National University of Computer & Emerging Sciences, Karachi
Artificial Intelligence-School of Computing
Fall 2024, Lab Manual - 03

Course Code (AI4002)	**Course: Computer Vision Lab**
Instructor(s):	**Sohail Ahmed**

Objectives:

1. Understand the concept of Linear & non-linear filtering and its applications in image processing.
2. Explore 1-D and 2-D sampling techniques and their significance in image representation.
3. Learn about Fourier transformations in 2-D and their role in image analysis.

1. Linear Filtering

Linear filtering is a fundamental concept in digital image processing that involves modifying an image by applying a linear operation called a filter or kernel. These filters are used to enhance or suppress certain features in an image, such as noise reduction, edge detection, and image sharpening. Linear filtering works by convolving the image with the filter kernel, which involves sliding the kernel over the image and computing a weighted sum of pixel values at each position.

Some key aspects of linear filtering in digital image processing:

Filter Kernel: A filter kernel is a small matrix, usually square, that defines the weights to be applied to the pixels in the image during convolution. The size of the kernel determines the extent of the neighborhood around each pixel that is considered during the filtering process. Common filter sizes include 3x3 and 5x5.

Convolution Operation: To apply a filter to an image, you slide the kernel over the entire image. At each position, you perform a point-wise multiplication between the kernel and the corresponding pixel values in the image and then sum up the results. This sum becomes the new value of the pixel at the center of the kernel.

Filter Types

- **Smoothing Filters:** These filters are used to reduce noise and blur an image. Common smoothing filters include the Gaussian filter and the mean filter.
- **Edge Detection Filters:** These filters highlight edges and boundaries in an image. Examples include the Sobel and Prewitt filters.
- **Sharpening Filters:** Sharpening filters enhance edges and fine details in an image. The Laplacian filter is an example.
- **Custom Filters:** You can create custom filters with specific weightings to achieve desired image processing effects.
- **Border Handling:** Handling image borders during convolution is important. There are different methods, such as zero-padding (setting border pixels to zero), mirror padding, or using only the valid part of the convolution result.

- **Filtering in Frequency Domain:** Convolution in the spatial domain can be computationally expensive, especially for large images and kernels. In some cases, it is more efficient to perform filtering in the frequency domain using techniques like the Fourier Transform.

- **Filtering Libraries:** Various programming languages and libraries provide functions for applying linear filters to images. Popular choices include OpenCV (Python), MATLAB, and image processing libraries in languages like C++ and Java.

Applications: Linear filtering is widely used in image enhancement, computer vision, medical image processing, and various other fields to extract valuable information from images or prepare them for further analysis.

2. Smoothing Filters

Smoothing linear filters in digital image processing are used to reduce noise, blur an image, and eliminate fine details. These filters work by averaging or weighting the pixel values in the neighborhood of each pixel to create a smoother and less noisy image.

1. **Mean Filter (Box Filter):**
   Working Mechanism: The mean filter replaces each pixel's value with the average of the pixel values in its neighborhood. It uses a square kernel (usually 3x3 or 5x5) and computes the average of the pixel values covered by the kernel.
2. **Gaussian Filter:**
   Working Mechanism: The Gaussian filter applies Gaussian smoothing to the image. It uses a Gaussian kernel, which emphasizes the central pixel and diminishes the influence of distant pixels. This creates a smoothing effect while preserving edges.

3. Working Mechanism for Different Edge Detection Filters

1. **Sobel Filter:**
   Working Mechanism: The Sobel filter calculates the gradient of an image to detect edges. It uses two 3x3 convolution kernels, one for detecting vertical edges and the other for horizontal edges. The gradient magnitude is computed as the square root of the sum of squares of the two gradients.

2. **Canny Edge Detector:**
   Working Mechanism: The Canny edge detector involves several steps, including Gaussian smoothing, gradient calculation, non-maximum suppression, and edge tracking by hysteresis. It identifies edges as regions where the gradient magnitude is above a certain threshold and connects them to form continuous edges.

3. **Laplacian of Gaussian (LoG):**
   Working Mechanism: The LoG filter first applies Gaussian smoothing to the image to reduce noise and then calculates the Laplacian (second derivative) of the smoothed image to find areas of rapid intensity change, which correspond to edges.

Smoothing Filters	Edge Detection Filters

import cv2
 	
# Load an image 	
image = cv2.imread('image.jpg')
 	
# Apply a 5x5 Mean (Box) Filter 	
mean_filtered_image = cv2.blur(image, (5, 5))
 	
# Apply Gaussian Smoothing 	
sigma = 1.5 # Adjust the sigma parameter for smoothing strength
gaussian_filtered_image = cv2.GaussianBlur(image, (0, 0), sigma)
 	
# Display the original and filtered images 	
cv2.imshow('Original Image', image)
cv2.imshow('Mean Filtered Image', mean_filtered_image)
 	
cv2.waitKey(0)
cv2.destroyAllWindows()
	import cv2
	
	# Load an image
	image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE) # Convert to grayscale for edge detection
	
	# Apply Sobel Filter to Detect Edges
	sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=5)
	sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=5)
	
	# Calculate the magnitude of the gradient
	edge_image = cv2.magnitude(sobel_x, sobel_y)
	
	# Apply Canny Edge Detector
	canny_edge_image = cv2.Canny(image, 100, 200) # Adjust threshold values as needed
	
	# Apply Gaussian Smoothing
	sigma = 1.5 # Adjust the sigma parameter for smoothing strength
	smoothed_image = cv2.GaussianBlur(image, (0, 0), sigma)
	
	# Apply Laplacian Filter for Edge Detection
	laplacian = cv2.Laplacian(smoothed_image, cv2.CV_64F)
	
	# Display the original and edge-detected images
	cv2.imshow('Original Image', image)
	cv2.imshow('Edge Image (Sobel)', edge_image.astype(np.uint8))
	
	cv2.waitKey(0)
	cv2.destroyAllWindows()

4. Sharpening Liner Filters and their Working Mechanism

Sharpening linear filters in digital image processing are used to enhance the fine details and edges in an image, making them appear more pronounced and crisp. These filters work by emphasizing the high-frequency components (such as edges) in the image while reducing the low-frequency components (such as smooth areas).

1. **Laplacian Filter:**

   Working Mechanism: The Laplacian filter enhances edges by highlighting areas where there is a rapid change in intensity values. It calculates the second derivative of the image, emphasizing regions with abrupt intensity transitions.

2. **Unsharp Masking (High-pass Filter):**

   Working Mechanism: Unsharp masking enhances details by subtracting a blurred version of the image from the original image. It accentuates the differences between neighboring pixels, effectively increasing contrast. The unsharp masking process involves convolving the image with a smoothing (blurring) kernel and then subtracting the smoothed image from the original image.

3. **High-Boost Filtering:**

   Working Mechanism: High-boost filtering is an extension of unsharp masking. It applies a weighted version of the Laplacian filter to the original image. The weight factor controls the degree of sharpening.
   High-Boost = A * Original - Blurred
   Here, "A" is a user-defined constant that determines the strength of sharpening.

4. **Gradient-based Filters (Sobel, Prewitt, Scharr):**

   Working Mechanism: Gradient-based filters, like Sobel, Prewitt, and Scharr, can also be used for sharpening. By applying these filters, you calculate the gradient of the image, which represents the rate of change of intensity. Edges are enhanced because they have high gradients.

import cv2
import numpy as np
 	
# Load an image 	
image = cv2.imread('image.jpg')
 	
# Laplacian Filter for Sharpening 	
laplacian = cv2.Laplacian(image, cv2.CV_64F)
sharpened_image_laplacian = cv2.add(image, laplacian)
sharpened_image_laplacian = np.clip(sharpened_image_laplacian, 0, 255).astype(np.uint8)
 	
# Unsharp Masking (High-pass Filter) for Sharpening 	
blurred_image = cv2.GaussianBlur(image, (5, 5), 0)
sharpened_image_unsharp = cv2.addWeighted(image, 2, blurred_image, -1, 0)
 	
# High-Boost Filtering for Sharpening 	
A = 2 # Adjust this value for the desired sharpening strength
	The Laplacian filter enhances edges by calculating the Laplacian of the image.
	Unsharp masking subtracts a blurred version of the image from the original to enhance details.
	High-boost filtering allows you to adjust the sharpening strength using the "A" parameter.
	You can adjust the parameters and experiment with different values of "A" to achieve the desired sharpening effect.

sharpened_image_high_boost = cv2.addWeighted(image, A + 1, blurred_image, -A, 0)
 	
# Display the original and sharpened images 	
cv2.imshow('Original Image', image)
cv2.imshow('Laplacian Sharpened Image', sharpened_image_laplacian)
cv2.imshow('Unsharp Masking Sharpened Image', sharpened_image_unsharp)
cv2.imshow('High-Boost Sharpened Image', sharpened_image_high_boost)
 	
cv2.waitKey(0)
cv2.destroyAllWindows()	

5. Border Handling Linear filtering

When applying linear filters to images in Python using libraries like OpenCV, you may need to handle the borders of the image. The border handling methods determine how to handle pixels near the image edges where the filter kernel extends beyond the image boundaries.

import cv2
import numpy as np
 	
# Load an image 	
image = cv2.imread('image.jpg')
 	
# Define a simple 3x3 kernel 	
kernel = np.array([[0, 1, 0],
[1, 5, 1],
[0, 1, 0]], dtype=np.float32) # Example kernel, you can define your own
 	
# Normalize the kernel to ensure the sum of its elements is 1 	
kernel /= kernel.sum()
 	
# Linear filtering with various border handling methods 	
filtered_image_replicate = cv2.filter2D(image, -1, kernel, borderType=cv2.BORDER_REPLICATE)
filtered_image_constant = cv2.filter2D(image, -1, kernel, borderType=cv2.BORDER_CONSTANT, borderValue=(0, 0, 0))
filtered_image_wrap = cv2.filter2D(image, -1, kernel, borderType=cv2.BORDER_WRAP)
filtered_image_reflect = cv2.filter2D(image, -1, kernel, borderType=cv2.BORDER_REFLECT)
	BORDER_REPLICATE: It replicates the border pixels to extend the image. This method is useful when you want to maintain the border pixel values.
	
	BORDER_CONSTANT: It pads the image with a constant value (specified by borderValue). This is useful when you want to set a specific background color around the image.
	
	BORDER_WRAP: It wraps the image around as if it's a torus, allowing the filter to continue from one edge to the opposite edge.
	
	BORDER_REFLECT: It reflects the border pixels, which can help reduce artifacts in the filtered image.

```
# Display the results
cv2.imshow('Original Image', image)
cv2.imshow('Filtered (Replicate)', filtered_image_replicate)
cv2.imshow('Filtered (Constant)', filtered_image_constant)
cv2.imshow('Filtered (Wrap)', filtered_image_wrap)
cv2.imshow('Filtered (Reflect)', filtered_image_reflect)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

6. Non-Linear Filtering:

Explanation: Non-linear filters are image processing techniques that use a non-linear mathematical function to process pixel values. Unlike linear filters (e.g., blurring, sharpening), non-linear filters consider neighboring pixel values in a non-linear way.

Real-Life Example: Removing salt-and-pepper noise from a photograph is a common real-life application of non-linear filtering.

1. **Median Filter:**
   The median filter replaces each pixel's value with the median value of the pixel values in its neighborhood. It's effective for salt-and-pepper noise removal
2. **Minimum Filter (Min Filter):**
   The minimum filter replaces each pixel's value with the minimum value among the pixel values in its neighborhood.
3. **Maximum Filter (Max Filter):**
   The maximum filter replaces each pixel's value with the maximum value among the pixel values in its neighborhood.
4. **Bilateral Filter:**
   The bilateral filter is a non-linear filter that smooths an image while preserving edges.

```
# Apply Median Filter
median_filtered_image = cv2.medianBlur(image, 5)  # Adjust kernel size as needed

kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
# Apply Minimum Filter
min_filtered_image = cv2.erode(image, kernel)

# Apply Maximum Filter
max_filtered_image = cv2.dilate(image, kernel)

# Apply Bilateral Filter
bilateral_filtered_image = cv2.bilateralFilter(image, d=9, sigmaColor=75, sigmaSpace=75)
```

Characteristic	Linear Filters	Non-Linear Filters
Linearity	Linear filters apply weighted sums of pixel values. The output at a pixel is a linear combination of its neighbors. Examples include mean, Gaussian, Sobel, etc.	Non-linear filters use non-linear operations on pixel values, making the output dependent on pixel ranking or other non-linear criteria. Examples include median, min, max, bilateral, etc.
Noise Reduction	Effective for reducing Gaussian noise and blurring an image while preserving linear structures.	Effective for removing non-Gaussian noise like salt-and-pepper noise and preserving details and edges.
Edge Preservation	Linear filters may smooth or blur edges in an image, making them less distinct.	Non-linear filters are better at preserving edges and details while reducing noise.
Computational Complexity	Generally computationally less expensive.	Can be more computationally intensive, especially with large kernels or adaptive filtering.
Common Examples	Mean filter, Gaussian filter, Sobel filter, Laplacian filter, etc.	Median filter, Min filter, Max filter, Bilateral filter, Adaptive median filter, Non-Local Means (NLM) filter, etc.
Application	Commonly used for tasks like basic smoothing, blurring, and gradient calculation.	Used for tasks like noise reduction, detail preservation, and edge enhancement.
Sensitivity to Noise	May not be effective against impulse noise (e.g., salt-and-pepper noise).	Effective against impulse noise and other non-Gaussian noise types.
Control Parameters	Typically controlled by filter size and kernel weights.	Controlled by filter size and the specific non-linear operation (e.g., median, min, max).

7. 1-D & 2-D Sampling

1-D (one-dimensional) sampling is the process of capturing discrete samples or data points along a single continuous signal or function. It involves selecting specific points from a continuous signal at regular intervals. These samples can be used to represent and analyze the original signal. In essence, 1-D sampling converts a continuous signal into a discrete form.

Example:

Consider a simple 1-D signal representing temperature measurements recorded every hour throughout a day. The temperature values are sampled at hourly intervals, resulting in a discrete set of data points. These discrete samples can be used for various purposes, such as calculating daily average temperature, detecting trends, or generating temperature graphs.

Applications:

- Audio processing: Sampling audio signals to create digital audio.
- Environmental monitoring: Sampling sensor data like temperature, humidity, or pollution levels.
- Stock market data: Sampling stock prices at specific time intervals for analysis.

2-D Sampling:

2-D (two-dimensional) sampling extends the concept of sampling to 2-D images or grids. It involves selecting discrete data points (pixels) from a continuous 2-D image at regular intervals. Each data point represents the intensity or color at that location in the image. 2-D sampling is essential in digital image processing and computer vision.

Example:

Consider a digital photograph taken with a camera. The camera's sensor samples the incoming light by dividing the image into a grid of pixels. Each pixel represents the color and intensity of the scene at that specific location. This discrete grid of pixels forms the digital image.

Applications:

- Digital image processing: Sampling images for manipulation, enhancement, and analysis.
- Computer vision: Extracting features from images for object recognition and tracking.
- Satellite imagery: Sampling Earth's surface to create digital maps.

```
import numpy as np
import matplotlib.pyplot as plt
# Continuous signal (e.g., temperature readings)
continuous_signal = np.array([23.5, 24.0, 23.8, 24.2, 23.7, 24.5, 23.9, 23.6, 24.1, 24.3])

# Sampling at hourly intervals
sampled_indices = np.arange(0, len(continuous_signal))
sampled_signal = continuous_signal[sampled_indices]

# Create a time axis for plotting
time_axis = np.arange(0, len(continuous_signal))

# Plot the continuous signal and sampled points
plt.figure(figsize=(10, 4))
plt.subplot(2, 1, 1)
plt.plot(time_axis, continuous_signal, marker='o', linestyle='-', color='b')
plt.title("Continuous 1-D Signal")
plt.xlabel("Time (hours)")
plt.ylabel("Temperature")

plt.subplot(2, 1, 2)
plt.stem(sampled_indices, sampled_signal, basefmt=" ", markerfmt="ro", linefmt="-r")
plt.title("Sampled 1-D Signal")
plt.xlabel("Sample Index")
plt.ylabel("Temperature")

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

8. Fourier Transformation in 2D

The 2D Fourier Transform is a mathematical transformation used in image processing and signal analysis to represent an image in the frequency domain. It decomposes an image into a sum of sinusoidal functions, each with a specific frequency and phase. This transformation is valuable for tasks such as filtering, compression, and feature extraction. Here's how the 2D Fourier Transform works:

Forward 2D Fourier Transform:

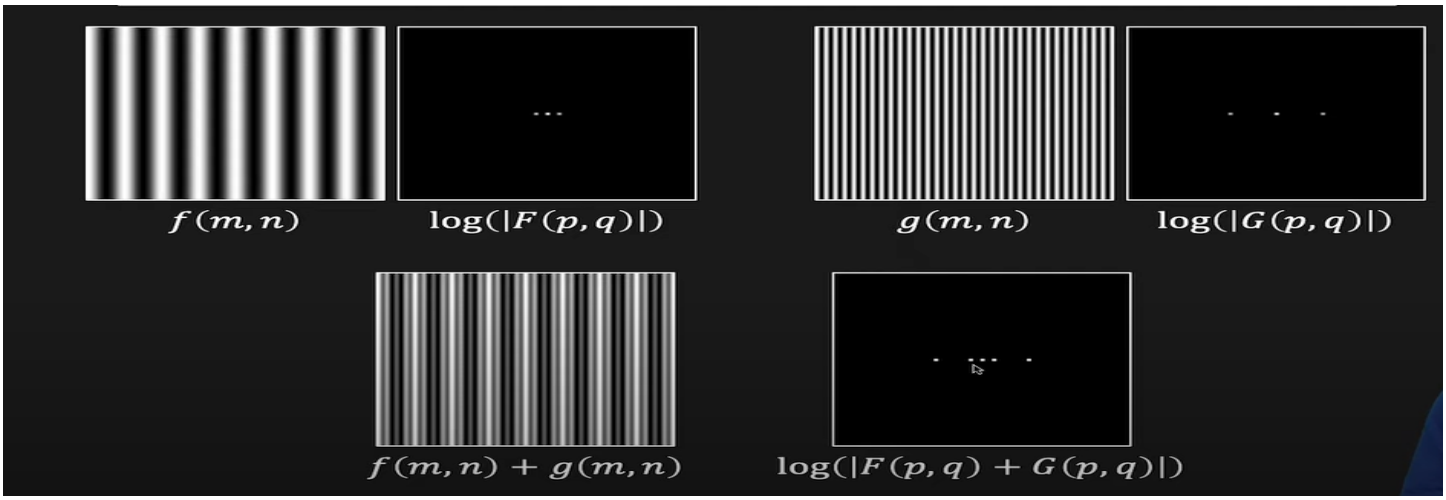
- **Input Image:** Start with an input image, typically represented as a 2D array of pixel values. The image can be in grayscale or color.
- **Spatial Domain:** In the spatial domain, the input image represents variations in intensity or color across its two dimensions (rows and columns). This domain is where we perceive the image visually.
- **Frequency Domain:** The 2D Fourier Transform converts the image from the spatial domain to the frequency domain. In the frequency domain, the image is represented as a 2D array of complex numbers, where each number corresponds to a specific frequency and phase.
- **Frequency Components:** Each complex number in the frequency domain represents a sinusoidal wave. The magnitude of the complex number represents the amplitude (strength) of that wave, and the phase represents its position relative to the origin.

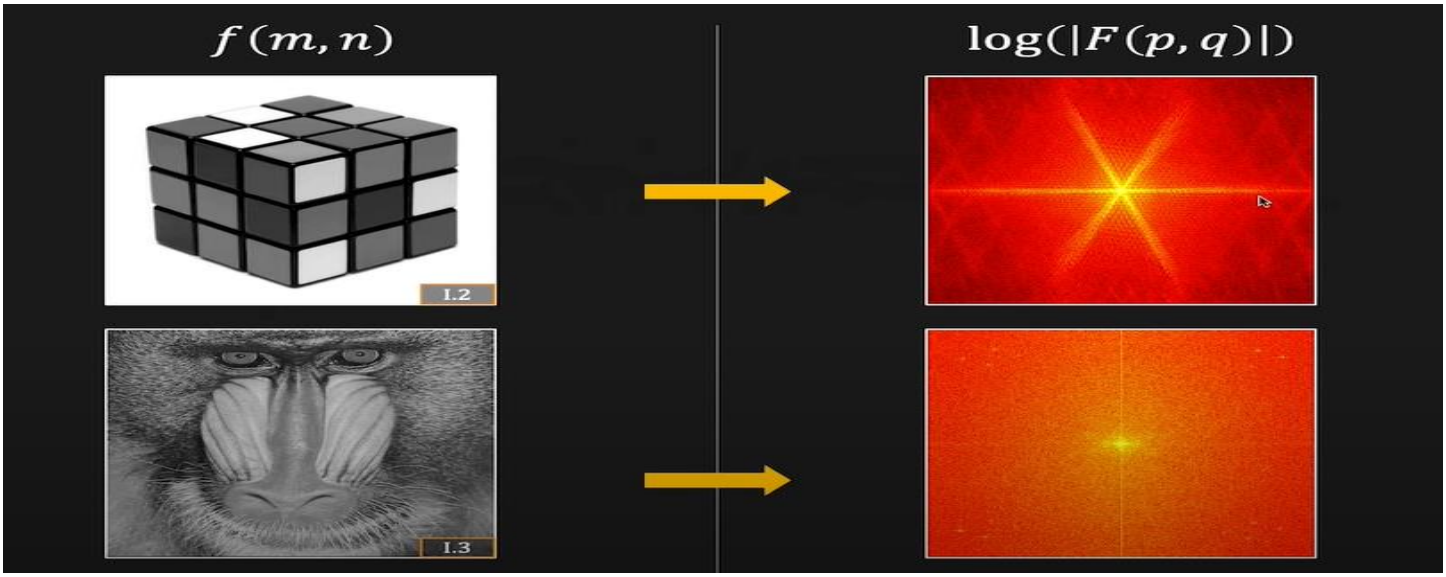
Inverse 2D Fourier Transform:

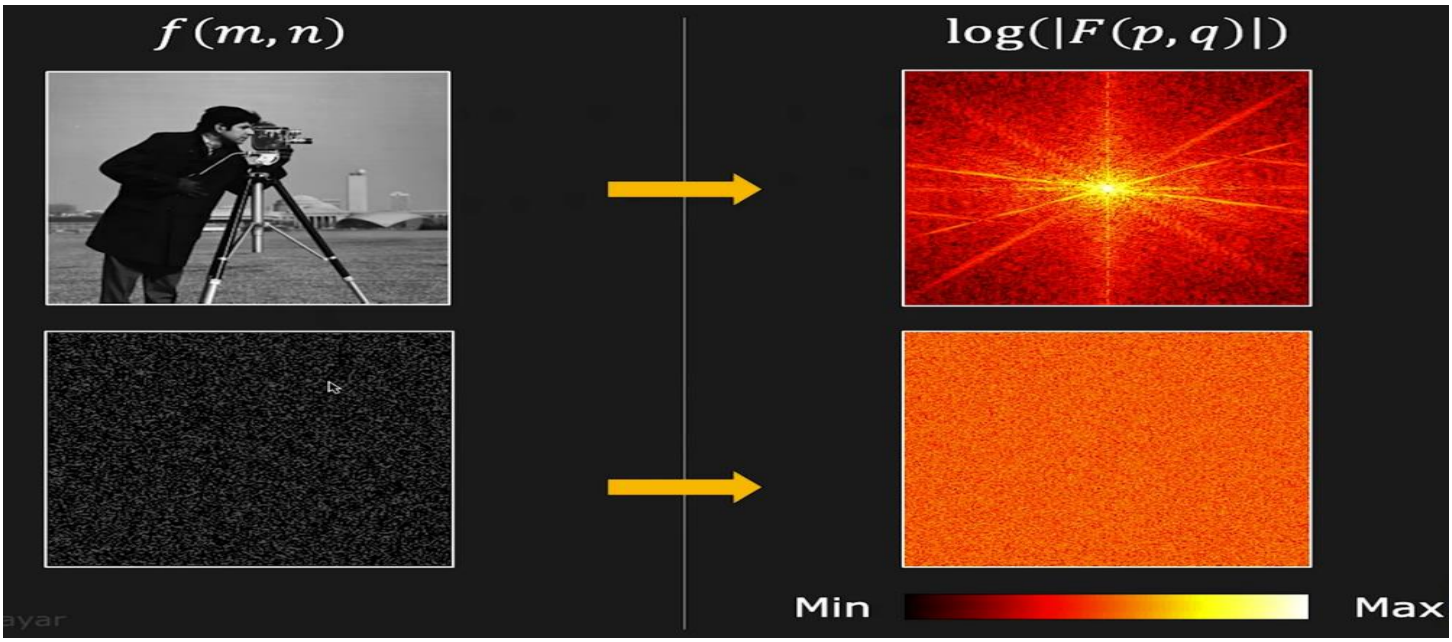
- **Frequency Domain Data:** Start with an image in the frequency domain, represented as a 2D array of complex numbers.
- **Inverse Transform:** Apply the inverse 2D Fourier Transform to the frequency domain data to convert it back to the spatial domain.
- **Reconstructed Image:** The result is a reconstructed image that should closely resemble the original input image. This reconstructed image can be used for various image processing tasks.

Applications of the 2D Fourier Transform in image processing include:

- **Filtering:** Filtering an image in the frequency domain allows for operations like smoothing (low-pass filtering) and edge enhancement (high-pass filtering).
- **Compression:** Transforming an image into the frequency domain can help reduce redundancy and compress the image data.
- **Feature Extraction:** Analyzing the frequency components of an image can aid in feature extraction and pattern recognition.
- **Image Restoration:** Dealing with issues like noise and blurriness by manipulating the frequency domain representation.
- **Image Registration:** Aligning and matching images by comparing their frequency components.


$f(m,n)$
$\log(|F(p,q)|)$
$g(m,n)$
$\log(|G(p,q)|)$
$f(m,n) + g(m,n)$
$\log(|F(p,q) + G(p,q)|)$


$f(m,n)$
$\log(|F(p,q)|)$
I.2
I.3


$f(m,n)$
$\log(|F(p,q)|)$
Min
Max

Forward 2D Fourier Transform

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Load the image (grayscale)
image = cv2.imread('input_image.jpg', cv2.IMREAD_GRAYSCALE)

# Perform the 2D Fourier Transform
# converts the image from the spatial domain to the frequency domain
fourier_transform = np.fft.fft2(image)

# shifting the zero frequency components to the center is a common practice in 2D Fourier Transforms to enhance the
# interpretability and usefulness of the frequency domain representation, especially for tasks like filtering, analysis, and
# visualization
# Shift the zero frequency components to the center
# We shift the zero frequency components to the center of the spectrum using np.fft.fftshift().
# This makes it easier to visualize the spectrum

fourier_transform_shifted = np.fft.fftshift(fourier_transform)

# Calculate the magnitude spectrum
magnitude_spectrum = np.log(np.abs(fourier_transform_shifted) + 1)  # Avoid log(0) by adding 1

# Display the original image and magnitude spectrum
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')

plt.subplot(1, 2, 2)
plt.imshow(magnitude_spectrum, cmap='gray')
plt.title('Magnitude Spectrum (log-scaled)')

plt.show()
```

Inverse 2D Fourier Transform:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Create synthetic frequency domain data (e.g., magnitude spectrum)
# For this example, we'll create a simple pattern in the frequency domain
# Replace this with your actual frequency domain data
frequency_domain_data = np.zeros((256, 256), dtype=float)
frequency_domain_data[100:120, 150:170] = 255.0

# Perform the inverse 2D Fourier Transform
spatial_domain_image = np.fft.ifft2(np.fft.ifftshift(frequency_domain_data)).real

# Display the reconstructed spatial domain image
plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.imshow(frequency_domain_data, cmap='gray')
plt.title('Frequency Domain Data')

plt.subplot(1, 2, 2)
plt.imshow(spatial_domain_image, cmap='gray')
plt.title('Reconstructed Image')
plt.show()
```

9. Implementation of 2D Fourier Transform in image processing for Different Applications

Low Pass Filtering

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image (grayscale)
image = cv2.imread('input_image.jpg',
cv2.IMREAD_GRAYSCALE)

# Perform the 2D Fourier Transform
fourier_transform = np.fft.fft2(image)

# Shift the zero frequency components to the center
fourier_transform_shifted = np.fft.fftshift(fourier_transform)

# Create a low-pass filter mask
rows, cols = image.shape
center_row, center_col = rows // 2, cols // 2
```

Image Dimensions:
Let the dimensions of the input image be rows (number of rows) and cols (number of columns).

Center Coordinates:
Calculate the coordinates of the center of the image:
center_row = rows / 2 (assuming integer division, so it rounds down to the nearest integer)
center_col = cols / 2

Cutoff Frequency:
Define a cutoff frequency, denoted as cutoff_frequency. This value determines the size of the low-pass filter and controls the extent of low-frequency information that will be retained.

cutoff_frequency = 30 # Adjust as needed
mask = np.zeros((rows, cols), dtype=np.uint8)
mask[center_row - cutoff_frequency:center_row + cutoff_frequency + 1,
center_col - cutoff_frequency:center_col + cutoff_frequency + 1] = 1
 	
# Apply the mask to the Fourier Transform 	
filtered_fourier_transform = fourier_transform_shifted * mask
 	
# Perform the inverse Fourier Transform 	
filtered_image = np.abs(np.fft.ifft2(np.fft.ifftshift(filtered_fourier_transform)))
 	
# Display the filtered image 	
plt.imshow(filtered_image, cmap='gray')
plt.title('Low-Pass Filtered Image')
plt.show()
	Initialize Mask:
	Create an empty mask (filter) with the same dimensions as the input image. The mask is initially filled with zeros:
	mask is a matrix of size rows x cols, where each element is initially set to 0.
	
	Define Filter Region:
	Set a square region within the mask to 1. This square region is centered at the image's center and has a size determined by the cutoff_frequency:
	
	The region defined with 1 values represents the pass region (low-pass filter), allowing low-frequency components to pass through.
	Pixels outside this square region remain 0, indicating that high-frequency components are suppressed.
	The region is defined as follows:
	
	For the rows: center_row - cutoff_frequency to center_row + cutoff_frequency
	For the columns: center_col - cutoff_frequency to center_col + cutoff_frequency

Mathematically

For each element (i, j) in the mask:

```
  if (center_row - cutoff_frequency <= i <= center_row + cutoff_frequency) and
    (center_col - cutoff_frequency <= j <= center_col + cutoff_frequency):
    mask(i, j) = 1
  else:
    mask(i, j) = 0
```

Image Compression

```
# Load the image (grayscale)
image = cv2.imread('input_image.jpg', cv2.IMREAD_GRAYSCALE)
# Perform the 2D Fourier Transform
-------
# Set a threshold to retain only significant frequency components
threshold = 1000  # Adjust as needed
filtered_fourier_transform[abs(filtered_fourier_transform) < threshold] = 0
# Perform the inverse Fourier Transform to obtain the compressed image
compressed_image = np.abs(np.fft.ifft2(np.fft.ifftshift(filtered_fourier_transform)))
```

```
# Display the compressed image
plt.imshow(compressed_image, cmap='gray')
plt.title('Compressed Image')
plt.show()
```

Feature Extraction (Magnitude Spectrum)

```
# Calculate the magnitude spectrum
--------
# Display the magnitude spectrum
plt.imshow(magnitude_spectrum, cmap='gray')
plt.title('Magnitude Spectrum (log-scaled)')
plt.show()
```

Image Restoration (Inverse Filtering)

```
# Load the degraded image (grayscale)
degraded_image = cv2.imread('degraded_image.jpg', cv2.IMREAD_GRAYSCALE)

# Perform the 2D Fourier Transform for both the original and degraded images
--------
-------

# Compute the inverse filter in the frequency domain
epsilon = 1e-6
inverse_filter = np.divide(fourier_transform_original, fourier_transform_degraded + epsilon)

# Apply the inverse filter to the degraded image in the frequency domain
restored_image_frequency_domain = fourier_transform_degraded * inverse_filter

# Perform the inverse Fourier Transform to obtain the restored image
restored_image = np.abs(np.fft.ifft2(restored_image_frequency_domain))

# Display the degraded and restored images
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(degraded_image, cmap='gray')
plt.title('Degraded Image')

plt.subplot(1, 2, 2)
plt.imshow(restored_image, cmap='gray')
plt.title('Restored Image')
plt.show()
```

Image Registration (Phase Correlation)

```
# Load the reference and target images (grayscale)
reference_image = cv2.imread('reference_image.jpg', cv2.IMREAD_GRAYSCALE)
target_image = cv2.imread('target_image.jpg', cv2.IMREAD_GRAYSCALE)

# Perform the 2D Fourier Transform for both images
----
----



# Calculate the cross-power spectrum
cross_power_spectrum = fourier_transform_reference * np.conj(fourier_transform_target)

# Calculate phase correlation
phase_correlation = np.fft.ifft2(cross_power_spectrum / (np.abs(cross_power_spectrum) + 1e-6))

# Find the peak in the phase correlation to estimate translation
shifted_peak = np.unravel_index(np.argmax(np.abs(phase_correlation)), phase_correlation.shape)
x_shift, y_shift = shifted_peak[1], shifted_peak[0]

# Apply translation to align the images
aligned_image = np.roll(target_image, (y_shift, x_shift), axis=(0, 1))

# Display the reference and aligned images
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(reference_image, cmap='gray')
plt.title('Reference Image')

plt.subplot(1, 2, 2)
plt.imshow(aligned_image, cmap='gray')
plt.title('Aligned Image')
plt.show()
```

Image registration, particularly using Phase Correlation, is a technique used in image processing to align or match two or more images, often referred to as the "reference" and "target" images. The primary goal is to find the transformation (e.g., translation, rotation, scaling) that aligns the target image with the reference image

- **Reference Image and Target Image:** You start with two images: the "reference image" and the "target image." The reference image is considered the fixed or known image, and the target image is the one you want to align with the reference image.

- **2D Fourier Transform:** Both the reference and target images undergo a 2D Fourier Transform (2D FFT) separately. The 2D FFT converts the images from the spatial domain (pixel values) into the frequency domain. In the frequency domain, you can analyze the phase and magnitude spectra of the images.

- **Phase Correlation**: The key concept in Phase Correlation is that the phase of the frequency components in the two images is used to estimate the relative displacement (translation) between them. This phase information is invariant to changes in lighting conditions and image contrast, making Phase Correlation robust for image alignment.

- **Cross-Power Spectrum:** Calculate the cross-power spectrum between the Fourier transforms of the reference and target images. The cross-power spectrum measures how well the two images are correlated across different translations. It highlights peaks in the spectrum, which correspond to translation offsets.

- **Inverse Fourier Transform:** Apply the inverse Fourier Transform (2D IFFT) to the cross-power spectrum to obtain the phase correlation map in the spatial domain. The phase correlation map will have a peak at the location corresponding to the translation between the reference and target images.

- **Peak Detection:** Find the peak in the phase correlation map. The peak's location (usually the coordinates of the maximum value) indicates the translation offset needed to align the target image with the reference image.

- **Alignment:** Apply the calculated translation offset to the target image to align it with the reference image. Common transformations include translation, rotation, and scaling.

- **Output:** The aligned target image is the registered result, aligned with the reference image.

10. Hybrid Images (Computational Photography)

Hybrid images are a fascinating concept in image processing and computer vision that combines two different images, one with high spatial frequencies (fine details) and another with low spatial frequencies (coarse features), to create a single image that appears different at different viewing distances. The primary idea behind hybrid images is that human vision has different sensitivity to spatial frequencies at various distances. Here's how they work:

High-Pass and Low-Pass Filtering: Two input images are prepared: one contains high-frequency components (fine details), and the other contains low-frequency components (coarse features). High-pass filtering extracts the fine details from the first image, and low-pass filtering extracts the coarse features from the second image. These filters can be Gaussian filters or other frequency-domain filters.

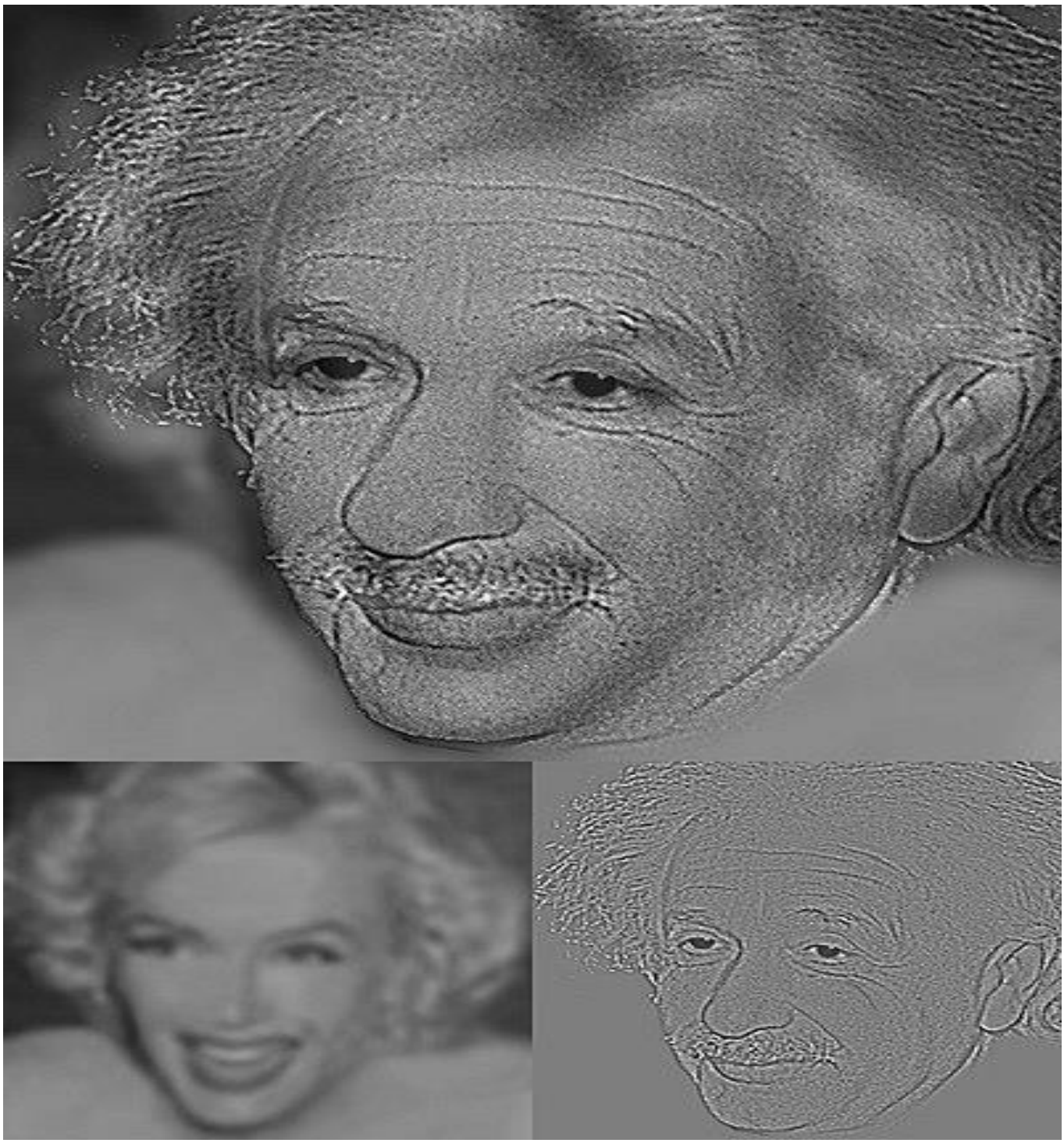
Combining Images: The filtered high-frequency image and the filtered low-frequency image are combined pixel-wise. The pixel values of the resulting image are calculated by adding the corresponding pixel values from both filtered images. The resulting image contains both coarse and fine details.

Visual Perception: The human visual system perceives images differently depending on viewing distance. When you look at a hybrid image from a distance, you primarily perceive the low-frequency components, making it appear as one image. As you get closer to the image, the high-frequency components become more noticeable, revealing the details present in the high-pass filtered image.

Visual Illusion: Hybrid images create a visual illusion where the image's interpretation changes with viewing distance. This effect can be used for various creative and scientific purposes, such as art, perception studies, and image recognition experiments.

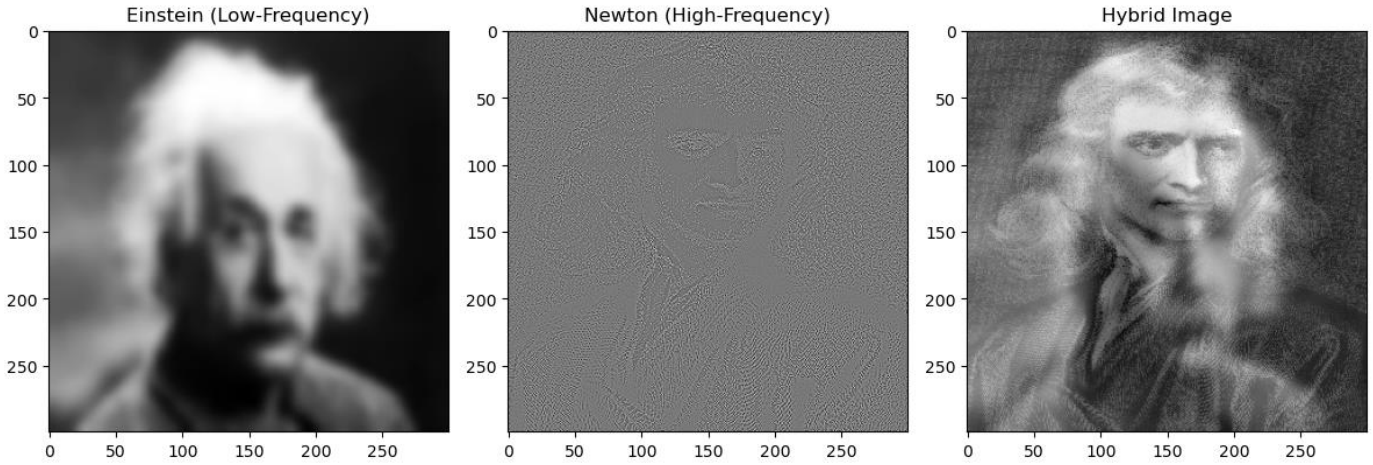
Common examples of hybrid images include faces that appear to switch between two different people when viewed from different distances or images that appear as both animals and everyday objects, depending on the viewing distance.

Hybrid images highlight the importance of understanding the human visual system's sensitivity to different spatial frequencies and its impact on perception. They provide insights into how we perceive images and can be used for artistic and scientific purposes to explore visual perception phenomena. (https://manavm3.web.illinois.edu/cs445/proj1/)




```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the two images
einstein_image = cv2.imread('ents.jpg')
einstein_image =  cv2.resize(einstein_image, (300,300))
newton_image = cv2.imread('ntn.jpg')
newton_image = cv2.resize(newton_image,(300,300))

# Convert the images to grayscale
einstein_gray = cv2.cvtColor(einstein_image, cv2.COLOR_BGR2GRAY)
newton_gray = cv2.cvtColor(newton_image, cv2.COLOR_BGR2GRAY)

# Apply Gaussian blur to Einstein (low-pass filter)
einstein_low_pass = cv2.GaussianBlur(einstein_gray, (25, 25), 0)

# Subtract the low-pass image from Newton (high-pass filter)
#newton_high_pass = newton_gray - einstein_low_pass
#newton_high_pass = cv2.Laplacian(newton_gray, cv2.CV_64F, (3,3))
# Add the low-pass Einstein and high-pass Newton to create the hybrid image
# Perform 2D Fourier Transform

fourier_transform = np.fft.fft2(newton_gray)

# Shift zero frequency components to the center
fourier_transform_shifted = np.fft.fftshift(fourier_transform)

# Define the size of the high-pass filter kernel (e.g., a Laplacian kernel)
kernel_size = 5

# Create a high-pass filter mask
rows, cols = newton_gray.shape
center_row, center_col = rows // 2, cols // 2
mask = np.ones((rows, cols), dtype=np.uint8)
mask[center_row - kernel_size:center_row + kernel_size + 1,
    center_col - kernel_size:center_col + kernel_size + 1] = 0
```

```
# Apply the mask to the Fourier Transform
filtered_fourier_transform = fourier_transform_shifted * mask

# Perform the inverse Fourier Transform to obtain the high-pass image
high_pass_image = np.fft.ifft2(np.fft.ifftshift(filtered_fourier_transform)).real

hybrid_image = einstein_low_pass + high_pass_image

# Display the original images and the hybrid image
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.imshow(einstein_low_pass, cmap='gray')
plt.title('Einstein (Low-Frequency)')

plt.subplot(1, 3, 2)
plt.imshow(newton_high_pass, cmap='gray')
plt.title('Newton (High-Frequency)')

plt.subplot(1, 3, 3)
plt.imshow(hybrid_image, cmap='gray')
plt.title('Hybrid Image')

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



Tasks

1. Complete the above codes where the lines are missing.
2. **Linear Filtering:**
    - Implement a Gaussian blur filter using convolution for image smoothing.
    - Apply a Sobel filter to perform edge detection on a grayscale image.
    - Perform image sharpening using the Laplacian filter.

- Implement a mean filter for noise reduction in an image.

3. **Non-Linear Filtering:**
   - Develop a median filter for removing salt-and-pepper noise from an image.
   - Apply a max filter to perform dilation on a binary image.
   - Implement a min filter to perform erosion on a binary image.
   - Create a bilateral filter for edge-preserving smoothing.
   - Implement an adaptive median filter for noise reduction while preserving edges.
4. **Fourier Transformations**:
   - Calculate the 1D Fourier Transform of a signal and visualize its magnitude and phase spectra.
   - Apply a 2D Fourier Transform to an image and display its magnitude spectrum.
   - Implement a high-pass filter in the frequency domain to emphasize edges in an image.
   - Perform image compression using the Fourier Transformation
5. **Hybrid Images:**
   - Create a hybrid image from two input images with different spatial frequencies.
   - Experiment with different combinations of high-pass and low-pass filters for hybrid image creation.
   - Generate a hybrid image that exhibits a strong visual illusion when viewed from different distances.
   - Investigate how changing the filter parameters affects the perception of a hybrid image.
   - Analyze the trade-offs between high and low-frequency components in hybrid images for various applications.