AIU.
ALAMEIN
INTERNATIONAL UNIVERSITY
جامعة العلمين الدولية
QUALITY ASSURANCE CENTER

Image Processing (CSE281)

Fall 2025/2026

Dr. Essam Abdellatef

Contact Number: 012 8 192 55 90

Email: eabdellatef@Aiu.edu.eg

Discrete Fourier Transform

Discrete Fourier Transform (DFT) is used to analyze Periodic, Discrete-Time signals.

- **Discrete-Time:** The signal is not a continuous curve but a sequence of samples taken at regular intervals.
- **Periodic:** The DFT inherently treats its finite input as one period of a periodic signal.
- **The Result:** A set of discrete frequencies.

Discrete Fourier Transform

For images, we use the Discrete Fourier Transform (DFT) because:

- **Images are discrete:** They are made of pixels, not a continuous function.
- **Images are finite:** They have a fixed width and height.
- The classic ***Fourier Transform and Fourier Series*** are mathematical concepts that help us understand what the DFT is doing, but they are not directly applicable to a grid of pixels.

Discrete Fourier Transform

Think of a grayscale image as a 2D signal where the brightness of a pixel is its amplitude. The DFT decomposes this image into its underlying 2D sinusoidal waves.

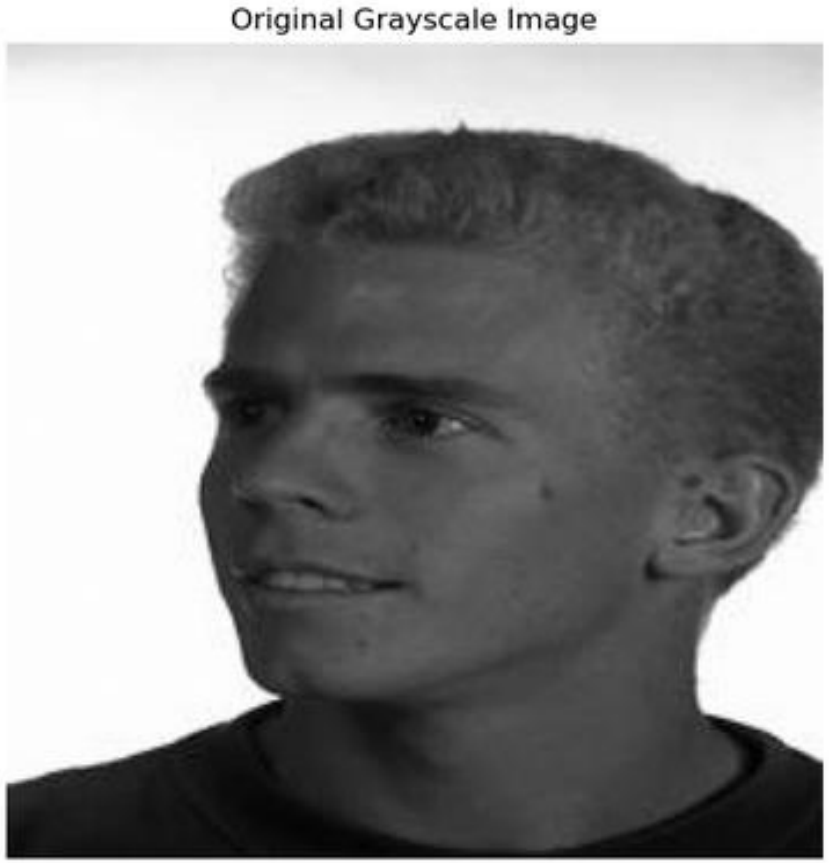
- ***The Spatial Domain:*** This is the original image, defined by pixel intensities I(x, y) at coordinates (x, y).

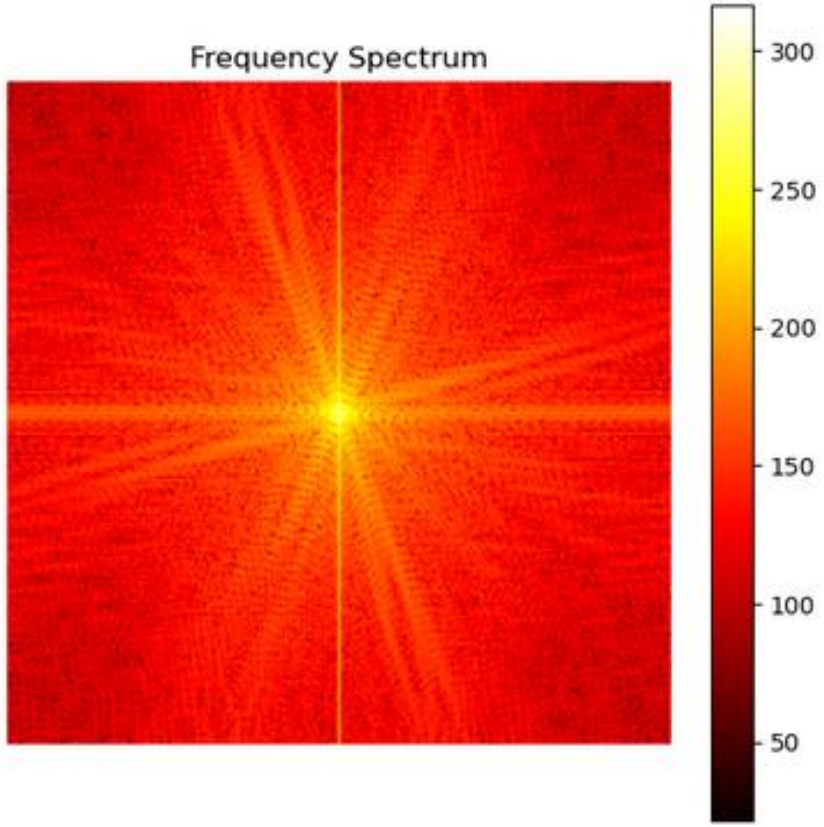
- ***The Frequency Domain (DFT Output):*** The DFT transforms the image into a new representation, ***F(u, v),*** which tells us about the frequencies present.

  - ***Low Frequencies*** represent slow changes in intensity (smooth areas)

  - ***High Frequencies*** represent rapid changes in intensity (edges, noise, and fine details).

Discrete Fourier Transform





Discrete Fourier Transform

The ***DFT transforms*** a finite sequence of ***N complex numbers (time domain)*** into another ***finite sequence of N complex numbers (frequency domain).***

$$X(k) = \sum_{n=0}^{N-1} x(n).e^{-j2\pi kn/N} \qquad k = 0,1,2,3,\ldots,N-1$$

IDFT Formula:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k).e^{j2\pi kn/N} \qquad n = 0,1,2,3,\ldots,N-1$$

Discrete Fourier Transform

Twiddle Factor:

$$W_N = e^{-j\frac{2\pi}{N}}$$

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{kn}$$

Discrete Fourier Transform

Find the ***DFT*** of the signal:

$$x(n) = \begin{cases} 1 & 0 \le n \le L-1 \\ 0 & \text{Otherwise} \end{cases}$$

Solution:

The DFT general equation:

$$X(k) = \sum_{n=0}^{L-1} 1.e^{-j2\pi\, nk/L}$$

$$= 1 + e^{-j2\pi\, k/L} + (e^{-j2\pi\, k/L})^2 + (e^{-j2\pi\, k/L})^3 + \ldots + (e^{-j2\pi\, k/L})^{L-1}$$

Discrete Fourier Transform

Find the DFT of the sequence x(n) = {1, 2, 3, 4}.

For k = 0: $X[0] = 1 + 2 + 3 + 4 = 10$

For k = 1: $X[1] = 1 - 2j - 3 + 4j = -2 + 2j$

For k = 2: $X[2] = 1 \cdot 1 + 2 \cdot (-1) + 3 \cdot 1 + 4 \cdot (-1) = 1 - 2 + 3 - 4 = -2$

For k = 3: $X[3] = 1 \cdot 1 + 2 \cdot (j) + 3 \cdot (-1) + 4 \cdot (-j) = 1 + 2j - 3 - 4j = -2 - 2j$

Final DFT Result: $X[k] = \{10, -2 + 2j, -2, -2 - 2j\}$

Discrete Fourier Transform

DFT Formula using W:

$$X[k] = \sum_{n=0}^{3} x[n] \cdot W_4^{kn}$$

$$X[0] = \sum_{n=0}^{3} x[n] \cdot W_4^{0} = 1 \cdot 1 + 2 \cdot 1 + 3 \cdot 1 + 4 \cdot 1 = 10$$

Discrete Fourier Transform

$$X[1] = \sum_{n=0}^{3} x[n] \cdot W_4^n$$

$$= 1 \cdot W_4^0 + 2 \cdot W_4^1 + 3 \cdot W_4^2 + 4 \cdot W_4^3$$

$$= 1 \cdot 1 + 2 \cdot (-j) + 3 \cdot (-1) + 4 \cdot (j)$$

$$= 1 - 2j - 3 + 4j = -2 + 2j$$

Discrete Fourier Transform

$$X[2] = \sum_{n=0}^{3} x[n] \cdot W_4^{2n}$$

$$= 1 \cdot W_4^0 + 2 \cdot W_4^2 + 3 \cdot W_4^4 + 4 \cdot W_4^6$$

Since $W_4^4 = (W_4^1)^4 = (-j)^4 = 1$ and $W_4^6 = W_4^2 = -1$:

$$= 1 \cdot 1 + 2 \cdot (-1) + 3 \cdot 1 + 4 \cdot (-1)$$

$$= 1 - 2 + 3 - 4 = -2$$

Discrete Fourier Transform

$$X[3] = \sum_{n=0}^{3} x[n] \cdot W_4^{3n}$$

$$= 1 \cdot W_4^0 + 2 \cdot W_4^3 + 3 \cdot W_4^6 + 4 \cdot W_4^9$$

Since $W_4^6 = W_4^2 = -1$ and $W_4^9 = W_4^1 = -j$:

$$= 1 \cdot 1 + 2 \cdot (j) + 3 \cdot (-1) + 4 \cdot (-j)$$

$$= 1 + 2j - 3 - 4j = -2 - 2j$$

Discrete Fourier Transform

$$X[k] = \{10,\ -2+2j,\ -2,\ -2-2j\}$$

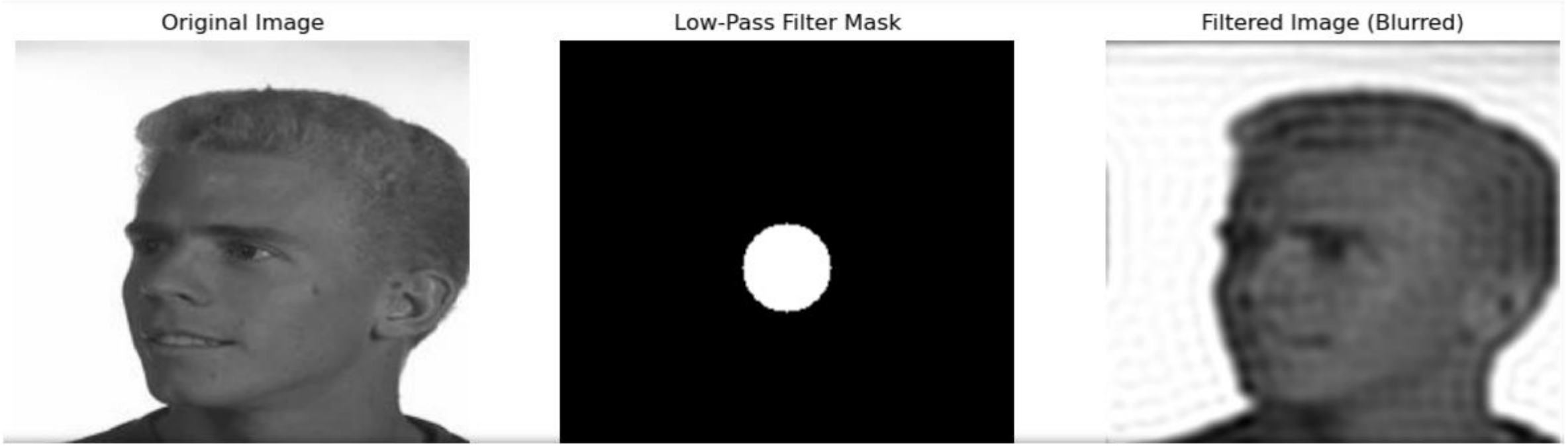
The DFT can also be expressed using the DFT matrix:

$$\begin{bmatrix} X[0] \\ X[1] \\ X[2] \\ X[3] \end{bmatrix} = \begin{bmatrix} W_4^0 & W_4^0 & W_4^0 & W_4^0 \\ W_4^0 & W_4^1 & W_4^2 & W_4^3 \\ W_4^0 & W_4^2 & W_4^4 & W_4^6 \\ W_4^0 & W_4^3 & W_4^6 & W_4^9 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Discrete Fourier Transform

```python
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

def low_pass_filter(image_path, cutoff_ratio=0.1):
    img = Image.open(image_path).convert('L')
    img_array = np.array(img)
    dft = np.fft.fft2(img_array)
    dft_shift = np.fft.fftshift(dft)
    rows, cols = img_array.shape
    crow, ccol = rows // 2, cols // 2
    mask = np.zeros((rows, cols))
    cutoff = int(min(rows, cols) * cutoff_ratio)
    y, x = np.ogrid[:rows, :cols]
    mask_area = (x - ccol)**2 + (y - crow)**2 <= cutoff**2
    mask[mask_area] = 1
    filtered_dft = dft_shift * mask
    filtered_ishift = np.fft.ifftshift(filtered_dft)
    filtered_array = np.fft.ifft2(filtered_ishift).real
    filtered_array = np.uint8(np.clip(filtered_array, 0, 255))
    return Image.fromarray(filtered_array), mask

image_path = "F:\\13.jpg"
filtered_img, mask = low_pass_filter(image_path, 0.1)
```

Discrete Fourier Transform



Discrete Fourier Transform

Original Image (Spatial Domain): Contains both low and high frequencies

Apply DFT → Convert to Frequency Domain

- DFT organizes frequencies with LOW frequencies at CENTER
- HIGH frequencies at EDGES/CORNERS

Create LPF Mask:

- Center (low frequencies) = 1 (PASS)
- Edges (high frequencies) = 0 (BLOCK)

Multiply DFT × Mask → Only low frequencies remain

Apply Inverse DFT → Blurred image

Discrete Fourier Transform

```python
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

def high_pass_filter(image_path, cutoff_ratio=0.1):
    img = Image.open(image_path).convert('L')
    img_array = np.array(img)
    dft = np.fft.fft2(img_array)
    dft_shift = np.fft.fftshift(dft)
    rows, cols = img_array.shape
    crow, ccol = rows // 2, cols // 2
    mask = np.ones((rows, cols))
    cutoff = int(min(rows, cols) * cutoff_ratio)
    y, x = np.ogrid[:rows, :cols]
    mask_area = (x - ccol)**2 + (y - crow)**2 <= cutoff**2
    mask[mask_area] = 0
    filtered_dft = dft_shift * mask
    filtered_ishift = np.fft.ifftshift(filtered_dft)
    filtered_array = np.fft.ifft2(filtered_ishift).real
    filtered_array = np.uint8(np.clip(filtered_array, 0, 255))
    return Image.fromarray(filtered_array), mask

image_path = "F:\\13.jpg"
filtered_img, mask = high_pass_filter(image_path, 0.1)
```

Discrete Fourier Transform

Original Image

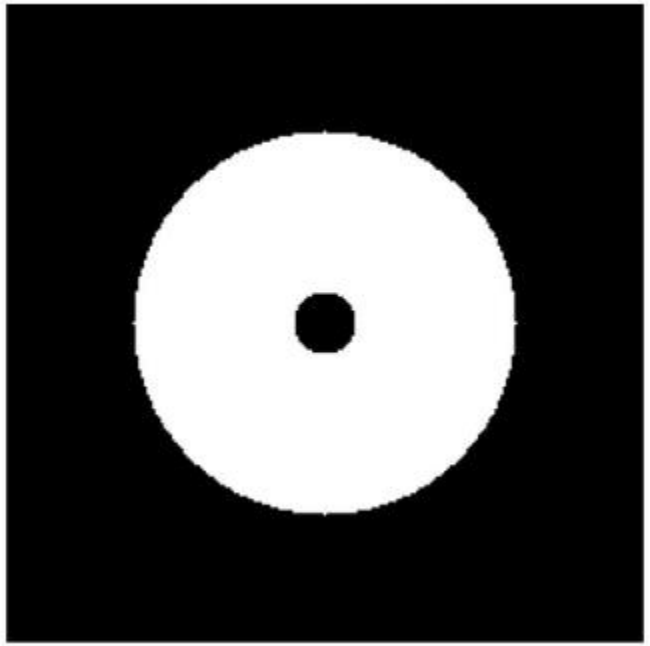
High-Pass Filter Mask

Filtered Image (Edges)

Discrete Fourier Transform

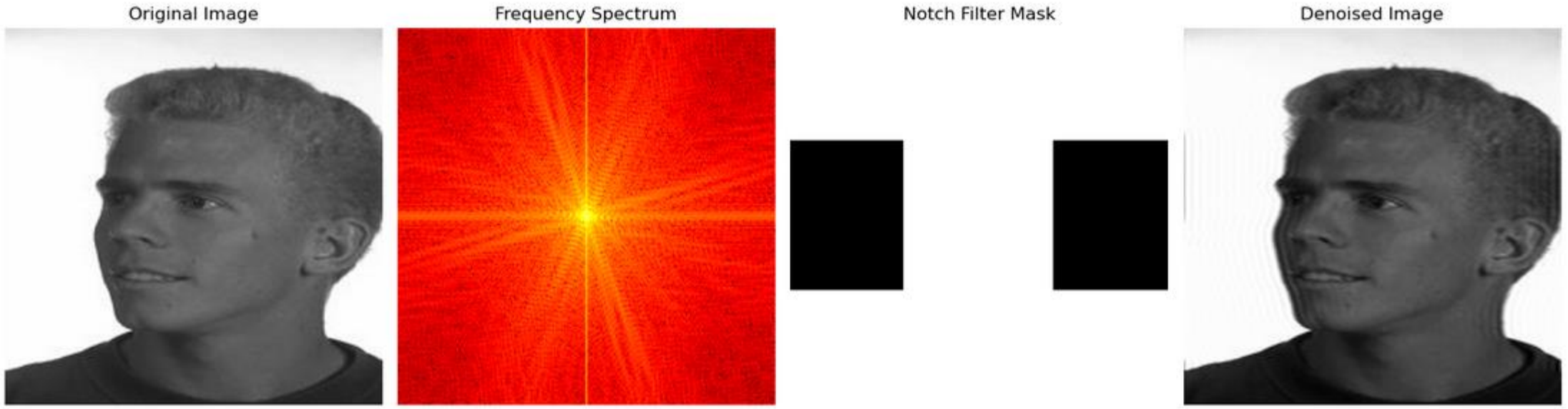
Original Image

Band-Pass Filter Mask



Filtered Image (Mid-Frequencies)

Discrete Fourier Transform



Discrete Fourier Transform

NOTCH FILTER STEPS:

Original Image (Spatial Domain): Contains image + periodic noise patterns

Apply DFT → Convert to Frequency Domain

- ❑ DFT shows bright spots at specific frequencies representing periodic noise

Create Notch Filter Mask:

- ❑ Specific frequency locations (noise spots) = 0 (BLOCK)
- ❑ All other frequencies = 1 (PASS)

Multiply DFT × Mask → Remove specific noise frequencies while preserving other frequencies

Apply Inverse DFT → Clean image with periodic noise removed

Image Encryption

XOR Encryption

- Uses bitwise XOR operation between image pixels and a secret key.
- Each pixel value (0 - 255) is XORed with the key value.

Image Encryption

```python
import numpy as np
from PIL import Image

def encrypt_image(image_path, key):
    img = Image.open(image_path)
    img_array = np.array(img)
    encrypted_array = img_array ^ key
    encrypted_img = Image.fromarray(encrypted_array)
    return encrypted_img

def decrypt_image(encrypted_img, key):

    encrypted_array = np.array(encrypted_img)
    decrypted_array = encrypted_array ^ key
    decrypted_img = Image.fromarray(decrypted_array)
    return decrypted_img
```

Image Encryption

```
image_path = "F:\\13.jpg"
secret_key = 123
encrypted = encrypt_image(image_path, secret_key)
decrypted = decrypt_image(encrypted, secret_key)

plt.figure()
plt.subplot(1, 2, 1)
plt.imshow(encrypted)
plt.title('Encrypted Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(decrypted)
plt.title('Decrypted Image')
plt.axis('off')
```

Image Encryption

Encrypted Image

Decrypted Image

Measuring Performance After Attacks

```python
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

def encrypt_image(image_path, key):
    img = Image.open(image_path)
    img_array = np.array(img)
    encrypted_array = img_array ^ key
    encrypted_img = Image.fromarray(encrypted_array)
    return encrypted_img, img_array

def decrypt_image(encrypted_img, key):
    encrypted_array = np.array(encrypted_img)
    decrypted_array = encrypted_array ^ key
    decrypted_img = Image.fromarray(decrypted_array)
    return decrypted_img, decrypted_array
```

Measuring Performance After Attacks

```python
# Apply attacks
def apply_attacks(image_array):
    attacks = {}

    noise = np.random.randint(0, 50, image_array.shape)
    noisy_image = np.clip(image_array + noise, 0, 255).astype(np.uint8)
    attacks['noise'] = noisy_image

    blurred = image_array.copy().astype(float)
    for i in range(1, image_array.shape[0]-1):
        for j in range(1, image_array.shape[1]-1):
            blurred[i,j] = np.mean(image_array[i-1:i+2, j-1:j+2])
    attacks['blur'] = blurred.astype(np.uint8)

    # Cropping attack (remove 20% from each side)
    h, w = image_array.shape[:2]
    crop_h, crop_w = int(h * 0.2), int(w * 0.2)
    cropped = image_array[crop_h:h-crop_h, crop_w:w-crop_w]
    # Resize back to original
    from PIL import Image
    cropped_img = Image.fromarray(cropped)
    attacks['crop'] = np.array(cropped_img.resize((w, h)))
    return attacks
```

Measuring Performance After Attacks

```python
# Calculate metrics
def calculate_metrics(original, attacked):
    mse = np.mean((original.astype(float) - attacked.astype(float)) ** 2)

    if mse == 0:
        psnr = 100
    else:
        psnr = 20 * np.log10(255.0 / np.sqrt(mse))

    correlation = np.corrcoef(original.flatten(), attacked.flatten())[0,1]
    hist_orig, _ = np.histogram(original, bins=256, range=(0,255))
    hist_att, _ = np.histogram(attacked, bins=256, range=(0,255))

    prob_orig = hist_orig / np.sum(hist_orig)
    prob_att = hist_att / np.sum(hist_att)

    entropy_orig = -np.sum(prob_orig * np.log2(prob_orig + 1e-10))
    entropy_att = -np.sum(prob_att * np.log2(prob_att + 1e-10))

    return {
        'MSE': mse,
        'PSNR': psnr,
        'Correlation': correlation,
        'Entropy_Original': entropy_orig,
        'Entropy_Attacked': entropy_att
    }
```

Measuring Performance After Attacks

```
image_path = "F:\\13.jpg"
secret_key = 123

encrypted, original_array = encrypt_image(image_path, secret_key)
decrypted, decrypted_array = decrypt_image(encrypted, secret_key)

encrypted_array = np.array(encrypted)
attacks = apply_attacks(encrypted_array)

results = {}
for attack_name, attacked_encrypted in attacks.items():
    # Decrypt the attacked image
    attacked_decrypted, attacked_dec_array = decrypt_image(
        Image.fromarray(attacked_encrypted), secret_key
    )

    metrics = calculate_metrics(original_array, attacked_dec_array)
    results[attack_name] = metrics
```

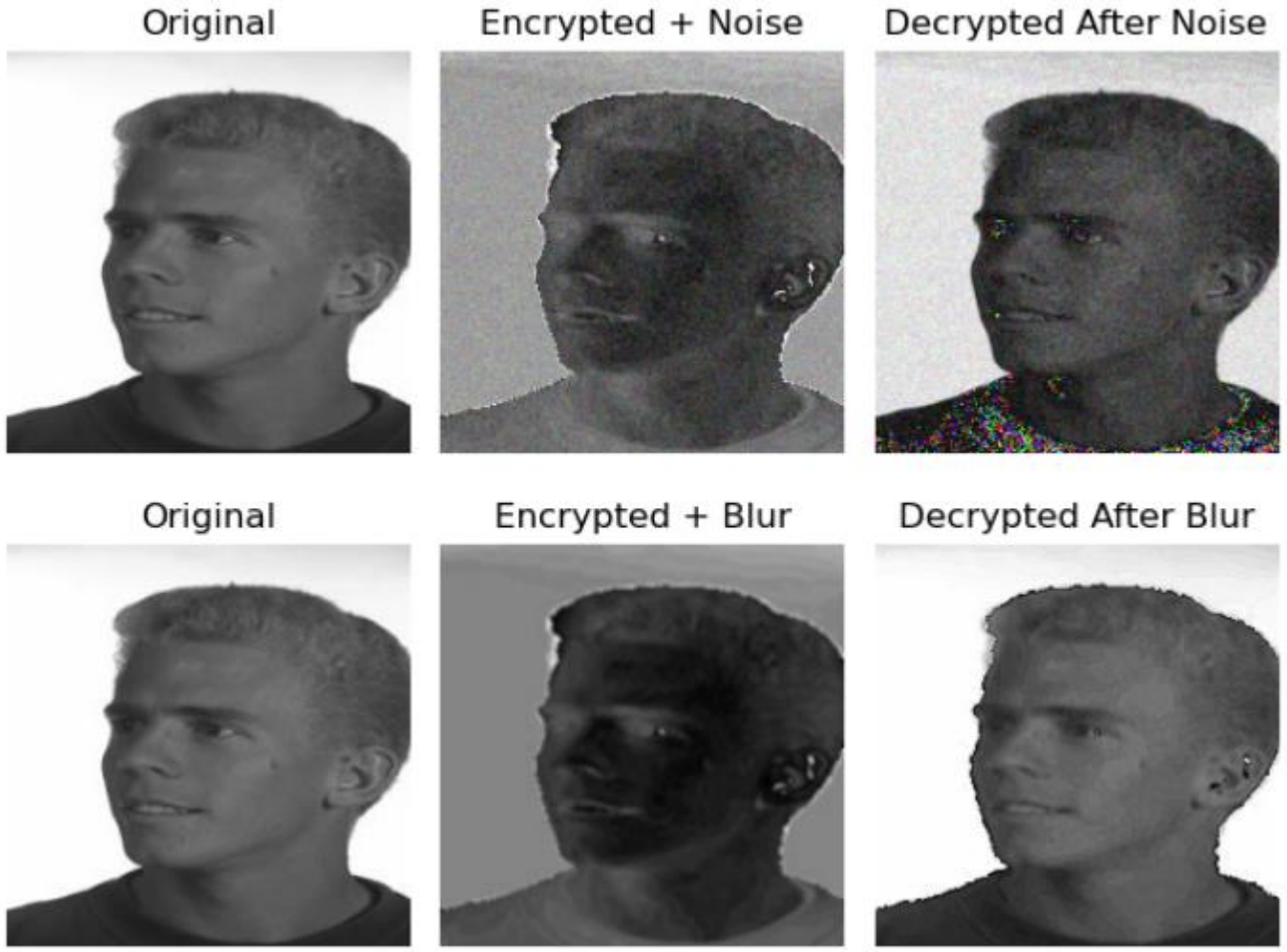
Measuring Performance After Attacks

```
plt.figure()
plt.subplot(1, 3, 1)
plt.imshow(original_array, cmap='gray')
plt.title('Original')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(attacked_encrypted, cmap='gray')
plt.title(f'Encrypted + {attack_name.title()}')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(attacked_dec_array, cmap='gray')
plt.title(f'Decrypted After {attack_name.title()}')
plt.axis('off')
plt.tight_layout()
plt.show()
```

Measuring Performance After Attacks

```python
# Print results
print("=" * 60)
print("ATTACK RESULTS:")
print("=" * 60)
for attack_name, metrics in results.items():
    print(f"\n{attack_name.upper()} ATTACK:")
    print(f"  MSE: {metrics['MSE']:.2f}")
    print(f"  PSNR: {metrics['PSNR']:.2f} dB")
    print(f"  Correlation: {metrics['Correlation']:.4f}")
    print(f"  Original Entropy: {metrics['Entropy_Original']:.4f}")
    print(f"  Attacked Entropy: {metrics['Entropy_Attacked']:.4f}")
```

Measuring Performance After Attacks



Measuring Performance After Attacks

Measuring Performance After Attacks

```
NOISE ATTACK:
  MSE: 1385.83
  PSNR: 16.71 dB
  Correlation: 0.9337
  Original Entropy: 6.2130
  Attacked Entropy: 7.4225

BLUR ATTACK:
  MSE: 261.71
  PSNR: 23.95 dB
  Correlation: 0.9811
  Original Entropy: 6.2130
  Attacked Entropy: 6.1661

CROP ATTACK:
  MSE: 8986.54
  PSNR: 8.59 dB
  Correlation: 0.1940
  Original Entropy: 6.2130
  Attacked Entropy: 6.1206
```