Fourier Transform Analysis and Image Filtering

Introduction

The concept of the Fourier Transform was introduced by the French mathematician Joseph Fourier in the early 19th century. Fourier proposed that any periodic function could be expressed as a sum of sine and cosine functions, a breakthrough that laid the foundation for signal processing and analysis.

The Discrete Fourier Transform (DFT), a numerical method for computing the Fourier Transform for discrete signals, became widely used with the advent of digital computing in the mid-20th century. It was first applied to image processing by researchers in the 1960s, revolutionizing how images are analyzed and manipulated.

Use of Fourier Transform on Images

The Discrete Fourier Transform (DFT) is a powerful mathematical tool used in image processing for several reasons:

- **Transformation to Frequency Domain:**
  - The DFT converts spatial information (pixel intensities) into frequency components, allowing us to analyze the image in terms of its frequency content.
- **Pattern Recognition:**
  - It helps to identify repetitive patterns, edges, and textures that may not be easily visible in the spatial domain.
- **Filtering:**
  - Frequency domain filtering allows for selective enhancement or suppression of specific frequency ranges, making it easier to remove noise or enhance features.
- **Efficient Processing:**
  - Operations such as convolution become simpler in the frequency domain, as they can be performed via multiplication.

What We Observe After Applying DFT

When we apply the DFT to an image, we can observe the following:

- **Magnitude Spectrum:**
  - The DFT reveals the magnitude spectrum, showing the contribution of each frequency to the image. This representation allows us to visualize high-frequency and low-frequency components.
- **Phase Information:**

  - In addition to magnitudes, we obtain phase information, which is crucial for reconstructing the original image after filtering.

- **Noise Identification:**
  - High-frequency components in the magnitude spectrum often correspond to noise and fine details, making it easier to target them during filtering.

Benefits of Using Fourier Transform in Image Processing

Using the Fourier Transform in image processing provides numerous advantages:

- **Enhanced Filtering Capabilities:**
  - Enables the design and application of various filters (low-pass, high-pass, band-pass) to achieve desired image quality.
- **Improved Image Analysis:**
  - Facilitates better understanding of image structures and textures, aiding in segmentation and classification tasks.
- **Efficient Compression:**
  - By transforming images into the frequency domain, we can efficiently compress images, retaining significant features while reducing data size.
- **Restoration of Degraded Images:**
  - Allows for effective restoration of images that have been degraded by noise or blurring through inverse filtering techniques.

Background on Fourier Transform and Filters

2D Discrete Time Fourier Transform (DTFT)

The Discrete Time Fourier Transform (DTFT) represents an image as a sum of complex exponentials of varying magnitudes, frequencies, and phases. For an image $f(x,y)$ of size $M \times N$, the DTFT can be computed using the following equation:

$$F(w_1, w_2) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-jw_1 x} e^{-jw_2 y}$$

Here, $F(w_1, w_2)$ is a complex-valued continuous function that is periodic in both w_1 and w_2 with a period of 2π. Because of the periodicity, usually only the range $-\pi \leq (w_1, w_2) \leq \pi$ is displayed.

The component $F(0,0)$ is the sum of all the values of the image $f(x,y)$. For this reason, $F(0,0)$ is often called the DC component of the Fourier transform.

article amsmath

Magnitude and Phase Spectrum

Since $F(w_1, w_2)$ is a complex-valued function, the magnitude $|F(w_1, w_2)|$ is known as **the Magnitude Spectrum**, and the phase $\angle F(w_1, w_2)$ is known as the Phase Spectrum.

The inverse transform exists. We can recover the image $f(x, y)$ from its spectrum $F(w_1, w_2)$ using the following equation:

$$f(x,y) = \frac{1}{(2\pi)^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} F(w_1, w_2) e^{jw_1 x} e^{jw_2 y} \, dw_1 \, dw_2$$

However, since $F(w_1, w_2)$ is a continuous function of w, this computation is not suitable for computers, as computers are discrete devices.

Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) of an image is given by:

$$\mathcal{F}(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi\left(\frac{ux}{M} + \frac{vy}{N}\right)}$$

where $f(x, y)$ is the original image and $\mathcal{F}(u, v)$ represents the frequency domain.

Inverse Discrete Fourier Transform (IDFT)

To convert the image back to the spatial domain, we use the Inverse Discrete Fourier Transform:

$$f(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} \mathcal{F}(u,v) e^{j2\pi\left(\frac{ux}{M} + \frac{vy}{N}\right)}$$

Fast Fourier Transform (FFT)

The FFT, on the other hand, is a class of algorithms designed to compute the DFT in a much more efficient manner, which reduces the complexity to

$$O(N \log N).$$

This efficiency gain is achieved through a divide-and-conquer approach:

- **Divide:** Split the input sequence into smaller sequences (typically even and odd indexed samples).
- **Conquer:** Recursively compute the DFT of the smaller sequences.
- **Combine:** Merge the results to obtain the DFT of the original sequence.

Key Differences Between DFT and FFT

Efficiency:

- DFT: $O(N^2)$ complexity makes it impractical for large datasets.
- FFT: $O(N \log N)$ complexity allows it to handle large datasets efficiently.

Implementation:

- DFT: Straightforward to implement but slow for large N.

- FFT: More complex to implement due to the recursive nature and the need for bit-reversal in input ordering.

Use Cases:

- DFT: Suitable for small datasets or when a straightforward implementation is required.
- FFT: Widely used in applications like signal processing, image analysis, and solving partial differential equations due to its speed.

Zero Padding

Padding involves adding zeros to expand the image matrix size. This process improves frequency resolution, allowing filters to align better with the image:

$$f_{\text{padded}}(m,n) = \begin{cases} f(m,n) & \text{if } 0 \leq m < M, 0 \leq n < N \\ 0 & \text{otherwise} \end{cases}$$

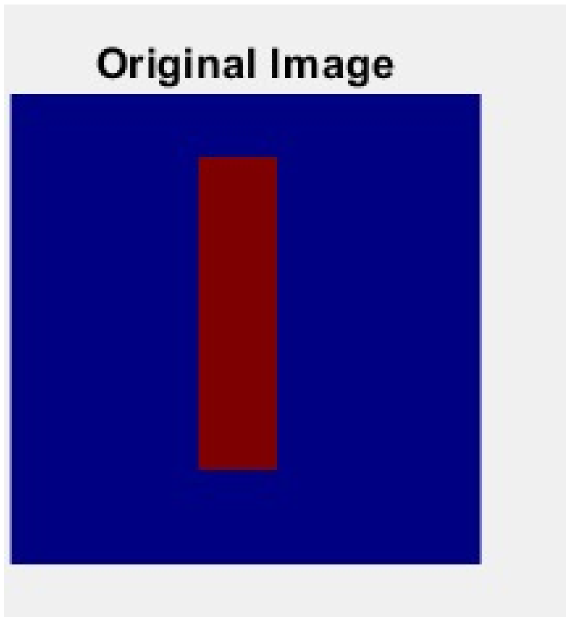
By expanding the image, padding facilitates better filtering and reduces edge artifacts, especially in convolution-based processes.

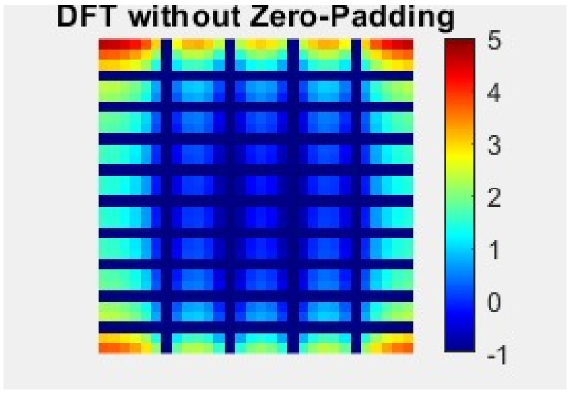
Visualizing the Discrete Fourier Transform

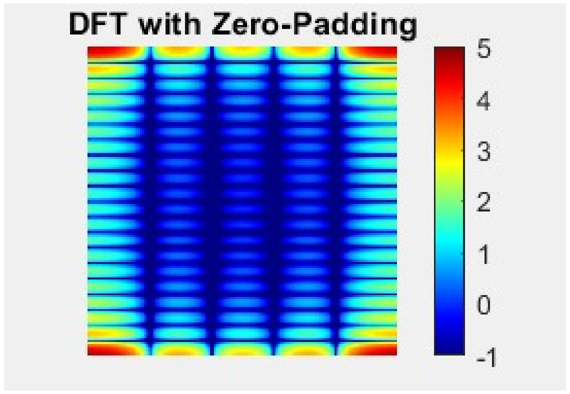
The following MATLAB code demonstrates the creation of a binary image, the application of the Discrete Fourier Transform (DFT) with and without zero-padding, and the centering of the zero-frequency component.

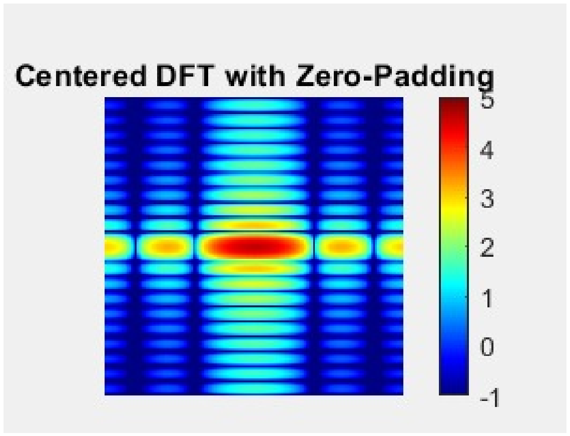
```
% MATLAB code as provided
% Create a binary image with a white rectangle on a black background
f = zeros(30,30);         % Initialize a 30x30 matrix with all zeros (
    black background)
f(5:24,13:17) = 1;        % Set a rectangular area to 1 (white rectangle)
    in the center
subplot(2,2,1);           % Set position for first plot in a 2x2 grid
imshow(f, "InitialMagnification", "fit"); % Display the binary image
colormap(gca, gray);      % Set colormap to gray for the current axis only
title('Original␣Image');

% Compute and display the Discrete Fourier Transform (DFT) without zero-
    padding
F = fft2(f);              % Perform 2D Fourier Transform on the original
    image
F_log = log(abs(F));      % Take the log of the magnitude to improve
    visibility
subplot(2,2,2);           % Set position for second plot
imshow(F_log, [-1 5], "InitialMagnification", "fit"); % Display the DFT
    without padding
colormap(jet);            % Apply a jet colormap for better contrast
colorbar;                 % Add a color bar for reference
title('DFT␣without␣Zero-Padding');

% Compute and display the DFT with zero-padding for higher frequency
    resolution
F_padded = fft2(f, 256, 256); % Perform 2D Fourier Transform with zero-
    padding to 256x256
F_padded_log = log(abs(F_padded)); % Take the log of the magnitude
```

```
subplot(2,2,3);              % Set position for third plot
imshow(F_padded_log, [-1 5], "InitialMagnification", "fit"); % Display the
    DFT with padding
colormap(jet);               % Apply colormap for better visibility
colorbar;                    % Add color bar for reference
title('DFT␣with␣Zero-Padding');

% Shift the zero-frequency component to the center of the spectrum and
   display it
F_shifted = fftshift(F_padded); % Shift the quadrants so zero-frequency is
    at the center
F_shifted_log = log(abs(F_shifted)); % Take the log of the magnitude
subplot(2,2,4);              % Set position for fourth plot
imshow(F_shifted_log, [-1 5], "InitialMagnification", "fit"); % Display
   the centered DFT
colormap(jet);               % Apply colormap for visibility
colorbar;                    % Add color bar for reference
title('Centered␣DFT␣with␣Zero-Padding');
```









Converting an image to grayscale

Converting an image to grayscale reduces complexity, focusing on intensity rather than color. Grayscale images are essential for Fourier Transform operations since they simplify frequency analysis.

Filter Types

Different filters serve unique purposes:

- **Ideal Low-Pass Filter**: Removes high frequencies, retaining smooth features.
- **Gaussian Low-Pass Filter**: Provides a smoother transition between retained and removed frequencies, reducing ringing effects.
- **Gaussian High-Pass Filter**: Emphasizes edges and fine details by suppressing low frequencies.
- **Butterworth Low-Pass Filter**: Balances sharpness and smoothness, with adjustable cutoff steepness.

MATLAB Code

MATLAB Code

```matlab
% Sample MATLAB Code for Image Processing
[filename, pathname] = uigetfile('*.*', 'Select a grayscale Image');
filewithpath = fullfile(pathname, filename);
img = imread(filewithpath);

% Convert to grayscale if necessary
if size(img, 3) == 3
    img = rgb2gray(img);
end

% Add Gaussian noise
noiseVariance = 0.02;
noisyImg = imnoise(img, 'gaussian', 0, noiseVariance);

% Fourier Transform
F = fft2(double(noisyImg));
Fs = fftshift(F);

% Apply filters
cutoff = 30; % Example cutoff value
H = gaussianLowPassFilter(size(noisyImg, 1), size(noisyImg, 2), cutoff);
Fsf = F .* H; % Filtered frequency domain image

% Inverse Fourier Transform
fimg = ifft2(fftshift(Fsf));
imgr = uint8(real(fimg));

% Visualization
figure;
subplot(1, 2, 1); imshow(noisyImg); title('Noisy Image');
subplot(1, 2, 2); imshow(imgr); title('Filtered Image');
```

Filter Functions

Ideal Low-Pass Filter

```
function H = idealLowPassFilter(rows, cols, cutoff)
    [u, v] = meshgrid(-floor(cols/2):floor((cols-1)/2), -floor(rows/2):floor((rows-1)/2));
    D = sqrt(u.^2 + v.^2);
    H = double(D <= cutoff);
end
```

Gaussian Low-Pass Filter

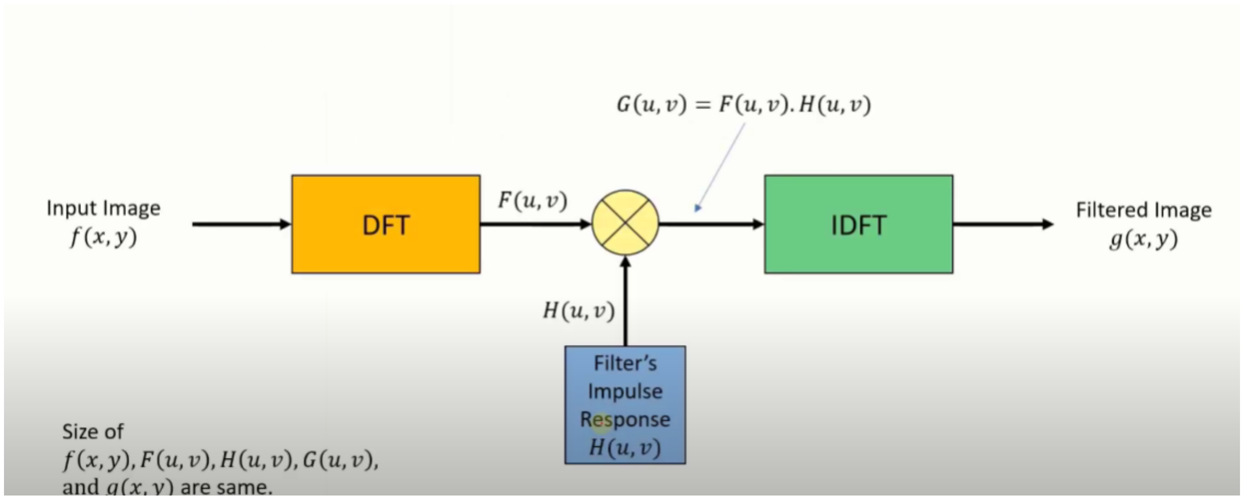
```
function H = gaussianLowPassFilter(rows, cols, cutoff)
    [u, v] = meshgrid(-floor(cols/2):floor((cols-1)/2), -floor(rows/2):floor((rows-1)/2));
    D = sqrt(u.^2 + v.^2);
    H = exp(-D.^2 / (2 * cutoff^2));
end
```

Gaussian High-Pass Filter

```
function H = gaussianHighPassFilter(rows, cols, cutoff)
    [u, v] = meshgrid(-floor(cols/2):floor((cols-1)/2), -floor(rows/2):floor((rows-1)/2));
    D = sqrt(u.^2 + v.^2);
    H = 1 - exp(-D.^2 / (2 * cutoff^2));
end
```

Butterworth Low-Pass Filter

```
function H = butterworthLowPassFilter(rows, cols, cutoff, n)
    [u, v] = meshgrid(-floor(cols/2):floor((cols-1)/2), -floor(rows/2):floor((rows-1)/2));
    D = sqrt(u.^2 + v.^2);
    H = 1 ./ (1 + (D ./ cutoff).^(2 * n));
end
```


$G(u,v) = F(u,v).H(u,v)$
Input Image $f(x,y)$
DFT
$F(u,v)$
IDFT
Filtered Image $g(x,y)$
$H(u,v)$
Filter's Impulse Response $H(u,v)$
Size of $f(x,y), F(u,v), H(u,v), G(u,v)$, and $g(x,y)$ are same.

Explanation of the Steps

Step 1: Selecting and Loading an Image

The code prompts the user to select an image file using:

```
[filename, pathname] = uigetfile('*.*', 'Select a grayscale Image');
filewithpath = fullfile(pathname, filename);
img = imread(filewithpath);
```

Step 2: Converting to Grayscale

If the image is in RGB format, it is converted to grayscale:

```
if size(img, 3) == 3
    img = rgb2gray(img);
end
```

Step 3: Adding Gaussian Noise

Gaussian noise is added to simulate real-world imperfections:

```
noiseVariance = 0.02;
noisyImg = imnoise(img, 'gaussian', 0, noiseVariance);
```

Step 4: Fourier Transform

We calculate the 2D Fourier Transform to analyze the image in the frequency domain:

$$\mathcal{F}(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}$$

```
F = fft2(double(noisyImg));
Fs = fftshift(F);
```

Step 5: Applying Frequency Domain Filters

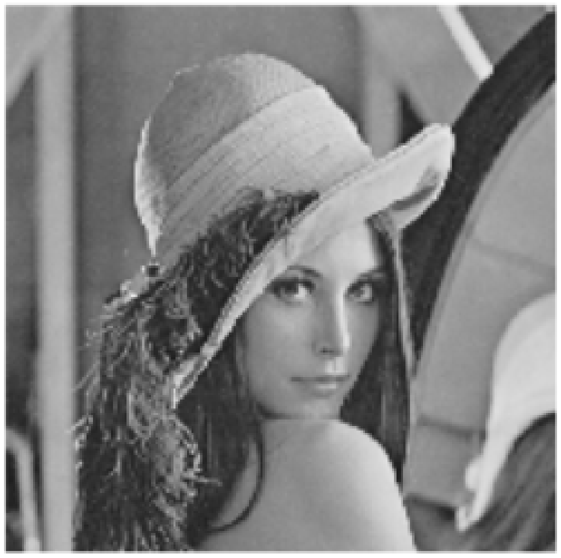
The code allows users to choose from four types of frequency filters.

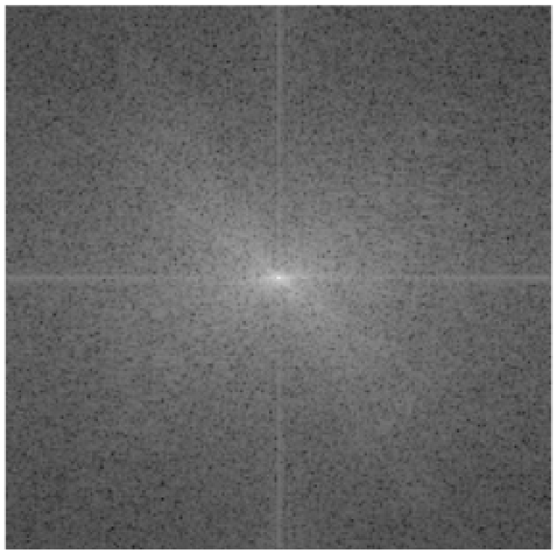
Step 6: Inverse Fourier Transform and Visualization

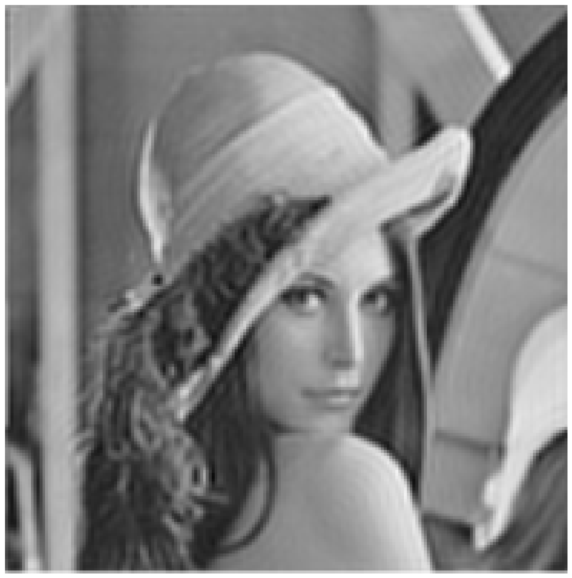
Finally, we apply the Inverse Fourier Transform and display the results.

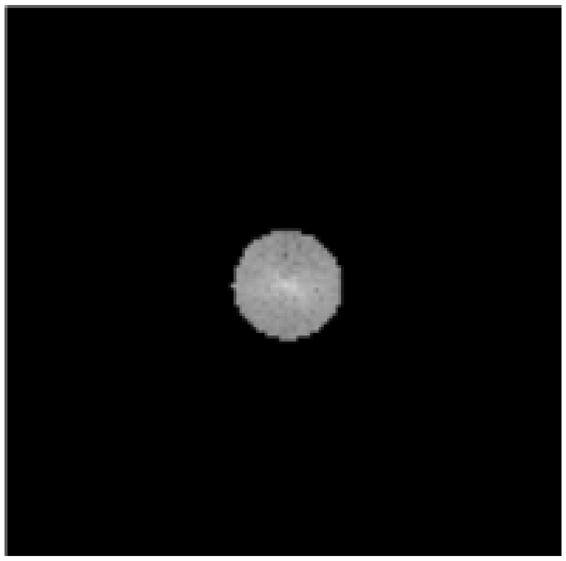
```
fimg = ifft2(fftshift(Fsf));
imgr = uint8(real(fimg));
```

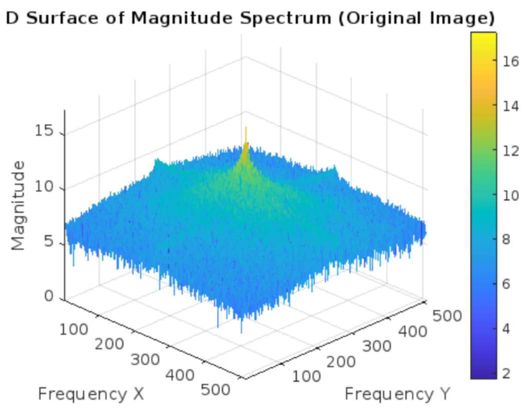
Results:


Original Image

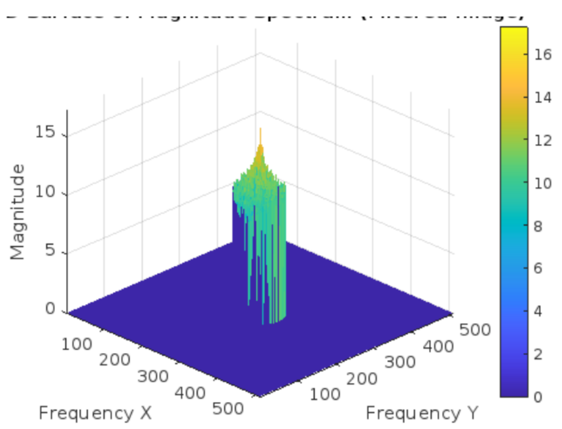

Magnitude Spectrum of Original Image


Filtered Image


Magnitude Spectrum of Filtered Image



3D View of Original Image



3D View of Filtered Image