Digital signal generation and time domain processing case4

Digital Signal Processing Experiment

Student Name: Liu Xingyan
Student Number: 202264690069
Professional Class: 22 Artificial Intelligence Class 1

Instructor: Ning Gengxin

Starting Semester: 2023-2024 second semester of the academic year

School of Future Technology

2024-05-25

1 Finite Impulse Response Digital Filter Design

1.1 Experimental Purpose

1. To deepen the understanding of the common specifications used in digital filter design.
2. To learn the design methods for digital filters, specifically Finite Impulse Response (FIR) filters.

1.2 Experimental Principle

1.2.1 low-pass Finite Impulse Response (FIR)

The design of a low-pass Finite Impulse Response (FIR) digital filter involves several critical specifications which determine its performance in various frequency bands. The specifications are defined as follows:

- **Passband Edge Frequency** (f_p): The frequency at which the passband ends and the transition band begins. It is the highest frequency at which the gain is within the specified passband ripple.
- **Stopband Edge Frequency** (f_s): The frequency at which the stopband begins. It is the lowest frequency at which the gain is within the specified stopband attenuation.
- **Passband Ripple** (δ_p): The allowable variation in the filter's gain in the passband, typically expressed in dB. It reflects the maximum allowable deviation of the filter response from unity within the passband.
- **Peak Passband Ripple** (Δ_p): The maximum deviation in the gain within the passband, often equivalent to δ_p but specified in terms of absolute magnitude rather than a logarithmic scale.
- **Stopband Ripple** (δ_s), with a minimum stopband attenuation (Δ_s): Defines the minimum attenuation required in the stopband, indicating the filter's effectiveness in suppressing unwanted frequencies. δ_s is the ripple in the stopband, usually a very small number close to zero in the linear scale, indicating that the amplitude of the filter's response in the stopband should be very low.

Mathematically, the filter design involves formulating constraints based on these specifications for the desired frequency response $H(e^{j\omega})$. The design objective can typically be represented by an optimization problem:

$$\begin{aligned}
\min_{b} \quad & \max\left\{\left||H(e^{j\omega})| - 1\right| : \omega \in [0, \omega_p]\right\} + \lambda \cdot \max\left\{|H(e^{j\omega})| : \omega \in [\omega_s, \pi]\right\} \\
\text{subject to} \quad & |H(e^{j\omega})| \leq 1 + \delta_p \quad \text{for} \quad \omega \in [0, \omega_p], \\
& |H(e^{j\omega})| \leq \delta_s \quad \text{for} \quad \omega \in [\omega_s, \pi],
\end{aligned}$$

where λ is a weighting factor balancing the trade-off between the passband and stopband specifications, $\omega_p = 2\pi f_p / f_s$ and $\omega_s = 2\pi f_s / f_s$ are the normalized passband and stopband edge frequencies, respectively. The filter coefficients b are optimized to meet these constraints while minimizing the overall error between the actual and desired filter response.

The Window Method for designing Finite Impulse Response (FIR) filters is a popular approach due to its simplicity and effectiveness. The primary concept involves shaping the ideal filter's impulse response using a window function to achieve practical specifications.

1.2.2 Ideal Filter Response

The ideal impulse response $h_d(n)$ of a low-pass filter encapsulates the theoretically perfect characteristics that the filter would exhibit if there were no limitations due to practical implementation. This response is derived from the inverse Fourier transform of the filter's ideal frequency response $H_d(\omega)$.

Mathematical Derivation

The ideal frequency response of a low-pass filter is a rectangular function in the frequency domain, which ideally retains all frequencies below a certain cutoff and attenuates all frequencies above it. Mathematically, it can be expressed as:

$$H_d(\omega) = \begin{cases} 1 & \text{if } |\omega| \leq \omega_c, \\ 0 & \text{if } |\omega| > \omega_c, \end{cases} \tag{1}$$

where ω_c is the cutoff frequency. The corresponding ideal impulse response is given by the inverse Fourier transform of $H_d(\omega)$:

$$h_d(n) = \int_{-\omega_c}^{\omega_c} \frac{1}{2\pi} e^{j\omega n} \, d\omega = \frac{1}{2\pi} \int_{-\omega_c}^{\omega_c} e^{j\omega n} \, d\omega. \tag{2}$$

Sinc Function

The evaluation of the integral leads to the sinc function, which is defined as the sine of a number divided by the number itself. Thus, the impulse response becomes:

$$h_d(n) = \frac{1}{\pi n} \left[e^{j\omega_c n} - e^{-j\omega_c n} \right] = \frac{2}{\pi n} \sin(\omega_c n) = \frac{\omega_c}{\pi} \text{sinc}\left(\frac{\omega_c n}{\pi} \right), \tag{3}$$

where the sinc function is defined as:

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}. \tag{4}$$

This function exhibits the property of an ideal low-pass filter by producing zero crossings at integer multiples of the inverse of the cutoff frequency, except at the origin where it equals one.

Properties of the Sinc Function

The sinc function's behavior in the time domain exhibits oscillations that decay with $\frac{1}{n}$, providing the necessary attenuation in the frequency domain to meet the ideal response characteristics. It is characterized by its sharp central peak at the origin and its rapid decay, which ensures minimal ringing in the practical implementation of the filter.

1.2.3 Window Functions

Window functions play a pivotal role in the design of Finite Impulse Response (FIR) filters by truncating the ideally infinite impulse response, thereby making it finite and manageable for practical implementations. This truncation is necessary to realize the filter physically, but it introduces phenomena such as spectral leakage and scalloping loss, which the window functions aim to control and minimize.

Purpose of Window Functions

The primary purpose of a window function $w(n)$ is to taper the ends of the impulse response, thus reducing the side lobes in the frequency spectrum that result from abrupt truncation. By doing so, the window functions enhance the filter's performance by smoothing transitions between passbands and stopbands.

Commonly Used Window Functions

Several window functions are used in FIR filter design, each with its own characteristics and applications. These include:

- **Rectangular Window:** The simplest form of windowing is the rectangular window, defined as:

$$w(n) = 1 \quad \text{for } 0 \leq n \leq N-1, \tag{5}$$

  where N is the length of the filter. This window offers no tapering and results in significant spectral leakage.

- **Hamming Window:** The Hamming window provides a smoother transition than the rectangular window and is defined by:

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right), \tag{6}$$

  It is widely used due to its ability to reduce the nearest side lobe.

- **Hanning Window:** Also known as the Hann window, it is similar to the Hamming window but with a cosine squared taper:

$$w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{N-1}\right), \tag{7}$$

  The Hanning window offers better sidelobe attenuation than the Hamming window but at the cost of a slightly wider main lobe.

- **Blackman Window:** The Blackman window provides even greater attenuation of side lobes, using a combination of cosine terms:

$$w(n) = 0.42 - 0.5\cos\left(\frac{2\pi n}{N-1}\right) + 0.08\cos\left(\frac{4\pi n}{N-1}\right), \tag{8}$$

  making it suitable for applications requiring very low side lobe levels.

Selection Criteria

The choice of window function depends on the specific requirements of the filter design, particularly the trade-off between main lobe width (resolution) and side lobe level (attenuation). More aggressive windows like the Blackman offer superior side lobe suppression at the expense of frequency resolution.

1.2.4 Design Procedure for FIR Filters Using the Window Method

The design of Finite Impulse Response (FIR) filters using the window method is a systematic process that involves several key steps. Each step is crucial for achieving a balance between the ideal theoretical performance and practical limitations.

Step-by-Step Design Procedure

The procedure for designing an FIR filter by windowing the ideal impulse response can be delineated as follows:

1. **Specification of Design Parameters:** Initially, define the essential filter parameters:
   - Desired cutoff frequency ω_c, which delineates the boundary between the passband and the stopband.
   - Filter length N, which impacts the filter's performance, including its ability to approximate the ideal response and the resolution of the frequency response.
2. **Computation of the Ideal Impulse Response:** The ideal impulse response $h_d(n)$ for a low-pass filter is calculated using:
$$h_d(n) = \frac{2\omega_c}{\pi}\text{sinc}\left(\frac{2\omega_c n}{\pi}\right), \tag{9}$$
where $\text{sinc}(x) = \frac{\sin(x)}{x}$. This response is infinitely long and non-causal.
3. **Selection of the Window Function:** Choose a window function $w(n)$ to taper the infinite duration of $h_d(n)$ and to control spectral leakage. The selection is based on the trade-off between the main lobe width and the side lobe attenuation:
   - **Rectangular window** for no side lobe control but sharp cutoff.
   - **Hamming or Hanning window** for a good balance between main lobe width and side lobe level.
   - **Blackman window** for excellent side lobe suppression.
4. **Application of the Window:** The actual filter coefficients are obtained by element-wise multiplication of the ideal impulse response with the window function:
$$h(n) = h_d(n) \cdot w(n), \tag{10}$$
for $n = 0, 1, \ldots, N - 1$. This step results in a causal and realizable FIR filter whose length is finite and determined by N.

Analytical Considerations

The performance of the designed FIR filter is influenced by the choice of N and the type of window. A larger N can offer better approximation to the ideal response by reducing the width of the main lobe and increasing the roll-off rate of the filter's transition band. However, it also increases the computational complexity. The type of window affects the attenuation of side lobes, thereby influencing the filter's ability to suppress unwanted frequencies.

1.2.5 Design Specifications for FIR Digital Filter

Using MATLAB software, the design specifications for the FIR digital filter are as follows:

Passband Specifications:

- Passband Edge Frequency: $\Omega_{p1} = 0.45\pi$
- Passband Edge Frequency: $\Omega_{p2} = 0.65\pi$
- Passband Ripple: $\alpha_p \leq 1$ dB

Stopband Specifications:

- Stopband Edge Frequency: $\Omega_{s1} = 0.3\pi$
- Stopband Edge Frequency: $\Omega_{s2} = 0.75\pi$
- Stopband Attenuation: $\alpha_s \geq 40$ dB

1.3 Experimental Design and Procedure

The objective of this experiment is to design a Finite Impulse Response (FIR) bandpass filter using MATLAB. The filter is designed to meet specified passband and stopband frequencies and attenuation requirements using the ‘fir1‘ function and a Hamming window.

1.3.1 Specification of Filter Parameters

The filter's design parameters are defined as follows:

- Sampling Frequency $F_s = 1000$ Hz, which is the rate at which the input signal is sampled.
- Passband frequencies $F_{p1} = 0.45 \times F_s/2$ Hz and $F_{p2} = 0.65 \times F_s/2$ Hz, defining the frequency range the filter should pass.
- Stopband frequencies $F_{s1} = 0.3 \times F_s/2$ Hz and $F_{s2} = 0.75 \times F_s/2$ Hz, defining the frequencies the filter should attenuate.
- Passband ripple $A_p = 1$ dB and Stopband attenuation $A_s = 40$ dB, specifying the allowable deviations within the passband and the minimum attenuation in the stopband.

1.3.2 Filter Order Estimation

The order of the filter N is crucial for achieving the desired filter performance and is estimated using the empirical formula:

$$N = \left\lceil \frac{A_s - 8}{2.285 \times (\text{transition\_width}/F_s)} \right\rceil, \tag{11}$$

where transition_width is the minimum of the differences between passband and stopband frequencies.

1.3.3 Filter Design

The FIR bandpass filter is designed using MATLAB's ‘fir1‘ function with the calculated order N and a Hamming window. The normalized passband frequencies are calculated as $f_1 = F_{p1}/(F_s/2)$ and $f_2 = F_{p2}/(F_s/2)$:

```
b = fir1(N, [f1 f2], 'bandpass', hamming(N + 1));
```

1.3.4 Filter Analysis

The characteristics of the designed filter are analyzed using various tools:

- Frequency and phase responses are displayed using MATLAB's `fvtool`.
- Impulse response is visualized to assess the filter's stability and causality.

1.3.5 Results Visualization

Graphs are plotted to visually assess the filter's magnitude and phase responses over the specified frequency range and to observe the impulse response, providing insights into the filter's behavior in the time domain.

```
    specificationsFs = 1000;
    Fp1 = 0.45 * Fs/2;
    Fp2 = 0.65 * Fs/2;
    Fs1 = 0.3 * Fs/2;
    Fs2 = 0.75 * Fs/2;
    Ap = 1;
    As = 40;

    transitionWidth = min([Fp1 - Fs1, Fs2 - Fp2]);
    N = ceil((As - 8) / (2.285 * (transitionWidth / Fs)));

    % Filter design using fir1 with Hamming window
    f1 = Fp1 / (Fs/2);
    f2 = Fp2 / (Fs/2);
    b = fir1(N, [f1 f2], 'bandpass', hamming(N + 1));

    fvtool(b, 1, 'Analysis', 'freq');
    fvtool(b, 1, 'Analysis', 'phase');
    fvtool(b, 1, 'Analysis', 'impulse');

    h = figure;
    freqz(b, 1, 1024, Fs);
    title('Magnitude and Phase Response');
    xlabel('Frequency (Hz)');
    ylabel('Magnitude (dB)');
    h = figure;
    impz(b, 1, 512, Fs);
    title('Impulse Response');
    xlabel('Samples');
    ylabel('Amplitude');
```

Listing 1: Design FIR filter using the window function method

2 Infinite Impulse Response (IIR) Digital Filter Design

Infinite Impulse Response (IIR) digital filters are a fundamental component in digital signal processing, distinguished by their ability to achieve sharp frequency response characteristics with relatively low filter orders. Unlike Finite Impulse Response (FIR) filters, IIR filters have feedback components, allowing the impulse response to theoretically extend indefinitely.

2.1 Fundamental Concepts

The design of IIR filters is fundamentally linked to analog filter design techniques due to the mature theories and well-established methodologies in analog domain. The main principle in IIR digital filter design involves translating an analog filter's specifications into the digital domain using transformations that preserve the frequency response characteristics.

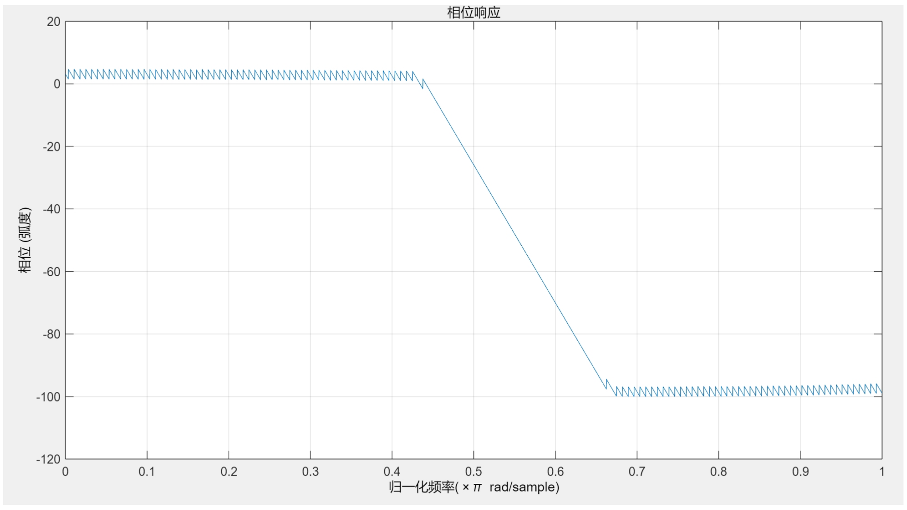

Figure 1: Phase Response

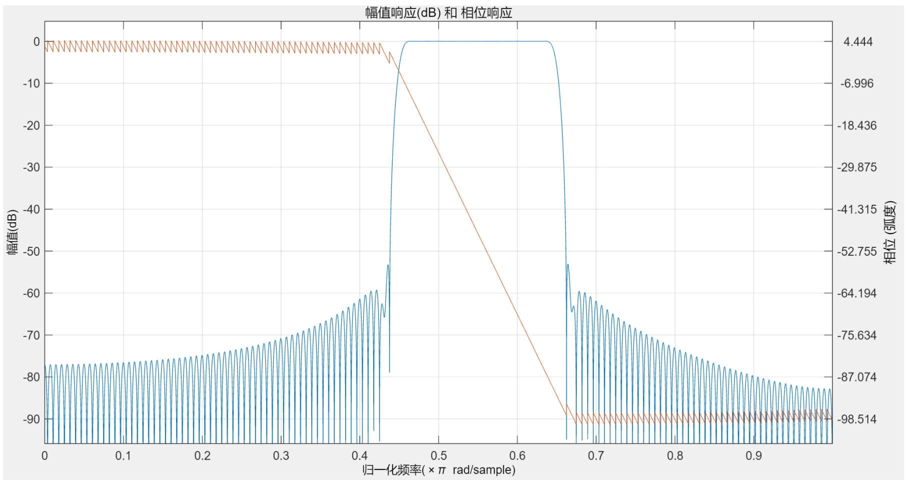

Figure 2: Amplitude response and phase response

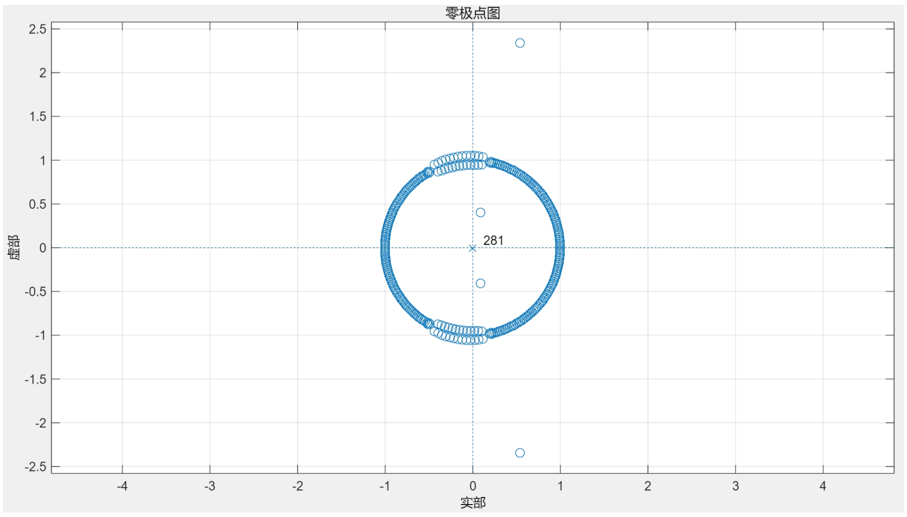

Figure 3: Pole-Zero Diagram

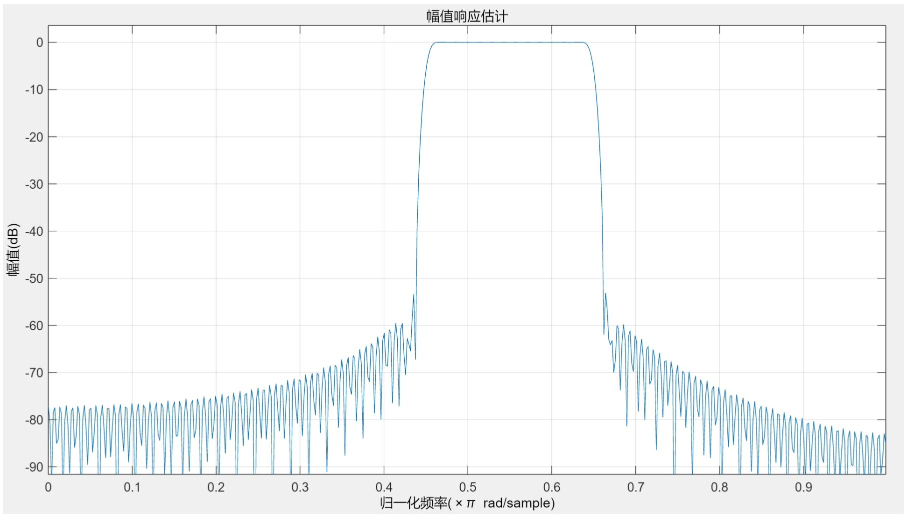

Figure 4: Amplitude response estimation

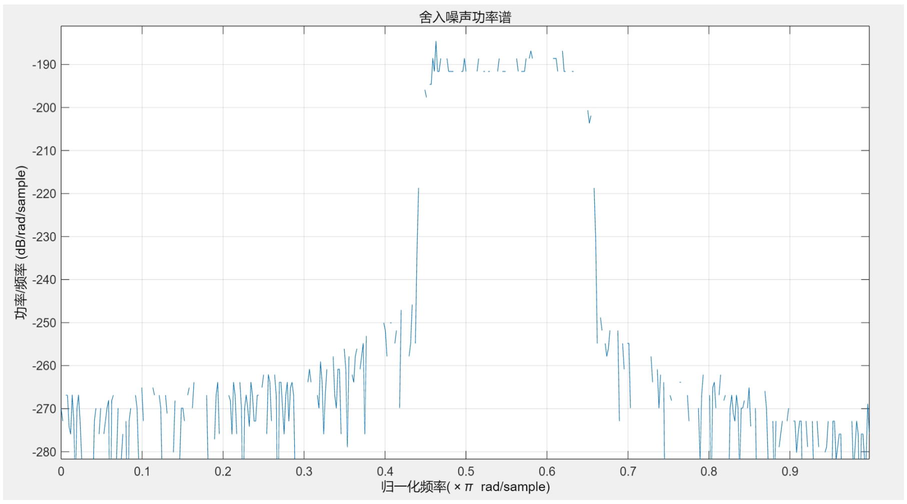

Figure 5: Rounded Noise Power Spectrum

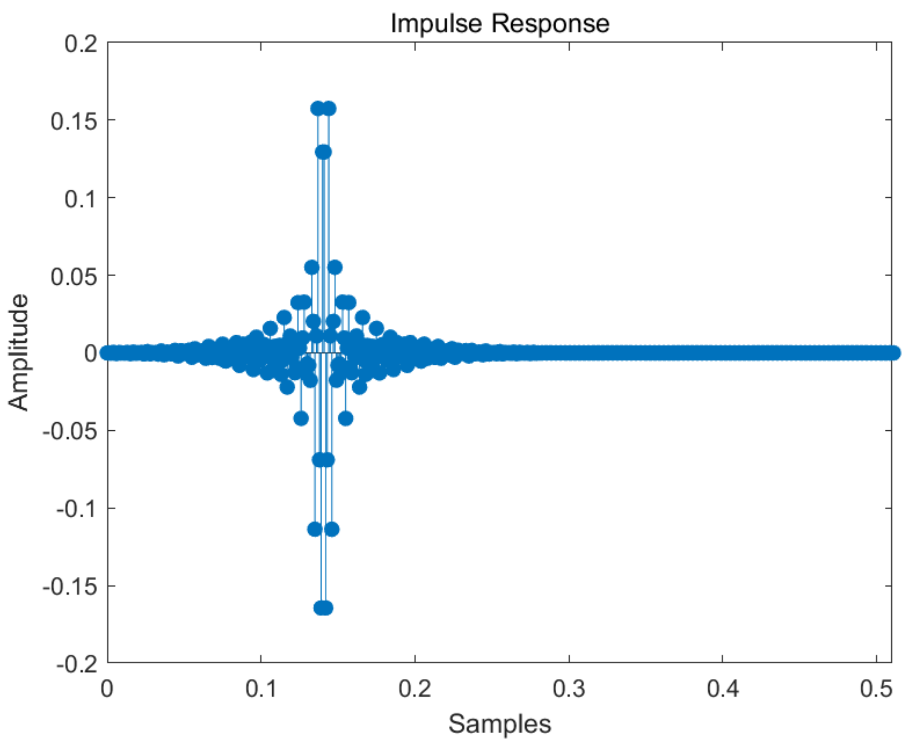

Figure 6: Impulse Response

Transfer Function

The transfer function $H(z)$ of an IIR filter is expressed as a ratio of polynomials in z^{-1}:

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-k}}, \tag{12}$$

where b_k are the feedforward coefficients, a_k are the feedback coefficients, M is the order of the numerator, and N is the order of the denominator. This function dictates the filter's frequency and phase response.

2.1.1 Design Methodologies

Key methodologies in IIR filter design include:

- **Bilinear Transformation:** A technique to convert an analog filter design (usually described by a Laplace transform $H(s)$) into a digital filter design $H(z)$. It maps the s-plane into the z-plane preserving the stability and frequency response:

$$s = \frac{2}{T}\frac{1 - z^{-1}}{1 + z^{-1}}, \tag{13}$$

  where T is the sampling period.

- **Impulse Invariance Method:** This method involves sampling the continuous-time impulse response of the analog filter and using it directly in the digital filter design. The method is exact only for specific types of filters where aliasing does not degrade the frequency response.

2.2 IIR Filter Design Using Bilinear Transformation

One of the cornerstone methodologies in IIR filter design is the Bilinear Transformation. This technique is pivotal in transitioning from analog filter designs, often represented in the Laplace transform domain, to digital filter implementations suitable for digital signal processing.

Principle of Bilinear Transformation

The Bilinear Transformation is a method used to convert the transfer function of an analog filter, $H(s)$, into a digital filter's transfer function, $H(z)$. The transformation is designed to map the complex s-plane into the z-plane, thereby translating the frequency response of the analog filter into the digital domain while preserving the filter's stability characteristics.

2.2.1 Mathematical Foundation

The Bilinear Transformation is mathematically defined as:

$$s = \frac{2}{T}\frac{1 - z^{-1}}{1 + z^{-1}}, \tag{14}$$

where s represents a complex variable in the Laplace transform (continuous-time), z represents a complex variable in the Z-transform (discrete-time), and T is the sampling period which links the continuous and discrete-time frameworks.

2.2.2 Mapping and Frequency Warping

The transformation ensures a one-to-one mapping from the s-plane to the z-plane, specifically mapping the left half of the s-plane (stability region for continuous systems) into the inside of the unit circle in the z-plane (stability region for discrete systems). This mapping is crucial as it preserves the stability of the analog filter when converted into the digital format.

However, a significant aspect of the Bilinear Transformation is the non-linear frequency warping that occurs due to the nature of the mapping. This warping can alter the frequency response characteristics of the filter, particularly near the Nyquist frequency. To counteract this effect, pre-warping techniques are often employed during the design phase to adjust critical frequencies of the analog filter prior to the transformation.

2.2.3 Frequency Pre-warping

To compensate for frequency warping, the critical frequencies in the analog domain are pre-warped using the formula:

$$\Omega = \frac{2}{T}\tan\left(\frac{\omega T}{2}\right), \tag{15}$$

where Ω represents the pre-warped analog frequencies and ω represents the critical frequencies in the digital domain. This adjustment ensures that the desired frequency response is accurately achieved in the digital filter.

2.2.4 Implementation

The practical implementation of the Bilinear Transformation in digital filter design involves substituting s in the analog filter's transfer function with the transformation equation to derive $H(z)$. This process converts the filter's poles and zeros from the s-plane to the z-plane, effectively designing the digital filter.

2.2.5 Frequency Warping and Pre-warping

Due to the nonlinear mapping in bilinear transformation, frequency warping occurs, which may alter the filter's response at different frequencies. Pre-warping compensates for this effect by adjusting the analog filter's cutoff frequencies before transformation:

$$\Omega_c = \frac{2}{T}\tan\left(\frac{\omega_c T}{2}\right), \tag{16}$$

where ω_c is the desired digital cutoff frequency, and Ω_c is the pre-warped analog frequency.

2.2.6 Stability and Causality

One of the key advantages of IIR filters is their recursive nature, which allows for an infinite impulse response using only a finite number of coefficients. However, the designer must ensure the stability of the filter, typically by guaranteeing that all poles of the transfer function are within the unit circle in the z-plane.

2.3 Filter Design Specifications

Using MATLAB, design a band-pass filter with the following specifications:

2.3.1 Passband Specifications:

- Lower passband edge frequency, Ω_{p1}: 0.45π radians per sample.
- Upper passband edge frequency, Ω_{p2}: 0.65π radians per sample.
- Maximum passband ripple, α_p: ≤ 1 dB.

2.3.2 Stopband Specifications:

- Lower stopband edge frequency, Ω_{s1}: 0.3π radians per sample.
- Upper stopband edge frequency, Ω_{s2}: 0.8π radians per sample.
- Minimum stopband attenuation, α_s: ≥ 40 dB.

2.4 Experimental Content: Designing and Analyzing a Band-Pass Butterworth Filter

This experiment involves using MATLAB to design, analyze, and visualize the frequency and impulse responses of a band-pass Butterworth filter. The design specifications are carefully chosen to meet predefined frequency and attenuation requirements.

2.4.1 Filter Specifications and Design

The Butterworth filter, known for its maximally flat magnitude response within the passband, is designed with the following specifications:

- Sampling frequency, $F_s = 2000$ Hz.
- Normalized passband frequencies, $N_p = [0.45, 0.65]$, translating to actual frequencies using $W_p = N_p \times \frac{F_s}{2}$.
- Normalized stopband frequencies, $N_s = [0.3, 0.8]$, calculated as $W_s = N_s \times \frac{F_s}{2}$.
- Passband ripple, $R_p = 1$ dB.

- Stopband attenuation, $R_s = 40$ dB.

The MATLAB function `buttord` is utilized to determine the minimum filter order n and the natural frequency W_n that meet the attenuation and ripple specifications:

```
[n, Wn] = buttord(Wp, Ws, Rp, Rs, 's');
```

Subsequently, the `butter` function designs the filter coefficients:

```
[b, a] = butter(n, Wn, 'bandpass', 's');
```

2.4.2 Filter Analysis and Visualization

To analyze and visualize the filter's performance:

- **Frequency Response:** Using the `fvtool`, the frequency response is displayed with a logarithmic frequency scale. The magnitude and phase responses are also plotted across a defined range of frequencies:

  ```
  [freq, mag] = freqz(b, a, 2048, Fs);
  plot(freq, 20*log10(abs(mag)));
  ```

- **Phase Response:** The phase response is unwrapped and plotted to understand the filter's phase characteristics across the frequency spectrum:

  ```
  [h, w] = freqz(b, a, 2048, Fs);
  phase = unwrap(angle(h));
  frequenciesHz = w * (Fs / (2 * pi));
  plot(frequenciesHz, phase);
  ```

- **Impulse Response:** The impulse response provides insights into the filter's behavior in the time domain:

  ```
  impz(b, a);
  ```

```
    Fs = 2000;
    Np = [0.45 0.65];
    Ns = [0.3 0.8];
    Rp = 1;
    Rs = 40;

    Wp = Np * Fs / 2;
    Ws = Ns * Fs / 2;


    [n, Wn] = buttord(Wp, Ws, Rp, Rs, 's');

    [b, a] = butter(n, Wn, 'bandpass', 's');

    if isa(b, 'dfilt.df2t')
    [b, a] = tf(b);
    end

```

```
    h = fvtool(b, a, 'Fs', Fs);
    set(h, 'Analysis', 'freq');
    set(h, 'FrequencyScale', 'log');

    figure;
    [freq, mag] = freqz(b, a, 2048, Fs);
    plot(freq, 20*log10(abs(mag)));
    title('Magnitude Response');
    xlabel('Frequency (Hz)');
    ylabel('Magnitude (dB)');
    if isa(b, 'digitalFilter')
    [b, a] = tf(b);
    end

    figure;
    [h, w] = freqz(b, a, 2048, Fs);
    phase = unwrap(angle(h));
    frequenciesHz = w * (Fs / (2 * pi));
    plot(frequenciesHz, phase);
    title('Phase Response');
    xlabel('Frequency (Hz)');
    ylabel('Phase (radians)');
    figure;
    impz(b, a);
    title('Impulse Response');
    xlabel('Samples');
    ylabel('Amplitude');
```

Listing 2: Design IIR filter by bilinear transformation method

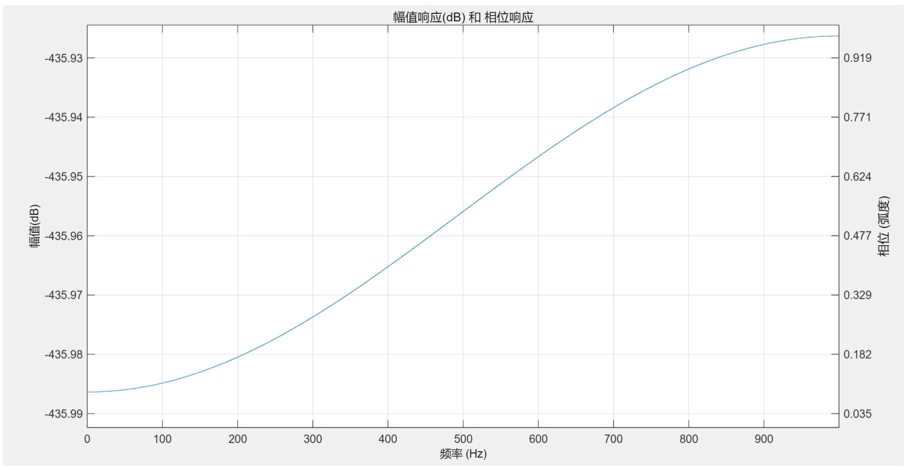

Figure 7: Phase Response

Summary of FIR Filter Design Experiment

The FIR filter design experiment focused on applying theoretical concepts and practical methodologies to design a Finite Impulse Response filter using MATLAB. The primary objectives of the experiment were to deepen the understanding of digital filter specifications and to learn the design process of FIR filters, specifically targeting low-pass filters.

Key steps in the experiment included:

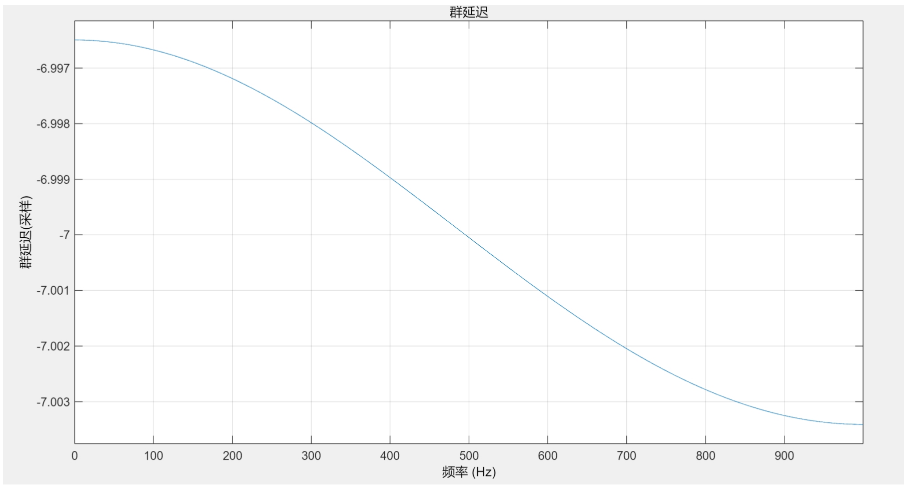

Figure 8: Phase Response

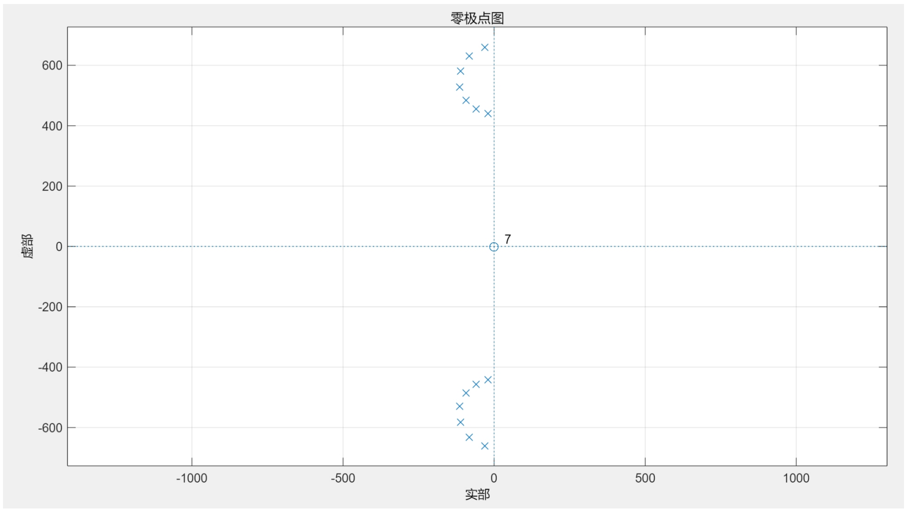

Figure 9: Phase Response

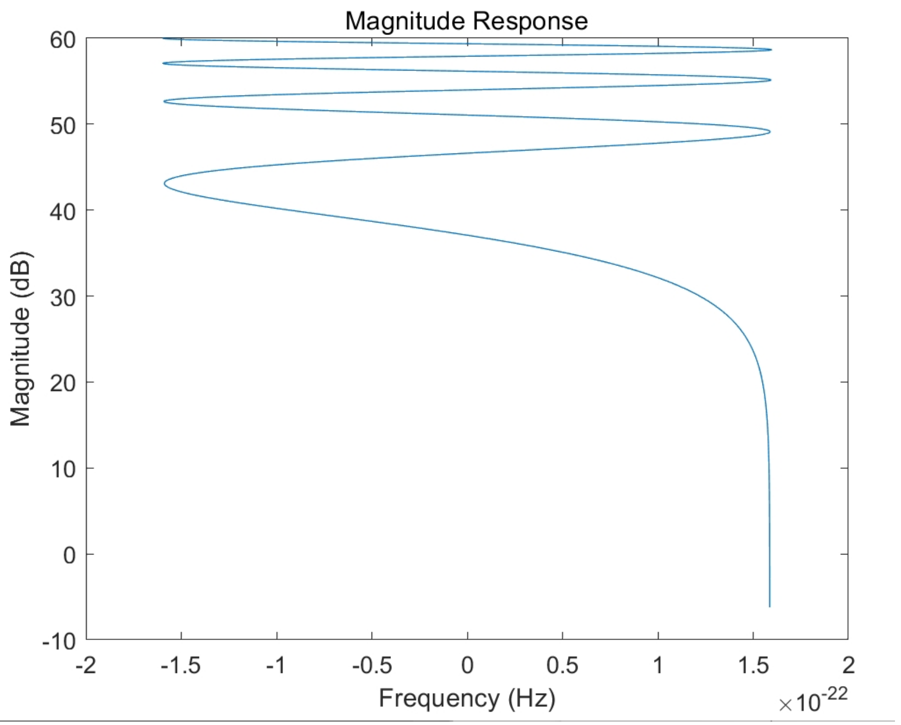

Figure 10: Phase Response

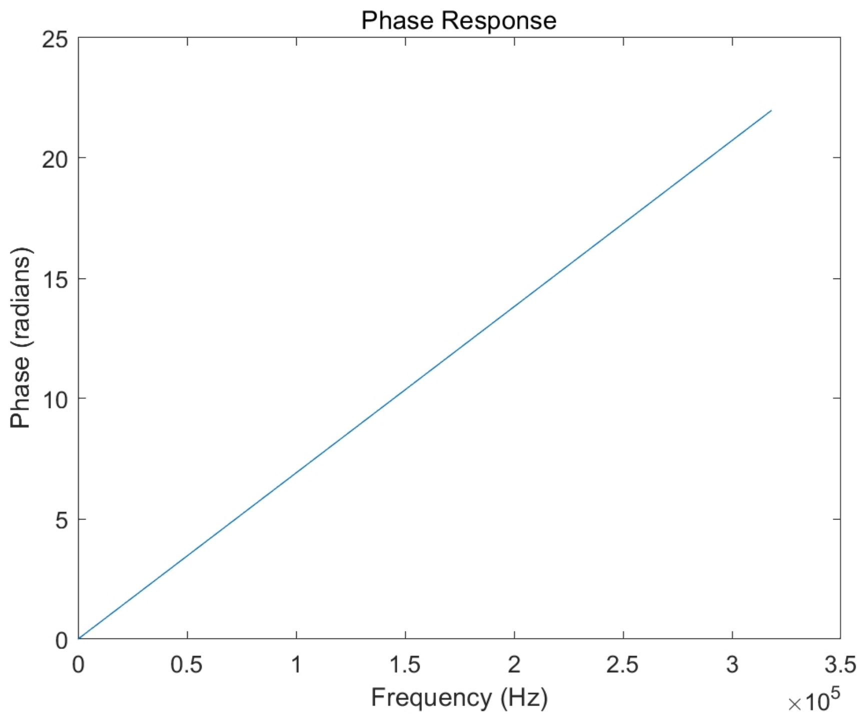

Figure 11: Phase Response

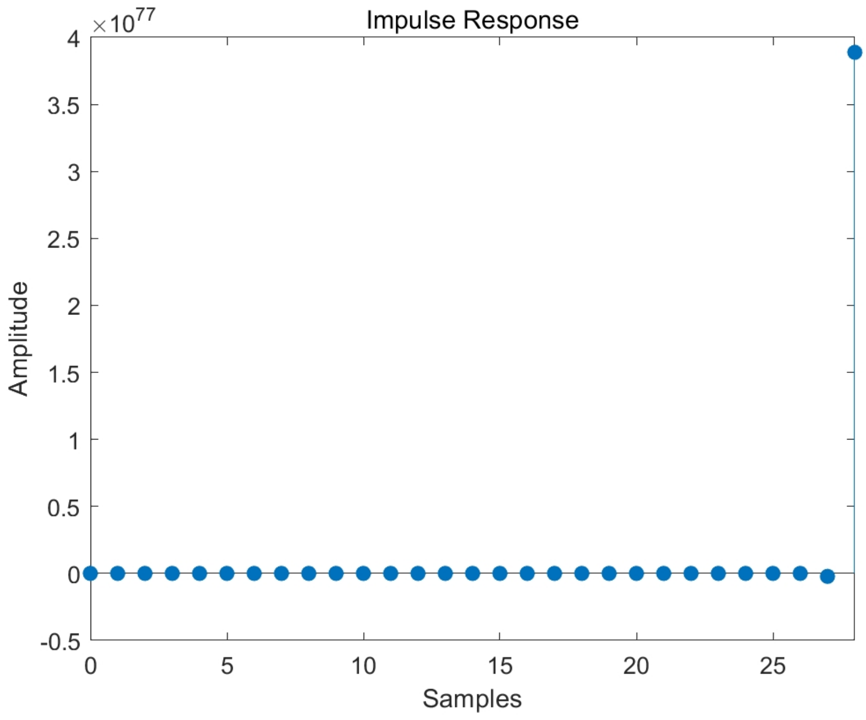

Figure 12: Phase Response

- Defining filter specifications such as passband and stopband frequencies, along with allowable ripples and attenuations.
- Using window functions to shape the ideal impulse response, thereby managing trade-offs between main lobe width and side lobe attenuation.
- Implementing the filter design in MATLAB using the `fir1` function combined with a Hamming window, reflecting a practical approach to filter design that balances performance and complexity.

The outcomes demonstrated the effectiveness of the window method in designing filters that meet specified requirements and provided practical insights into the impact of different window functions on filter performance.

Summary of IIR Filter Design Experiment

The IIR filter design experiment was aimed at understanding and implementing Infinite Impulse Response filter design techniques, with a focus on the bilinear transformation method. The experiment bridged the gap between analog and digital filter design, emphasizing the translation of analog filter specifications into digital filters without losing the essential characteristics of the filter response.

Essential aspects covered in the experiment included:

- Employing the bilinear transformation to convert analog filter designs into digital implementations, ensuring stability and appropriate frequency response.
- Addressing challenges such as frequency warping through pre-warping techniques to maintain the accuracy of the filter's frequency response.
- Analyzing the designed filter's performance using MATLAB tools like `fvtool` and `freqz` to visually and quantitatively assess the magnitude and phase responses.

The experiment highlighted the precision required in designing IIR filters and provided a comprehensive understanding of how transformations from the analog to the digital domain are managed.

Conclusion

Both experiments provided valuable practical experience in digital filter design, illustrating the distinct characteristics and design approaches for FIR and IIR filters. The FIR filter experiment emphasized practical filter implementation using window methods, while the IIR filter experiment focused on theoretical concepts such as bilinear transformation and the challenges of digital implementation. Collectively, these experiments underline the importance of detailed specification and careful method selection in the successful design and application of digital filters in signal processing.

3 Experimental Design: Convolution Processing of Image Signals

3.1 Experimental Purpose

1. To familiarize with common functions and methods of image processing.
2. To develop the ability to solve problems by reviewing literature.
3. To understand image convolution, convolution kernels (operators), and their common uses.

3.2 Experimental Requirements

Given a two-dimensional grayscale or color image:

1. Implement operations using different convolution kernels on the image.
2. Add noise to the image and apply low-pass filtering, high-pass filtering, edge detection, and Gaussian filtering.
3. Apply special effects processing on images, such as blur, mosaic, and various other special effects.
4. Resize the image at any multiple and rotate it at any angle.
5. Create a graphical interactive interface for better user experience.
6. Implement more innovative and creative features for bonus points.

3.3 Experimental Principles

Grayscale Conversion

The initial step in the image processing pipeline involves converting a color image to grayscale. This transformation is crucial because it reduces the computational complexity by collapsing the three color channels (red, green, and blue) into a single channel, thereby simplifying subsequent processing steps without significantly compromising the information content for many applications.

The grayscale conversion is performed using the function `rgb2gray`. This function calculates a weighted sum of the red (I_R), green (I_G), and blue (I_B) channels based on their contributions to human visual perception. The conversion formula is expressed as:

$$I_{\text{gray}} = 0.2989 \cdot I_R + 0.5870 \cdot I_G + 0.1140 \cdot I_B$$

This formula is derived from the luminance component of the YIQ color space, which is designed to reflect the human eye's sensitivity to different wavelengths. The weights 0.2989, 0.5870, and 0.1140 are empirically determined coefficients that approximate the perceptual importance of the red, green, and blue components, respectively. Specifically, the green channel is given the highest weight because the human eye is most sensitive to green light, while the red channel has a moderate contribution and the blue channel has the least.

Mathematically, the transformation can be seen as a dot product of the color vector $\mathbf{I}(x, y) = [I_R(x, y), I_G(x, y), I_B(x, y)]^T$ with the weight vector $\mathbf{w} = [0.2989, 0.5870, 0.1140]^T$:

$$I_{\text{gray}}(x, y) = \mathbf{w}^T\mathbf{I}(x, y) = 0.2989 \cdot I_R(x, y) + 0.5870 \cdot I_G(x, y) + 0.1140 \cdot I_B(x, y)$$

Where (x, y) denotes the pixel coordinates. This operation is applied to each pixel in the image, resulting in a new single-channel image I_{gray} of the same spatial dimensions as the original color image.

The importance of grayscale conversion in image processing lies in its ability to retain essential structural and intensity information while discarding color information that may not be necessary for tasks such as edge detection, segmentation, or morphological operations. By focusing on intensity values, algorithms can be simplified and computational resources can be conserved, which is particularly beneficial for real-time applications and large datasets.

In summary, the grayscale conversion serves as a foundational preprocessing step that prepares the image for more advanced processing techniques by reducing its dimensionality and emphasizing luminance variations. This step leverages the perceptual characteristics of human vision to maintain the critical aspects of the visual information in a more compact form.

Convolution Kernels

Convolution kernels, also known as filters, are fundamental tools in image processing. They are used to perform operations such as blurring, sharpening, and edge detection by modifying pixel values based on their neighbors. Mathematically, the convolution operation for a kernel K applied to an image I is defined as:

$$(I * K)(x, y) = \sum_{i=-m}^{m} \sum_{j=-n}^{n} I(x + i, y + j) \cdot K(i, j)$$

where K is a $(2m + 1) \times (2n + 1)$ matrix and $I(x, y)$ is the pixel value at position (x, y).

The specific convolution kernels used in this experiment are detailed below:

Identity Kernel The identity kernel leaves the image unchanged, acting as a baseline for comparison. It is represented by:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

This kernel maintains the original pixel value and does not alter the image.

Sharpening Kernel The sharpening kernel enhances image details by amplifying high-frequency components. It accentuates edges by subtracting the neighboring pixel values, thus making the transitions between different regions more pronounced. The kernel is:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The central value 5 is the original pixel value weighted heavily, while the -1 values subtract the surrounding pixels' contributions.

Convolution Operation

The convolution operation is a fundamental technique in image processing used to apply various filters to an image. Mathematically, convolution involves sliding a kernel K over the image I and computing the sum of the element-wise multiplications at each position. The discrete two-dimensional convolution of an image I with a kernel K is defined as:

$$(I * K)(x, y) = \sum_{i=-m}^{m} \sum_{j=-n}^{n} I(x + i, y + j) \cdot K(m + i, n + j)$$

where:

- $I(x, y)$ represents the pixel value at position (x, y) in the image.
- $K(m + i, n + j)$ represents the kernel value at position (i, j).
- The kernel K is typically a small matrix of size $(2m + 1) \times (2n + 1)$.

In our implementation, we use the `conv2` function to perform the convolution. This function accepts the grayscale image and the kernel as inputs, along with the 'same' option to ensure the output image retains the same dimensions as the input image. The 'same' option pads the image borders appropriately to preserve the original size.

Padding and Boundary Handling Padding is essential when applying convolution to ensure that the filter can be applied to the edge pixels. Various padding strategies can be employed, such as zero-padding, replication-padding, or reflection-padding. For the 'same' option, zero-padding is commonly used, which extends the image by adding zeros around the border. Formally, if I is an image of size $M \times N$ and K is a kernel of size $(2m + 1) \times (2n + 1)$, the padded image I_p is given by:

$$I_p(x, y) = \begin{cases} I(x, y) & \text{if } 1 \leq x \leq M \text{ and } 1 \leq y \leq N \\ 0 & \text{otherwise} \end{cases}$$

Efficiency Considerations The computational complexity of convolution is proportional to the product of the image and kernel sizes, i.e., $O(M \times N \times (2m+1) \times (2n+1))$. This can be computationally expensive for large images or kernels. Various optimization techniques, such as using the Fast Fourier Transform (FFT) for convolution, can be employed to reduce computational costs, particularly for larger kernels.

Implementation in MATLAB In MATLAB, the `conv2` function simplifies this process. The syntax for performing convolution with a kernel K on an image I while preserving the output size is:

$$\text{output} = \text{conv2}(I, K, 'same')$$

where `output` is the resulting convolved image. The 'same' argument ensures the dimensions of `output` match those of the input image I.

In summary, the convolution operation is a powerful tool in image processing, enabling the application of various filters through mathematical manipulation of pixel values. Understanding the underlying mathematical principles and efficient implementation strategies is crucial for effective image processing.

Gaussian Noise Addition

Gaussian noise is a common noise model used in image processing, representing statistical noise having a probability density function (PDF) equal to that of the normal distribution, also known as the Gaussian distribution. In this experiment, Gaussian noise is added to the grayscale image to simulate real-world noise conditions.

The addition of Gaussian noise can be mathematically described as follows:

$$I_{\text{noisy}}(x, y) = I_{\text{gray}}(x, y) + n(x, y)$$

where $I_{\text{noisy}}(x, y)$ is the intensity of the noisy image at pixel (x, y), $I_{\text{gray}}(x, y)$ is the intensity of the original grayscale image at pixel (x, y), and $n(x, y)$ is the Gaussian noise added to the pixel. The noise $n(x, y)$ follows a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ with mean $\mu = 0$ and variance $\sigma^2 = 0.01$:

$$n(x, y) \sim \mathcal{N}(0, 0.01)$$

The PDF of the Gaussian distribution is given by:

$$P(n) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(n - \mu)^2}{2\sigma^2}\right)$$

For this experiment, the mean μ is zero, and the variance σ^2 is 0.01, indicating that the noise values are centered around zero with a standard deviation of $\sigma = \sqrt{0.01} = 0.1$. The introduction of Gaussian noise adds variability to the pixel intensities, simulating imperfections that occur in real-world image acquisition processes due to sensor noise, transmission errors, or other factors.

The `imnoise` function in MATLAB is used to add this noise to the grayscale image. It modifies each pixel intensity $I_{\text{gray}}(x, y)$ by adding a random value $n(x, y)$ drawn from the specified Gaussian distribution. The resulting noisy image can be represented as a matrix where each element has been perturbed by a corresponding noise value.

This process is crucial for testing the robustness of image processing algorithms under realistic conditions where noise is present. By adding Gaussian noise, we can evaluate the effectiveness of various filtering techniques in noise reduction and edge preservation, which are essential for applications such as image enhancement, segmentation, and feature extraction.

Convolution Kernels

Convolution kernels are used to filter the image. The Gaussian kernel smooths the image by averaging pixel values with their neighbors:

$$K_{\text{gaussian}} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

The high-pass kernel enhances edges by emphasizing the central pixel and subtracting neighboring pixel values:

$$K_{\text{highpass}} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Filtering and Edge Detection

Filtering and edge detection are critical steps in image processing for noise reduction and feature enhancement. The convolution operation with different kernels serves to apply these filters effectively.

Low-Pass Filtering Low-pass filtering is performed using a Gaussian kernel, which smooths the image by averaging the pixel values with their neighbors. This process helps in reducing noise while preserving the overall structure of the image. The Gaussian kernel K_{gaussian} is defined as:

$$K_{\text{gaussian}} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

The convolution operation with the Gaussian kernel for low-pass filtering is mathematically expressed as:

$$I_{\text{lowpass}}(x, y) = \sum_{i=-1}^{1}\sum_{j=-1}^{1} I_{\text{noisy}}(x+i, y+j) \cdot K_{\text{gaussian}}(i+1, j+1)$$

where $I_{\text{noisy}}(x, y)$ is the intensity of the noisy image at pixel (x, y), and $K_{\text{gaussian}}(i+1, j+1)$ are the weights of the Gaussian kernel.

High-Pass Filtering High-pass filtering is used to enhance the details and edges of an image by amplifying the high-frequency components. The high-pass kernel K_{highpass} is designed to highlight the intensity variations by subtracting the neighboring pixel values from the central pixel value:

$$K_{\text{highpass}} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

The convolution operation with the high-pass kernel for enhancing image details is given by:

$$I_{\text{highpass}}(x, y) = \sum_{i=-1}^{1}\sum_{j=-1}^{1} I_{\text{gray}}(x+i, y+j) \cdot K_{\text{highpass}}(i+1, j+1)$$

where $I_{\text{gray}}(x, y)$ is the intensity of the original grayscale image at pixel (x, y), and $K_{\text{highpass}}(i+1, j+1)$ are the weights of the high-pass kernel.

Edge Detection with Sobel Operator Edge detection is a technique used to identify and locate sharp discontinuities in an image. The Sobel operator is commonly used for this purpose, as it calculates the gradient of the image intensity at each pixel, which corresponds to the edges.

The Sobel operator uses two kernels to compute the gradient in the x and y directions:

$$K_{\text{sobel\_x}} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad K_{\text{sobel\_y}} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The gradient in the x direction, $G_x(x, y)$, is calculated as:

$$G_x(x, y) = \sum_{i=-1}^{1}\sum_{j=-1}^{1} I_{\text{gray}}(x+i, y+j) \cdot K_{\text{sobel\_x}}(i+1, j+1)$$

Similarly, the gradient in the y direction, $G_y(x, y)$, is computed as:

$$G_y(x, y) = \sum_{i=-1}^{1}\sum_{j=-1}^{1} I_{\text{gray}}(x+i, y+j) \cdot K_{\text{sobel\_y}}(i+1, j+1)$$

The magnitude of the gradient at each pixel, which represents the edge strength, is given by:

$$G(x, y) = \sqrt{G_x(x, y)^2 + G_y(x, y)^2}$$

This magnitude image highlights the edges in the original image, with higher values indicating stronger edges. The Sobel operator is effective in detecting edges while also smoothing the image to reduce noise.

In summary, filtering and edge detection are fundamental processes in image processing. The low-pass filter reduces noise, the high-pass filter enhances details, and the Sobel operator identifies edges by computing the image gradient.

Mean Squared Error (MSE) Calculation

The mean squared error is used to quantify the difference between the original and processed images. It is defined as:

$$\mathrm{MSE}(I_1, I_2) = \frac{1}{MN} \sum_{x=1}^{M} \sum_{y=1}^{N} (I_1(x, y) - I_2(x, y))^2$$

where I_1 and I_2 are the original and processed images, respectively, and M and N are the dimensions of the images.

Gaussian Blur

Gaussian blur is used to smooth images by reducing noise and detail. The Gaussian kernel K_{gaussian} used in this experiment is created using the `fspecial` function with a size of 5×5 and a standard deviation $\sigma = 2$:

$$K_{\text{gaussian}} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

This kernel is applied to the image using convolution:

$$I_{\text{gaussian}}(x, y) = \sum_{i=-2}^{2} \sum_{j=-2}^{2} I_{\text{gray}}(x + i, y + j) \cdot K_{\text{gaussian}}(i + 3, j + 3)$$

Motion Blur

Motion blur is an effect that simulates the blurring that occurs when a camera or the object being photographed moves during the exposure. This blur can be modeled by convolving the image with a motion blur kernel. The motion blur kernel K_{motion} is created using the `fspecial` function in MATLAB, which generates a linear filter corresponding to the specified motion parameters.

The motion blur kernel can be described mathematically as follows:

$$K_{\text{motion}}(x, y) = \begin{cases} \frac{1}{L} & \text{if } (x, y) \text{ lies on the motion path} \\ 0 & \text{otherwise} \end{cases}$$

where L is the length of the motion path. For this experiment, $L = 20$ pixels, and the motion is at an angle of 45 degrees.

The kernel K_{motion} can be represented as a matrix where non-zero values are distributed along a line segment of length L at a specified angle. The matrix elements corresponding to the motion path are assigned the value $\frac{1}{L}$, ensuring that the sum of all elements in the kernel equals 1, preserving the overall image brightness.

The convolution of the image $I_{\text{gray}}(x, y)$ with the motion blur kernel K_{motion} can be expressed as:

$$I_{\text{motion\_blur}}(x, y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} I_{\text{gray}}(x + i, y + j) \cdot K_{\text{motion}}(i, j)$$

where k is the half-size of the kernel, i.e., $k = \left\lfloor \frac{L}{2} \right\rfloor$.

In this particular case, the motion blur kernel is generated for a motion length of 20 pixels at an angle of 45 degrees. The effective kernel K_{motion} can be visualized as:

$$K_{\text{motion}} = \frac{1}{20} \begin{bmatrix} 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & 1 & 0 & \cdots & 0 & 0 \\ \vdots & 1 & 0 & \cdots & 0 & 0 & \vdots \\ 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \vdots & 0 & 0 & \cdots & 1 & 0 \\ \vdots & 0 & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & 1 \end{bmatrix}$$

This kernel is applied to the grayscale image using convolution, resulting in an image where the motion blur effect is evident along the specified direction. The convolution operation effectively averages the pixel values along the motion path, creating a streaking effect that simulates the appearance of motion blur.

By using the `fspecial` function and the `imfilter` function in MATLAB, the motion blur can be efficiently applied, demonstrating the impact of camera or object movement on the captured image.

Mosaic Effect

The mosaic effect is a technique used to transform an image by reducing its resolution in localized regions, giving it a pixelated or tiled appearance. This is achieved by dividing the image into non-overlapping blocks and replacing each block with its average intensity value. This process effectively reduces the detail within each block, creating a visual effect reminiscent of a mosaic.

Mathematically, the mosaic effect can be described as follows. Consider an image $I_{\text{gray}}(x, y)$ of dimensions $M \times N$. We divide this image into blocks of size $B \times B$, where $B = 10$ in this case. For each block, we compute the average intensity and assign this value to all pixels within the block.

Formally, let (i, j) denote the top-left corner of a block, and (k, l) denote the pixel coordinates within the block. The average intensity for the block starting at (i, j) is given by:

$$\bar{I}(i, j) = \frac{1}{B^2} \sum_{k=i}^{i+B-1} \sum_{l=j}^{j+B-1} I_{\text{gray}}(k, l)$$

where $B = 10$. This average intensity $\bar{I}(i, j)$ is then assigned to all pixels within the block:

$$I_{\text{mosaic}}(i : i + B - 1, j : j + B - 1) = \bar{I}(i, j)$$

In explicit terms, the assignment can be written as:

$$I_{\text{mosaic}}(k, l) = \bar{I}(i, j) \quad \text{for } i \leq k < i + B \text{ and } j \leq l < j + B$$

This process is repeated for all blocks in the image. The resulting image I_{mosaic} retains the overall structure of the original image but with reduced detail within each block, creating the characteristic mosaic effect.

To ensure a comprehensive understanding, let's expand on the steps:

1. Block Division: The image is divided into blocks of size $B \times B$. Each block contains B^2 pixels.

2. Average Calculation: For each block, the average intensity is calculated by summing the intensities of all pixels within the block and dividing by the total number of pixels B^2:

$$\bar{I}(i, j) = \frac{1}{100} \sum_{k=i}^{i+9} \sum_{l=j}^{j+9} I_{\text{gray}}(k, l)$$

3. Pixel Assignment: All pixels within the block are then set to this average intensity value:

$$I_{\text{mosaic}}(i : i + 9, j : j + 9) = \bar{I}(i, j)$$

The result is a new image I_{mosaic} where each block of size 10×10 pixels has a uniform intensity value, representing the average of the original intensities within that block.

This method effectively reduces the resolution locally, which can be useful in various applications such as artistic effects, data obfuscation, or preparing images for further processing steps that require reduced detail.

Emboss Effect

The emboss effect is an image processing technique that highlights the edges and transitions in an image, creating a three-dimensional (3D) appearance. This effect is achieved through convolution with an emboss kernel that emphasizes intensity changes in specific directions. The resulting image appears as if the details are raised or etched, providing a textured look.

The emboss kernel K_{emboss} used in this experiment is designed to detect edges by computing the difference in pixel intensities along a certain direction. The kernel is defined as:

$$K_{\text{emboss}} = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

This kernel emphasizes positive changes in intensity in the diagonal direction from the top-left to the bottom-right. The values in the kernel are chosen such that negative weights (-2 and -1) are assigned to pixels preceding the central pixel, while positive weights (1 and 2) are assigned to pixels following the central pixel. The central pixel is assigned a weight of 1.

The convolution operation for applying the emboss effect is mathematically expressed as:

$$I_{\text{emboss}}(x, y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} I_{\text{gray}}(x + i, y + j) \cdot K_{\text{emboss}}(i + 2, j + 2)$$

where $I_{\text{emboss}}(x, y)$ is the intensity of the embossed image at pixel (x, y), and $I_{\text{gray}}(x + i, y + j)$ is the intensity of the grayscale image at the neighboring pixel $(x + i, y + j)$.

To better understand this operation, let's break it down:

1. Kernel Application: For each pixel (x, y) in the grayscale image, the emboss kernel K_{emboss} is centered on (x, y). The intensity of the pixels within the kernel's window is multiplied by the corresponding kernel values.

2. Sum of Products: The products of the pixel intensities and the kernel values are summed to compute the new intensity value for the pixel (x, y) in the embossed image.

3. Edge Emphasis: The kernel's design ensures that areas with rapid intensity changes (edges) result in high absolute values in the output, thus highlighting these regions. Positive changes in intensity (lighter to darker transitions) are given more weight, creating the raised effect.

The emboss effect is widely used in graphic design and image editing to create visually appealing textures and to emphasize specific features within an image, making it a valuable tool in various applications.

Sharpening

Sharpening is a technique used in image processing to enhance the edges and fine details of an image, making it appear more defined and crisp. This is achieved by emphasizing the high-frequency components, which correspond to rapid intensity changes.

The sharpening kernel K_{sharpen} is designed to highlight the differences between a pixel and its neighboring pixels. It is defined as:

$$K_{\text{sharpen}} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

This kernel has a central value of 5, which significantly amplifies the central pixel's intensity. The surrounding values of -1 subtract the neighboring pixel intensities, thus enhancing the contrast between the central pixel and its neighbors.

The convolution operation to apply the sharpening effect is expressed mathematically as:

$$I_{\text{sharpen}}(x, y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} I_{\text{gray}}(x + i, y + j) \cdot K_{\text{sharpen}}(i + 2, j + 2)$$

where $I_{\text{sharpen}}(x, y)$ is the intensity of the sharpened image at pixel (x, y), and $I_{\text{gray}}(x + i, y + j)$ is the intensity of the grayscale image at the neighboring pixel $(x + i, y + j)$.

To elaborate on the process:

1. Kernel Centering: For each pixel (x, y) in the grayscale image, the sharpening kernel K_{sharpen} is centered on (x, y). The intensity values of the pixels within the kernel's window are multiplied by the corresponding kernel values.

2. Summation: The products of the pixel intensities and the kernel values are summed to compute the new intensity value for the pixel (x, y) in the sharpened image.

Sharpening is a fundamental technique in image processing used to improve the visual quality of images. It is particularly useful in applications where clarity and detail are essential, such as in medical imaging, satellite image analysis, and various fields of computer vision.

By applying the sharpening kernel through convolution, we enhance the image's edges, making it more visually appealing and informative.

Image Resizing

Image resizing involves changing the dimensions of an image. This can be done by scaling the image up or down. The resizing operation is performed using interpolation methods to estimate the pixel values in the resized image.

Doubling the Image Size To double the size of the grayscale image, we use the `imresize` function with a scaling factor of 2. The resized image I_{double} has dimensions twice that of the original image:

$$I_{\text{double}}(x, y) = I_{\text{gray}}\left(\frac{x}{2}, \frac{y}{2}\right)$$

where (x, y) are the pixel coordinates in the resized image, and the function `imresize` performs interpolation to determine the intensity values.

Halving the Image Size To half the size of the grayscale image, we use the `imresize` function with a scaling factor of 0.5. The resized image I_{half} has dimensions half that of the original image:

$$I_{\text{half}}(x, y) = I_{\text{gray}}(2x, 2y)$$

Here, (x, y) are the pixel coordinates in the resized image, and interpolation is used to compute the intensity values.

Image Rotation

Image rotation involves rotating an image by a specified angle around its center. This is achieved by applying a geometric transformation that maps the coordinates of the original image to the new coordinates after rotation.

Rotation by 45 Degrees The image is rotated by 45 degrees using the `imrotate` function. The new coordinates (x', y') of a pixel after rotation are given by:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

where $\theta = 45°$.

Rotation by 90 Degrees The image is rotated by 90 degrees using the same transformation matrix with $\theta = 90°$:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Rotation by 180 Degrees The image is rotated by 180 degrees using the transformation matrix with $\theta = 180°$:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

These transformations result in images that are rotated by the specified angles around the center of the original image.

These image transformations are fundamental techniques in image processing, each serving a unique purpose in manipulating and analyzing images.

3.4 Methodology

```matlab
image = imread('Lenna.png');
gray_image = rgb2gray(image);
identity_kernel = [0 0 0; 0 1 0; 0 0 0];
sharpening_kernel = [0 -1 0; -1 5 -1; 0 -1 0];
edge_detection_kernel = [-1 -1 -1; -1 8 -1; -1 -1 -1];
gaussian_kernel = [1 2 1; 2 4 2; 1 2 1] / 16;

identity_image = conv2(double(gray_image), identity_kernel, 'same');
sharpened_image = conv2(double(gray_image), sharpening_kernel, 'same');
edge_detected_image = conv2(double(gray_image), edge_detection_kernel, '
    same');
gaussian_blurred_image = conv2(double(gray_image), gaussian_kernel, 'same
    ');

figure;
subplot(2, 3, 1);
imshow(gray_image);
```

```
title('Original Image');

subplot(2, 3, 2);
imshow(uint8(identity_image));
title('Identity Filter');

subplot(2, 3, 3);
imshow(uint8(sharpened_image));
title('Sharpened Image');

subplot(2, 3, 4);
imshow(uint8(edge_detected_image));
title('Edge Detection');

subplot(2, 3, 5);
imshow(uint8(gaussian_blurred_image));
title('Gaussian Blur');

imwrite(uint8(identity_image), 'identity_image.png');
imwrite(uint8(sharpened_image), 'sharpened_image.png');
imwrite(uint8(edge_detected_image), 'edge_detected_image.png');
imwrite(uint8(gaussian_blurred_image), 'gaussian_blurred_image.png');
```

Listing 3: Image Processing with Convolution Kernels

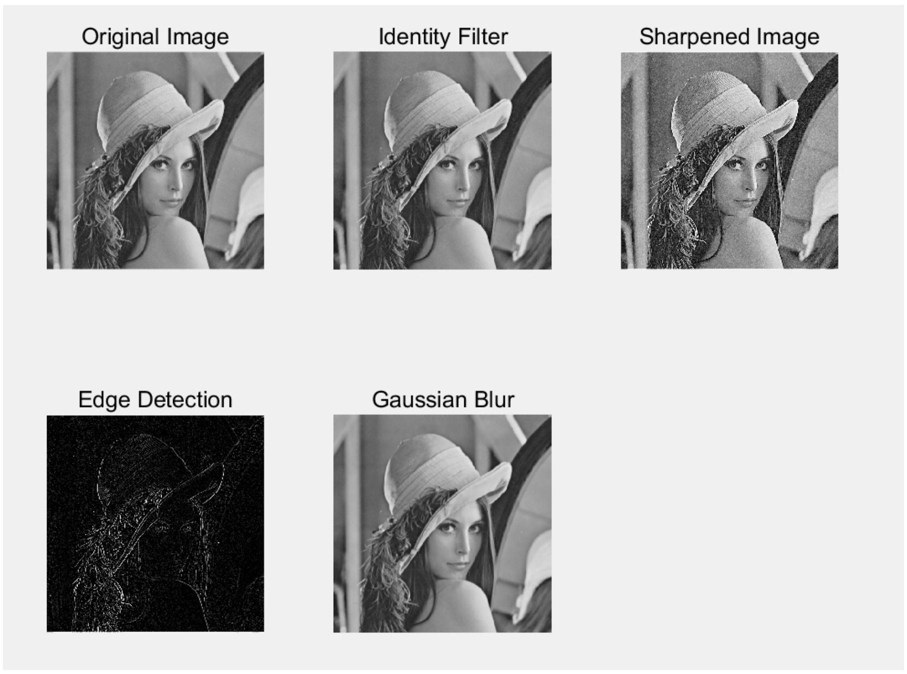

Figure 13: Image Processing with Convolution Kernels

```
image = imread('Lenna.png');
gray_image = rgb2gray(image);


noisy_image = imnoise(gray_image, 'gaussian', 0, 0.01);

gaussian_kernel = [1 2 1; 2 4 2; 1 2 1] / 16;
```

```
high_pass_kernel = [-1 -1 -1; -1 9 -1; -1 -1 -1];


low_pass_filtered_image = imfilter(noisy_image, gaussian_kernel, '
    replicate');

high_pass_filtered_image = imfilter(gray_image, high_pass_kernel, '
    replicate');

sobel_x = [-1 0 1; -2 0 2; -1 0 1];
sobel_y = [-1 -2 -1; 0 0 0; 1 2 1];
edges_x = imfilter(double(gray_image), sobel_x, 'replicate');
edges_y = imfilter(double(gray_image), sobel_y, 'replicate');
edges = sqrt(edges_x.^2 + edges_y.^2);

original_noisy_mse = immse(double(noisy_image), double(gray_image));
low_pass_mse = immse(double(low_pass_filtered_image), double(gray_image))
    ;
high_pass_mse = immse(double(high_pass_filtered_image), double(gray_image
    ));

figure;
subplot(2, 3, 1);
imshow(gray_image);
title('Original Image');

subplot(2, 3, 2);
imshow(noisy_image);
title(sprintf('Noisy Image (MSE: %.2f)', original_noisy_mse));

subplot(2, 3, 3);
imshow(low_pass_filtered_image);
title(sprintf('Low-pass Filtered (MSE: %.2f)', low_pass_mse));

subplot(2, 3, 4);
imshow(high_pass_filtered_image);
title(sprintf('High-pass Filtered (MSE: %.2f)', high_pass_mse));

subplot(2, 3, 5);
imshow(edges, []);
title('Edge Detection (Sobel)');

imwrite(noisy_image, 'noisy_image.png');
imwrite(low_pass_filtered_image, 'low_pass_filtered_image.png');
imwrite(high_pass_filtered_image, 'high_pass_filtered_image.png');
imwrite(uint8(edges), 'edges.png');
```

Listing 4: Image Processing with Noise Addition and Filtering

```
image = imread('Lenna.png');

gray_image = rgb2gray(image);

gaussian_kernel = fspecial('gaussian', [5 5], 2);
```

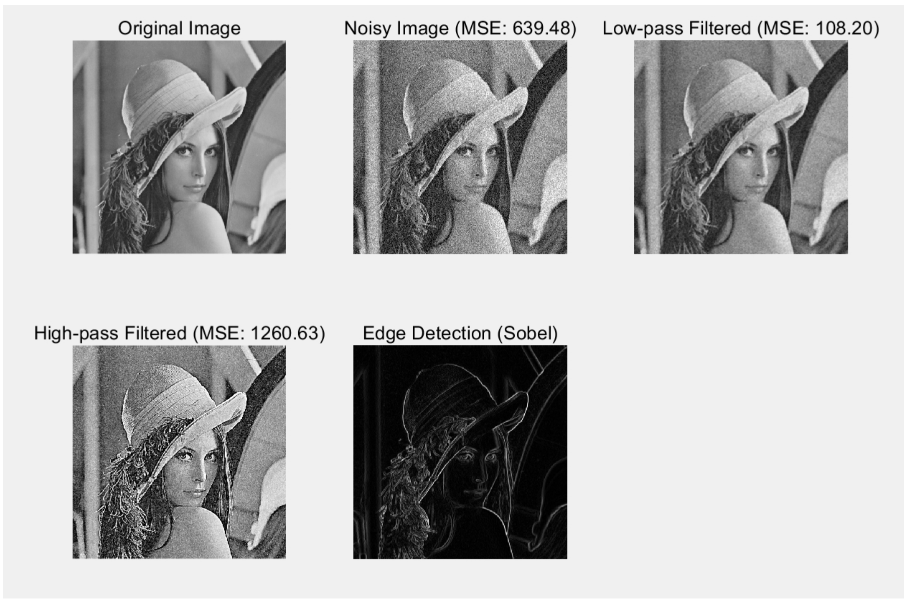
Original Image
Noisy Image (MSE: 639.48)
Low-pass Filtered (MSE: 108.20)
High-pass Filtered (MSE: 1260.63)
Edge Detection (Sobel)

Figure 14: Image Processing with Noise Addition and Filtering

```
motion_kernel = fspecial('motion', 20, 45);
sharpening_kernel = [0 -1 0; -1 5 -1; 0 -1 0];
emboss_kernel = [-2 -1 0; -1 1 1; 0 1 2];

gaussian_blurred_image = imfilter(gray_image, gaussian_kernel, 'replicate
    ');

motion_blurred_image = imfilter(gray_image, motion_kernel, 'replicate');

block_size = 10;
[m, n] = size(gray_image);
mosaic_image = gray_image;
for i = 1:block_size:m
for j = 1:block_size:n
block = gray_image(i:min(i+block_size-1, m), j:min(j+block_size-1, n));
avg_color = mean(block(:));
mosaic_image(i:min(i+block_size-1, m), j:min(j+block_size-1, n)) =
    avg_color;
end
end

embossed_image = imfilter(gray_image, emboss_kernel, 'replicate');

sharpened_image = imfilter(gray_image, sharpening_kernel, 'replicate');

figure;
subplot(2, 3, 1);
imshow(gray_image);
title('Original Image');

subplot(2, 3, 2);
imshow(gaussian_blurred_image);
title('Gaussian Blur');

```

```
subplot(2, 3, 3);
imshow(motion_blurred_image);
title('Motion Blur');

subplot(2, 3, 4);
imshow(mosaic_image);
title('Mosaic Effect');

subplot(2, 3, 5);
imshow(embossed_image);
title('Emboss Effect');

subplot(2, 3, 6);
imshow(sharpened_image);
title('Sharpened Image');

imwrite(gaussian_blurred_image, 'gaussian_blurred_image.png');
imwrite(motion_blurred_image, 'motion_blurred_image.png');
imwrite(mosaic_image, 'mosaic_image.png');
imwrite(embossed_image, 'embossed_image.png');
imwrite(sharpened_image, 'sharpened_image.png');
```

Listing 5: Image Processing with Various Filters and Effects

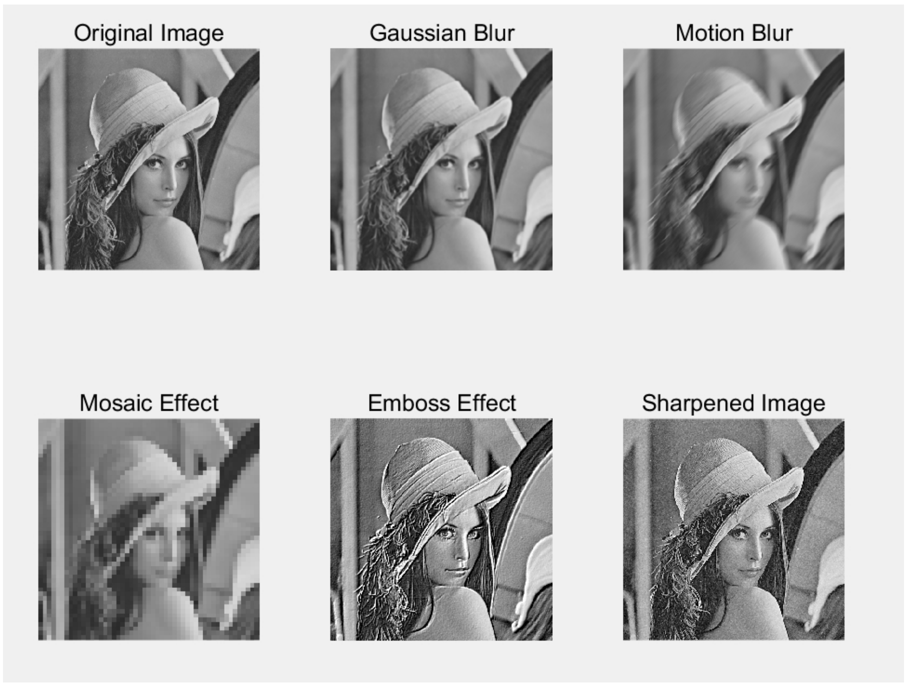

Figure 15: Image Processing with Various Filters and Effects

```
image = imread('Lenna.png');

gray_image = rgb2gray(image);

resized_image_double = imresize(gray_image, 2);
resized_image_half = imresize(gray_image, 0.5);

```

```
rotated_image_45 = imrotate(gray_image, 45);
rotated_image_90 = imrotate(gray_image, 90);
rotated_image_180 = imrotate(gray_image, 180);

figure;
subplot(2, 3, 1);
imshow(gray_image);
title('Original Image');

subplot(2, 3, 2);
imshow(resized_image_double);
title('Resized Image (Double)');

subplot(2, 3, 3);
imshow(resized_image_half);
title('Resized Image (Half)');

subplot(2, 3, 4);
imshow(rotated_image_45);
title('Rotated Image (45 degrees)');

subplot(2, 3, 5);
imshow(rotated_image_90);
title('Rotated Image (90 degrees)');

subplot(2, 3, 6);
imshow(rotated_image_180);
title('Rotated Image (180 degrees)');

imwrite(resized_image_double, 'resized_image_double.png');
imwrite(resized_image_half, 'resized_image_half.png');
imwrite(rotated_image_45, 'rotated_image_45.png');
imwrite(rotated_image_90, 'rotated_image_90.png');
imwrite(rotated_image_180, 'rotated_image_180.png');
```

Listing 6: Image Resizing and Rotation

Introduction

The implementation of a real-time filter application using MATLAB. The application allows users to apply Gaussian blur and sharpening filters to a grayscale image, with adjustable parameters for the Gaussian blur sigma and the sharpening strength.

Implementation Details

Main Figure Creation The main figure window is created using the `uifigure` function, which provides a user interface for the application:

```
fig = uifigure('Name', 'Real-Time Filter Application', 'Position', [100, 100, 800, 600
```

Loading and Displaying the Image The image is loaded and converted to grayscale. Two UI axes are created to display the original and processed images:

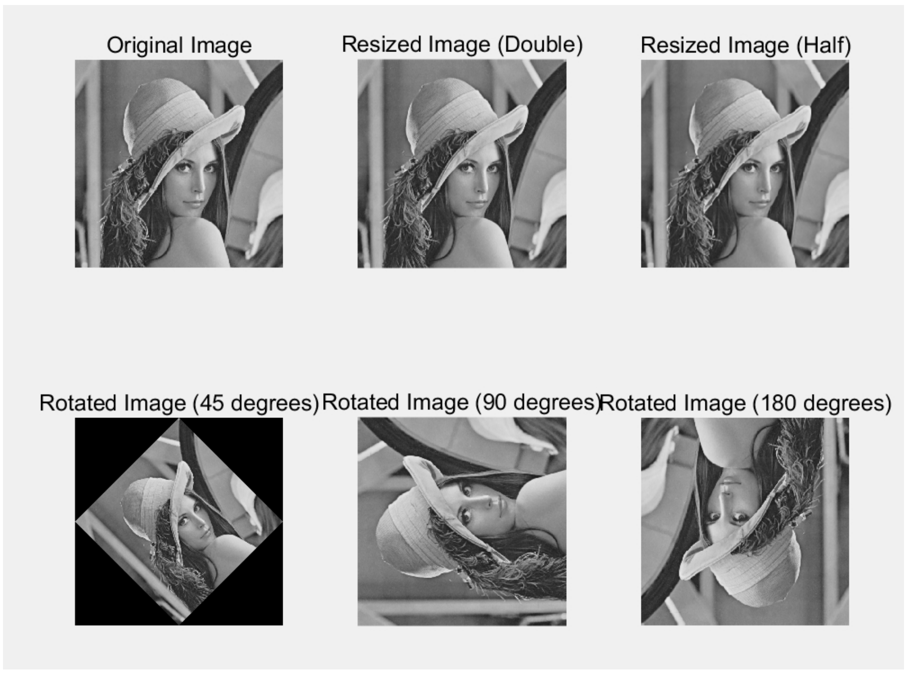
Original Image
Resized Image (Double)
Resized Image (Half)
Rotated Image (45 degrees)
Rotated Image (90 degrees)
Rotated Image (180 degrees)

Figure 16: Image Resizing and Rotation

```
image = imread('Lenna.png');
gray_image = rgb2gray(image);

original_ax = uiaxes(fig, 'Position', [50, 300, 300, 250]);
imshow(gray_image, 'Parent', original_ax);
title(original_ax, 'Original Image');

processed_ax = uiaxes(fig, 'Position', [450, 300, 300, 250]);
imshow(gray_image, 'Parent', processed_ax);
title(processed_ax, 'Processed Image');
```

Creating Sliders for Adjustable Parameters Two sliders are created for adjusting the Gaussian blur sigma and the sharpening strength. Labels are added to indicate the purpose of each slider:

```
uilabel(fig, 'Position', [50, 200, 100, 22], 'Text', 'Gaussian Sigma:');
sigma_slider = uislider(fig, 'Position', [160, 210, 150, 3], 'Limits', [0.1, 10], 'Val

uilabel(fig, 'Position', [50, 150, 100, 22], 'Text', 'Sharpening:');
sharpen_slider = uislider(fig, 'Position', [160, 160, 150, 3], 'Limits', [0, 5], 'Valu
```

Button to Apply Filters A button is created to apply the filters based on the current values of the sliders. When the button is pressed, the `apply_filters` callback function is executed:

```
apply_button = uibutton(fig, 'Position', [50, 100, 100, 30], 'Text', 'Apply Filters',
```

Callback Function for Applying Filters The `apply_filters` function retrieves the current slider values, applies the Gaussian blur and sharpening filters, and updates the processed image:

```
function apply_filters
sigma = sigma_slider.Value;
```

```
sharpen_strength = sharpen_slider.Value;

gaussian_kernel = fspecial('gaussian', [5 5], sigma);
blurred_image = imfilter(gray_image, gaussian_kernel, 'replicate');

sharpening_kernel = [0 -1 0; -1 (5 + sharpen_strength) -1; 0 -1 0];
sharpened_image = imfilter(blurred_image, sharpening_kernel, 'replicate');

imshow(sharpened_image, 'Parent', processed_ax);
title(processed_ax, sprintf('Processed Image (Sigma: %.1f, Sharpening: %.1f)', sigma,
end
```

The real-time filter application allows users to dynamically adjust the parameters for Gaussian blur and sharpening, providing immediate visual feedback. This implementation demonstrates the practical use of MATLAB's GUI capabilities and image processing functions to create an interactive tool for image enhancement.

```matlab
function real_time_filter_app

fig = uifigure('Name', 'Real-Time Filter Application', 'Position', [100,
    100, 800, 600]);

image = imread('Lenna.png');
gray_image = rgb2gray(image);

original_ax = uiaxes(fig, 'Position', [50, 300, 300, 250]);
imshow(gray_image, 'Parent', original_ax);
title(original_ax, 'Original Image');

processed_ax = uiaxes(fig, 'Position', [450, 300, 300, 250]);
imshow(gray_image, 'Parent', processed_ax);
title(processed_ax, 'Processed Image');

uilabel(fig, 'Position', [50, 200, 100, 22], 'Text', 'Gaussian Sigma:');
sigma_slider = uislider(fig, 'Position', [160, 210, 150, 3], 'Limits',
    [0.1, 10], 'Value', 1);

uilabel(fig, 'Position', [50, 150, 100, 22], 'Text', 'Sharpening:');
sharpen_slider = uislider(fig, 'Position', [160, 160, 150, 3], 'Limits',
    [0, 5], 'Value', 1);

apply_button = uibutton(fig, 'Position', [50, 100, 100, 30], 'Text', '
    Apply Filters', 'ButtonPushedFcn', @(btn, event) apply_filters);

function apply_filters
sigma = sigma_slider.Value;
sharpen_strength = sharpen_slider.Value;

gaussian_kernel = fspecial('gaussian', [5 5], sigma);
blurred_image = imfilter(gray_image, gaussian_kernel, 'replicate');

sharpening_kernel = [0 -1 0; -1 (5 + sharpen_strength) -1; 0 -1 0];
sharpened_image = imfilter(blurred_image, sharpening_kernel, 'replicate')
    ;
```

```

imshow(sharpened_image, 'Parent', processed_ax);
title(processed_ax, sprintf('Processed Image (Sigma: %.1f, Sharpening:
    %.1f)', sigma, sharpen_strength));
end
end
```

Listing 7: Real-Time Filter Application

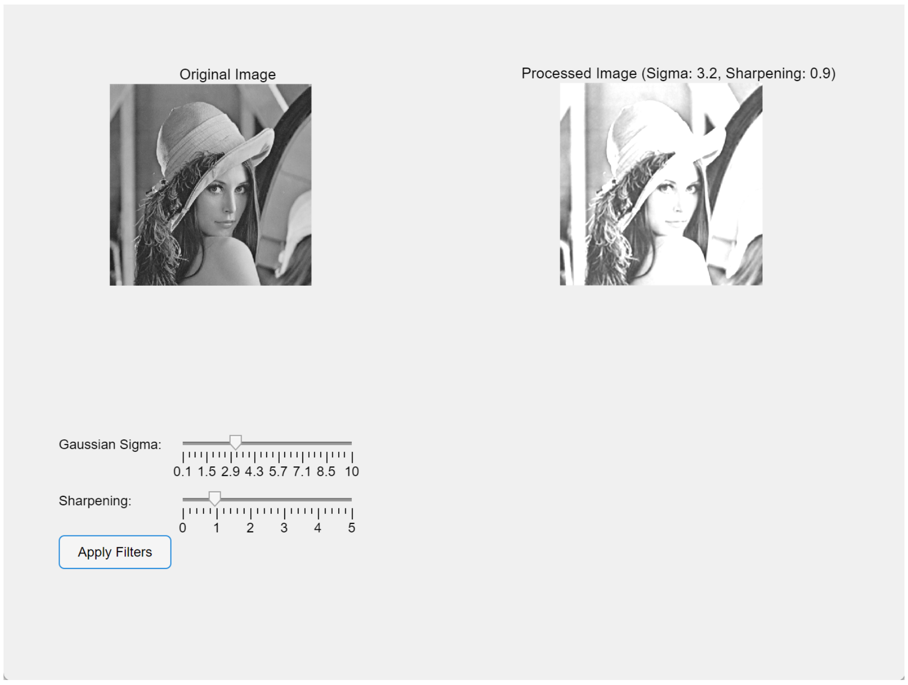

Figure 17: Real-Time Filter Application

3.5 Conclusion

The experiment successfully demonstrated the implementation of various image processing techniques using MATLAB. The real-time filter application provided an interactive platform for exploring the effects of different filters, enhancing the learning experience. This experiment reinforced the understanding of fundamental image processing concepts and their practical applications.