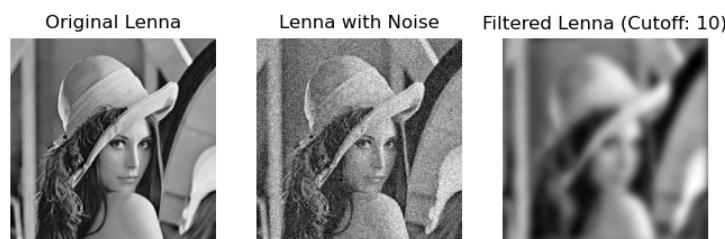
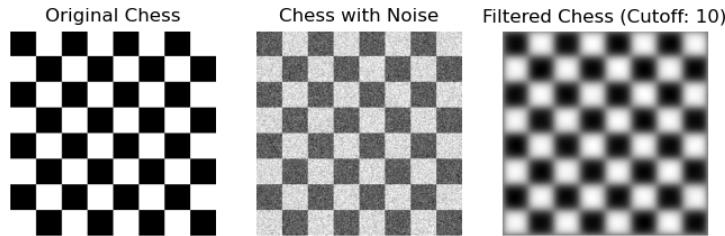
Lenna RMS Error for Gaussian Filter (Cutoff: 30): 12.8030

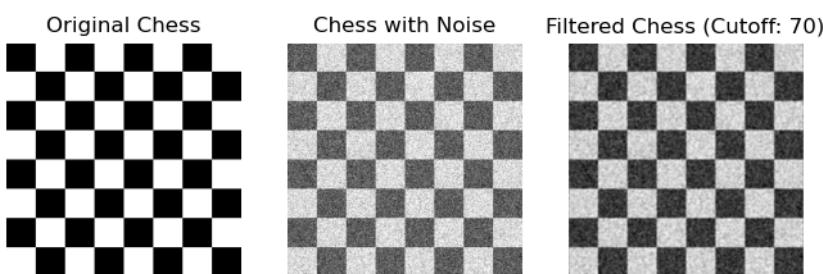
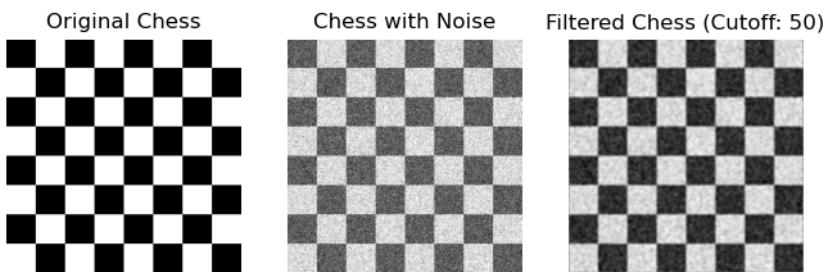
Lenna RMS Error for Gaussian Filter (Cutoff: 50): 11.0281

Lenna RMS Error for Gaussian Filter (Cutoff: 70): 11.925

Butterworth Low-Pass Filter

Butterworth Filter เป็น Low-Pass Filter ที่ให้การเปลี่ยนแปลงที่เรียบเนียนของความถี่จากพาร์สันไปยังพาร์สต์ด โดยไม่มีการสั่น (Ripple) ในห้องพาร์สต์ดและพาร์สต์ด เช่นเดียวกับ Ideal Filter และภาพที่ได้จากการใช้ Butterworth Filter แสดงให้เห็น Noise ที่ลดลงโดยรักษารายละเอียดของภาพในระดับหนึ่ง ซึ่งจะขึ้นอยู่กับค่า cutoff frequency และ order ของ filter





Chess RMS Error for cutoff 10: 31.1686
Chess RMS Error for cutoff 30: 18.0358
Chess RMS Error for cutoff 50: 14.6185
Chess RMS Error for cutoff 70: 14.0121

Lenna RMS Error for cutoff 10: 22.1625
Lenna RMS Error for cutoff 30: 12.6205
Lenna RMS Error for cutoff 50: 11.0807
Lenna RMS Error for cutoff 70: 12.2034

ภาคผนวก

Github : [Donteatpineappleonpizza/DIPH_W2_640612097](https://github.com/Donteatpineappleonpizza/DIPH_W2_640612097)

1.1-1.2

```

from skimage.io import imread
import numpy as np
import matplotlib.pyplot as plt
from numpy.fft import fft2, fftshift

def pad_to_next_power_of_2(image):
    """
    Pad the image to the next power of 2 in both dimensions.
    """
    m, n = image.shape
    M, N = 2**np.ceil(np.log2([m,n])).astype(int)
    padded = np.zeros((M,N), dtype=image.dtype)

    startx, starty = (M - m) // 2, (N - n) // 2 # Calculate starting points

    padded[startx:startx+m, starty:starty+n] = image # Place image in the
center
    return padded

# Load the image
image_path = 'Cross.pgm'
image = imread(image_path)

# Pad the image
padded_image = pad_to_next_power_of_2(image)

# Display the original and padded images
fig, ax = plt.subplots(1, 2, figsize=(12, 6))

# Original image
ax[0].imshow(image, cmap='gray')
ax[0].set_title('Original Image')
ax[0].axis('on')

# Padded image
ax[1].imshow(padded_image, cmap='gray')
ax[1].set_title('Padded Image')
ax[1].axis('on')

```

```

plt.show()

# Perform Fourier Transform on the padded image
fft_image = fft2(padded_image)
fft_shifted = fftshift(fft_image) # Shift the zero frequency component to
the center

# Calculate amplitude and phase spectra
amplitude_spectrum = np.abs(fft_shifted)
phase_spectrum = np.angle(fft_shifted)

# Display the amplitude and phase spectra
fig, ax = plt.subplots(1, 2, figsize=(12, 6))

# Amplitude spectrum
ax[0].imshow(np.log1p(amplitude_spectrum), cmap='gray')
ax[0].set_title('Amplitude Spectrum')
ax[0].axis('off')

# Phase spectrum
ax[1].imshow(phase_spectrum, cmap='gray')
ax[1].set_title('Phase Spectrum')
ax[1].axis('off')

plt.show()

```

```

from numpy.fft import ifft2, ifftshift

def shift_image_via_fft(fft_data, dx, dy):
    """
    Shift the image by dx and dy using phase manipulation in the frequency
    domain.

    Parameters:
    - fft_data: The FFT of the image.
    - dx: The shift in the x direction.
    - dy: The shift in the y direction.

    Returns:
    """

```

Parameters:

- `fft_data`: The FFT of the image.
- `dx`: The shift in the x direction.
- `dy`: The shift in the y direction.

Returns:

```

- The inverse FFT of the shifted image.

"""

(rows, cols) = fft_data.shape
M, N = np.meshgrid(np.arange(cols), np.arange(rows))

# Create the phase shift array
phase_shift = np.exp(-2j * np.pi * ((dx * M / cols) + (dy * N / rows)))

# Apply the phase shift
shifted_fft_data = fft_data * phase_shift

# Perform inverse FFT
shifted_image = ifft2(shifted_fft_data)

return shifted_image

# Shift the image by (20,30) in the x and y directions, respectively
shifted_image = shift_image_via_fft(fft_shifted, 20, 30)

# Plot the shifted image
plt.figure(figsize=(6, 6))
plt.imshow(np.abs(shifted_image), cmap='gray')
plt.title('Shifted Image')
plt.axis('on')
plt.show()

```

1.3

```

import cv2
from numpy.fft import fft2, ifft2, fftshift, ifftshift
import numpy as np
import matplotlib.pyplot as plt
# Correcting the padding issue
# Ensuring the image is no larger than 256x256 for padding, otherwise, it's
resized to fit.

# Read the PGM image directly using cv2.imread
image_path = 'Cross.pgm'
ima = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

```

```

# Rotate the image by 30 degrees
rows, cols = ima.shape
M = cv2.getRotationMatrix2D((cols / 2, rows / 2), 30, 1)
rotated_ima = cv2.warpAffine(ima, M, (cols, rows))

def pad_to_size(image, target_size=256):
    rows, cols = image.shape
    if rows > target_size or cols > target_size:
        # If the image is larger than the target size, resize it down
        image = cv2.resize(image, (target_size, target_size),
interpolation=cv2.INTER_AREA)
    else:
        # Calculate padding to center the image within the target size
        pad_vert = (target_size - rows) // 2
        pad_horiz = (target_size - cols) // 2
        image = np.pad(image, ((pad_vert, target_size - rows - pad_vert),
                             (pad_horiz, target_size - cols - pad_horiz))),
        mode='constant', constant_values=0)
    return image

# Pad the rotated image properly
pad_rotated_image = pad_to_size(rotated_ima)

# Compute the Fourier transform of the padded, rotated image
rotated_imagefft = fft2(np.float32(pad_rotated_image))

# Compute the amplitude spectrum and shift it to center
amplitude_spectrum_rotated = np.log1p(np.abs(rotated_imagefft)) # Use
log1p for better visualization
amplitude_spectrum_rotated_shifted = fftshift(amplitude_spectrum_rotated)

# Compute the phase spectrum and shift it to center
phase_spectrum_rotated = np.angle(rotated_imagefft)
phase_spectrum_rotated_shifted = fftshift(phase_spectrum_rotated)

# Plotting the original, rotated image alongside its amplitude and phase
spectrum
plt.figure(figsize=(10, 5))

```

```
# The rotated image
plt.subplot(1, 3, 1)
plt.imshow(rotated_ima, cmap='gray')
plt.title('Rotated Image (30 degrees)')
plt.axis('off')

# Amplitude spectrum of the rotated image
plt.subplot(1, 3, 2)
plt.imshow(amplitude_spectrum_rotated_shifted, cmap='gray')
plt.title('Amplitude spectrum rotated')
plt.axis('off')

# Phase spectrum of the rotated image
plt.subplot(1, 3, 3)
plt.imshow(phase_spectrum_rotated_shifted, cmap='gray')
plt.title('Phase spectrum rotated')
plt.axis('off')

plt.tight_layout()
plt.show()
```

1.4

```
from skimage.io import imread
from skimage.transform import resize
import numpy as np
import matplotlib.pyplot as plt

# Load the image using skimage.io.imread
image_path = 'Cross.pgm'
original_image = imread(image_path)

# Downsample the image to 100x100 using skimage.transform.resize
# Note: resize function needs the scale in (rows, cols) which matches
# (height, width)
downsampled_image_sk = resize(original_image, (100, 100), mode='reflect',
anti_aliasing=True)

# Compute the 2D Fourier Transform of the downsampled image
ft_image_sk = np.fft.fft2(downscaled_image_sk)
ft_image_shifted_sk = np.fft.fftshift(ft_image_sk)
```

```
# Compute amplitude and phase spectra
amplitude_spectrum_sk = np.abs(ft_image_shifted_sk)
phase_spectrum_sk = np.angle(ft_image_shifted_sk)

# Plotting the results
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Original downsampled image using skimage
axes[0].imshow(downscaled_image_sk, cmap='gray')
axes[0].set_title('Downsampled Image (skimage)')
axes[0].axis('on')

# Amplitude Spectrum
axes[1].imshow(np.log1p(amplitude_spectrum_sk), cmap='gray')
axes[1].set_title('Amplitude Spectrum (skimage)')
axes[1].axis('off')

# Phase Spectrum
axes[2].imshow(phase_spectrum_sk, cmap='gray')
axes[2].set_title('Phase Spectrum (skimage)')
axes[2].axis('off')

# Show the plots
plt.show()
```

1.5

```
import cv2
from numpy.fft import fft2, ifft2, fftshift, ifftshift
import numpy as np
import matplotlib.pyplot as plt
# Corrected code snippet with cv2.imread used to read the PGM image for
both operations

# Read the PGM image directly using cv2.imread
image_path = 'Cross.pgm'
ima = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# Pad the image to 256x256 (closest 2^n)
pad_size = (256 - ima.shape[0]) // 2
```

```
padimage = np.pad(ima, ((pad_size, pad_size), (pad_size, pad_size)), mode='constant', constant_values=0)

# Compute the Fourier transform
imagefft = fft2(np.float32(padimage))

# Set the amplitude data to one for the phase-only image
shiftedfft_one_amp = np.exp(1j * np.angle(imagefft))

# Inverse shift the Fourier transform for the phase-only image
shift_idft_one_amp = ifftshift(shiftedfft_one_amp)

# Inverse Fourier transform for the phase-only image
image_one_amp = ifft2(shift_idft_one_amp)

# Take the real part of the image for the phase-only image
image_one_amp = np.real(image_one_amp)

# Display the phase-only result
plt.figure(figsize=(6, 6))
plt.imshow(image_one_amp, cmap='gray')
plt.title('Image with no amplitude')
plt.axis('on')
plt.show()

# Set the phase to zero for the amplitude-only image
shiftedfft_zero_phase = np.abs(imagefft)

# Inverse shift the Fourier transform for the amplitude-only image
shift_idft_zero_phase = ifftshift(shiftedfft_zero_phase)

# Inverse Fourier transform for the amplitude-only image
image_zero_phase = ifft2(shift_idft_zero_phase)

# Take the real part of the image for the amplitude-only image
image_zero_phase = np.real(image_zero_phase)

# Display the amplitude-only result
plt.figure(figsize=(6, 6))
plt.imshow(image_zero_phase, cmap='gray')
plt.title('Image with no phase')
```

```
plt.axis('on')
plt.show()
```

1.6

```
import cv2
from numpy.fft import fft2, ifft2, fftshift, ifftshift
import numpy as np
import matplotlib.pyplot as plt
# Corrected code snippet with cv2.imread used to read the PGM image for
both operations

# Read the PGM image directly using cv2.imread
image_path = 'Lenna.pgm'
ima = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# Pad the image to 256x256 (closest 2^n)
pad_size = (256 - ima.shape[0]) // 2
padimage = np.pad(ima, ((pad_size, pad_size), (pad_size, pad_size)), mode='constant', constant_values=0)

# Compute the Fourier transform
imagefft = fft2(np.float32(padimage))

# Set the amplitude data to one for the phase-only image
shiftedfft_one_amp = np.exp(1j * np.angle(imagefft))

# Inverse shift the Fourier transform for the phase-only image
shift_idft_one_amp = ifftshift(shiftedfft_one_amp)

# Inverse Fourier transform for the phase-only image
image_one_amp = ifft2(shift_idft_one_amp)

# Take the real part of the image for the phase-only image
image_one_amp = np.real(image_one_amp)

# Display the phase-only result
plt.figure(figsize=(6, 6))
plt.imshow(image_one_amp, cmap='gray')
plt.title('Image with no amplitude')
plt.axis('on')
```

```

plt.show()

# Set the phase to zero for the amplitude-only image
shiftedfft_zero_phase = np.abs(imagefft)

# Inverse shift the Fourier transform for the amplitude-only image
shift_idft_zero_phase = ifftshift(shiftedfft_zero_phase)

# Inverse Fourier transform for the amplitude-only image
image_zero_phase = ifft2(shift_idft_zero_phase)

# Take the real part of the image for the amplitude-only image
image_zero_phase = np.real(image_zero_phase)

# Display the amplitude-only result
plt.figure(figsize=(6, 6))
plt.imshow(image_zero_phase, cmap='gray')
plt.title('Image with no phase')
plt.axis('on')
plt.show()

```

1.7

```

import numpy as np
import matplotlib.pyplot as plt
from numpy.fft import fft2, ifft2, fftshift
import cv2

# Define a function to perform convolution in the spatial domain
def spatial_domain_convolution(image, kernel):
    # Kernel needs to be flipped for convolution
    kernel_flipped = np.flipud(np.fliplr(kernel))
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2

    # Pad the image with zeros on all sides
    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width,
    pad_width)), mode='constant')

```

```

# Initialize the output array
output = np.zeros_like(image)

# Perform the convolution using the flipped kernel
for i in range(image_height):
    for j in range(image_width):
        # Extract the current region of interest
        region = padded_image[i:i+kernel_height, j:j+kernel_width]
        output[i, j] = np.sum(region * kernel_flipped)

return output

# Define a function to perform convolution in the frequency domain
def frequency_domain_convolution(image, kernel):
    # Get the size of the image and kernel
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape

    # Pad the kernel to be the same size as the image
    padded_kernel = np.zeros((image_height, image_width))
    padded_kernel[:kernel_height, :kernel_width] = kernel
    # Shift the kernel to the center
    padded_kernel = fftshift(padded_kernel)

    # Compute the FFT of the image and the padded kernel
    image_fft = fft2(image)
    kernel_fft = fft2(padded_kernel)

    # Perform element-wise multiplication
    convolved_fft = image_fft * kernel_fft

    # Compute the inverse FFT to get the convolved image
    convolved_image = np.real(ifft2(convolved_fft))

return convolved_image

# Define a 3x3 averaging kernel
kernel = np.ones((3, 3), np.float32) / 9.0

# Read the chess image
chess_image_path = 'Chess.pgm'

```

```

chess_image = cv2.imread(chess_image_path, cv2.IMREAD_GRAYSCALE)

# Perform spatial domain convolution
spatial_blurred = spatial_domain_convolution(chess_image, kernel)

# Perform frequency domain convolution
frequency_blurred = frequency_domain_convolution(chess_image, kernel)

# Normalize the output images
spatial_blurred_normalized = np.clip(spatial_blurred, 0,
255).astype(np.uint8)
frequency_blurred_normalized = np.clip(frequency_blurred, 0,
255).astype(np.uint8)

# Plot the results
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
axes[0].imshow(spatial_blurred_normalized, cmap='gray')
axes[0].set_title('Spatial Domain Blur')
axes[0].axis('off')

axes[1].imshow(frequency_blurred_normalized, cmap='gray')
axes[1].set_title('Frequency Domain Blur')
axes[1].axis('off')

plt.tight_layout()
plt.show()

```

2.1

```

from matplotlib import image
from numpy.fft import fft2, ifft2, fftshift, ifftshift
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread

# Load the image
image_path = 'Cross.pgm'
image = imread(image_path)

def apply_ideal_low_pass_filter(image, cutoff_frequency):
    # Perform FFT

```

```

f_transform = fft2(image)
f_transform_shifted = fftshift(f_transform)

# Create a low-pass filter mask with the cutoff frequency
rows, cols = image.shape
center_row, center_col = rows // 2, cols // 2
x, y = np.ogrid[:rows, :cols]
mask = np.sqrt((x - center_row)**2 + (y - center_col)**2) <=
cutoff_frequency
filtered = f_transform_shifted * mask

# Inverse FFT
f_ishift = ifftshift(filtered)
img_back = ifft2(f_ishift)
img_back = np.abs(img_back)

return img_back

# Convert image to grayscale numpy array if not already
image_np = np.array(image)

# Apply the ideal low-pass filter with different cutoff frequencies
cutoff_frequencies = [10, 30, 50, 70]
fig, axs = plt.subplots(1, len(cutoff_frequencies) + 1, figsize=(10, 5))

# Original image
axs[0].imshow(image_np, cmap='gray')
axs[0].set_title('Original')
axs[0].axis('off')

# Filtered images
for i, cutoff in enumerate(cutoff_frequencies):
    filtered_image = apply_ideal_low_pass_filter(image_np, cutoff)
    axs[i+1].imshow(filtered_image, cmap='gray')
    axs[i+1].set_title(f'Cutoff: {cutoff}')
    axs[i+1].axis('off')

plt.tight_layout()
plt.show()

def apply_gaussian_low_pass_filter(image, sigma):

```

```

# Perform FFT
f_transform = fft2(image)
f_transform_shifted = fftshift(f_transform)

# Create a Gaussian low-pass filter
rows, cols = image.shape
center_row, center_col = rows // 2, cols // 2
x, y = np.ogrid[:rows, :cols]
distance = np.sqrt((x - center_row)**2 + (y - center_col)**2)
gaussian_mask = np.exp(-(distance**2) / (2*(sigma**2)))

# Apply filter
filtered = f_transform_shifted * gaussian_mask

# Inverse FFT
f_ishift = ifftshift(filtered)
img_back = ifft2(f_ishift)
img_back = np.abs(img_back)

return img_back

# Sigma values for Gaussian low-pass filter
sigma_values = [10, 30, 50, 70]
fig, axs = plt.subplots(1, len(sigma_values) + 1, figsize=(10, 5))

# Original image
axs[0].imshow(image_np, cmap='gray')
axs[0].set_title('Original')
axs[0].axis('off')

# Filtered images with Gaussian filter
for i, sigma in enumerate(sigma_values):
    filtered_image = apply_gaussian_low_pass_filter(image_np, sigma)
    axs[i+1].imshow(filtered_image, cmap='gray')
    axs[i+1].set_title(f'Sigma: {sigma}')
    axs[i+1].axis('off')

plt.tight_layout()
plt.show()

```

2.2

```

import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread
from numpy.fft import fft2, ifft2, fftshift, ifftshift

def apply_low_pass_filter(image, cutoff_frequency):
    # Fourier Transform
    f_transform = fft2(image)
    f_transform_shifted = fftshift(f_transform)

    # Create a low-pass filter mask
    rows, cols = image.shape
    center_row, center_col = rows // 2, cols // 2
    X, Y = np.ogrid[:rows, :cols]
    mask = np.sqrt((X - center_row)**2 + (Y - center_col)**2) <=
cutoff_frequency
    filtered = f_transform_shifted * mask

    # Inverse Fourier Transform
    f_ishift = ifftshift(filtered)
    img_back = ifft2(f_ishift)
    img_filtered = np.abs(img_back)

    return img_filtered

def calculate_rms_error(original, filtered):
    return np.sqrt(np.mean((original - filtered) ** 2))

# Load the images
chess_noise = imread('Chess_noise.pgm')
lenna_noise = imread('Lenna_noise.pgm')
chess = imread('Chess.pgm')
lenna = imread('Lenna.pgm')

cutoff_frequencies = [10, 30, 50, 70]

# Process and calculate RMS for each cutoff frequency
for cutoff in cutoff_frequencies:
    chess_filtered = apply_low_pass_filter(chess_noise, cutoff)
    lenna_filtered = apply_low_pass_filter(lenna_noise, cutoff)

```

```

chess_rms = calculate_rms_error(chess, chess_filtered)
lenna_rms = calculate_rms_error(lenna, lenna_filtered)

print(f'Chess RMS Error for cutoff {cutoff}: {chess_rms}')
print(f'Lenna RMS Error for cutoff {cutoff}: {lenna_rms}')

# Adjustments for better inline display
plt.rcParams['figure.figsize'] = [8, 8] # or another size that fits your
screen

# Process and calculate RMS for each cutoff frequency and visualize
for cutoff in cutoff_frequencies:
    chess_filtered = apply_low_pass_filter(chess_noise, cutoff)
    lenna_filtered = apply_low_pass_filter(lenna_noise, cutoff)

    chess_rms = calculate_rms_error(chess, chess_filtered)
    lenna_rms = calculate_rms_error(lenna, lenna_filtered)

    # Print RMS error
    print(f'Chess RMS Error for cutoff {cutoff}: {chess_rms:.4f}')
    print(f'Lenna RMS Error for cutoff {cutoff}: {lenna_rms:.4f}')

# Visualization
fig, axs = plt.subplots(2, 3)

axs[0, 0].imshow(chess, cmap='gray')
axs[0, 0].set_title('Original Chess')
axs[0, 0].axis('off')

axs[0, 1].imshow(chess_noise, cmap='gray')
axs[0, 1].set_title('Chess with Noise')
axs[0, 1].axis('off')

axs[0, 2].imshow(chess_filtered, cmap='gray')
axs[0, 2].set_title(f'Filtered Chess (Cutoff: {cutoff})')
axs[0, 2].axis('off')

axs[1, 0].imshow(lenna, cmap='gray')
axs[1, 0].set_title('Original Lenna')
axs[1, 0].axis('off')

```

```

axs[1, 1].imshow(lenna_noise, cmap='gray')
axs[1, 1].set_title('Lenna with Noise')
axs[1, 1].axis('off')

axs[1, 2].imshow(lenna_filtered, cmap='gray')
axs[1, 2].set_title(f'Filtered Lenna (Cutoff: {cutoff})')
axs[1, 2].axis('off')

plt.show()

```

2.2_Gaussian

```

import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread
from numpy.fft import fft2, ifft2, fftshift, ifftshift

def apply_gaussian_filter(image, cutoff_frequency):
    # Fourier Transform
    f_transform = fft2(image)
    f_transform_shifted = fftshift(f_transform)

    rows, cols = image.shape
    center_row, center_col = rows // 2, cols // 2

    # Create a Gaussian filter mask
    X, Y = np.ogrid[:rows, :cols]
    distance = np.sqrt((X - center_row)**2 + (Y - center_col)**2)
    sigma = cutoff_frequency / np.sqrt(2*np.log(2))  # Convert cutoff
frequency to sigma
    gaussian_mask = np.exp(-distance**2 / (2*sigma**2))

    # Apply the Gaussian mask to the shifted Fourier spectrum
    filtered = f_transform_shifted * gaussian_mask

    # Inverse Fourier Transform
    f_ishift = ifftshift(filtered)
    img_back = ifft2(f_ishift)
    img_filtered = np.abs(img_back)

```

```

    return img_filtered

def calculate_rms_error(original, filtered):
    return np.sqrt(np.mean((original - filtered) ** 2))

# Load the images (replace 'path_to_your_image' with the actual paths)
chess_noise = imread('Chess_noise.pgm')
lenna_noise = imread('Lenna_noise.pgm')
chess = imread('Chess.pgm')
lenna = imread('Lenna.pgm')

cutoff_frequencies = [10, 30, 50, 70]

# Adjustments for better inline display
plt.rcParams['figure.figsize'] = [8, 8] # Adjust based on your display
needs

# Process, calculate RMS for each cutoff frequency, and visualize
for cutoff in cutoff_frequencies:
    chess_filtered = apply_gaussian_filter(chess_noise, cutoff)
    lenna_filtered = apply_gaussian_filter(lenna_noise, cutoff)

    chess_rms = calculate_rms_error(chess, chess_filtered)
    lenna_rms = calculate_rms_error(lenna, lenna_filtered)

    # Print RMS error
    print(f'Chess RMS Error for Gaussian Filter (Cutoff: {cutoff}):'
{chess_rms:.4f}')
    print(f'Lenna RMS Error for Gaussian Filter (Cutoff: {cutoff}):'
{lenna_rms:.4f}')

    # Visualization
    fig, axs = plt.subplots(2, 3)

    axs[0, 0].imshow(chess, cmap='gray')
    axs[0, 0].set_title('Original Chess')
    axs[0, 0].axis('off')

    axs[0, 1].imshow(chess_noise, cmap='gray')
    axs[0, 1].set_title('Chess with Noise')

```

```

axs[0, 1].axis('off')

axs[0, 2].imshow(chess_filtered, cmap='gray')
axs[0, 2].set_title(f'Filtered Chess (Cutoff: {cutoff})')
axs[0, 2].axis('off')

axs[1, 0].imshow(lenna, cmap='gray')
axs[1, 0].set_title('Original Lenna')
axs[1, 0].axis('off')

axs[1, 1].imshow(lenna_noise, cmap='gray')
axs[1, 1].set_title('Lenna with Noise')
axs[1, 1].axis('off')

axs[1, 2].imshow(lenna_filtered, cmap='gray')
axs[1, 2].set_title(f'Filtered Lenna (Cutoff: {cutoff})')
axs[1, 2].axis('off')

plt.show()

```

2.2 Butterworth

```

import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread
from numpy.fft import fft2, ifft2, fftshift, ifftshift

def apply_butterworth_filter(image, cutoff_frequency, order=2):
    # Fourier Transform
    f_transform = fft2(image)
    f_transform_shifted = fftshift(f_transform)

    rows, cols = image.shape
    center_row, center_col = rows // 2, cols // 2

    # Create a Butterworth filter mask
    X, Y = np.ogrid[:rows, :cols]
    distance = np.sqrt((X - center_row)**2 + (Y - center_col)**2)
    butterworth_mask = 1 / (1 + (distance / cutoff_frequency)**(2*order))

    # Apply the Butterworth mask to the shifted Fourier spectrum

```

```

filtered = f_transform_shifted * butterworth_mask

# Inverse Fourier Transform
f_ishift = ifftshift(filtered)
img_back = ifft2(f_ishift)
img_filtered = np.abs(img_back)

return img_filtered


def calculate_rms_error(original, filtered):
    return np.sqrt(np.mean((original - filtered) ** 2))

def calculate_rms_error(original, filtered):
    return np.sqrt(np.mean((original - filtered) ** 2))

# Load the images
chess_noise = imread('Chess_noise.pgm')
lenna_noise = imread('Lenna_noise.pgm')
chess = imread('Chess.pgm')
lenna = imread('Lenna.pgm')

cutoff_frequencies = [10, 30, 50, 70]

# Process and calculate RMS for each cutoff frequency
for cutoff in cutoff_frequencies:
    chess_filtered = apply_butterworth_filter(chess_noise, cutoff)
    lenna_filtered = apply_butterworth_filter(lenna_noise, cutoff)

    chess_rms = calculate_rms_error(chess, chess_filtered)
    lenna_rms = calculate_rms_error(lenna, lenna_filtered)

    print(f'Chess RMS Error for cutoff {cutoff}: {chess_rms}')
    print(f'Lenna RMS Error for cutoff {cutoff}: {lenna_rms}')

# Adjustments for better inline display
plt.rcParams['figure.figsize'] = [8, 8] # or another size that fits your screen

# Process and calculate RMS for each cutoff frequency and visualize
for cutoff in cutoff_frequencies:

```

```

chess_filtered = apply_butterworth_filter(chess_noise, cutoff)
lenna_filtered = apply_butterworth_filter(lenna_noise, cutoff)

chess_rms = calculate_rms_error(chess, chess_filtered)
lenna_rms = calculate_rms_error(lenna, lenna_filtered)

# Print RMS error
print(f'Chess RMS Error for cutoff {cutoff}: {chess_rms:.4f}')
print(f'Lenna RMS Error for cutoff {cutoff}: {lenna_rms:.4f}')

# Visualization
fig, axs = plt.subplots(2, 3)

axs[0, 0].imshow(chess, cmap='gray')
axs[0, 0].set_title('Original Chess')
axs[0, 0].axis('off')

axs[0, 1].imshow(chess_noise, cmap='gray')
axs[0, 1].set_title('Chess with Noise')
axs[0, 1].axis('off')

axs[0, 2].imshow(chess_filtered, cmap='gray')
axs[0, 2].set_title(f'Filtered Chess (Cutoff: {cutoff})')
axs[0, 2].axis('off')

axs[1, 0].imshow(lenna, cmap='gray')
axs[1, 0].set_title('Original Lenna')
axs[1, 0].axis('off')

axs[1, 1].imshow(lenna_noise, cmap='gray')
axs[1, 1].set_title('Lenna with Noise')
axs[1, 1].axis('off')

axs[1, 2].imshow(lenna_filtered, cmap='gray')
axs[1, 2].set_title(f'Filtered Lenna (Cutoff: {cutoff})')
axs[1, 2].axis('off')

plt.show()

```