Digital signal generation and time domain processing case2

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

2024-05-06

Contents

1 Common Discrete Signal and Its Implementation

1.1 Experiment Purpose

The objective of this experiment is twofold:

1. To reinforce the conceptual understanding of common discrete signals by exploring their properties and implementations.
2. To acquire practical skills in generating fundamental discrete-time signals in the time domain using MATLAB, facilitating a deeper grasp of theoretical concepts through hands-on experience.

1.2 Principles

The foundation of digital signal processing lies in the understanding and manipulation of basic discrete signals. These signals are the building blocks for more complex operations and systems in the digital realm.

1.2.1 Unit Sample Sequence

The unit sample sequence, also known as the discrete-time impulse or the unit impulse, is denoted by $\delta[n]$ and serves as the digital equivalent of the Dirac delta function. It is mathematically defined as:

$$\delta[n] = \begin{cases} 1, & n = 0, \\ 0, & n \neq 0, \end{cases}$$

where n is an integer representing the discrete time index. This sequence is pivotal in the analysis and characterization of digital systems, serving as an identity element in the context of convolution.

1.2.2 Unit Step Sequence

The unit step sequence is a fundamental sequence in discrete-time systems, represented by $\mu[n]$ and defined as a cumulative sum of the unit sample sequence:

$$\mu[n] = \sum_{k=-\infty}^{n} \delta[k] = \begin{cases} 1, & n \geq 0, \\ 0, & n < 0. \end{cases}$$

The unit step sequence acts as a Heaviside function in discrete time and is extensively used to represent signals that switch on at a certain point in time.

1.2.3 Sinusoidal Sequence

A sinusoidal sequence is one of the most commonly encountered sequences in signal processing. It represents a harmonically oscillating signal with a constant amplitude and can be expressed as:

$$x[n] = A \sin\left(\frac{2\pi n f}{F_s} + \phi\right),$$

where:

- A denotes the amplitude,
- f is the frequency of the sinusoid,
- F_s represents the sampling frequency, and
- ϕ is the phase of the sinusoidal sequence at $n = 0$.

1.2.4 Complex Exponential Sequence

The complex exponential sequence is a critical signal in the analysis of linear time-invariant (LTI) systems. It is defined as:

$$x[n] = Ae^{j(\omega n+\phi)},$$

where j is the imaginary unit, $\omega = 2\pi f$ is the angular frequency, and ϕ is the phase. The complex exponential sequence encompasses both growth and oscillation, and its Euler's formula representation connects the complex exponentials with sinusoidal sequences.

1.2.5 Real Exponential Sequence

The real exponential sequence is a manifestation of the exponential function in discrete-time signal processing. It takes the form:

$$x[n] = a^n,$$

where a is a real constant. The behavior of the sequence is fundamentally determined by the value of a: it grows when $|a| > 1$, decays when $|a| < 1$, and remains constant when $|a| = 1$.

1.3 Experimental Tasks and Their Implementations

1.3.1 Unit Sample Sequence

Using MATLAB, generate a unit sample sequence in the -10 to 10 interval through the following function, as shown in fig. 1. In this function, the program judges whether each value in the input sequence n is 0 and generates a corresponding unit sample sequence.

```matlab
n = -10:10;
delta = zeros(size(n));
delta(n == 0) = 1;
figure;
stem(n, delta, 'filled', 'MarkerSize', 8, 'LineWidth', 2, 'Color', 'b');
title('Unit Sample Sequence \delta[n]');
xlabel('n');
ylabel('\delta[n]');
grid on;
axis([-10 10 -0.1 1.1]);
```

Listing 1: MATLAB Code for Unit Sample Sequence

1.3.2 Unit Step Sequence

Using MATLAB, generate a unit step sequence in the -10 to 10 interval through the following function, as shown in fig. 2. In this function, the program judges whether each value in the input sequence n is greater or equal to 0 and generates a corresponding unit step sequence.

```matlab
n = -10:10;
u = n >= 0;
figure;
stem(n, u, 'filled', 'MarkerSize', 8, 'LineWidth', 2, 'Color', 'b');
title('Unit Step Sequence u[n]');
xlabel('n');
ylabel('u[n]');
grid on;
axis([-10 10 -0.1 1.1]);
```

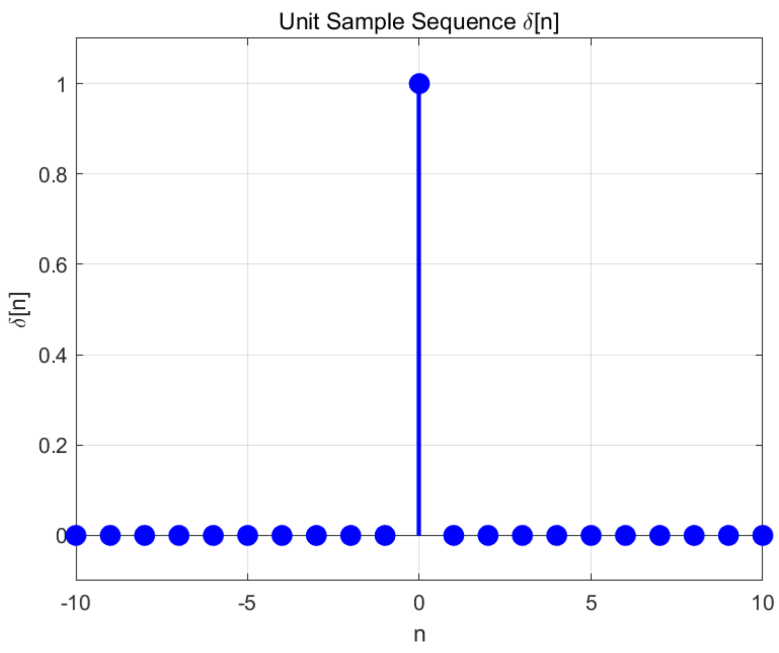


Figure 1: Visualization of the Unit Sample Sequence.

Listing 2: MATLAB Code for Unit Step Sequence

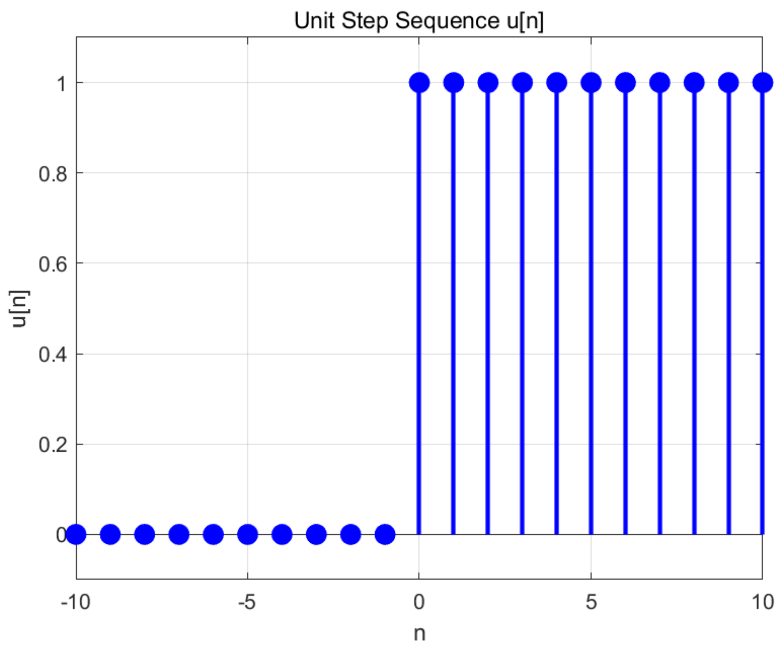


Figure 2: Visualization of the Unit Step Sequence.

1.3.3 Sinusoidal Sequence

This MATLAB script generates and plots a sine sequence, defined over 50 points with an amplitude of 1 and a frequency of 0.1. It visualizes the sine wave using a stem plot, enhancing clarity with blue markers.

```
N = 50;
A = 1;
```

```
f = 0.05;

n = 0:N-1;
sine_seq = A * sin(2 * pi * f * n);

figure;
stem(n, sine_seq, 'filled', 'MarkerSize', 8, 'LineWidth', 2, 'Color', 'b'
    );
title('Sine Sequence');
xlabel('n');
ylabel('Amplitude');
grid on;
axis([0 N-1 -1.1*A 1.1*A]);
```

Listing 3: MATLAB Code for Sinusoidal Sequence

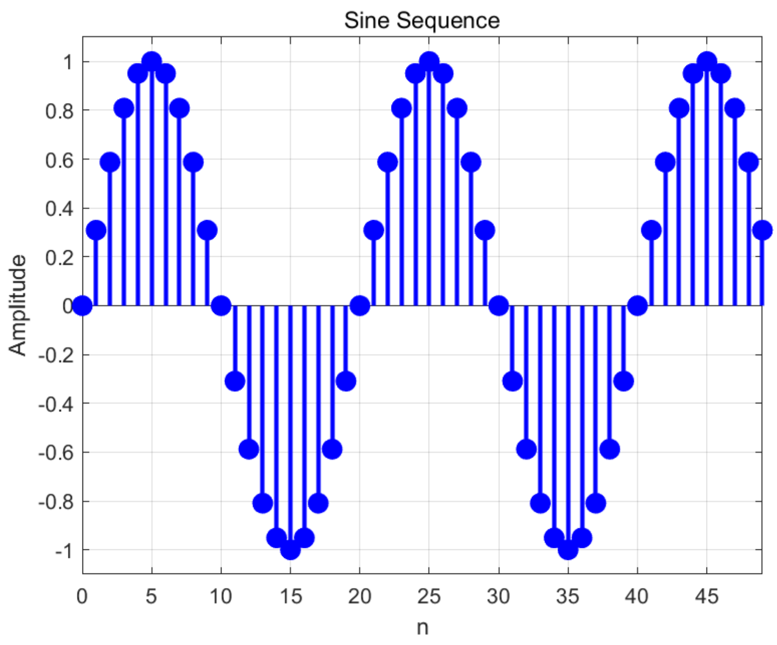


Figure 3: Visualization of the Sinusoidal Sequence.

1.3.4 Complex Exponential Sequence

This script visualizes a complex exponential sequence of $N = 30$ with reduced real part $a = 0.05$ and imaginary part $b = 0.3$. It uses 2D and 3D drawing comparison methods to facilitate a deeper understanding of images and parameters

```
N = 30;
a = 0.05;
b = 0.3;

n = 0:N-1;
comp_exp = exp((a + 1i * b) * n);

figure;
hold on;
plot3(n, real(comp_exp), imag(comp_exp), 'b-', 'LineWidth', 2);
```

```
stem3(n, real(comp_exp), imag(comp_exp), 'filled', 'LineWidth', 2, '
    MarkerSize', 8, 'MarkerEdgeColor', 'b', 'MarkerFaceColor', 'b');
hold off;

title('Complex Exponential Sequence');
xlabel('n');
ylabel('Real Part');
zlabel('Imaginary Part');
grid on;

set(gca, 'Color', 'w');
colormap([0 0 1]);
```

Listing 4: MATLAB Code for Complex Exponential Sequence Part1

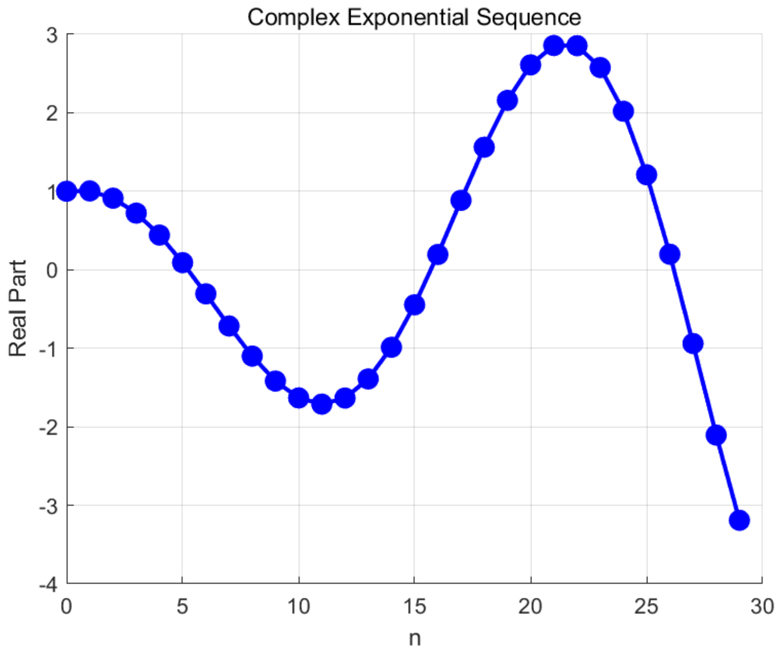


Figure 4: Visualization of the Complex Exponential Sequence Part1.

```
N = 30;
a = 0.05;
b = 0.3;

n = 0:N-1;
comp_exp = exp((a + 1i * b) * n);

figure;
hold on;

plot3(n, real(comp_exp), imag(comp_exp), 'b-', 'LineWidth', 2);
stem3(n, real(comp_exp), imag(comp_exp), 'MarkerFaceColor', 'b', '
    MarkerEdgeColor', 'b', 'LineWidth', 2, 'MarkerSize', 8);

title('3D Visualization of Complex Exponential Sequence');
xlabel('Index n');
ylabel('Real Part');
```

```
zlabel('Imaginary Part');
grid on;

view(-35, 45);

set(gca, 'Color', 'w');
colormap([0 0 1]);
hold off;
```

Listing 5: MATLAB Code for Complex Exponential Sequence Part2

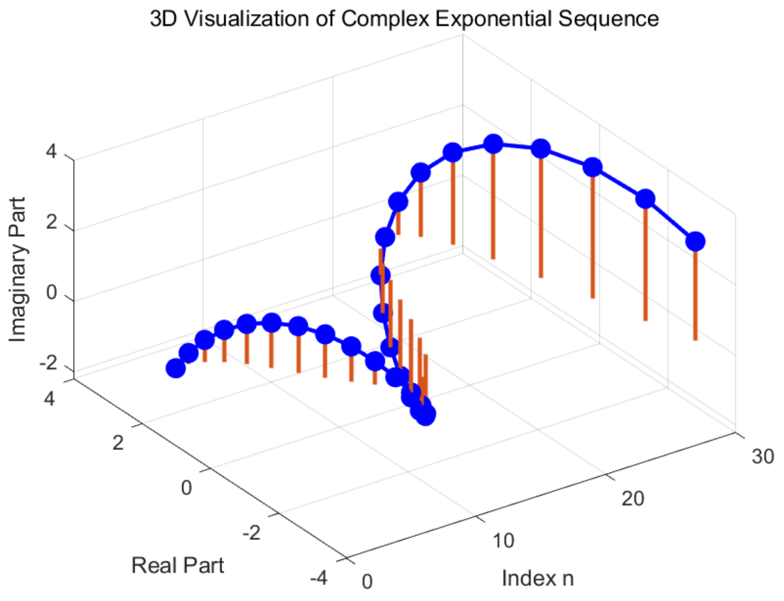


Figure 5: Visualization of the Complex Exponential Sequence Part2.

1.3.5 Real Exponential Sequence

In MATLAB, a real exponential sequence can be generated through the following function. Using a = 1.1, the sequence is shown in fig. 6.

```
N = 50;
a = 1.1;

n = 0:N-1;
real_exp = a.^n;
figure;
stem(n, real_exp, 'filled', 'MarkerSize', 8, 'LineWidth', 2, '
    MarkerEdgeColor', 'b', 'MarkerFaceColor', 'b');

title('Real Exponential Sequence');
xlabel('n');
ylabel('Value');
grid on;
axis([0 N-1 0 max(real_exp)*1.1]);

set(gca, 'Color', 'w');
colormap([0 0 1]);
```

Listing 6: MATLAB Code for Real Exponential Sequence

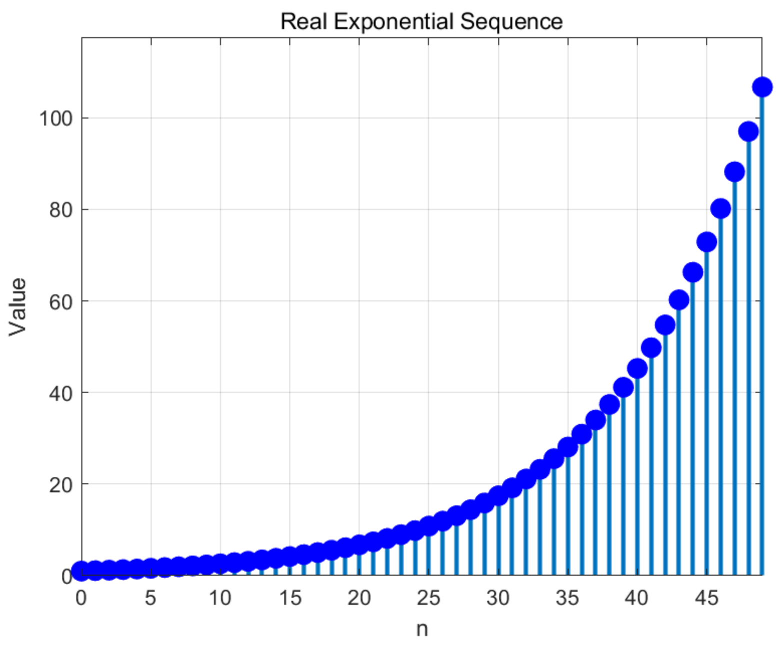


Figure 6: Visualization of the Real Exponential Sequence.

2 Analysis of Discrete Systems in Time Domain

Time-domain analysis of discrete systems focuses on understanding the behavior of signals and systems over time. It involves the examination of input and output signals, system stability, and the response of systems to various input types such as step, impulse, and sinusoidal inputs.

2.1 Experiment Purpose

The objective of this experiment is twofold:

1. Get familiar with and master representation of discrete systems using difference equations.
2. Gain a better understanding of impulse response and convolution analysis methods.

2.2 Principles

2.2.1 Discrete-Time Signals and Systems

A discrete-time system is defined by its response to an input signal. The input signal $x[n]$ and the output signal $y[n]$ are related by the system's characteristic equation or transfer function. For linear time-invariant (LTI) systems, the relationship is typically characterized by a difference equation:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k] \tag{1}$$

where b_k and a_k are the system coefficients, and M and N are the orders of the system.

2.2.2 Impulse Response and Convolution

The impulse response $h[n]$ of a discrete system is its output when the input is an impulse signal $\delta[n]$. The output of the system to any arbitrary input can then be calculated by the convolution sum:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k] \tag{2}$$

This equation signifies the superposition principle where the output is a weighted sum of past inputs scaled by the impulse response.

2.2.3 Step Response

The step response of a system is its output when the input is a unit step function $u[n]$. It is related to the impulse response by the cumulative sum:

$$s[n] = \sum_{k=-\infty}^{n} h[k] \tag{3}$$

This response is critical in understanding how a system behaves from an initially at-rest state.

2.2.4 Stability of Discrete Systems

A discrete system is stable if its response to any bounded input results in a bounded output. In terms of the system's impulse response, this is conditionally true if:

$$\sum_{n=-\infty}^{\infty} |h[n]| < \infty \tag{4}$$

This condition ensures that the system's output will not diverge over time.

2.2.5 Frequency Response

The frequency response of a system provides insight into how different frequency components of an input signal are modified by the system. It is derived from the z-transform of the impulse response:

$$H(z) = \sum_{n=-\infty}^{\infty} h[n]z^{-n} \tag{5}$$

evaluated on the unit circle $z = e^{j\omega}$, where ω is the angular frequency.

2.3 Core Problem

2.3.1 Program1 Analysis

```
% Given sequences
h = [3 2 1 -2 1 0 -4 0 3]; % impulse response
x = [1 -2 3 -4 3 2 1]; % input sequence

% Convolution using conv function
y_conv = conv(h, x);
n = 0:length(y_conv)-1;

```

```
% Padding x with zeros and using filter function
x_padded = [x zeros(1, length(h)-1)];
y_filter = filter(h, 1, x_padded);

% Compute the absolute error between the two outputs
absolute_error = abs(y_conv - y_filter);
mae = mean(absolute_error); % Mean Absolute Error
rmse = sqrt(mean(absolute_error.^2)); % Root Mean Squared Error

% Plot the results
figure;

subplot(3,1,1);
stem(n, y_conv, 'r', 'filled');
xlabel('Time index n');
ylabel('Amplitude');
title('Output Obtained by Convolution');
grid on;

subplot(3,1,2);
stem(n, y_filter, 'b', 'filled');
xlabel('Time index n');
ylabel('Amplitude');
title('Output Generated by Filtering');
grid on;

subplot(3,1,3);
stem(n, absolute_error, 'k', 'filled');
xlabel('Time index n');
ylabel('Absolute Error');
title(['Absolute Error between Convolution and Filtering | MAE: ',
    num2str(mae), ' | RMSE: ', num2str(rmse)]);
grid on;

% Displaying the evaluation indicators
disp(['Mean Absolute Error (MAE): ', num2str(mae)]);
disp(['Root Mean Squared Error (RMSE): ', num2str(rmse)]);
```

Listing 7: MATLAB Code for Program $P2_7$

System 1: $\mathbf{y[n] + 0.75y[n-1] + 0.125y[n-2] = x[n] - x[n-1]}$

1. Unit Impulse Response $h[n]$

The difference equation for the unit impulse response $h[n]$ of the system is as follows:

$$h[n] + 0.75h[n-1] + 0.125h[n-2] = \delta[n] - \delta[n-1]$$

Taking the Z-transform of both sides, we obtain:

$$H(z) + 0.75H(z)z^{-1} + 0.125H(z)z^{-2} = 1 - z^{-1}$$

Simplifying this expression gives:

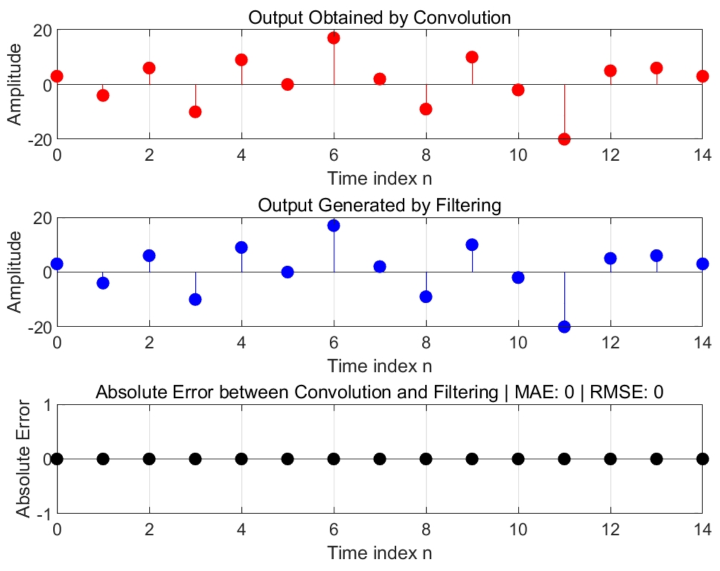


Figure 7: Comparison results of conv and filter

$$H(z) = \frac{1 - z^{-1}}{1 + 0.75z^{-1} + 0.125z^{-2}}$$

Using partial fractions, we find:

$$H(z) = \frac{0.5}{1 + 0.5z^{-1}} + \frac{-0.5}{1 - 0.25z^{-1}}$$

Taking the inverse Z-transform, we derive the unit impulse response:

$$h[n] = (-0.5^n + 0.25^n)\,u[n] - \left(-0.5^{n-1} + 0.25^{n-1}\right)u[n-1]$$

2. Unit Step Response $s[n]$

The difference equation for the unit step response $s[n]$ of the system is:

$$s[n] + 0.75s[n-1] + 0.125s[n-2] = u[n] - u[n-1]$$

Taking the Z-transform of both sides, we obtain:

$$S(z) = \frac{1 - z^{-1}}{1 + 0.75z^{-1} + 0.125z^{-2}} \cdot \frac{1}{1 - z^{-1}}$$

Simplifying this expression gives:

$$S(z) = \frac{0.5}{(1 + 0.5z^{-1})(1 - z^{-1})} + \frac{-0.5}{(1 - 0.25z^{-1})(1 - z^{-1})}$$

Using partial fractions, we find:

$$S(z) = \frac{0.5}{1 - z^{-1}}\left(\frac{1}{1 + 0.5z^{-1}} - \frac{1}{1 - 0.25z^{-1}}\right)$$

Taking the inverse Z-transform, we derive the unit step response:

$$s[n] = (-0.5^n + 0.25^n)\,u[n] + \frac{1}{1.5}\left(-0.5^{n-1} + 0.25^{n-1}\right)u[n-1]$$

System 2: $\mathbf{y[n] = 0.25\{x[n-1] + x[n-2] + x[n-3] + x[n-4]\}}$

1. Unit Impulse Response $h[n]$

Given the equation for this system:

$$y[n] = 0.25\{x[n-1] + x[n-2] + x[n-3] + x[n-4]\}$$

The unit impulse response $h[n]$:

$$h[n] = 0.25\{\delta[n-1] + \delta[n-2] + \delta[n-3] + \delta[n-4]\}$$

2. Unit Step Response $s[n]$

Given the equation for this system:

$$y[n] = 0.25\{u[n-1] + u[n-2] + u[n-3] + u[n-4]\}$$

The unit step response $s[n]$:

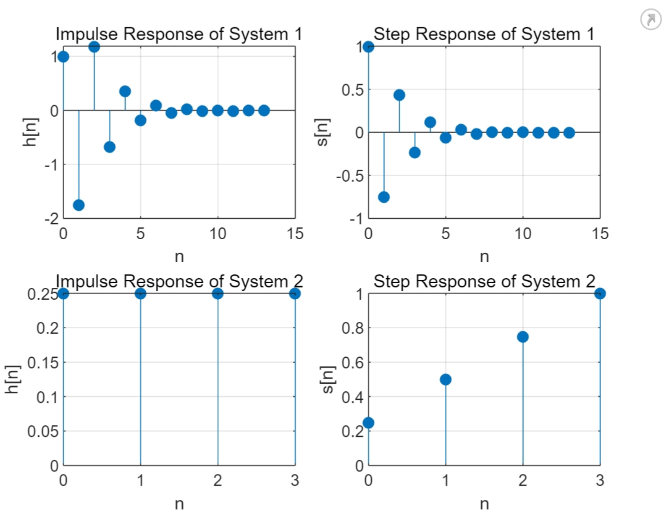
$$s[n] = 0.25\{u[n-1] + u[n-2] + u[n-3] + u[n-4]\}$$

2.3.2 Program2 Analysis

```matlab
clc;
clear;
close all;

b1 = [1, -1];
a1 = [1, 0.75, 0.125];
b2 = [0.25, 0.25, 0.25, 0.25];
a2 = 1;

% Impulse responses
[h1_imp, t1_imp] = impz(b1, a1);
[h2_imp, t2_imp] = impz(b2, a2);

% Step responses
h1_step = filter(b1, a1, ones(size(h1_imp)));
h2_step = filter(b2, a2, ones(size(h2_imp)));

figure;

subplot(2, 2, 1);
stem(t1_imp, h1_imp, 'filled');
title('Impulse Response of System 1');
xlabel('n');
ylabel('h[n]');
grid on;

subplot(2, 2, 2);
stem(t1_imp, h1_step, 'filled');
title('Step Response of System 1');
xlabel('n');
```

```
ylabel('s[n]');
grid on;

subplot(2, 2, 3);
stem(t2_imp, h2_imp, 'filled');
title('Impulse Response of System 2');
xlabel('n');
ylabel('h[n]');
grid on;

subplot(2, 2, 4);
stem(t2_imp, h2_step, 'filled');
title('Step Response of System 2');
xlabel('n');
ylabel('s[n]');
grid on;

disp('Theoretical impulse response of System 1:');
disp(h1_imp');
disp('Theoretical step response of System 1:');
disp(h1_step');

disp('Theoretical impulse response of System 2:');
disp(h2_imp');
disp('Theoretical step response of System 2:');
disp(h2_step');
```

Listing 8: MATLAB Code for 2.2.4.2



```
Theoretical impulse response of System 1:
    1.0000   -1.7500    1.1875   -0.6719    0.3555   -0.1826    0.0925   -0.0466

Theoretical step response of System 1:
    1.0000   -0.7500    0.4375   -0.2344    0.1211   -0.0615    0.0310   -0.0156

Theoretical impulse response of System 2:
    0.2500    0.2500    0.2500    0.2500
Theoretical step response of System 2:
    0.2500    0.5000    0.7500    1.0000
```

Figure 8: System analysis diagram

3 Application experiment: Speech gene period estimation

3.1 Experiment Purpose

We use ”passion can withstand the long years”

```
[a_signal, a_fs] = audioread('a.M4A');
[o_signal, o_fs] = audioread('o.M4A');
[e_signal, e_fs] = audioread('e.M4A');
[sentence_signal, sentence_fs] = audioread('sentence.M4A');

[a_pitch_periods, a_lags] = estimate_pitch_period(a_signal, a_fs);
[o_pitch_periods, o_lags] = estimate_pitch_period(o_signal, o_fs);
[e_pitch_periods, e_lags] = estimate_pitch_period(e_signal, e_fs);
[sentence_pitch_periods, sentence_lags] = estimate_pitch_period(
    sentence_signal, sentence_fs);

visualize_pitch(a_signal, a_fs, a_pitch_periods, a_lags, 'a');
visualize_pitch(o_signal, o_fs, o_pitch_periods, o_lags, 'o');
visualize_pitch(e_signal, e_fs, e_pitch_periods, e_lags, 'e');
visualize_pitch(sentence_signal, sentence_fs, sentence_pitch_periods,
    sentence_lags, 'sentence');

function [pitch_periods, lags] = estimate_pitch_period(signal, fs)
    frame_size = 0.02;
    frame_length = round(frame_size * fs);
    overlap = frame_length / 2;
    [signal, fs] = preprocess_signal(signal, fs);

    frames = buffer(signal, frame_length, overlap, 'nodelay');
    num_frames = size(frames, 2);
    pitch_periods = zeros(1, num_frames);
    lags = cell(1, num_frames);

    for i = 1:num_frames
        frame = frames(:, i);
        [ac, lags{i}] = xcorr(frame);
        ac = ac(lags{i} >= 0);
        lags{i} = lags{i}(lags{i} >= 0);

        [~, idx] = max(ac(2:end));
        pitch_periods(i) = lags{i}(idx + 1);
    end
end

function [signal, fs] = preprocess_signal(signal, fs)
    if size(signal, 2) == 2
        signal = mean(signal, 2);
    end

    target_fs = 16000;
    if fs ~= target_fs
        signal = resample(signal, target_fs, fs);
        fs = target_fs;
```

```matlab
    end
end

function visualize_pitch(signal, fs, pitch_periods, lags, label)
    frame_size = 0.02;
    frame_length = frame_size * fs;
    overlap = frame_length / 2;
    frame_time = (1:length(pitch_periods)) * (frame_length - overlap) /
       fs;
    num_frames = length(pitch_periods);
    max_frames = min(50, num_frames);

    figure;
    subplot(3, 1, 1);
    time_axis = (1:length(signal)) / fs;
    plot(time_axis, signal);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title([label, ' - Original Signal']);

    subplot(3, 1, 2);
    stem(frame_time, pitch_periods / fs, 'filled');
    xlabel('Time (s)');
    ylabel('Pitch Period (s)');
    title([label, ' - Pitch Periods']);

    subplot(3, 1, 3);
    hold on;
    for i = 1:max_frames
        frame_start = (i-1) * (frame_length - overlap) + 1;
        frame_end = frame_start + frame_length - 1;
        if frame_end > length(signal)
            continue;
        end
        frame1 = signal(frame_start:frame_end);
        frame2 = signal((frame_start+pitch_periods(i)):min((frame_end+
           pitch_periods(i)), length(signal)));
        [cross_corr, cross_lags] = xcorr(frame1, frame2);
        plot(cross_lags / fs, cross_corr);
    end
    xlabel('Lag (s)');
    ylabel('Cross-Correlation');
    title([label, ' - Cross-Correlation']);
    hold off;
end
```

Listing 9: Autocorrelation method MATLAB Code

```matlab
[a_signal, a_fs] = audioread('a.M4A');
[o_signal, o_fs] = audioread('o.M4A');
[e_signal, e_fs] = audioread('e.M4A');
[sentence_signal, sentence_fs] = audioread('sentence.M4A');

a_pitch_periods = hps_pitch_period(a_signal, a_fs);
```

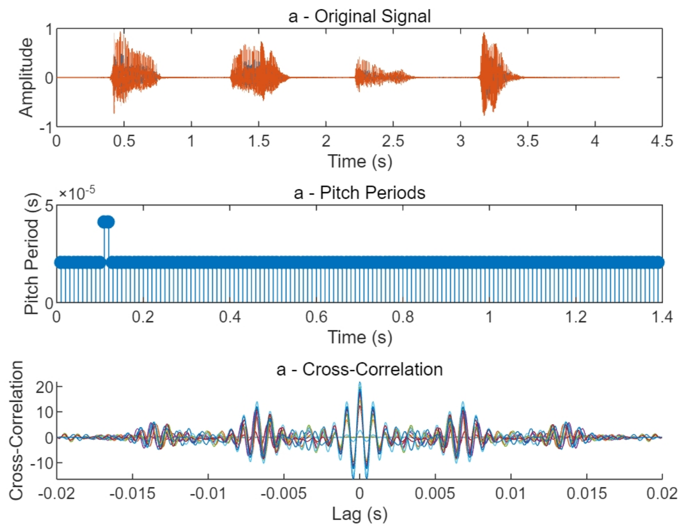


Figure 9: My voice of different a by Autocorrelation method

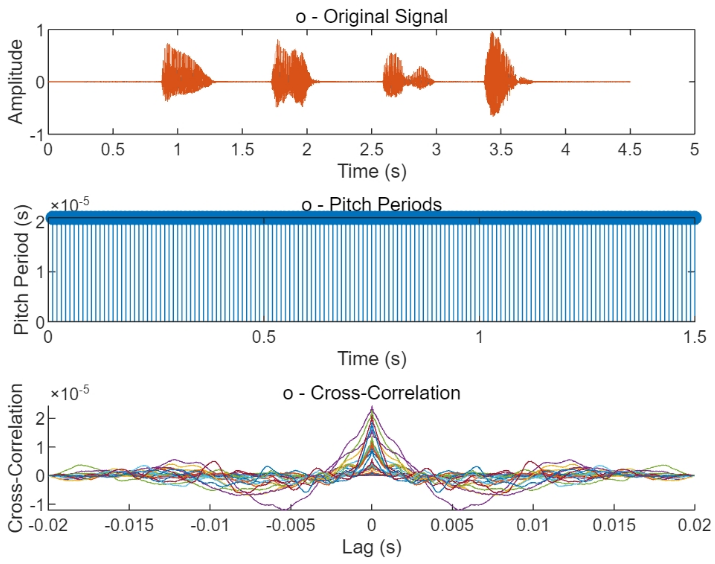


Figure 10: My voice of different o by Autocorrelation method

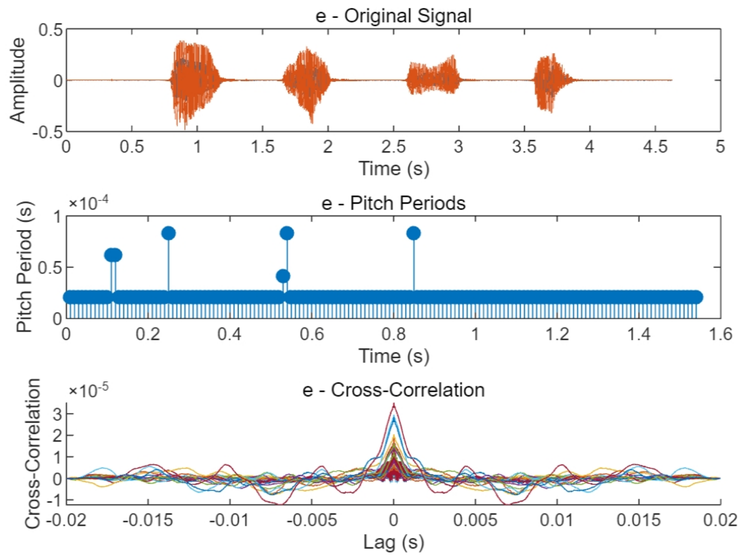


Figure 11: My voice of different e by Autocorrelation method

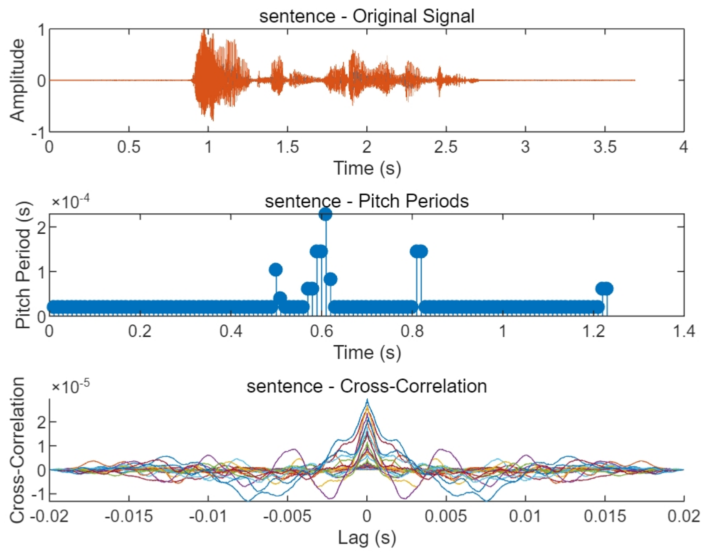


Figure 12: My voice of passion can withstand the long years by Autocorrelation method

```
o_pitch_periods = hps_pitch_period(o_signal, o_fs);
e_pitch_periods = hps_pitch_period(e_signal, e_fs);
sentence_pitch_periods = hps_pitch_period(sentence_signal, sentence_fs);

visualize_pitch(a_signal, a_fs, a_pitch_periods, 'a');
visualize_pitch(o_signal, o_fs, o_pitch_periods, 'o');
visualize_pitch(e_signal, e_fs, e_pitch_periods, 'e');
visualize_pitch(sentence_signal, sentence_fs, sentence_pitch_periods, '
    sentence');

function pitch_periods = hps_pitch_period(signal, fs)
    frame_size = 0.02;
    frame_length = round(frame_size * fs);
    overlap = frame_length / 2;
    [signal, fs] = preprocess_signal(signal, fs);

    frames = buffer(signal, frame_length, overlap, 'nodelay');
    num_frames = size(frames, 2);
    pitch_periods = zeros(1, num_frames);

    for i = 1:num_frames
        frame = frames(:, i);
        pitch_periods(i) = hps(frame, fs);
    end
end

function [signal, fs] = preprocess_signal(signal, fs)
    if size(signal, 2) == 2
        signal = mean(signal, 2);
    end

    target_fs = 16000;
    if fs ~= target_fs
        signal = resample(signal, target_fs, fs);
        fs = target_fs;
    end
end

function pitch_period = hps(frame, fs)
    N = length(frame);
    spectrum = abs(fft(frame, N));
    spectrum = spectrum(1:N/2);

    max_harmonics = 5;
    hps_spectrum = spectrum;
    for harmonic = 2:max_harmonics
        downsampled = spectrum(1:harmonic:end);
        hps_spectrum = hps_spectrum(1:length(downsampled)) .* downsampled
            ;
    end

    [~, idx] = max(hps_spectrum);
    pitch_period = fs / idx;
```

```
end

function visualize_pitch(signal, fs, pitch_periods, label)
    frame_size = 0.02;
    frame_length = frame_size * fs;
    overlap = frame_length / 2;
    frame_time = (1:length(pitch_periods)) * (frame_length - overlap) /
       fs;

    figure;
    subplot(2, 1, 1);
    time_axis = (1:length(signal)) / fs;
    plot(time_axis, signal);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title([label, ' - Original Signal']);

    subplot(2, 1, 2);
    stem(frame_time, pitch_periods, 'filled');
    xlabel('Time (s)');
    ylabel('Pitch Period (s)');
    title([label, ' - Pitch Periods']);
end
```

Listing 10: MATLAB Code for Harmonic Product Spectrum (HPS)

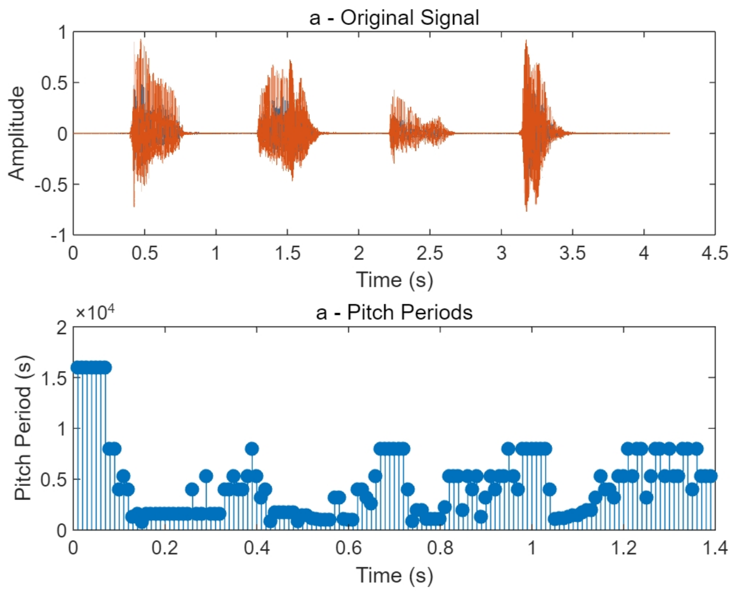


Figure 13: My voice of different a by Harmonic Product Spectrum (HPS)

4 Application experiment: Hearing test signal generator

4.1 Introduction

Human hearing is typically sensitive to frequencies between 20 Hz and 20,000 Hz. As people age, their ability to hear higher frequencies generally diminishes. This project aimed to create an application

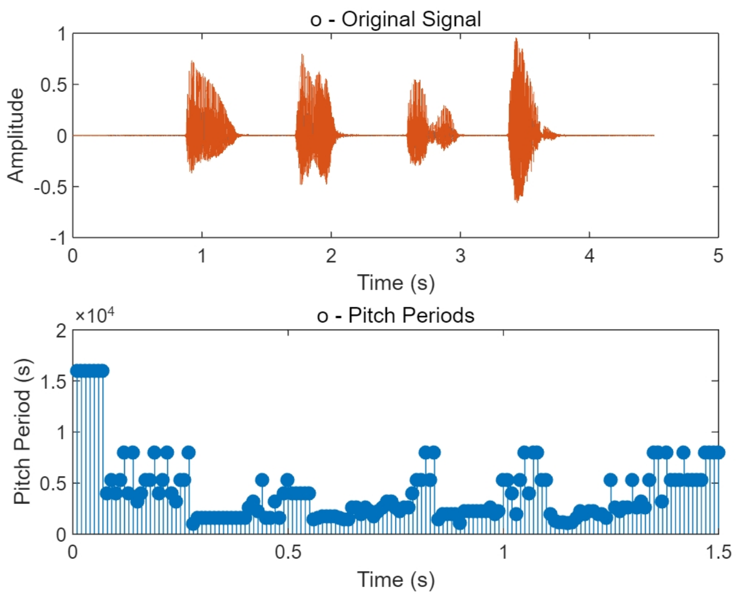


Figure 14: My voice of different o by Harmonic Product Spectrum (HPS)

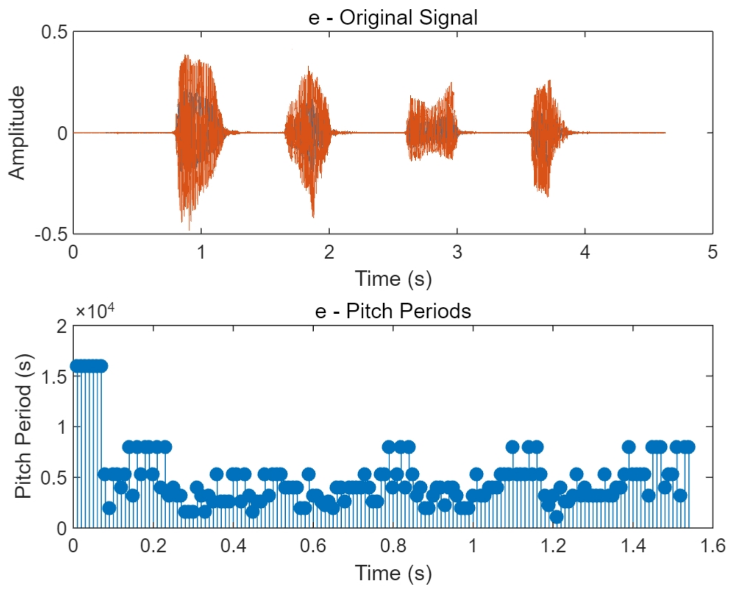


Figure 15: My voice of different e by Harmonic Product Spectrum (HPS)

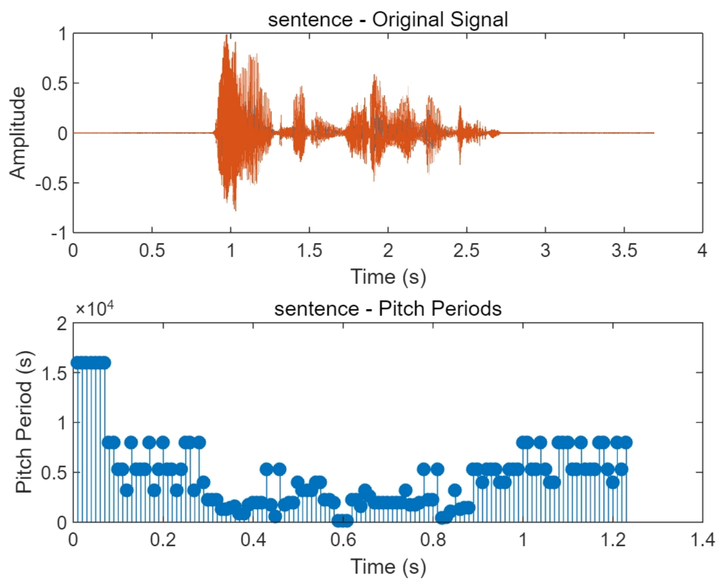


Figure 16: My voice of passion can withstand the long years by Harmonic Product Spectrum (HPS)

that generates audio tones for testing hearing sensitivity across different frequencies. The primary objectives were to implement an easy-to-use interface, allow frequency sweeping, and record the results.

4.2 Methodology

The application was built using Python, utilizing the Tkinter library for the graphical user interface (GUI). The audio tones were generated using the `pydub` and `simpleaudio` libraries. The GUI allowed users to set a specific frequency and volume or conduct a frequency sweep across the typical hearing range.

The function `generate_tone` creates a sine wave at a specified frequency and volume using the `pydub` library. The generated tone is then played using `simpleaudio`.

```python
def generate_tone(frequency, volume, duration=1000):
    sine_wave = Sine(frequency).to_audio_segment(duration=duration)
    sine_wave = sine_wave - (100 - volume)
    play_obj = sa.play_buffer(sine_wave.raw_data,
                              num_channels=sine_wave.channels,
                              bytes_per_sample=sine_wave.sample_width,
                              sample_rate=sine_wave.frame_rate)
    play_obj.wait_done()
```

The GUI allowed users to input a frequency and volume using sliders and entry boxes. Buttons triggered actions such as playing the tone, conducting a frequency sweep, saving the results, and resetting the settings.

The frequency sweep function iterated through frequencies from 20 Hz to 20,000 Hz in 100 Hz increments, asking the user if they could hear each frequency. The responses were recorded.

To enhance user experience, the frequency sweep was executed in a separate thread to keep the GUI responsive. This was implemented using Python's `threading` module.

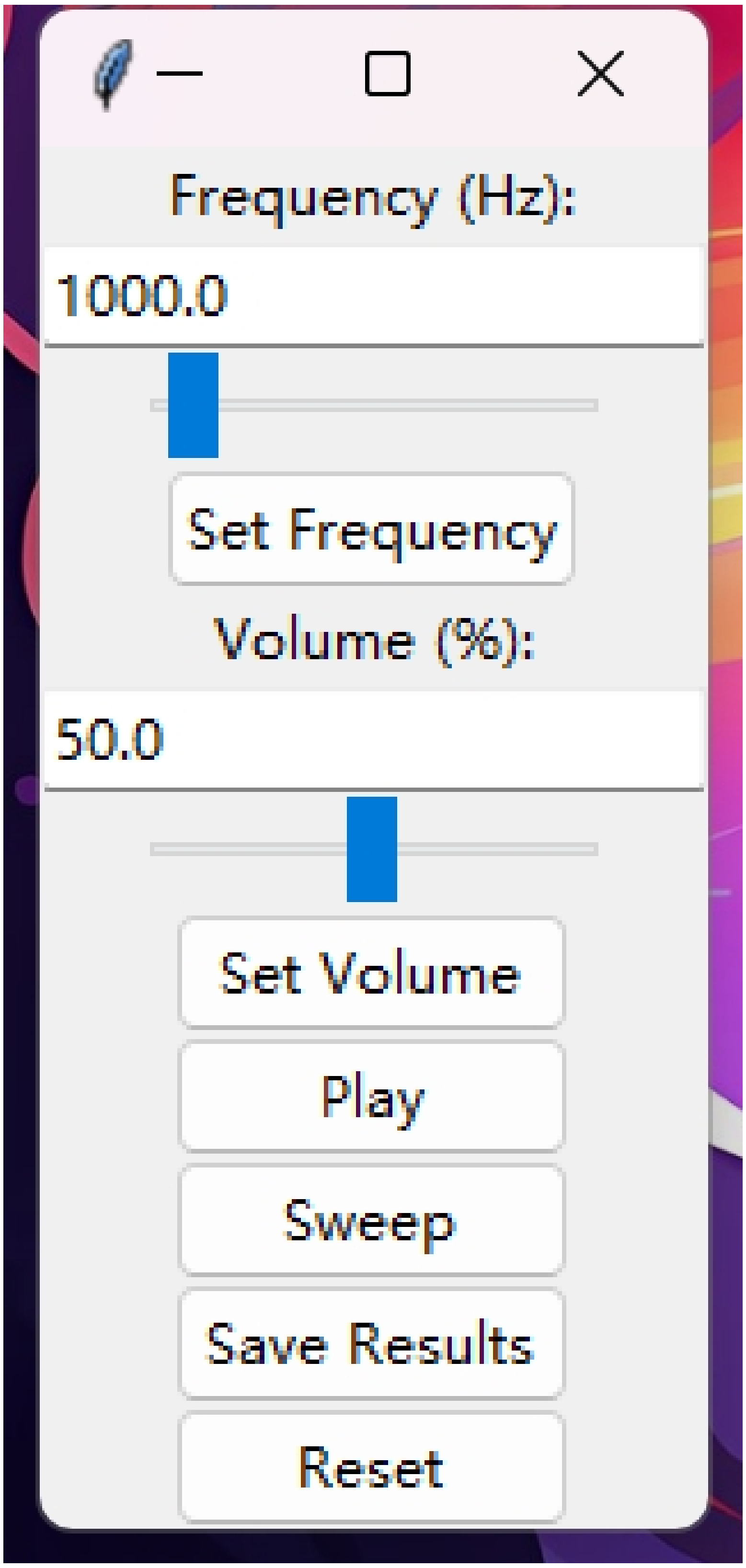


Figure 17: Hearing Test Signal Generator GUI

The application was tested on several individuals to determine their hearing range. The results were saved to a text file and indicated that hearing ability declined for higher frequencies, particularly among older participants.

The application successfully generated audio tones and conducted frequency sweep tests. However, the user experience during the sweep could be further improved. The current implementation uses pop-up dialogs, which may become repetitive. A custom dialog or visual feedback would enhance usability.

The hearing test signal generator achieved its objectives of providing an interface for testing hearing sensitivity. Future work could include refining the user experience and adding functionality for testing different ear sensitivity.

[fragile]

```
import tkinter as tk
from tkinter import ttk, messagebox
from pydub.generators import Sine
import simpleaudio as sa
import time
def generate_tone(frequency: float, volume: float, duration: int = 1000):
    sine_wave = Sine(frequency).to_audio_segment(duration=duration)
    sine_wave = sine_wave - (100 - volume)
    play_obj = sa.play_buffer(sine_wave.raw_data,
                              num_channels=sine_wave.channels,
                              bytes_per_sample=sine_wave.sample_width,
                              sample_rate=sine_wave.frame_rate)
    play_obj.wait_done()
def play_sound():
    frequency = frequency_var.get()
    volume = volume_var.get()
    generate_tone(frequency, volume)

def sweep_frequencies():
    results = []
    for freq in range(20, 20001, 100):
        generate_tone(freq, volume_var.get())
        if messagebox.askyesno("Frequency Check", f"Did you hear {freq} Hz?"):
            results.append((freq, "Yes"))
        else:
            results.append((freq, "No"))
    global hearing_results
    hearing_results = results
    messagebox.showinfo("Sweep Complete", "Frequency sweep complete.")

def save_results():
    if not hearing_results:
        messagebox.showerror("No Results", "No hearing test results to save.")
        return
    file_path = "hearing_test_results.txt"
    with open(file_path, "w") as file:
        for freq, heard in hearing_results:
            file.write(f"Frequency: {freq} Hz, Heard: {heard}\n")
    messagebox.showinfo("Results Saved", f"Results saved to {file_path}.")

def reset_settings():
    frequency_var.set(1000.0)
    volume_var.set(50.0)
    hearing_results.clear()

def set_frequency():
    try:
        frequency_var.set(float(frequency_entry.get()))
    except ValueError:
        messagebox.showerror("Invalid Input", "Please enter a valid frequency.")

def set_volume():
    try:
        volume_var.set(float(volume_entry.get()))
    except ValueError:
        messagebox.showerror("Invalid Input", "Please enter a valid volume.")

root = tk.Tk()
root.title("Hearing Test Signal Generator")

frequency_var = tk.DoubleVar(value=1000.0)
volume_var = tk.DoubleVar(value=50.0)
hearing_results = []

ttk.Label(root, text="Frequency (Hz):").pack()
frequency_entry = ttk.Entry(root, textvariable=frequency_var)
frequency_entry.pack()

frequency_slider = ttk.Scale(root, from_=20, to=20000, variable=frequency_var,
    orient='horizontal')
frequency_slider.pack()

set_freq_button = ttk.Button(root, text="Set Frequency", command=set_frequency)
set_freq_button.pack()

ttk.Label(root, text="Volume (%):").pack()
volume_entry = ttk.Entry(root, textvariable=volume_var)
volume_entry.pack()

volume_slider = ttk.Scale(root, from_=0, to=100, variable=volume_var, orient='
    horizontal')
volume_slider.pack()

set_vol_button = ttk.Button(root, text="Set Volume", command=set_volume)
set_vol_button.pack()

play_button = ttk.Button(root, text="Play", command=play_sound)
play_button.pack()

sweep_button = ttk.Button(root, text="Sweep", command=sweep_frequencies)
sweep_button.pack()

save_button = ttk.Button(root, text="Save Results", command=save_results)
save_button.pack()

reset_button = ttk.Button(root, text="Reset", command=reset_settings)
reset_button.pack()

root.mainloop()
```