AIN SHAMS UNIVERISTY FACULTY OF ENGINEERING i-CREDIT HOURS ENGINEERING PROGRAMS COMPUTER ENGINEERING AND SOFTWARE SYSTEMS PROGRAM

ECE 251: Signals and Systems Fundamentals

Project Report - Fall 2024

Presented By:

Omar Mohamed Mostafa 22P0197

Ahmed Abbady Mohamed 22P0308

Ahmed Wael Raafat 22P0221

Ezzeldin ismail 22p0141

Anas Mansour 22u0005

Table of Contents

Contribution Table

Omar Mohamed Mostafa 22p0197	Steps (1-5)
Ahmed Abbady Mohamed 22P0308	Steps (6-10)
Ahmed Wael Raafat 22P0221	Step(11-15)
Ezzeldin ismail 22p0141	Steps(16-20)
Anas Mansour 22u0005	Steps(21-25)

Abstract

This project aims to give a practical approach to principles taught in the “ECE 251: Signals and Systems Fundamentals” course in GNU Octave (MATLAB) with a more computation approach allowing for illustration of analyzes of signals in time domain, frequency domain with figures, and filtering of those signals to demonstrate Butterworth filters (low and high) application for frequency separation.

Introduction

This project focuses on the generation, analysis, and filtering of a signal composed of four distinct frequency components (500 Hz, 1000 Hz, 1500 Hz, and 2000 Hz). The signal is sampled at 10 kHz (Fs) and processed using various digital signal processing techniques.

Key objectives include:

1. Signal generation and analysis in both time and frequency domains
2. Implementation of Butterworth filters for frequency separation
3. Energy analysis and verification of Parseval's theorem
4. Practical application through audio file generation and visualization

Project

```
clc;
clearvars;
close all;
```

- This allows compiler to clear Command Window and variables to allow for clean compilation.
- In command window a command of “pkg load signal” is needed on old version of matlab or any version of GNU Octave. (if error is given write “pkg install -forge signals) which downloads necessary libraries for some functions required in this project)
- Headphone warning before listening to any .wav files.

Step 1:

1. (4%) Generate the signal $x(t)$ defined as follows:

$$x(t) = \cos(2\pi f_1 t) + \cos(2\pi f_2 t) + \cos(2\pi f_3 t) + \cos(2\pi f_4 t)$$

where $f_1 = 500$ Hz, $f_2 = 1000$ Hz, $f_3 = 1500$ Hz, and $f_4 = 2000$ Hz.

```
% Step 1: Define the frequencies and create the time vector
freq = [500 1000 1500 2000];    % Frequency components
Fs = 10000;
t = linspace(0, 1, Fs);      % Time vector (1 second duration, 10 kHz sampling)
% Generate the signal
x = cos(2*pi*freq(1)*t) + cos(2*pi*freq(2)*t) + cos(2*pi*freq(3)*t) + cos(2*pi*freq(4)*t);
```

- A freq array is made which stores all the required frequencies indicated in the project requirements which is then each frequency in array is retrieved to each respective cosine functions to then sum up the 4 cosine functions.

- Using a sample rate of 10KHz (Fs), a vector of t is created with 10000 values made from 0 to 1, which is them used in the summation of the 4 cosine functions to create signal x(t) which is stored in a vector (x) which takes sum with each t.

Step 2:

2. (4%) Store the generated signal $x(t)$ as an audio file with extension (*.wav)

```
% Step 2: Normalize the signal and save it to a .wav file
xN = x / max(abs(x));  % Normalize to ensure max value is 1
filename = 'x1(t).wav';
audiowrite(filename, xN, Fs); % Save as .wav file at 10 kHz sampling rate
```

First normalization of signal to ensure maximum amplitude is 1 as .wav files require values between -1 and 1 when saving an audio file, then a file with the name of “x1(t).wav” is made as a reservation for audio which is then wrote using audiowrite taking filename (destination = “x1(t).wav”), xN (normalized signal) and sampling rate as arguments (which is required in audiowrite() as it tells audio player how many samples to play per second, and without it the player wouldn’t know the correct playback speed).

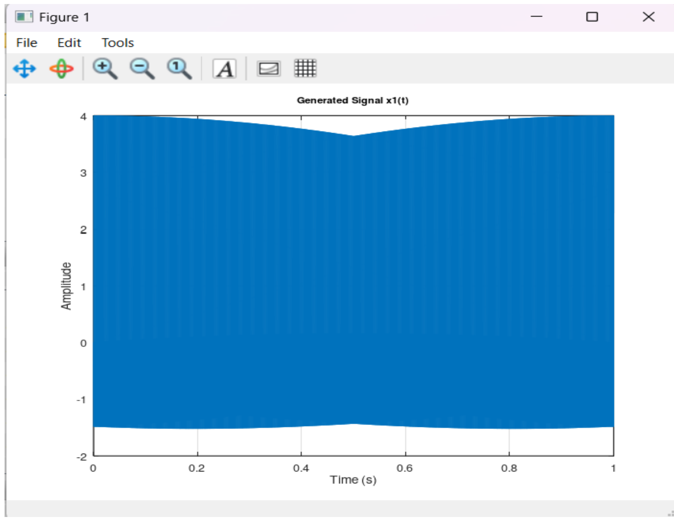
x1(t).wav

Step 3:

3. (4%) Plot the signal $x(t)$ versus time t.

```
% Step 3: Plot the signal in time domain
figure;
plot(t, x);
xlabel('Time (s)');
ylabel('Amplitude');
title('Generated Signal x(t)');
grid on;
```

Figure is used to create a new window which then contains a graph with t being the x-axis and x (summation of the 4 cosine functions) being the y-axis with it being generated using plot() function. Labels on x-axis and y-axis are made using xlabel and ylabel , and title is made to describe the figure all for clarity of figure.



Step 4:

4. (4%) Compute the energy of the signal $x(t)$.

```
% Step 4: Calculate energy in time domain
x_squared = x.^2;          % Square the signal to find |x(t)|^2
energy_time = trapz(t, x_squared);  % Integrate |x(t)|^2 over time
disp(['The energy of x1(t): ', num2str(energy_time)]); % Display time-domain energy
```

Continuous-Time Signal:

$$E_x^{time} = \int_{-\infty}^{\infty} |x(t)|^2 dt$$

- With consideration of this formula, firstly x is squared and stored in x_squared which is a vector taking the square of each sum in the vector x and to square every element a ".^2" is used for code clarity, then trapz calculates the area under the signal curve using trapezoidal method which approximates the area by connecting points with straight lines and summing the areas of resulting trapezoids, the result is stored in a variable (energy_time) that is then displayed in command window after casting it into a string.

Command Window

```
The energy of x1(t): 2
```

Step 5 + 6:

5. (4%) Compute the frequency spectrum $X(f)$ of this signal.

6. (4%) Plot the magnitude of $X(f)$ in the frequency range $-f_s/2 \leqslant f \leqslant f_s/2$, where f_s is the sampling frequency.

```
% Step 5: Compute the frequency spectrum using FFT
N = length(x);            % Number of samples
X_f = fft(x);             % Compute the Fourier Transform of the signal

% step 6: Normalize FFT by dividing by N
X_f_shifted = fftshift(X_f);  % Shift for plotting
X_f_magnitude = abs(X_f_shifted)/N;  % Get the magnitude of the FFT

% Plot the frequency spectrum (magnitude)
figure;% to create a window to popup
f = Fs*(-N/2:N/2-1)/N;  % Create frequency axis for plot
plot(f, X_f_magnitude);  % Use fftshift to center zero frequency in the plot
xlabel('Frequency (Hz)');
ylabel('Magnitude');
title('Frequency Spectrum of x(t)');
xlim([-5000, 5000]);  % Limit frequency axis to relevant range
grid on;
```

- fft() function (Fast Fourier Transform) computes the fourier transform of x converting it from a time domain to frequency domain stored in a vector of the same size (X_f) (size = 10000).
- By default FFT puts zero frequency at start of array, but theoretically there are negative frequencies as well as:

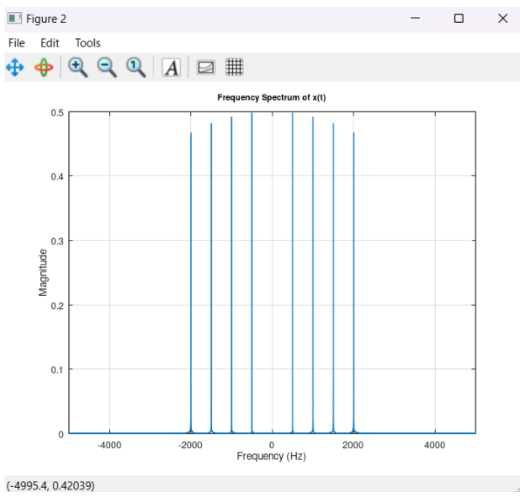
$$\cos x = \frac{e^{ix} + e^{-ix}}{2}$$

- There is a negative counterpart in cosine's euler identity which would have negative frequency, if frequencies start from 0 plot wouldn't be accurate as there would be no representation of negative frequencies, so to solve this ffshift() is used to shift the zero frequency to the center which helps visualize symmetry of frequencies.
- Lastly when plotting frequency spectrum x axis is the frequency and with constraint given in requirement. We need to create frequency axis (f = Fs*(-N/2: N/2-1)/N) , N being the length of x which is used for frequency resolution ($\Delta f = Fs/N$) as the spacing between frequency points is determined by number of samples (N), so dividing by N ensures correct scaling of frequency axis. N is also used in computation of X[k] (X_f_magnitude) as in MATLAB theoretically even if signal is CT (continuous time), as results are all stored and plotted in points therefore it is turned into a discrete signal. So DFT is performed with fft, dividing by N ensures amplitude of signal in

frequency domain is consistent with the time domain signal, so that magnitude is accurate with Parseval's theorem:

$$\sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2,$$

- Then a plot is made similar to step 3 but this time f being the x-axis and magnitude of x_f (X[K]) as y-axis and f as x axis with xlim limiting the x axis values represented in plot from -Fs/2 to Fs/2



Step 7:

7. (4%) Compute the Energy of the signal $x(t)$ from its frequency spectrum $X(f)$, and hence you can verify Parseval's theorem.

```
% Step 7: Compute energy from the frequency spectrum (Parseval's theorem)
energy_frequency = sum(abs(X_f).^2)/(N^2);  % Sum of squared FFT magnitudes
disp(['The energy of the signal from the frequency domain is: ', num2str(energy_frequency)]);
disp(['Parseval verfication: ', num2str(abs(energy_frequency - energy_time ))])
% the result should be 0, but its 0.0014 due to diff metholds.
```

To compute the energy of frequency domain signal, Parseval's theorem is used as mentioned in step 5+6 and dividing by N^2 is necessary as without it the frequency domain energy would be scaled incorrectly relative to time domain energy then for debugging the value is displayed in command window, then to verify Parseval's theorem the difference between energy in both domains should be very close to 0 with the difference being in numerical method.

```
The energy of x1(t): 2
The energy of the signal from the frequency domain is: 2.0014
Parseval verfication: 0.0014
```

As value is close to 0 we can verify that Parseval theorem is correct, and frequency and time domain signals are consistent with each other with correct scaling.

Step 8:

8. (4%) Design a Butterworth low-pass filter with filter order 20 and cut-off frequency of 1.25 kHz.

```
% Step 8: Design and apply a Butterworth low-pass filter
filter_order = 20;                          % Filter order
cutoff_frequency = 1250;                    % Cutoff frequency in Hz
Wn = cutoff_frequency / (Fs/2);             % Normalize cutoff frequency (0 to 1 range)
[b, a] = butter(filter_order, Wn, 'low'); % Design Butterworth low-pass filter
```

The butter function takes filter order, (Wn) cutoff frequency (which has to be between 0 and 1) , and filter type (‘low’, ‘high’, ‘bandpass’, or ‘bandstop’).

Constraints defined in requirements were put in separate variables (filter_order), (cutoff_frequency) then in Wn it Is Normalized to scale between 0 and 1 and in our case Wn

$$H(z) = \frac{B(z)}{A(z)}$$

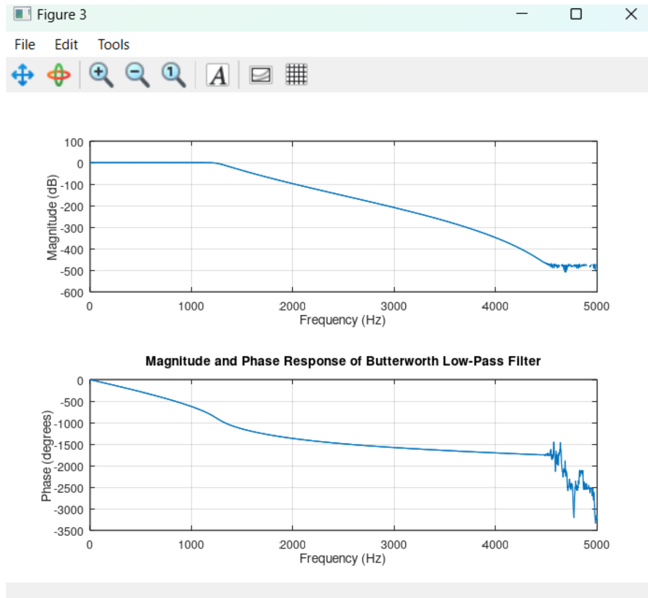
is 0.25. A low-pass Butterworth filter is used to allow signal with frequencies below 1250 to pass through while reducing amplitude of frequencies above 1250 which removes them from the signal. b stores the numerator coefficients of the filter transfer function, while a stores denominator coefficients of the filter transfer function.

Step 9:

9. (4%) Plot the magnitude and phase response of the Butterworth LPF you've designed.

```
% step 9 Plot the magnitude and phase response of the Butterworth LPF
figure;% to create a window to popup
freqz(b, a, 2048, Fs);
% 2048  give a very detailed plot (high resolution) but take slightly longer to compute.
title('Magnitude and Phase Response of Butterworth Low-Pass Filter');
```

As mentioned before, a figure is made to create a window and a title is used for clarity, this time instead of plot we use freq() which is used to compute frequency response of filter which takes usually 3 arguments but in this case we use 4 arguments which is b and a both from butter function, 2048 which is the number of points to use in sampling the frequency domain (as 2^n number increases , a more detailed frequency response is produced but as compute time cost), and lastly Fs which is the sampling frequency which is optional (syntax wise).



Step 10 + 11:

10. (4%) Apply the signal $x(t)$ to this Butterworth LPF and let's denote the output signal as $y_1(t)$.
11. (4%) Store the generated signal $y_1(t)$ as an audio file with extension (*.wav)

```
% Step 10: Apply the signal x(t) to the Butterworth LPF
y1 = filter(b, a, x);  % Filter the signal using the designed LPF

% STEP 11 :Store the filtered signal y1(t) as a .wav audio file
output_filename = 'y1(t).wav';
audiowrite(output_filename, y1 / max(abs(y1)), Fs); % Normalize and save as .wav
```

filter() function takes 3 arguments (b as defined in step 8) , (a as defined in step 8), and lastly x which is our signal in time domain, this filtered signal is stored is y1 which is a vector.

Then similar to step 2 an audio file is generated and stored in y1(t).wav

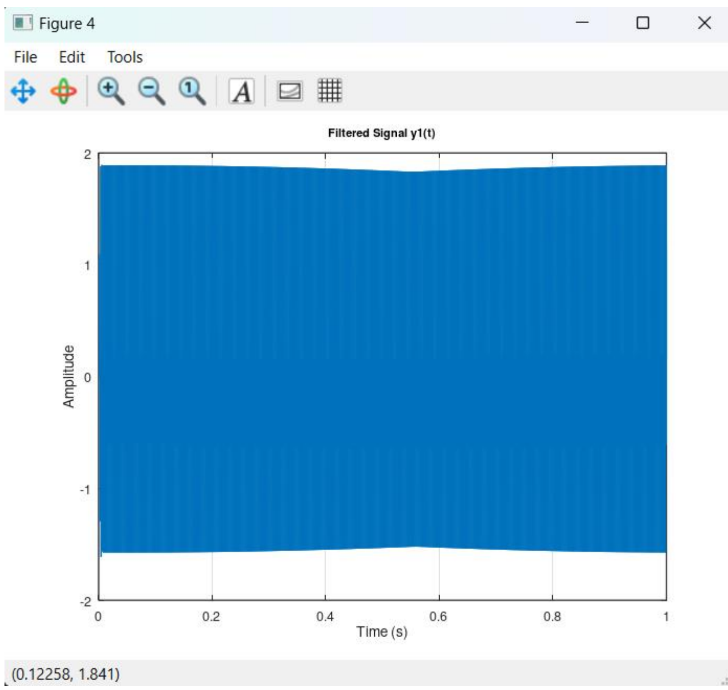
y1(t).wav

Step 12:

12. (4%) Plot the signal $y_1(t)$ versus time t.

```
%step 12 :Plot the filtered signal y1(t) in the time domain
figure;% to create a window to popup
plot(t, y1);
xlabel('Time (s)');
ylabel('Amplitude');
title('Filtered Signal y1(t)');
grid on;
```

As previously mentioned, here we used plot with t being the x-axis and y1 (filtered signal) as y-axis.



Step 13:

13. (4%) Compute the energy of the signal $y_1(t)$.

```
% Step 13: Compute the energy of the filtered signal y1(t)
y1_squared = y1.^2;                     % Square the filtered signal to find |y1(t)|^2
energy_y1 = trapz(t, y1_squared);       % Integrate |y1(t)|^2 over the time
disp(['The energy of the filtered signal y1(t) is: ', num2str(energy_y1)]); % Display the energy
```

Similar to step 4, energy of y1(t) was computed.

```
The energy of the filtered signal y1(t) is: 0.99836
```

Step 14 + 15:

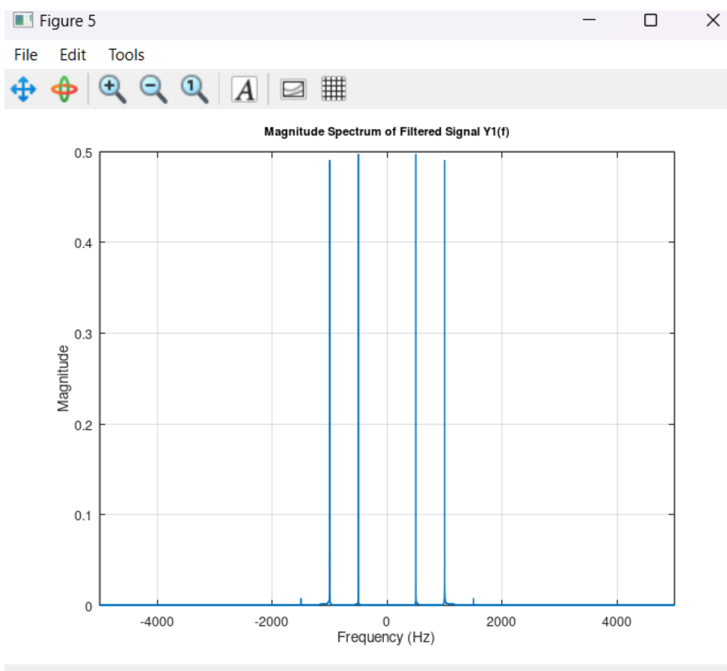
14. (4%) Compute the frequency spectrum $Y_1(f)$ of this signal.

15. (4%) Plot the magnitude of $Y_1(f)$ in the frequency range $-f_s/2 \leqslant f \leqslant f_s/2$.

```
% Step 14: Compute the frequency spectrum Y1 of the filtered signal
Y1_f = fft(y1);                      % Compute the Fourier Transform of the filtered signal
Y1_f_shifted = fftshift(Y1_f);       % Shift for plotting
Y1_f_magnitude = abs(Y1_f_shifted)/N; % Normalize the FFT magnitude

% Step 15: Plot the magnitude of Y1(f) in the frequency range -Fs/2 <= f <= Fs/2
figure;
plot(f, Y1_f_magnitude);             % Plot the magnitude spectrum
xlabel('Frequency (Hz)');
ylabel('Magnitude');
title('Magnitude Spectrum of Filtered Signal Y1(f)');
xlim([-5000, 5000]);                 % Restrict to the range -Fs/2 to Fs/2
grid on;
```

Similar to step 5 + 6 , Fourier transform of y1(t) was computed (Y1_f)and after correct scaling it was used to plot magnitude of Y1_f_magnitude after it was computed with it being the y axis and f which was previously declared in step 5 + 6 as constraints are the same, being the x-axis.



Step 16:

16. (4%) Compute the Energy of the signal $y_1(t)$ from its frequency spectrum $Y_1(f)$, and hence you can verify Parseval's theorem.

```
% Step 16: Compute the energy of y1(t) from its frequency spectrum and verify Parseval's theorem
energy_y1_frequency = sum(abs(Y1_f).^2)/(N^2); % Sum of squared FFT magnitudes
disp(['The energy of y1(t) from its frequency spectrum is: ', num2str(energy_y1_frequency)]);

% Verify Parseval's theorem
parseval_difference = abs(energy_y1 - energy_y1_frequency);
disp(['Parseval verification : ', num2str(parseval_difference)]);
% the result should be 0, but its 9.8143×10^-5 which is 0.000098143 due to diff metholds.

```

Similar to step 7 , energy in frequency domain is done using Parseval's theorem and then after correct scaling the difference between energy in time and frequency domain is computed to verify Parseval's theorem.

```
The energy of the filtered signal y1(t) is: 0.99836
The energy of y1(t) from its frequency spectrum is: 0.99826
Parseval verification : 9.8143e-05
```

9.8143e-05 is equivalent to 0.000098143 which means Parseval's theorem is valid and the difference is due to using different numerical methods.

Step 17 + 18:

17. (4%) Design a Butterworth high-pass filter with filter order 20 and cut-off frequency of 1.25 kHz.

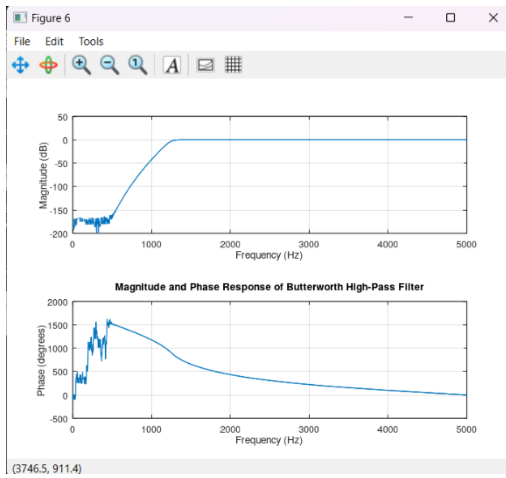
18. (4%) Plot the magnitude and phase response of the Butterworth HPF you've designed.

```
% Step 17: Design a Butterworth high-pass filter
filter_order_hp = 20;                    % Filter order
cutoff_frequency_hp = 1250;              % Cutoff frequency in Hz
Wn_hp = cutoff_frequency_hp / (Fs/2);    % Normalize cutoff frequency (0 to 1 range)
[b_hp, a_hp] = butter(filter_order_hp, Wn_hp, 'high'); % Design Butterworth high-pass filter

% step 18 Plot the magnitude and phase response of the high-pass filter
figure;
freqz(b_hp, a_hp, 2048, Fs);% 2048  give a very detailed plot (high resolution) but take slightly longer to compute.
title('Magnitude and Phase Response of Butterworth High-Pass Filter');

```

Exact same constraints and steps as steps 8 + 9 but difference being that in butter function the filter type is high which would allow passing for frequencies above 1250 and reduces amplitude of frequencies below 1250 (as frequency decrease reduction increase).



Step 19 + 20:

19. (4%) Apply the signal $x(t)$ to this Butterworth HPF and let's denote the output signal as $y_2(t)$.

20. (4%) Store the generated signal $y_2(t)$ as an audio file with extension (*.wav)

```
% Step 19: Apply the signal x(t) to the Butterworth HPF
y2 = filter(b_hp, a_hp, x);  % Filter the signal using the high-pass filter

% Step 20: Store the filtered signal y2(t) as a .wav audio file
output_filename_y2 = 'y2(t).wav';
audiowrite(output_filename_y2, y2 / max(abs(y2)), Fs); % Normalize and save as .wav
```

Same steps as 10 + 11 where filtered signal of x but this time filtered with nominator and denominator of Butterworth high pass filter stored in y2 then audio file like in step 2 is generated while doing the normalization inside the audiowrite function.

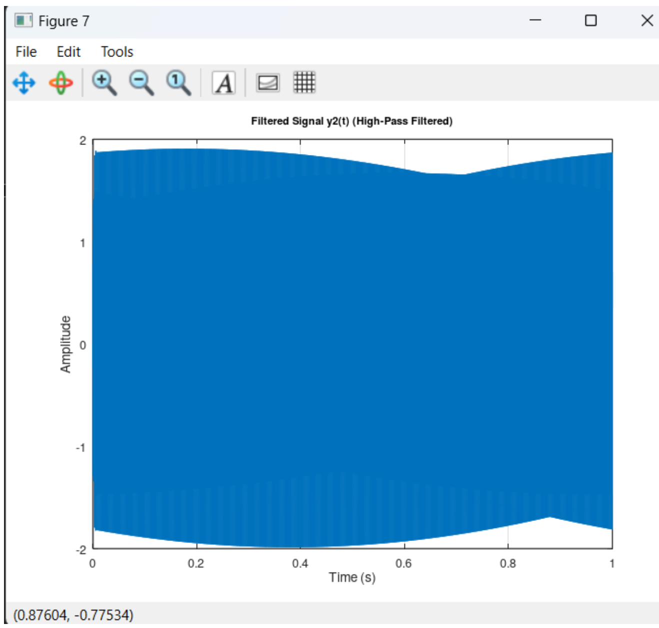
y2(t).wav

Step 21:

21. (4%) Plot the signal $y_2(t)$ versus time t.

```
% Step 21: Plot the filtered signal y2(t) in the time domain
figure;
plot(t, y2);
xlabel('Time (s)');
ylabel('Amplitude');
title('Filtered Signal y2(t) (High-Pass Filtered)');
grid on;
```

Similar to step 3 a graph is made with y2 being the y-axis and t being the x-axis.



Step 22:

22. (4%) Compute the energy of the signal $y_2(t)$.

```
% Step 22: Compute the energy of the filtered signal y2(t)
y2_squared = y2.^2;                    % Square the filtered signal to find |y2(t)|^2
energy_y2 = trapz(t, y2_squared);     % Integrate |y2(t)|^2 over the time vector
disp(['The energy of the filtered signal y2(t) is: ', num2str(energy_y2)]); % Display the energy
```

Similar to step 4, energy of y2(t) is computed.

```
The energy of the filtered signal y2(t) is: 0.99946
```

Step 23+24:

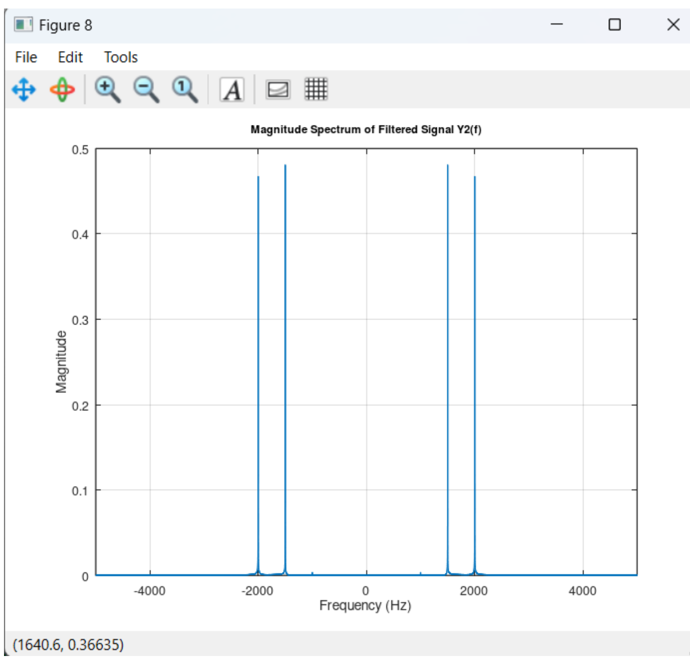
23. (4%) Compute the frequency spectrum $Y_2(f)$ of this signal.

24. (4%) Plot the magnitude of $Y_2(f)$ in the frequency range $-f_s/2 \leqslant f \leqslant f_s/2$.

```
% Step 23: Compute the frequency spectrum Y2(f) of the filtered signal
Y2_f = fft(y2);                         % Compute the Fourier Transform of the filtered signal
Y2_f_shifted = fftshift(Y2_f);          % Shift the spectrum for plotting
Y2_f_magnitude = abs(Y2_f_shifted)/N; % Normalize the FFT magnitude

% Step 24: Plot the magnitude of Y2(f) in the frequency range -Fs/2 <= f <= Fs/2
figure;
plot(f, Y2_f_magnitude);                % Plot the magnitude spectrum
xlabel('Frequency (Hz)');
ylabel('Magnitude');
title('Magnitude Spectrum of Filtered Signal Y2(f)');
xlim([-5000, 5000]);                    % Restrict to the range -Fs/2 to Fs/2
grid on;
```

Similar to step 14+ 15, with y2_f_magnitude being the y axis and x-axis being the f from step 5+ 6 , as there are the same constraints.



Step 25:

25. (4%) Compute the Energy of the signal $y_2(t)$ from its frequency spectrum $Y_2(f)$, and hence you can verify Parseval's theorem.

```
% Step 25: Compute the energy of y2(t) from its frequency spectrum and verify Parseval's theorem
energy_y2_frequency = sum(abs(Y2_f).^2)/(N^2); % Sum of squared FFT magnitudes
disp(['The energy of y2(t) from its frequency spectrum is: ', num2str(energy_y2_frequency)]);

% Verify Parseval's theorem
parseval_difference_y2 = abs(energy_y2 - energy_y2_frequency);
disp(['Parseval verification for y2(t): ', num2str(parseval_difference_y2)]);
```

Lastly, Similar to step 7 , energy in frequency domain is done using Parseval’s theorem and then after correct scaling the difference between energy in time and frequency domain is computed to verify Parseval's theorem.

```
The energy of the filtered signal y2(t) is: 0.99946
The energy of y2(t) from its frequency spectrum is: 0.99938
Parseval verification for y2(t): 7.5314e-05
```

7.5314e-05 is equivalent to 0.000075314 which means Parseval’s theorem is valid and the difference is due to using different numerical methods.

Conclusion:

The project demonstrated the implementation of principles taught in the course through practical application. The signal was effectively separated into low and high-frequency components using 20th-order Butterworth filters with 1250 Hz cutoff frequency. Energy calculations in both time and frequency domains showed minimal difference, validating Parseval's theorem with differences on the order of 10^-4 or less. The generated audio files and spectral analyses provide verification of the filtering effectiveness, while the figures offer insights into the signal's characteristics at each processing stage. This implementation serves as a practical benefit for understanding and applying filtering techniques in real-world applications.