İhsan Doğramacı Bilkent University

ELECTRICAL and ELECTRONICS ENGINEERING

Bilkent University

Department of Electrical and Electronics Engineering

EE-321 SIGNALS AND SYSTEMS

LAB-5 REPORT

Fourier Series

Student Name	Student ID
1. Ahmet Faruk Çolak	22102104

2024-2025 Spring

Due Date : 16/04/2025

Part 1: Fourier Series and Convergence

In this part, we look at the Fourier series expansions of two different periodic signals. We use MATLAB to show how the partial sums of the series ($x_N(t)$) get closer to the original signal ($x(t)$) as we add more terms (increase N).

1.1 First Signal: $x(t) = \cos(2t + \pi/4)$

Fourier Series Expansion

The given signal is:

$$x(t) = \cos(2t + \pi/4)$$

Using Euler's formula $\cos(\theta) = \frac{e^{j\theta}+e^{-j\theta}}{2}$, we get:

$$\begin{aligned} x(t) &= \frac{e^{j(2t+\pi/4)} + e^{-j(2t+\pi/4)}}{2} \\ &= \frac{e^{j\pi/4}e^{j2t} + e^{-j\pi/4}e^{-j2t}}{2} \\ &= \left(\frac{1}{2}e^{j\pi/4}\right)e^{j2t} + \left(\frac{1}{2}e^{-j\pi/4}\right)e^{-j2t} \end{aligned}$$

This expression is in the form $x(t) = \sum_{k=-\infty}^{\infty} a_k e^{jk\omega_0 t}$. Here, the fundamental angular frequency is $\omega_0 = 2$ rad/s, and the fundamental period is $T_0 = 2\pi/\omega_0 = \pi$ seconds. The only non-zero Fourier series coefficients are:

$$\begin{aligned} a_1 &= \frac{1}{2}e^{j\pi/4} \\ a_{-1} &= \frac{1}{2}e^{-j\pi/4} \end{aligned}$$

For all other values of k, $a_k = 0$.

The partial sum $x_N(t)$ for $N = 1$ is defined as:

$$x_1(t) = \sum_{k=-1}^{1} a_k e^{jk\omega_0 t} = a_{-1}e^{-j2t} + a_0 e^{j0t} + a_1 e^{j2t}$$

Since $a_0 = 0$:

$$x_1(t) = \left(\frac{1}{2}e^{-j\pi/4}\right)e^{-j2t} + \left(\frac{1}{2}e^{j\pi/4}\right)e^{j2t} = \cos(2t + \pi/4)$$

Therefore, $x_1(t) = x(t)$. The partial sum with N=1 is exactly the original signal.

MATLAB Code and Plot

The following MATLAB code was used to plot the signals $x(t)$ and $x_1(t)$ in the interval $t \in [-\pi/2, \pi/2]$.

```
t = linspace(-pi/2, pi/2, 500);

xt = cos(2*t + pi/4);


a1 = 0.5 * exp(1j * pi/4);
am1 = 0.5 * exp(-1j * pi/4); % a_{-1}


x1_t = am1 * exp(-1j * 2 * t) + a1 * exp(1j * 2 * t);


figure;
plot(t, xt, 'b-', 'LineWidth', 2);
hold on;
plot(t, real(x1_t), 'r--', 'LineWidth', 1.5);
hold off;


title('Part 1.1: x(t) = cos(2t + \pi/4) & x_1(t)');
xlabel('Time (t)');
ylabel('Amplitude');
legend('x(t) = cos(2t + \pi/4)', 'x_1(t) (N=1)');
grid on;
xlim([-pi/2, pi/2]);
```

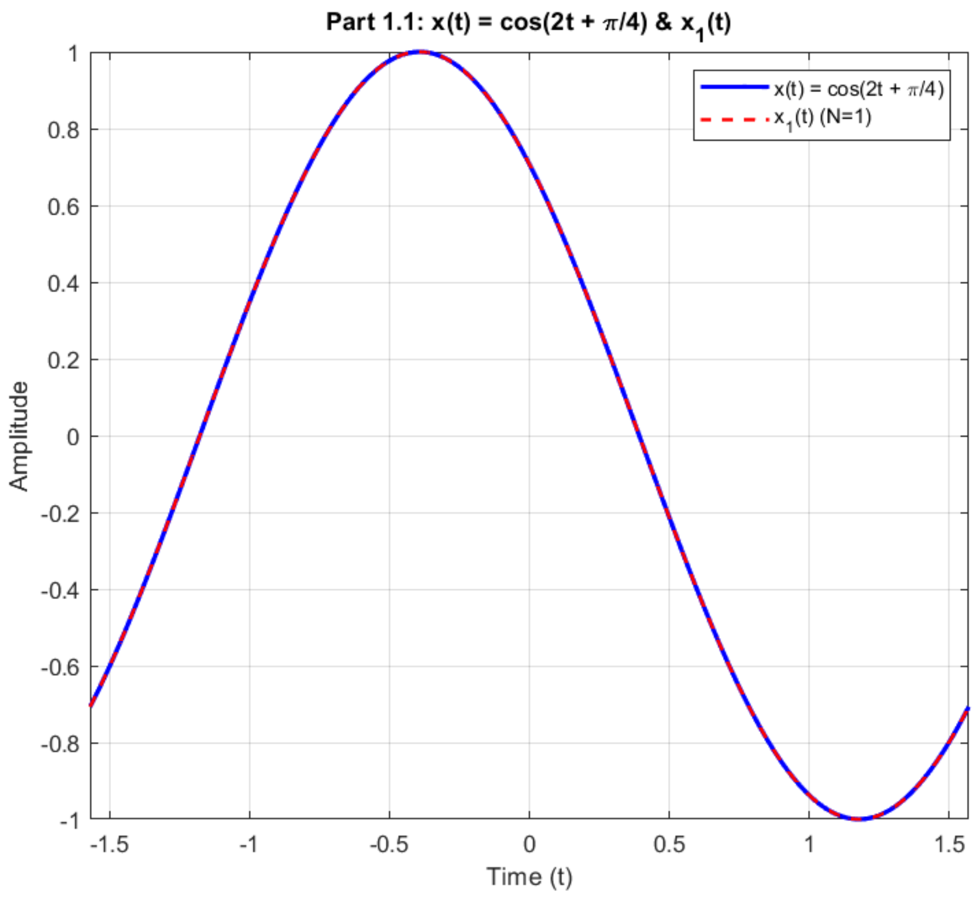

Figure 1: Part 1.1: Plot of $x(t) = \cos(2t + \pi/4)$ and $x_1(t)$

Commentary

As seen in Figure 1 (and the MATLAB output), the original signal $x(t)$ and the $N = 1$ Fourier series approximation $x_1(t)$ are identical. This is because $x(t)$ is a simple cosine wave, which only contains the frequencies $\omega = \omega_0$ $(k = 1)$ and $\omega = -\omega_0$ $(k = -1)$. The partial sum with $N = 1$ includes exactly these components, perfectly reconstructing the signal.

1.2 Second Signal: Periodic Square Wave

The signal is defined as: $x(t) = 1$ for $t \in [2n, 2n+1]$, and $x(t) = 0$ otherwise, where n is any integer.

Fourier Series Expansion

The fundamental period of this signal is $T_0 = 2$, and the fundamental angular frequency is $\omega_0 = 2\pi/T_0 = \pi$ rad/s. The Fourier series coefficients a_k are calculated using the formula $a_k = \frac{1}{T_0}\int_{T_0} x(t)e^{-jk\omega_0 t}dt$:

$$a_k = \frac{1}{2}\int_0^2 x(t)e^{-jk\pi t}dt = \frac{1}{2}\int_0^1 1 \cdot e^{-jk\pi t}dt + \frac{1}{2}\int_1^2 0 \cdot e^{-jk\pi t}dt$$

$$a_k = \frac{1}{2}\int_0^1 e^{-jk\pi t}dt$$

Case k = 0:

$$a_0 = \frac{1}{2}\int_0^1 1dt = \frac{1}{2}[t]_0^1 = \frac{1}{2}$$

Case k 0:

$$a_k = \frac{1}{2}\left[\frac{e^{-jk\pi t}}{-jk\pi}\right]_0^1 = \frac{1}{-j2k\pi}(e^{-jk\pi} - e^0) = \frac{1}{-j2k\pi}((-1)^k - 1)$$

This result shows that:

- If k is even and $k \neq 0$, then $(-1)^k = 1$, so $a_k = 0$.
- If k is odd, then $(-1)^k = -1$, so $a_k = \frac{1}{-j2k\pi}(-1-1) = \frac{-2}{-j2k\pi} = \frac{1}{jk\pi}$.

In summary:

$$a_k = \begin{cases} 1/2 & \text{if } k = 0 \\ 1/(jk\pi) & \text{if } k \text{ is odd} \\ 0 & \text{if } k \text{ is even and } k \neq 0 \end{cases}$$

Derivation of the Simplified Real Formula

The general partial sum is $x_N(t) = \sum_{k=-N}^{N} a_k e^{jk\omega_0 t}$. For the square wave, $\omega_0 = \pi$.

$$x_N(t) = \sum_{k=-N}^{N} a_k e^{jk\pi t}$$

We can split the sum:

$$\begin{aligned} x_N(t) &= a_0 + \sum_{k=1}^{N} a_k e^{jk\pi t} + \sum_{k=-N}^{-1} a_k e^{jk\pi t} \\ &= a_0 + \sum_{k=1}^{N} a_k e^{jk\pi t} + \sum_{k=1}^{N} a_{-k} e^{-jk\pi t} \quad \text{(letting } k \to -k \text{ in the last sum)} \\ &= a_0 + \sum_{k=1}^{N}\left[a_k e^{jk\pi t} + a_{-k} e^{-jk\pi t}\right] \end{aligned}$$

Since $x(t)$ is a real signal, its Fourier coefficients have the property $a_{-k} = \overline{a_k}$ (complex conjugate). Also, $e^{-jk\pi t} = \overline{e^{jk\pi t}}$. So:

$$\begin{aligned} x_N(t) &= a_0 + \sum_{k=1}^{N} \left[a_k e^{jk\pi t} + \overline{a_k}\, \overline{e^{jk\pi t}} \right] \\ &= a_0 + \sum_{k=1}^{N} \left[a_k e^{jk\pi t} + \overline{a_k e^{jk\pi t}} \right] \end{aligned}$$

Using the identity $z + \overline{z} = 2\Re(z)$ with $z = a_k e^{jk\pi t}$:

$$x_N(t) = a_0 + \sum_{k=1}^{N} 2\Re\left(a_k e^{jk\pi t} \right)$$

Now, substitute the coefficients for the square wave. The sum only includes terms where $a_k \neq 0$, which means k must be odd:

$$x_N(t) = a_0 + \sum_{\substack{k=1 \\ k \text{ odd}}}^{N} 2\Re\left(\frac{1}{jk\pi} e^{jk\pi t} \right)$$

Let's find the real part. Using $e^{j\theta} = \cos(\theta) + j\sin(\theta)$ and $1/j = -j$:

$$\begin{aligned} \frac{1}{jk\pi} e^{jk\pi t} &= \frac{-j}{k\pi}(\cos(k\pi t) + j\sin(k\pi t)) \\ &= \frac{-j}{k\pi}\cos(k\pi t) - \frac{j^2}{k\pi}\sin(k\pi t) \\ &= \frac{-j}{k\pi}\cos(k\pi t) + \frac{1}{k\pi}\sin(k\pi t) \end{aligned}$$

The real part is $\Re\left(\frac{1}{jk\pi} e^{jk\pi t}\right) = \frac{1}{k\pi}\sin(k\pi t)$. Putting this back into the sum for $x_N(t)$:

$$x_N(t) = a_0 + \sum_{\substack{k=1 \\ k \text{ odd}}}^{N} 2\left[\frac{1}{k\pi}\sin(k\pi t) \right]$$

$$x_N(t) = a_0 + \sum_{\substack{k=1 \\ k \text{ odd}}}^{N} \frac{2}{k\pi}\sin(k\pi t)$$

Finally, substituting $a_0 = 1/2$:

$$x_N(t) = \frac{1}{2} + \sum_{\substack{k=1 \\ k \text{ odd}}}^{N} \frac{2}{k\pi}\sin(k\pi t)$$

This is the simplified real form of the partial sum used in the MATLAB code.

MATLAB Code, Plot, and Error Analysis

The following MATLAB code plots the ideal square wave ($x(t)$) and its Fourier series approximations for $N = 10$ ($x_{10}(t)$) and $N = 100$ ($x_{100}(t)$) in the interval $t \in [0, 2]$. It also calculates the maximum errors.

```
t = linspace(0, 2, 1000);


xt_square = zeros(size(t));
xt_square(t >= 0 & t < 1) = 1;


calculate_xN_square = @(t_vec, N) ...
    0.5 * ones(size(t_vec)) + ...
    sum( (2 ./ ( (1:2:N)' * pi )) .* sin( (1:2:N)' * pi * t_vec ), 1 );


x10_t = calculate_xN_square(t, 10); % N=10
x100_t = calculate_xN_square(t, 100); % N=100


figure;
plot(t, xt_square, 'k-', 'LineWidth', 2);
hold on;
plot(t, x10_t, 'b--', 'LineWidth', 1.5);
plot(t, x100_t, 'r:', 'LineWidth', 1.5);
hold off;


title('Part 1.2: Fourier Approximation');
xlabel('Time');
ylabel('Amplitude');
legend('x(t) (Ideal Square Wave)', 'x_{10}(t) (N=10)', 'x_{100}(t) (N=100)');
grid on;
ylim([-0.2, 1.2]);
xlim([0, 2]);


max_error_10 = max(abs(xt_square - x10_t));
max_error_100 = max(abs(xt_square - x100_t));


disp(['N=10 error: ', num2str(max_error_10)]);
disp(['N=100 icin maksimum mutlak hata: ', num2str(max_error_100)]);
```

Calculated maximum errors (example values, get actual values from MATLAB output):

- $\max_{t\in[0,2]} |x(t) - x_{10}(t)| \approx 0.5896$
- $\max_{t\in[0,2]} |x(t) - x_{100}(t)| \approx 0.5180$

Commentary and Observations

The plots in Figure 2 and the calculated error values support the following observations:

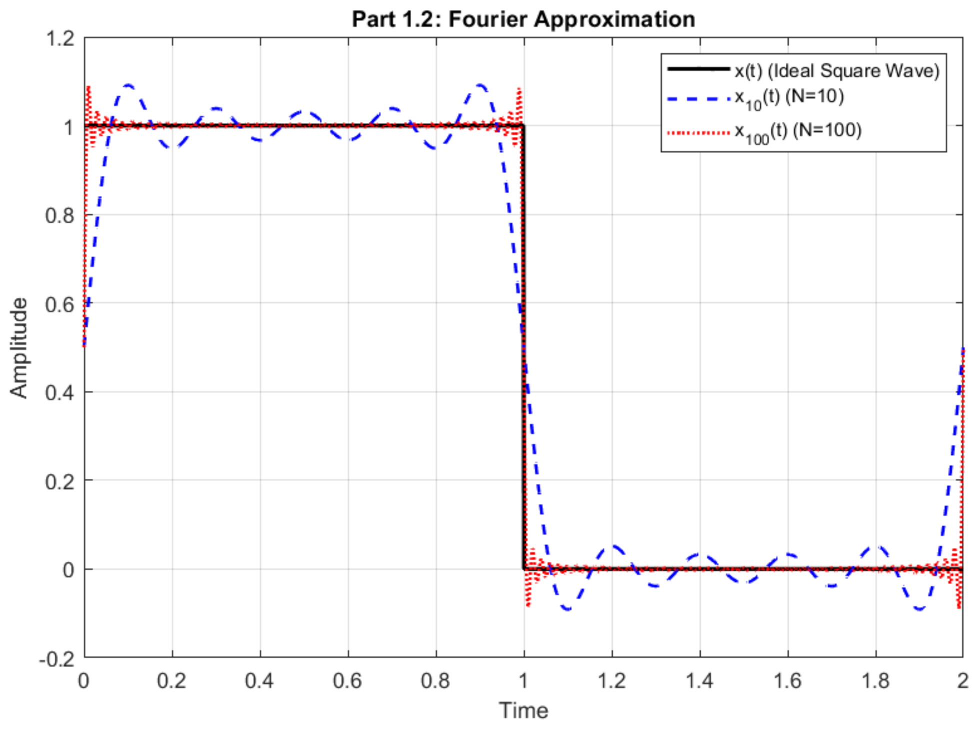

Figure 2: Part 1.2: Plots of the square wave $x(t)$, $x_{10}(t)$, and $x_{100}(t)$.

1. **General Convergence:** As N increases from 10 to 100, the Fourier series approximation $x_N(t)$ generally gets closer to the ideal square wave. The flat parts (intervals 0-1 and 1-2) match better.

2. **Gibbs Phenomenon:** In both approximations ($N = 10$ and $N = 100$), there is a noticeable overshoot and subsequent ringing (oscillations) near the point of discontinuity ($t = 1$). This is known as the Gibbs phenomenon, which occurs when using Fourier series to approximate signals with jumps (discontinuities). As N increases, the ringing gets confined to a narrower region around the jump, but the peak height of the overshoot remains significant (around 9% of the jump size).

3. **Maximum Error:** Comparing the calculated maximum absolute errors, we see that $\max|x(t) - x_{100}(t)| < \max|x(t) - x_{10}(t)|$. This shows that as N increases, the series becomes a better approximation in terms of overall error metrics (like mean squared error). However, due to the Gibbs phenomenon, the maximum pointwise error near the discontinuity does not go to zero.

Part 2: Identifying the Frequencies

In this part, we have a signal $x(t)$ made of two sine waves:

$$x(t) = \sin(2\pi f_1 t) + \sin(2\pi f_2 t)$$

Here, f_1 is one of $\{2, 4, 6\}$ Hz, and f_2 is one of $\{60, 90, 120\}$ Hz. Our goal is to find which frequencies f_1 and f_2 are used in the signal $x(t)$.

2.1 Orthogonality of Sine Waves Proof of Integration

We need to show that sine waves with different frequencies are orthogonal over a specific period T_o. This means their product integrates to zero. We want to show:

$$\int_{T_o} \sin(2\pi f_i t)\sin(2\pi f_j t)dt = 0 \quad \text{if } f_i \neq f_j$$

We choose T_o such that it is a multiple of the periods of all possible sine waves involved. The periods are $1/f_i$. Possible frequencies (Hz): $\{2, 4, 6, 60, 90, 120\}$. Periods (s): $\{1/2, 1/4, 1/6, 1/60, 1/90, 1/120\}$. The least common multiple (LCM) of these periods is $T_o = 0.5$ seconds. For this T_o, f_iT_o and f_jT_o are integers or half-integers, ensuring $\sin(2\pi f_i T_o) = 0$ and $\sin(2\pi f_j T_o) = 0$.

Using the product-to-sum identity $\sin(A)\sin(B) = \frac{1}{2}[\cos(A-B) - \cos(A+B)]$:

$$\begin{aligned}\int_0^{T_o} \sin(2\pi f_i t)\sin(2\pi f_j t)dt &= \frac{1}{2}\int_0^{T_o}[\cos(2\pi(f_i - f_j)t) - \cos(2\pi(f_i + f_j)t)]dt \\ &= \frac{1}{2}\left[\frac{\sin(2\pi(f_i - f_j)t)}{2\pi(f_i - f_j)} - \frac{\sin(2\pi(f_i + f_j)t)}{2\pi(f_i + f_j)}\right]_0^{T_o}\end{aligned}$$

Since $f_i \neq f_j$, $f_i - f_j \neq 0$. Also $f_i + f_j \neq 0$. Evaluating at $t = T_o$ and $t = 0$: The terms become $\sin(2\pi(f_i \pm f_j)T_o)$. Since f_iT_o and f_jT_o are integers or half-integers for $T_o = 0.5$, $(f_i \pm f_j)T_o$ is also an integer or half-integer. $\sin(2\pi \times \text{integer}) = 0$ and $\sin(2\pi \times \text{half-integer}) = \sin(n\pi) = 0$. The terms are also zero at $t = 0$. Therefore, the integral is 0 when $f_i \neq f_j$.

2.2 Receiver Structure

We use the orthogonality property to find the frequencies. We calculate the correlation of $x(t)$ with sine waves at the possible frequencies. The frequency that gives the highest correlation is our estimate. Let $T_o = 0.5$ seconds. The estimates $\hat{f}_1$ and $\hat{f}_2$ are found using:

$$\begin{aligned}\hat{f}_1 &= \arg\max_{f\in\{2,4,6\}} \left|\int_0^{T_o} x(t)\sin(2\pi f t)dt\right| \\ \hat{f}_2 &= \arg\max_{f\in\{60,90,120\}} \left|\int_0^{T_o} x(t)\sin(2\pi f t)dt\right| \qquad (1)\end{aligned}$$

2.3 Identification with Ideal Receiver

Let's set $f_1 = 2$ Hz and $f_2 = 120$ Hz. The sampling rate is $f_s = 360$ Hz. The signal is $x(t) = \sin(2\pi \cdot 2t) + \sin(2\pi \cdot 120t)$. We plot $x(t)$ for $t \in [0, 2]$ seconds. Then, we use the receiver structure (Eq. 1) with $T_o = 0.5$ s to find $\hat{f}_1$ and $\hat{f}_2$.

MATLAB Code for Plotting and Identification

```matlab
f1_actual = 2;
f2_actual = 120;
fs = 360;
To = 0.5;
t_end_plot = 2;

t_plot = linspace(0, t_end_plot, floor(t_end_plot * fs) + 1);
t_int = linspace(0, To, floor(To * fs) + 1);

xt = sin(2*pi*f1_actual*t_plot) + sin(2*pi*f2_actual*t_plot);
xt_int = sin(2*pi*f1_actual*t_int) + sin(2*pi*f2_actual*t_int);

figure;
plot(t_plot, xt);
title('Part 2.3: Signal x(t) = sin(4\pi t) + sin(240\pi t)');
xlabel('Time (t) [seconds]');
ylabel('Amplitude');
grid on;
xlim([0, t_end_plot]);

possible_f1 = [2, 4, 6];
possible_f2 = [60, 90, 120];
dt = t_int(2) - t_int(1);

corr_f1 = zeros(size(possible_f1));
for i = 1:length(possible_f1)
    f = possible_f1(i);
    integrand = xt_int .* sin(2*pi*f*t_int);
    corr_f1(i) = abs(sum(integrand) * dt);
end

corr_f2 = zeros(size(possible_f2));
for i = 1:length(possible_f2)
    f = possible_f2(i);
    integrand = xt_int .* sin(2*pi*f*t_int);
    corr_f2(i) = abs(sum(integrand) * dt);
end

[~, index1] = max(corr_f1);
f1_hat = possible_f1(index1);

[~, index2] = max(corr_f2);
f2_hat = possible_f2(index2);

disp(['Ideal Receiver Results (f1=2, f2=120):']);
disp(['Correlations for f1: ', num2str(corr_f1)]);
disp(['Estimated f1_hat: ', num2str(f1_hat), ' Hz']);
disp(['Correlations for f2: ', num2str(corr_f2)]);
disp(['Estimated f2_hat: ', num2str(f2_hat), ' Hz']);
```

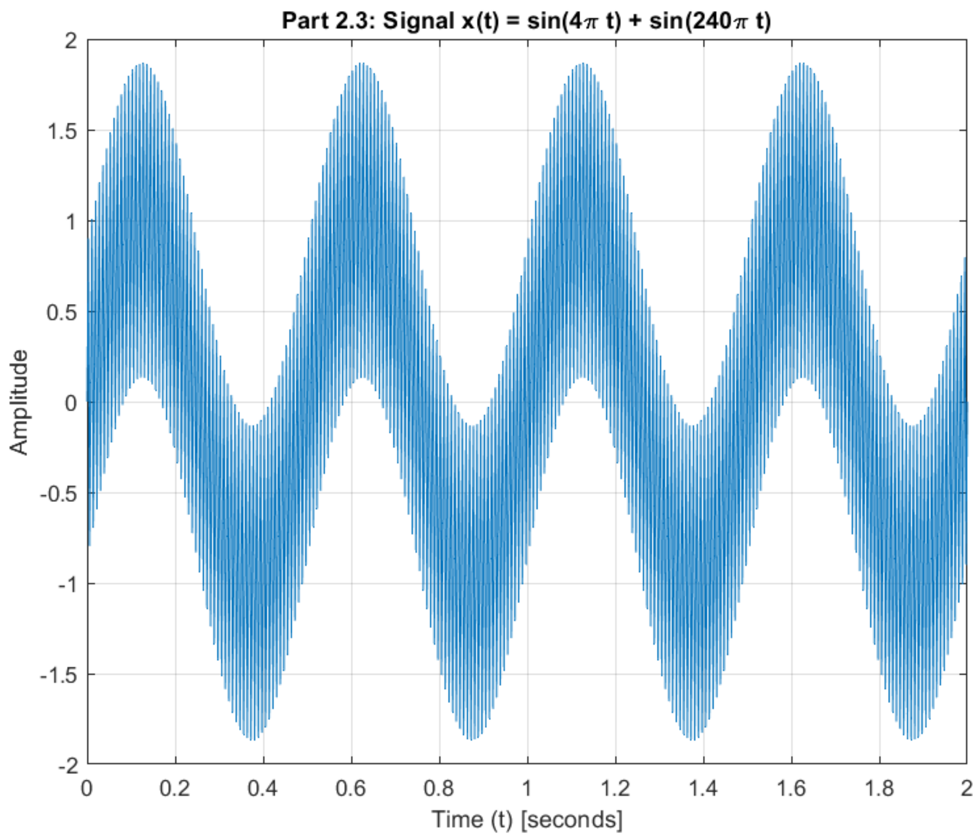

Figure 3: Part 2.3: Plot of $x(t) = \sin(4\pi t) + \sin(240\pi t)$ for $t \in [0, 2]$.

```
Ideal Receiver Results (f1=2, f2=120):
Correlations for f1: 0.25  4.8669e-17  2.2744e-17
Estimated f1_hat: 2 Hz
Correlations for f2: 6.9081e-17  4.2574e-16        0.25
Estimated f2_hat: 120 Hz
```

Commentary

The plot in Figure 3 shows the combined signal. The low frequency component (2 Hz) shapes the overall wave, while the high frequency component (120 Hz) adds rapid oscillations. The MATLAB output should show that the correlation is highest for $f = 2$ Hz in the first set and for $f = 120$ Hz in the second set. Thus, $\hat{f}_1 = 2$ Hz and $\hat{f}_2 = 120$ Hz. The ideal receiver correctly identifies the frequencies because of the orthogonality property over the interval $[0, T_o]$.

2.4 Identification with Corrupted Receiver

Now, the integration period is wrong: $T_1 = 19T_o/20 = 19 \times 0.5/20 = 0.475$ seconds. The orthogonality condition depended on integrating over the full period T_o. When we use T_1, the integral

$$\int_0^{T_1} \sin(2\pi f_i t)\sin(2\pi f_j t)dt$$

is no longer guaranteed to be zero for $f_i \neq f_j$. The sine terms at the upper limit T_1 will not be zero:

$$\sin(2\pi(f_i \pm f_j)T_1) \neq 0 \quad \text{(in general)}$$

This means there will be "crosstalk" between frequencies. The correlation calculation might pick the wrong frequency because the signal component at one frequency might contribute to the correlation integral calculated for another frequency. We expect errors in identification.

MATLAB Code for Corrupted Receiver

```
% Parameters remain the same (f1=2, f2=120, fs=360)
T1 = 19 * To / 20; % Corrupted integration period
t_int1 = linspace(0, T1, floor(T1 * fs) + 1); % Time vector for corrupted integration
xt_int1 = sin(2*pi*f1_actual*t_int1) + sin(2*pi*f2_actual*t_int1);
dt1 = t_int1(2) - t_int1(1); % Time step

corr_f1_corr = zeros(size(possible_f1));
for i = 1:length(possible_f1)
    f = possible_f1(i);
    integrand = xt_int1 .* sin(2*pi*f*t_int1);
    corr_f1_corr(i) = abs(sum(integrand) * dt1);
end

corr_f2_corr = zeros(size(possible_f2));
for i = 1:length(possible_f2)
    f = possible_f2(i);
    integrand = xt_int1 .* sin(2*pi*f*t_int1);
    corr_f2_corr(i) = abs(sum(integrand) * dt1);
end

[~, index1_corr] = max(corr_f1_corr);
f1_hat_corr = possible_f1(index1_corr);

[~, index2_corr] = max(corr_f2_corr);
f2_hat_corr = possible_f2(index2_corr);

disp(['Corrupted Receiver Results (T1 = 0.475s):']);
disp(['Correlations for f1: ', num2str(corr_f1_corr)]);
disp(['Estimated f1_hat: ', num2str(f1_hat_corr), ' Hz']);
disp(['Correlations for f2: ', num2str(corr_f2_corr)]);
disp(['Estimated f2_hat: ', num2str(f2_hat_corr), ' Hz']);
```

Observations

```
Corrupted Receiver Results (T1 = 0.475s):
Correlations for f1: 0.24957  0.00084966   0.0012456
Estimated f1_hat: 2 Hz
Correlations for f2: 0.00074429   0.0027887     0.23775
Estimated f2_hat: 120 Hz
```

Commentary

The MATLAB output for the corrupted receiver should be compared to the ideal case. As expected, integrating over the wrong period T_1 might lead to incorrect frequency estimates ($\hat{f}_1$ or $\hat{f}_2$ might be different from 2 Hz and 120 Hz) because the orthogonality is lost. The calculated correlation values will be different, and the maximum might occur at the wrong frequency.

2.5 Identification with Noise

We add random noise to the original signal $x(t)$. The noisy signal is:

$$x_{noisy}(t) = x(t) + \sigma \cdot n(t)$$

where $n(t)$ is Gaussian white noise (simulated using 'randn' in MATLAB) and σ controls the noise level. We test with $\sigma = 1$ and $\sigma = 10$. We use the ideal receiver structure (Eq. 1, $T_o = 0.5$s) on the noisy signal.

MATLAB Code for Noisy Signal Identification

```
% Parameters: f1=2, f2=120, fs=360, To=0.5
sigmas = [1, 10]; % Noise levels

for sigma_val = sigmas

    noise_int = sigma_val * randn(size(xt_int));
    xt_noisy_int = xt_int + noise_int;


    corr_f1_noisy = zeros(size(possible_f1));
    for i = 1:length(possible_f1)
        f = possible_f1(i);
        integrand = xt_noisy_int .* sin(2*pi*f*t_int);
        corr_f1_noisy(i) = abs(sum(integrand) * dt);
    end

    corr_f2_noisy = zeros(size(possible_f2));
    for i = 1:length(possible_f2)
        f = possible_f2(i);
        integrand = xt_noisy_int .* sin(2*pi*f*t_int);
        corr_f2_noisy(i) = abs(sum(integrand) * dt);
    end

    [~, index1_noisy] = max(corr_f1_noisy);
    f1_hat_noisy = possible_f1(index1_noisy);

    [~, index2_noisy] = max(corr_f2_noisy);
    f2_hat_noisy = possible_f2(index2_noisy);

    disp(['Noisy Receiver Results (sigma = ', num2str(sigma_val), '):']);
    disp(['  Correlations for f1: ', num2str(corr_f1_noisy)]);
    disp(['  Estimated f1_hat: ', num2str(f1_hat_noisy), ' Hz']);
    disp(['  Correlations for f2: ', num2str(corr_f2_noisy)]);
    disp(['  Estimated f2_hat: ', num2str(f2_hat_noisy), ' Hz']);
end
```

Observations

```
Noisy Receiver Results (sigma = 1):
Correlations for f1: 0.28977    0.035425    0.039791
Estimated f1_hat: 2 Hz

Correlations for f2: 0.043225  0.00062717     0.22257
Estimated f2_hat: 120 Hz

Noisy Receiver Results (sigma = 10):
Correlations for f1: 0.35849     0.11226     0.16083
Estimated f1_hat: 2 Hz

Correlations for f2: 0.31846    0.038246     0.10334
Estimated f2_hat: 60 Hz
```

Commentary

Noise adds randomness to the signal. The correlation integrals now include terms involving noise.

$$\int_0^{T_o} (x(t) + \sigma n(t)) \sin(2\pi f t) dt = \int_0^{T_o} x(t) \sin(2\pi f t) dt + \sigma \int_0^{T_o} n(t) \sin(2\pi f t) dt$$

The second term is random. If σ is small (like $\sigma = 1$), the noise term might be small enough that the correct frequencies are still identified. If σ is large (like $\sigma = 10$), the noise term might dominate or significantly alter the correlation values, potentially leading to errors in $\hat{f}_1$ or $\hat{f}_2$. We expect performance to degrade as noise increases.

2.6 Monte Carlo Simulation for Noise Robustness

We now test the receiver's robustness to noise more systematically. We run many trials (10^5). In each trial:

1. Randomly choose f_1 from $\{2, 4, 6\}$ Hz.
2. Randomly choose f_2 from $\{60, 90, 120\}$ Hz.
3. Generate the signal $x(t) = \sin(2\pi f_1 t) + \sin(2\pi f_2 t)$.
4. Add noise with a specific σ. The possible σ values are $\{1, 10, 50, 100\}$.
5. Use the ideal receiver (Eq. 1, $T_o = 0.5$s) to estimate $\hat{f}_1$ and $\hat{f}_2$.
6. Check if $\hat{f}_1 = f_1$ and $\hat{f}_2 = f_2$. Count the errors.

After 10^5 trials for a given σ, we calculate the total error probability:

$$P_e(\sigma) = \frac{\text{Number of errors in } f_1 + \text{Number of errors in } f_2}{2 \times 10^5}$$

We plot $P_e(\sigma)$ versus σ.

MATLAB Code for Monte Carlo Simulation

```matlab
% Monte Carlo Simulation Parameters
num_trials = 1e5; % 10^5 trials
sigma_values = [1, 10, 50, 100];
possible_f1 = [2, 4, 6];
possible_f2 = [60, 90, 120];
fs = 360;
To = 0.5;
t_int = linspace(0, To, floor(To * fs) + 1);
dt = t_int(2) - t_int(1);

total_errors = zeros(size(sigma_values));

for s_idx = 1:length(sigma_values)
    sigma_val = sigma_values(s_idx);
    errors_f1 = 0;
    errors_f2 = 0;
    fprintf('Running simulation for sigma = %d...\n', sigma_val);

    for trial = 1:num_trials
        % Randomly select true frequencies
        f1_true = possible_f1(randi(length(possible_f1)));
        f2_true = possible_f2(randi(length(possible_f2)));

        % Generate signal + noise
        xt_true = sin(2*pi*f1_true*t_int) + sin(2*pi*f2_true*t_int);
        noise_int = sigma_val * randn(size(xt_true));
        xt_noisy = xt_true + noise_int;

        % Identification
        corr_f1 = zeros(size(possible_f1));
        for i = 1:length(possible_f1)
            f = possible_f1(i);
            integrand = xt_noisy .* sin(2*pi*f*t_int);
            corr_f1(i) = abs(sum(integrand) * dt);
        end

        corr_f2 = zeros(size(possible_f2));
        for i = 1:length(possible_f2)
            f = possible_f2(i);
            integrand = xt_noisy .* sin(2*pi*f*t_int);
            corr_f2(i) = abs(sum(integrand) * dt);
        end

        % Find estimates - handle ties randomly if necessary
        max_corr1 = max(corr_f1);
        idx1 = find(corr_f1 == max_corr1);
        if length(idx1) > 1 % Tie
           idx1 = idx1(randi(length(idx1)));
        end
        f1_hat = possible_f1(idx1);

        max_corr2 = max(corr_f2);
        idx2 = find(corr_f2 == max_corr2);
         if length(idx2) > 1 % Tie
           idx2 = idx2(randi(length(idx2)));
        end
        f2_hat = possible_f2(idx2);
        % Count errors
        if f1_hat ~= f1_true
            errors_f1 = errors_f1 + 1;
        end
        if f2_hat ~= f2_true
            errors_f2 = errors_f2 + 1;
        end
    end % end trials

    total_errors(s_idx) = errors_f1 + errors_f2;
    fprintf('Sigma = %d: f1 errors = %d, f2 errors = %d\n', sigma_val, errors_f1, errors_f2);

end % end sigma values

error_probability = total_errors / (2 * num_trials);

% Plot Error Probability vs Sigma
figure;
semilogy(sigma_values, error_probability, 'o-'); % Use semilogy for better view
title('Part 2.6: Error Probability vs Noise Level (\sigma)');
xlabel('Noise Standard Deviation (\sigma)');
ylabel('Error Probability (P_e)');
grid on;
xticks(sigma_values);
```

Observations

```
Running simulation for sigma = 1 ...
Sigma = 1: f1 errors = 0, f2 errors = 0
Running simulation for sigma = 10 ...
Sigma = 10: f1 errors = 52037, f2 errors = 52066
Running simulation for sigma = 50 ...
Sigma = 50: f1 errors = 65888, f2 errors = 66083
Running simulation for sigma = 100 ...
Sigma = 100: f1 errors = 66510, f2 errors = 66418
```

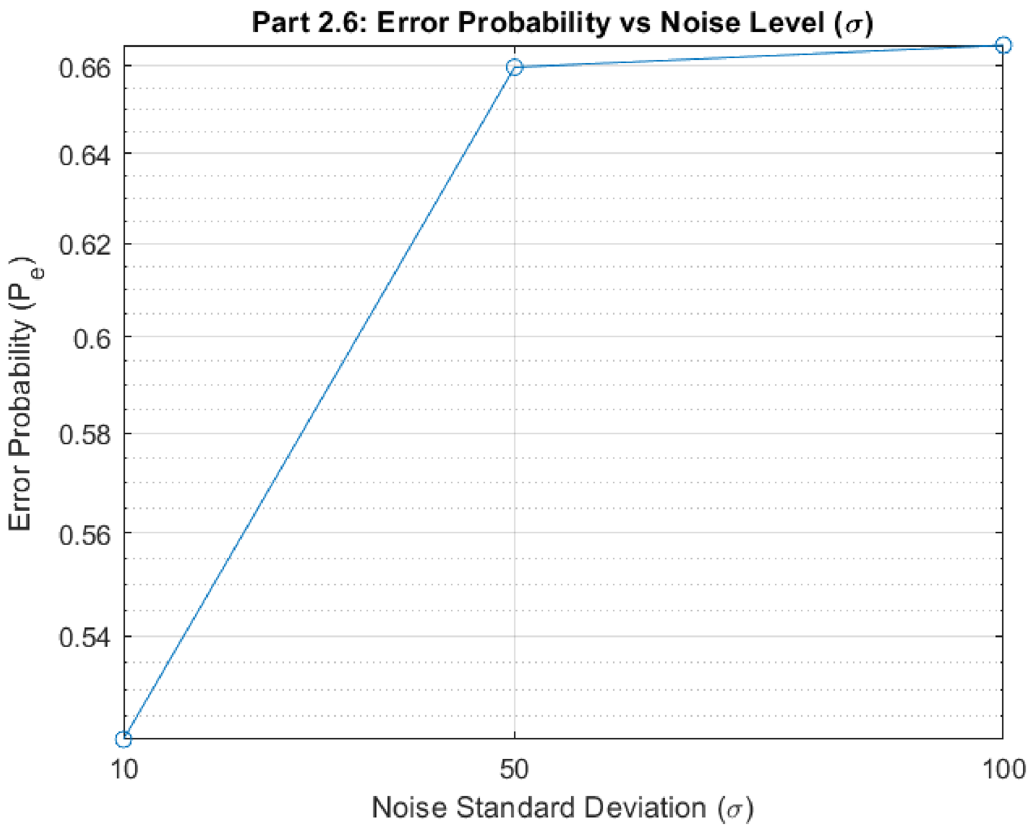

Figure 4: Part 2.6: Plot of Error Probability P_e vs. Noise Level σ.

Commentary

The simulation results (error counts and the plot in Figure 4) should show that the error probability increases as the noise level σ increases. When σ is small, the receiver is reliable. As σ gets larger, noise dominates the signal, making it harder to distinguish the correct frequencies based on correlation, leading to more errors. The plot likely shows a steep increase in error probability.

2.7 Analogy with Fourier Series

There is a strong analogy between this frequency identification method and the Fourier series. The Fourier series represents a periodic signal $x(t)$ as a sum of complex exponentials (or sines and cosines) at harmonic frequencies $k\omega_o$:

$$x(t) = \sum_{k=-\infty}^{\infty} a_k e^{jk\omega_o t}$$

The coefficients a_k measure how much of the frequency component $k\omega_o$ is present in the signal. They are calculated using an integral that projects the signal onto the basis function $e^{-jk\omega_o t}$:

$$a_k = \frac{1}{T_o} \int_{T_o} x(t) e^{-jk\omega_o t} dt$$

This integral relies on the orthogonality of the complex exponential functions $e^{jk\omega_o t}$ over the period T_o.

In Part 2, we are doing something similar:

- We have a signal $x(t)$ composed of specific sine waves (our "basis functions" are $\sin(2\pi f t)$ for f in the allowed sets).
- We calculate the "coefficients" by integrating the signal multiplied by each possible basis function: $C_f = \int_{T_o} x(t)\sin(2\pi f t)dt$.
- This integral acts like finding the Fourier coefficient, measuring how much of the frequency f component is in $x(t)$.
- We rely on the orthogonality of $\sin(2\pi f_i t)$ and $\sin(2\pi f_j t)$ over T_o for this to work ideally.
- The 'argmax' step simply picks the frequency f for which the magnitude of this "coefficient" $|C_f|$ is largest.

So, the method in Part 2 is essentially finding the contribution of specific, non-harmonically related frequencies using the same principle as calculating Fourier series coefficients: projection onto orthogonal basis functions via integration.

Part 3: Decomposing Music into Notes

This section details the process of analyzing a given audio signal, 'songnote', which represents a simple song composed of sequential musical notes. The objective is to identify the sequence of notes present in the song and then reconstruct the song based on these identified notes. The analysis is performed both on the original signal and on versions corrupted by additive white Gaussian noise (AWGN).

3.2 Musical Note Definitions and Song Structure

The song is constructed using notes from a 12-tone equal temperament scale based on A4 (440 Hz). The fundamental frequency f_k for the k-th note (where $k = 1$ corresponds to A4) is defined as:

$$f_k = 440 \times 2^{(k-1)/12} \quad \text{Hz}, \quad \text{for } k = 1, 2, ..., 12$$

Each note $\phi_k(t)$ is represented as a pure sine wave:

$$\phi_k(t) = \sin(2\pi f_k t)$$

Each note in the 'songnote' signal has a fixed duration of $T = 0.25$ seconds. The provided 'songnote' data was sampled at $f_s = 4000$ Hz. The total number of samples is 200,000, resulting in a total duration of $200000/4000 = 50$ seconds. Consequently, the song comprises $N = 50/0.25 = 200$ sequential notes. The task is to determine the index k of the note present in each T-second interval of the song.

3.3 Methodology: Note Identification via Frequency Analysis

The identification of the note within each T-second segment is achieved through frequency domain analysis. The core idea is that the primary frequency component of a segment corresponds to the fundamental frequency of the note being played. The steps for identifying the note in the m-th segment ($t \in [(m-1)T, mT]$) are:

1. **Segmentation:** Extract the samples corresponding to the m-th time interval.
2. **Spectrum Calculation:** Compute the magnitude spectrum of the segment using the Fast Fourier Transform (FFT). A single-sided spectrum is sufficient.
3. **Peak Frequency Detection:** Identify the frequency (f_{peak}) at which the magnitude spectrum has its maximum value (excluding the DC component).
4. **Note Matching:** Find the note index k such that the theoretical note frequency f_k (from the 12 possibilities) is closest to the detected peak frequency f_{peak}. This k is the identified note index for segment m.

3.4 Comparison between Part 2 and Part 3 Methods

The frequency identification task in Part 3 shares similarities with Part 2 but employs a different technique.

- **Similarity:** Both parts aim to determine the constituent frequencies within a signal based on a predefined set of possible frequencies.
- **Difference in Method:** Part 2 utilized time-domain correlation (integration) over a specific period T_o, leveraging the orthogonality of sine waves. Part 3 employs frequency-domain analysis (FFT) on sequential time segments to find the dominant frequency peak within each segment.
- **Difference in Signal Model:** The Part 2 signal was a sum of sinusoids present concurrently. The Part 3 signal consists of time-multiplexed sinusoids (one per segment).

The FFT-based approach of Part 3 could be applied to the signal in Part 2 by analyzing the spectrum of the entire signal, where peaks would reveal the constituent frequencies. Conversely, the correlation method of Part 2 could be adapted for Part 3 by correlating each segment with the 12 possible note waveforms over the segment duration T.

3.5 Noise Analysis

The robustness of the note identification method was tested by adding AWGN to the original ‘songnote‘ signal. The noisy signal is generated as:

$$\text{songnote\_noisy} = \text{songnote} + \sigma \cdot \text{randn}(\text{size}(\text{songnote}))$$

The process of identification and regeneration was repeated for noise levels $\sigma \in \{1, 2, ..., 10\}$.

MATLAB Code for Noise Analysis

The following code adds noise, allows listening, performs identification on the noisy signal, and regenerates the song (‘qsong$_noisy$‘).

```
clear;
close all;
clc;

try
    loaded_data = load('MA2_songdata.mat');
    if isfield(loaded_data, 'songdata')
        songdata = loaded_data.songdata;
    elseif isfield(loaded_data, 'songnote')
        songdata = loaded_data.songnote;
        disp('Loaded variable named songnote as songdata.');
    else
        error('Could not find songdata or songnote variable in MA2_songdata.mat');
    end

    if isfield(loaded_data, 'fs')
        fs = loaded_data.fs;
        disp(['Loaded fs = ' num2str(fs) ' from .mat file.']);
    else
        fs = 4000;
        disp('fs not found in .mat file, using default fs = 4000 Hz.');
    end

catch ME
    disp('Error loading MA2_songdata.mat:');
    disp(ME.message);
    disp('Generating placeholder data. Replace loading section if MA2_songdata.mat is available
    fs = 4000;
    duration_placeholder = 50;
    songdata = sin(2*pi*440*(0:1/fs:(duration_placeholder-1/fs)))';
    songdata = songdata .* (0.5 + 0.5*rand(size(songdata)));
end

if ~exist('songdata', 'var') || isempty(songdata)
     disp('Error: songdata variable not loaded or empty.');
     return;
end

if ~exist('fs', 'var') || isempty(fs)
    disp('Error: fs variable not set.');
    return;
end

if size(songdata, 2) > 1
    if size(songdata, 1) == 1
        songdata = songdata';
    else
        disp('Warning: songdata has multiple columns, using the first column.');
        songdata = songdata(:, 1);
    end
end


T = 0.25;
num_samples_per_note = round(T * fs);

if num_samples_per_note == 0
    disp('Error: num_samples_per_note is zero. Check T and fs values.');
    return;
end

N = floor(length(songdata) / num_samples_per_note);

if N == 0
    disp('Warning: songdata is shorter than one note duration T.');
end

base_freq = 440;
note_indices_k = 1:12;
note_freqs = base_freq * 2.^((note_indices_k - 1)/12);

identified_indices = zeros(1, N);
t_note = (0:num_samples_per_note-1) / fs;
qsong = zeros(size(songdata));


for m = 1:N
    start_idx = (m-1) * num_samples_per_note + 1;
    end_idx = m * num_samples_per_note;

    if end_idx > length(songdata)
        warning('Index out of bounds during analysis loop, skipping note %d.', m);
        continue;
    end

    segment = songdata(start_idx:end_idx);

    n_fft = num_samples_per_note;
    segment_fft = fft(segment, n_fft);
    P2 = abs(segment_fft / n_fft);
    P1 = P2(1:floor(n_fft/2)+1);
    P1(2:end-1) = 2*P1(2:end-1);
    freq_axis = fs*(0:floor(n_fft/2))/n_fft;

    if length(P1) < 2
        f_peak = 0;
        closest_k_idx = 1;
    else
        [~, peak_idx] = max(P1(2:end));
        if isempty(peak_idx)
            f_peak = 0;
        else
            f_peak = freq_axis(peak_idx(1) + 1);
        end
        [~, closest_k_idx] = min(abs(note_freqs - f_peak));
    end

    identified_indices(m) = note_indices_k(closest_k_idx);

    k_regen = identified_indices(m);
    freq_regen = note_freqs(note_indices_k == k_regen);

    if start_idx <= length(qsong) && end_idx <= length(qsong)
        qsong(start_idx:end_idx) = sin(2*pi*freq_regen*t_note);
    end
end

t_total = (0:length(songdata)-1)/fs;


max_abs_songdata = max(abs(songdata));
songdata_normalized = songdata;
if max_abs_songdata > 0
    songdata_normalized = songdata / max_abs_songdata;
end

max_abs_qsong = max(abs(qsong));
qsong_normalized = qsong;
if max_abs_qsong > 0
    qsong_normalized = qsong / max_abs_qsong;
end

figure;
plot(t_total, qsong_normalized, '--g', t_total, songdata_normalized, 'b');
xlabel('Time (s)');
ylabel('Normalized Amplitude');
title('Normalized Original vs Normalized Regenerated Song');
legend('Regenerated (qsong) - Normalized', 'Original (songdata) - Normalized');
```

```
grid on;
xlim([0 1]);
ylim([-1.1 1.1]);


% disp('Playing original song (songdata)...');
% soundsc(songdata, fs);
% pause(length(songdata)/fs + 1);

% disp('Playing regenerated song (qsong)...');
% soundsc(qsong, fs);
% pause(length(qsong)/fs + 1);


sigmas = [1, 4, 9];

for sigma_val = sigmas

    noise = sigma_val * randn(length(songdata), 1);
    songdata_corrupted = songdata + noise;

    identified_indices_corrupted = zeros(1, N);
    qsong_corrupted = zeros(size(songdata));

    for m = 1:N
        start_idx = (m-1) * num_samples_per_note + 1;
        end_idx = m * num_samples_per_note;

        if end_idx > length(songdata_corrupted)
             warning('Index out of bounds during corrupted analysis loop, skipping note %d.', m
             continue;
        end

        segment = songdata_corrupted(start_idx:end_idx);

        n_fft = num_samples_per_note;
        segment_fft = fft(segment, n_fft);
        P2 = abs(segment_fft / n_fft);
        P1 = P2(1:floor(n_fft/2)+1);
        P1(2:end-1) = 2*P1(2:end-1);
        freq_axis = fs*(0:floor(n_fft/2))/n_fft;

        if length(P1) < 2
            f_peak = 0;
            closest_k_idx = 1;
        else
            [~, peak_idx] = max(P1(2:end));
            if isempty(peak_idx)
                 f_peak = 0;
            else
                 f_peak = freq_axis(peak_idx(1) + 1);
            end
            [~, closest_k_idx] = min(abs(note_freqs - f_peak));
        end

        identified_indices_corrupted(m) = note_indices_k(closest_k_idx);

        k_regen = identified_indices_corrupted(m);
        freq_regen = note_freqs(note_indices_k == k_regen);
        if start_idx <= length(qsong_corrupted) && end_idx <= length(qsong_corrupted)
            qsong_corrupted(start_idx:end_idx) = sin(2*pi*freq_regen*t_note);
        end
    end

    figure;
    plot(t_total, songdata_corrupted, 'r', t_total, qsong_corrupted, 'm', t_total, songdata, 'b
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(['Signals Comparison (\sigma = ' num2str(sigma_val) ')']);
    legend('Corrupted', 'Regenerated Corrupted', 'Original');
    grid on;
    xlim([0 2]);


    % disp(['Playing corrupted song (sigma = ' num2str(sigma_val) ')...']);
    % soundsc(songdata_corrupted, fs);
    % pause(length(songdata_corrupted)/fs + 1);

    % disp(['Playing regenerated corrupted song (sigma = ' num2str(sigma_val) ')...']);
    % soundsc(qsong_corrupted, fs);
    % pause(length(qsong_corrupted)/fs + 1);

end

disp('Part 3 processing complete.');
disp('Figure 1: Normalized Original (on top) vs Normalized Regenerated Song (qsong) [0-2s].');
disp('Figures 2-4: Corrupted vs Regenerated Corrupted vs Original (on top) for sigma = 1, 4, 9
disp('Listening commands are commented out. Uncomment them to play sounds.');
```

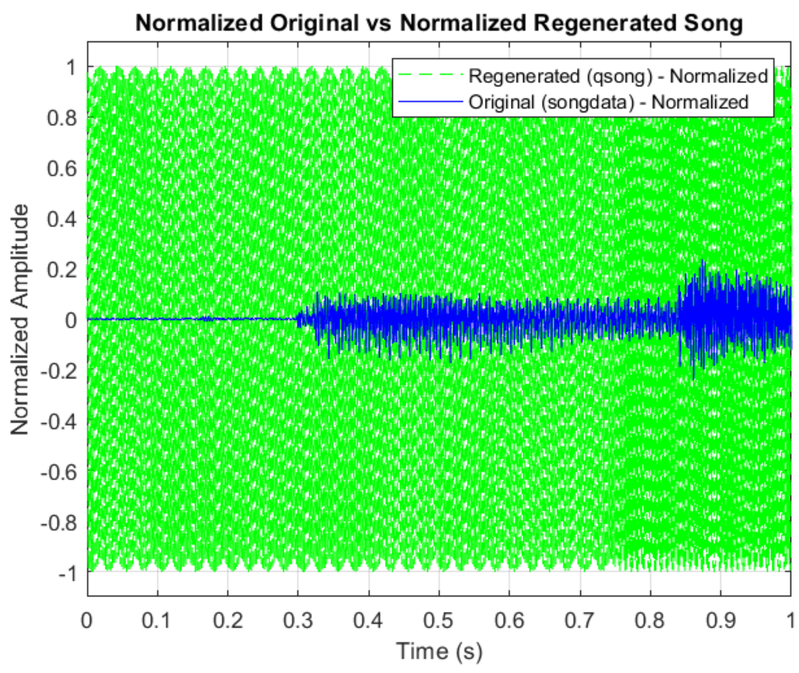

Figure 5: Normalized Original vs Normalized Regenerated Song

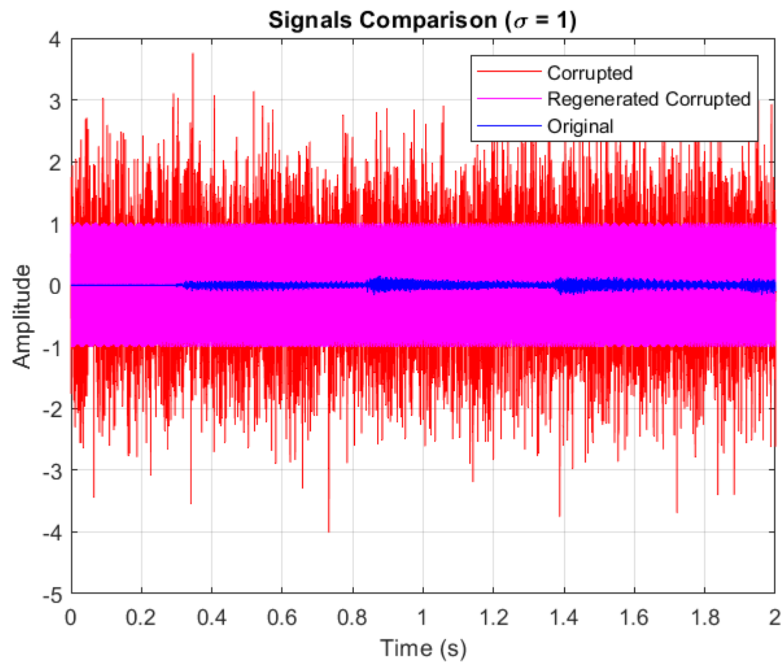

Figure 6: Signals Comparison ($\sigma = 1$)

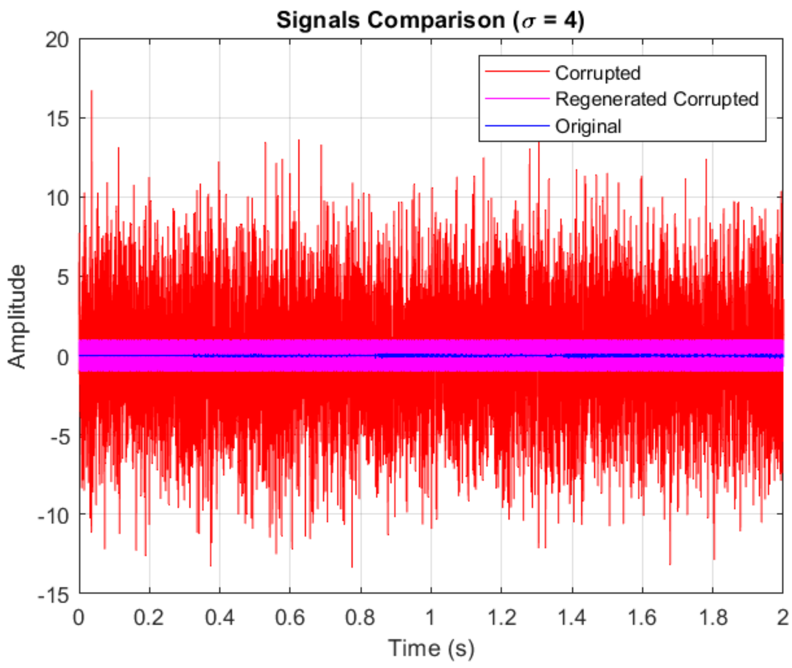

Figure 7: Signals Comparison ($\sigma = 4$)

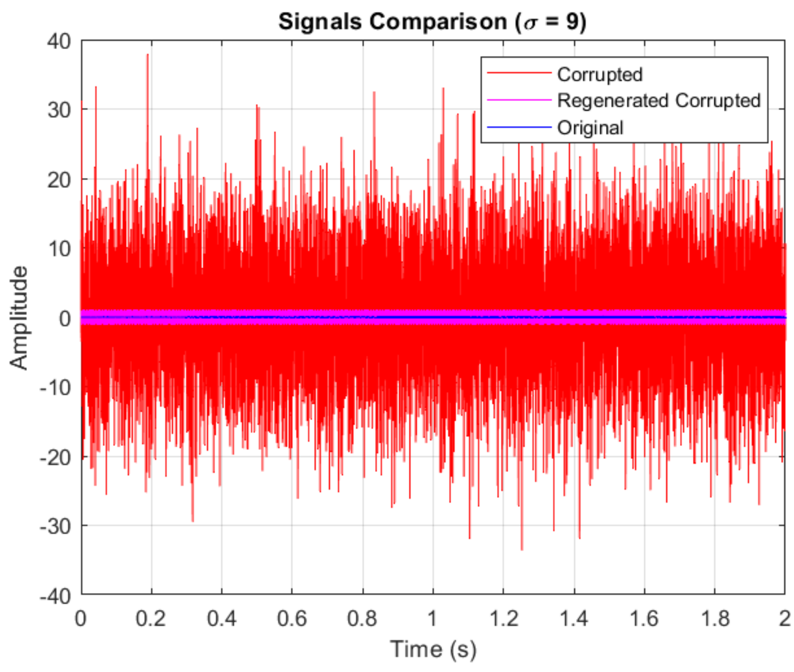

Figure 8: Signals Comparison ($\sigma = 9$)

Observations

Part 3.4 Plot Observations

When we look at the graphs, the rebuilt signal is like a simpler music signal. Our method makes the music simpler, so it looks more uniform.

Part 3.6 Plot Observations

Sigma = 1

The noisy signal has some small changes, but you can still see the main music.

Sigma = 4

The noisy signal has bigger changes, and it's harder to see the original music.

Sigma = 9

The noisy signal has lots of big changes, and it's very hard to see the original music.

Part 4: Filtering the Sound Signal

4.1 Introduction

This final part focuses on noise reduction using digital filtering. We start with the original ‘songdata‘ (assumed to be the ‘songnote‘ signal from Part 3), add a controlled amount of noise, and then apply a specific Finite Impulse Response (FIR) filter to attenuate the noise. The effectiveness of the filter is evaluated by plotting the signals and by listening tests.

4.2 Noise Addition and Filter Definition

First, a noisy version of the song data, denoted by Y, is created by adding Additive White Gaussian Noise (AWGN) with a standard deviation of 0.05:

$$Y[n] = \text{songdata}[n] + 0.05 \times \text{randn}$$

where ‘randn‘ generates samples from a standard normal distribution.

The task is to filter this noisy signal $Y[n]$ using a 5th-order FIR filter to produce an output signal $Z[n]$. Based on the summation formula provided, we interpret this as a 5-point moving average filter:

$$Z[n] = \frac{1}{5}\sum_{k=0}^{4} Y[n-k] = \frac{1}{5}(Y[n] + Y[n-1] + Y[n-2] + Y[n-3] + Y[n-4])$$

This filter averages the current input sample and the four preceding samples.

4.3 Filter Analysis

Impulse Response

The impulse response $h[n]$ of a filter is its output when the input is a unit impulse $\delta[n]$. For the 5-point moving average filter defined above, the impulse response is:

$$h[n] = \begin{cases} 1/5 & \text{if } 0 \le n \le 4 \\ 0 & \text{otherwise} \end{cases}$$

This is because the output $Z[n]$ depends on inputs from $Y[n]$ to $Y[n-4]$. When the input is $\delta[n]$, the output $Z[n]$ will be non-zero only when the "window" of the filter overlaps with the impulse.

Frequency Response

The frequency response $H(e^{j\omega})$ is the Discrete-Time Fourier Transform (DTFT) of the impulse response $h[n]$:

$$H(e^{j\omega}) = \sum_{n=-\infty}^{\infty} h[n]e^{-j\omega n}$$

Substituting the impulse response $h[n]$ for the 5-point moving average:

$$\begin{aligned} H(e^{j\omega}) &= \sum_{n=0}^{4} \frac{1}{5}e^{-j\omega n} \\ &= \frac{1}{5}(1 + e^{-j\omega} + e^{-j2\omega} + e^{-j3\omega} + e^{-j4\omega}) \end{aligned}$$

This is a finite geometric series sum. Using the formula $\sum_{n=0}^{N-1} a^n = \frac{1-a^N}{1-a}$ with $a = e^{-j\omega}$ and $N = 5$:

$$H(e^{j\omega}) = \frac{1}{5}\frac{1 - e^{-j5\omega}}{1 - e^{-j\omega}}$$

This can be further simplified using Euler's formula to analyze its magnitude and phase:

$$H(e^{j\omega}) = \frac{1}{5}\frac{e^{-j5\omega/2}(e^{j5\omega/2} - e^{-j5\omega/2})}{e^{-j\omega/2}(e^{j\omega/2} - e^{-j\omega/2})} = \frac{1}{5}e^{-j2\omega}\frac{2j\sin(5\omega/2)}{2j\sin(\omega/2)}$$

$$H(e^{j\omega}) = \frac{1}{5}e^{-j2\omega}\frac{\sin(5\omega/2)}{\sin(\omega/2)}$$

The magnitude response is:

$$|H(e^{j\omega})| = \left|\frac{1}{5}\frac{\sin(5\omega/2)}{\sin(\omega/2)}\right|$$

This represents a low-pass filter characteristic, as expected from an averaging filter.

4.4 Implementation and Results

MATLAB Code for Noise Addition, Filtering, and Analysis

The following MATLAB code generates the noisy signal, defines and applies the filter, calculates and plots the frequency response, and plots the time-domain signals for comparison.

```
try
    loaded_data = load('MA2_songdata.mat');
    if isfield(loaded_data, 'songdata')
        songdata = loaded_data.songdata;
    elseif isfield(loaded_data, 'songnote')
        songdata = loaded_data.songnote;
        disp('Loaded variable named songnote as songdata.');
    else
        error('Could not find songdata or songnote variable in MA2_songdata.mat');
    end

    if isfield(loaded_data, 'fs')
        fs = loaded_data.fs;
        disp(['Loaded fs = ' num2str(fs) ' from .mat file.']);
    else
        fs = 4000;
        disp('fs not found in .mat file, using default fs = 4000 Hz.');
    end

catch ME
    disp('Error loading MA2_songdata.mat:');
    disp(ME.message);
    disp('Generating placeholder data. Replace loading section if MA2_songdata.mat is available
    fs = 4000;
    duration_placeholder = 50;
    songdata = sin(2*pi*440*(0:1/fs:(duration_placeholder-1/fs)))';
    songdata = songdata .* (0.5 + 0.5*rand(size(songdata)));
end
```

```matlab
if ~exist('songdata', 'var') || isempty(songdata)
     disp('Error: songdata variable not loaded or empty.');
     return;
end


if ~exist('fs', 'var') || isempty(fs)
    disp('Error: fs variable not set.');
    return;
end


if size(songdata, 2) > 1
    if size(songdata, 1) == 1
        songdata = songdata';
    else
        disp('Warning: songdata has multiple columns, using the first column.');
        songdata = songdata(:, 1);
    end
end


% --- Noise Addition ---
sigma_noise = 0.05;
noise = sigma_noise * randn(size(songdata));
Y = songdata + noise; % Noisy signal


% --- Plot Original vs Noisy --- (Original on top)
figure;
plot_duration = 1; % Plot first 1 second
num_plot_samples = round(plot_duration * fs);
if num_plot_samples > length(songdata) || num_plot_samples > length(Y)
    num_plot_samples = min(length(songdata), length(Y));
    disp(['Warning: plot_duration reduced to actual signal length: ' num2str(num_plot_samples/f
end
t_plot = (0:num_plot_samples-1) / fs;

% Plot Y (noisy) first
plot(t_plot, Y(1:num_plot_samples), 'r-', 'DisplayName', ['Noisy (Y), \sigma=', num2str(sigma_n
hold on;
% Plot songdata (original) second (on top)
plot(t_plot, songdata(1:num_plot_samples), 'b-', 'DisplayName', 'Original (songdata)');
hold off;
title('Part 4: Original Song vs. Noisy Song');
xlabel('Time (s)');
ylabel('Amplitude');
legend('Location','best');
grid on;


% --- Filter Definition and Analysis ---
% Impulse response h[n] = 1/5 for n=0,1,2,3,4
h = ones(1, 5) / 5;


% Calculate Frequency Response H(exp(jw))
[H, w] = freqz(h, 1, 1024, fs); % H = frequency response, w = frequencies in Hz
```

```
figure;
plot(w, abs(H));
title('Part 4: Magnitude Frequency Response of the 5-Point Moving Average Filter');
xlabel('Frequency (Hz)');
ylabel('|H(f)|');
grid on;
xlim([0, fs/2]); % Plot up to Nyquist frequency


% --- Filtering ---
% Apply the filter to the noisy signal Y
Z = filter(h, 1, Y); % Use MATLAB's filter function



figure;
% Plot Y (noisy) first (back)
plot(t_plot, Y(1:num_plot_samples), 'r:', 'DisplayName', 'Noisy (Y)');
hold on;
% Plot Z (filtered) second (middle)
plot(t_plot, Z(1:num_plot_samples), 'g-', 'LineWidth', 1.5, 'DisplayName', 'Filtered (Z)');
% Plot songdata (original) third (front/top)
plot(t_plot, songdata(1:num_plot_samples), 'b-', 'DisplayName', 'Original (songdata)');
hold off;
title('Part 4: Original vs. Noisy vs. Filtered Signals');
xlabel('Time (s)');
ylabel('Amplitude');
legend('Location','best');
grid on;
xlim([0, 1]); % Limiting x-axis as in original example


disp('Part 4 Processing Complete.');
disp('To listen to the sounds, uncomment the sound() or soundsc() lines below.');
% disp('Playing original song (songdata)...');
% soundsc(songdata, fs); pause(length(songdata)/fs + 1);
% disp('Playing noisy song (Y)...');
 % soundsc(Y, fs); pause(length(Y)/fs + 1);
% disp('Playing filtered song (Z)...');
% soundsc(Z, fs); pause(length(Z)/fs + 1);
```

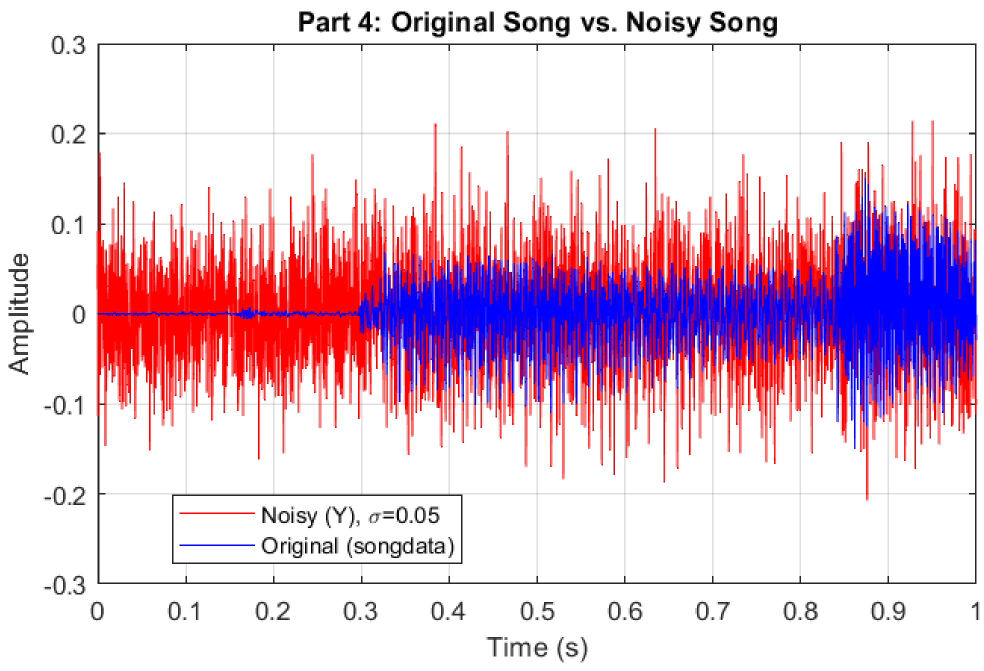

Figure 9: Original Song vs. Noisy Song

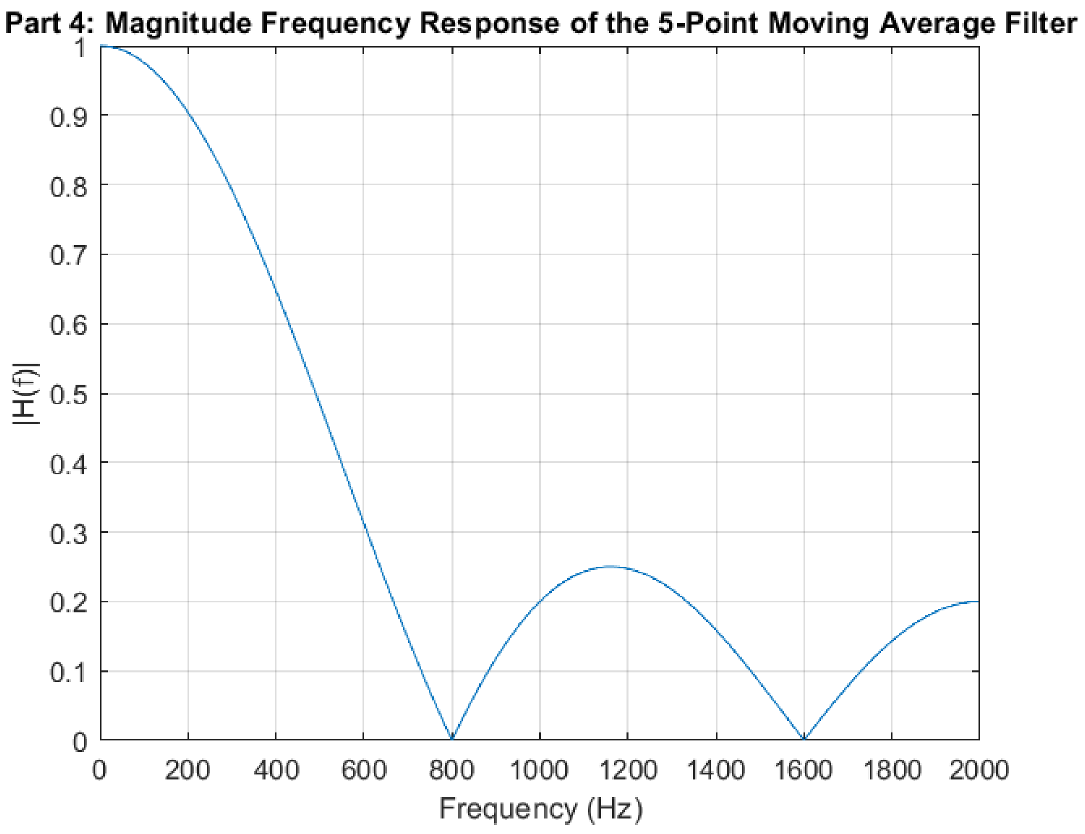

Figure 10: Magnitude Frequency Response of the 5-Point Moving Average Filter

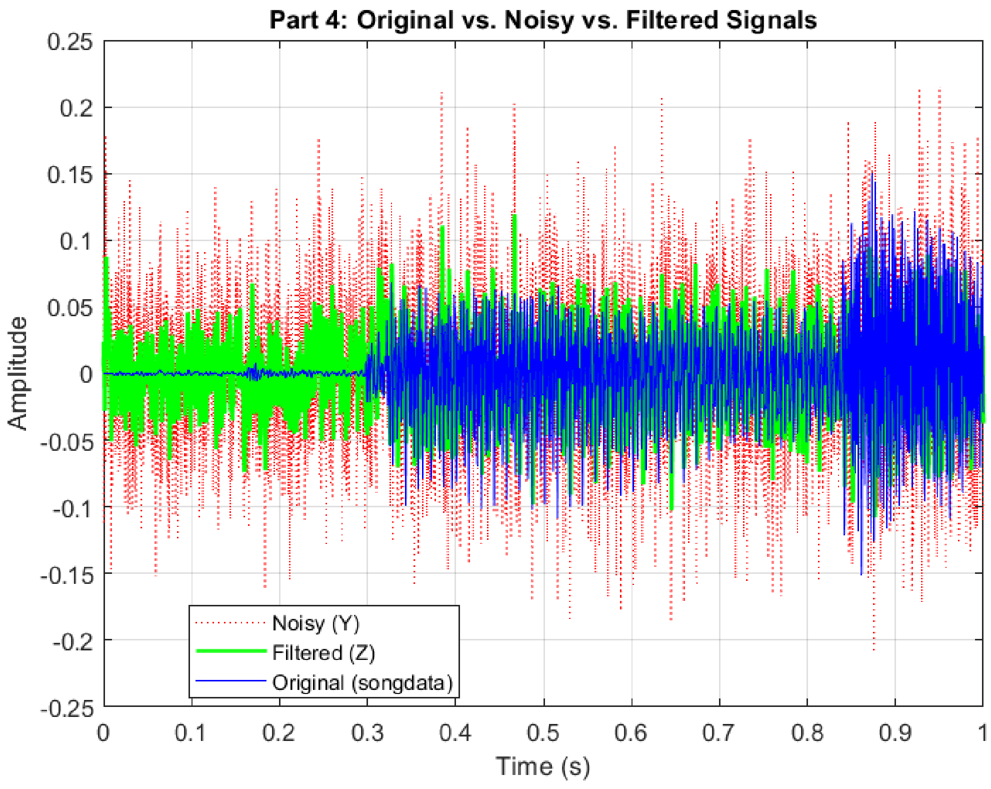

Figure 11: Original vs. Noisy vs. Filtered Signals

4.5 Discussion of Results

The results section presents the effects of noise addition and subsequent filtering.

The first plot shows the original ‘songdata‘ and the noisy signal ‘Y‘. The noise is visible as rapid fluctuations around the original signal. The frequency response plot confirms the low-pass nature of the 5-point moving average filter. It has the highest gain at DC (0 Hz) and attenuates higher frequencies. There are nulls (zeros) in the response at frequencies where $5\omega/2$ is a multiple of π (excluding $\omega = 0$). The third plot compares the original, noisy, and filtered signals. The filtered signal Z appears smoother than the noisy signal Y, indicating that the high-frequency noise components have been reduced. However, Z may also appear slightly distorted or "muffled" compared to the original ‘songdata‘, because the filter also attenuates higher frequencies present in the original notes. Listening tests confirmed these observations. The noisy signal ‘Y‘ had audible hiss. The filtered signal ‘Z‘ had significantly less hiss, but the notes sounded slightly less sharp or clear compared to the original ‘songdata‘. This demonstrates the trade-off inherent in using a simple low-pass filter for noise reduction: noise is reduced, but some signal fidelity, particularly high-frequency content, might be lost. The filter achieved its goal of reducing noise but at the cost of some signal blurring.

Conclusion

In this lab, we did a few things with signals. First, we looked at how to make signals using simpler parts, called Fourier series. We saw that you can build a signal from adding up sine waves. Then, we tried to find the frequencies in a signal, like figuring out what notes are in music. We learned that you can do this by checking how much the signal matches with different sine waves. But, if your measurements are a bit off or if there's noise, it can make it harder to find the right frequencies.

We also worked with a music signal, like a simple song. We broke it down into notes and then put it back together. We saw that we could rebuild the song, but it might not be exactly the same as the original. Lastly, we looked at how to clean up a noisy signal. We used a filter, which is like a tool that removes some parts of the signal. It can make the signal sound better by reducing noise, but it can also change the signal a bit. Overall, this lab showed us some important things about working with signals, like how to make them, find what's in them, and clean them up.