EEE3030 Signal Processing and Machine Learning

Semester 1 Report

Sahas Talasila *230057896*

Contents

Abstract

This report presents the findings and methodologies employed in the EEE3030 Signal Processing and Machine Learning course. It encompasses a comprehensive analysis of signal processing techniques, machine learning algorithms, and their applications in various domains. The report details the experimental setups, data analysis, and results obtained from implementing different models. Key insights and conclusions drawn from the study are also discussed, highlighting the effectiveness of the approaches used.

Introduction

Procedure, Results and Discussion

2.1 Task 1 Audio File Analysis and Transformation

This task aims to analyse the audio file (Sahas_Talasila.wav), as provided in the assignment folder. Time and frequency domain analysis of the audio file.

2.1.1 Task 1 - Audio File Information

Some basic information about the audio file needs to be extracted first. When we load a `.wav` file in MATLAB, we obtain two key pieces of information:

- The discrete-time signal samples: $x[n]$
- The sampling frequency: f_s (samples per second)

From these, we can determine several important properties that guide our signal analysis.

If N is the total number of samples, then the duration of the signal is shown by *Equation 1*, which is needed to understand the time span of the audio file:

$$T_{\text{duration}} = \frac{N}{f_s} \tag{1}$$

The frequency resolution (also called the bin width) determines the smallest distinguishable frequency difference in an FFT, which will be explained later on, but for now, we can calculate the frequency resolution using *Equation 2*.

The frequency resolution is needed because it tells us how finely we can analyse the frequency content of the signal. A higher resolution allows us to distinguish between closely spaced frequency components. A smaller resolution means that we can only see broader frequency bands, which may obscure important details in the signal's frequency contents. The equation for frequency resolution is given by *Equation 2*:

$$\Delta f = \frac{f_s}{N} \tag{2}$$

For example, with $f_s = 96\,\text{kHz}$ and $N = 96000$ and using *Equation 2*, the user would obtain:

$$\Delta f = 1\,\text{Hz}$$

The Nyquist theorem states that the highest frequency that can correctly represent is half the sampling frequency, using *Equation 3*:

$$f_{\text{Nyquist}} = \frac{f_s}{2} \tag{3}$$

For $f_s = 96\,\text{kHz}$:

$$f_{\text{Nyquist}} = 48\,\text{kHz}$$

This is sufficient for AM signals (which is the exact type of signal this report focuses on) with carriers in the tens of kHz and bandwidths of a few kHz.

Other pieces of information can be seen from the code below, for example, a visual plot of the signal in the time domain.

Code and Explanation

```
filename = 'Sahas_Talasila.wav';
[x, fs] = audioread(filename);

% If stereo, convert to mono by taking first channel
if size(x, 2) > 1
    x = x(:, 1);
    fprintf('Note: Stereo file detected, using first channel only.\n\n');
end

% audioread() returns a column vector, but we need row vectors throughout
if iscolumn(x)
    x = x';  % Transpose to row vector
end

%% Calculate basic signal properties
N = length(x);                      % Total number of samples
duration = N / fs;                  % Signal duration in seconds
freq_resolution = fs / N;           % Frequency resolution (bin width) in Hz
nyquist_freq = fs / 2;              % Maximum representable frequency

%% Calculate amplitude statistics
max_amplitude = max(x);
min_amplitude = min(x);
peak_to_peak = max_amplitude - min_amplitude;
rms_amplitude = sqrt(mean(x.^2));
```

```
26
27 %% Sub-task 1.2: Time Domain Analysis
28
29 % Create time vector
30 t = (0:N-1) / fs;
31
32 % Plotting Logic, removed for readability
```

Listing 1: MATLAB Code for Task 1 Part 1: Time Domain Analysis

Code Output and Explanation

The code above implements the ideas mentioned in the **Procedure** paragraph, by reading in the audio file and then converting it into a vector. Stereo channels have been accounted for as well, which could then affect our future outputs.

The following properties and plots (mentioned earlier) have been calculated:

- Sample count (needed for frequency domain analysis)
- Signal time duration (needed for future analysis)
- Frequency resolution (Used to distinguish between two closely spaced frequency components in a signal)
- Nyquist frequency
- Max and Min amplitudes of the signal
- Peak to peak amplitude
- RMS values
- Time domain output

Results and Discussion
Firstly, **Figure X** shows the terminal output, which can be used to verify the theory and procedure above.

FIGURE SHOWING THE INITIAL TERMINAL OUTPUT AND INFO STATED BELOW

Sampling frequency confirmation: With $f_s = 96000Hz$ this confirms the signal was recorded correctly and our subsequent analysis will use the correct frequency scaling.

Signal duration: Our output is 2.54 seconds.

Frequency resolution: The resolution is $0.39Hz$, which is a very fine resolution. For identifying an AM signal band with edges that could be at any frequency, a resolution of $1Hz$ or less is ideal. Our large value of N (244104 samples), ensures a higher quality output.

Amplitude statistics: The RMS amplitude gives us a baseline understanding of the signal power. The peak-to-peak value indicates the dynamic range, but are not needed directly for the future tasks. It provides a standardised method for measuring performance, giving us a baseline output.

Figure X also shows the signal in the time domain:

FIGURE SHOWING TIME DOMAIN PLOT

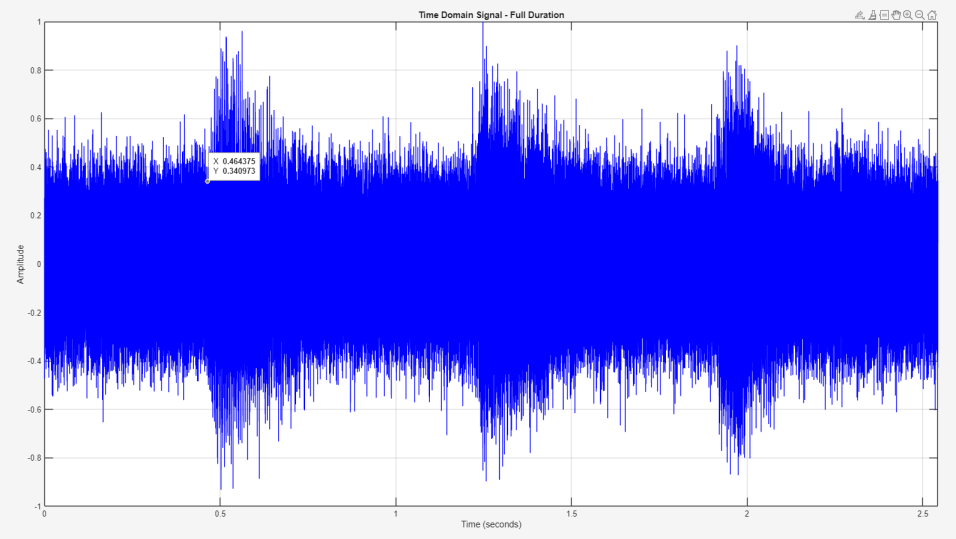

Figure 1: Initial Signal in Time Domain

From the above plot, it is easy to identify 3 significant peaks, at roughly 0.5, 1.2 and 1.9 seconds. A visual inspection (and common sense), can reveal the locations of the three characters. A lot of white noise is present in the signal as well, which is shown by the consistent and uniform oscillations. This highlights issues with this form of analysis as well. A reader cannot simply distinguish the carrier, minimum and maximum frequencies, which are needed if the message is to be decoded.

The next section will focus on the FFT method for a clearer view and understanding of the signal.

2.1.2 Task 1 - Frequency Domain Analysis

Procedure and Theory

The Discrete Fourier Transform (DFT) decomposes a discrete time-domain signal into its frequency components, allowing the easy identification of the carrier frequency f_c and determine the AM signal band. For a signal $x[n]$ of length N, the DFT is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-j\frac{2\pi kn}{N}}, \tag{4}$$

where $X[k]$ is the complex spectrum at bin k, and the exponential term represents a complex sinusoid at that frequency. Large values of $X[k]$ indicate strong frequency content in the signal (*Equation X*).

The Fast Fourier Transform (FFT) computes the same result as the DFT but reduces the complexity from N^2 to $N \log_2 N$, as used in MATLAB's `fft()`.

Each frequency bin maps to a physical frequency:

$$f[k] = \frac{k f_s}{N}, \tag{5}$$

so $k = 0$ corresponds to DC, $k = N/2$ to the Nyquist frequency, and bins above $N/2$ represent negative frequencies.

The FFT magnitude scales with N, so amplitude normalisation is performed using:

$$|X[k]|_{\text{norm}} = \frac{|X[k]|}{N}. \tag{6}$$

For real signals, the spectrum is symmetric, so a single-sided spectrum is obtained by keeping bins 0 to $N/2$ and doubling all non-DC and non-Nyquist bins:

$$|X[k]|_{\text{ss}} = \begin{cases} \dfrac{|X[k]|}{N}, & k = 0 \text{ or } k = \frac{N}{2}, \\ \dfrac{2|X[k]|}{N}, & \text{otherwise.} \end{cases}$$

To visualise components spanning different magnitudes, the spectrum is converted to decibels:

$$X_{\text{dB}}[k] = 20 \log_{10}(|X[k]|_{\text{norm}}),$$

where a tenfold increase in amplitude corresponds to +20 dB. This scaling highlights both the carrier and sidebands as well as lower-level noise components.

Code and Explanations

To supplement the steps above, *Listing X* shows the code used to implement it.

```
%% Sub-task 1.2: Frequency Domain Analysis - FFT Implementation

% Compute the FFT of the signal
X = fft(x);

% The FFT output is complex - compute the magnitude
X_magnitude = abs(X);

% Normalise by dividing by N to get correct amplitude scaling
X_normalised = X_magnitude / N;

% Create single-sided spectrum (positive frequencies only)
% We need bins from 0 (DC) to N/2 (Nyquist)
num_bins_single_sided = floor(N/2) + 1;
X_single_sided = X_normalised(1:num_bins_single_sided);

% Double the amplitude for all bins except DC and Nyquist
% This accounts for the energy in the negative frequencies we
   discarded
X_single_sided(2:end-1) = 2 * X_single_sided(2:end-1);

% Create the frequency vector for the single-sided spectrum
```

```
% Each bin k corresponds to frequency f = k * fs / N
f = (0:num_bins_single_sided-1) * fs / N;

% Convert to decibels for logarithmic scaling
% We add eps (smallest positive number) to avoid log(0) = -
   infinity
X_dB = 20 * log10(X_single_sided + eps);

% Create figure for the frequency spectrum (removed for
   readability)

% Plot with logarithmic (dB) scaling (removed for readability)

% Display frequency domain statistics (removed for readability)
```

Listing 2: MATLAB Code for Task 1 Part 2: Frequency Domain Analysis

Using the `fft()` function, output X is a complex vector of the same length as x (N elements). Each element X[k] contains both magnitude and phase information about the frequency component at bin k. This will relate to the frequency resolution later on.

The magnitude has been calculated and then normalised for ease of understanding and stops spectral peaks from appearing.

The lines X - Y show that from DC (bin 1 in MATLAB's 1-indexing) to Nyquist (bin N/2+1). The next line extracts these bins. The third line doubles all amplitudes except DC and Nyquist - this discards the negative frequency half of the spectrum, losing half of the signal energy. For real signals, the negative frequencies are mirror images of the positive frequencies, so doubling recovers the correct total amplitude. DC and Nyquist are not doubled because they don't have mirror images.

Finally for the code, plots can be viewed showing the signal in the frequency domain. **Figure X** shows the signal in the frequency domain without dB scaling, and **Figure Y** implements dB scaling

Results and Discussion
First, the plots will be discussed, and then the terminal output.

FIGURE SHOWING FREQUENCY DOMAIN SIGNAL NO SCALING (older image used instead here)

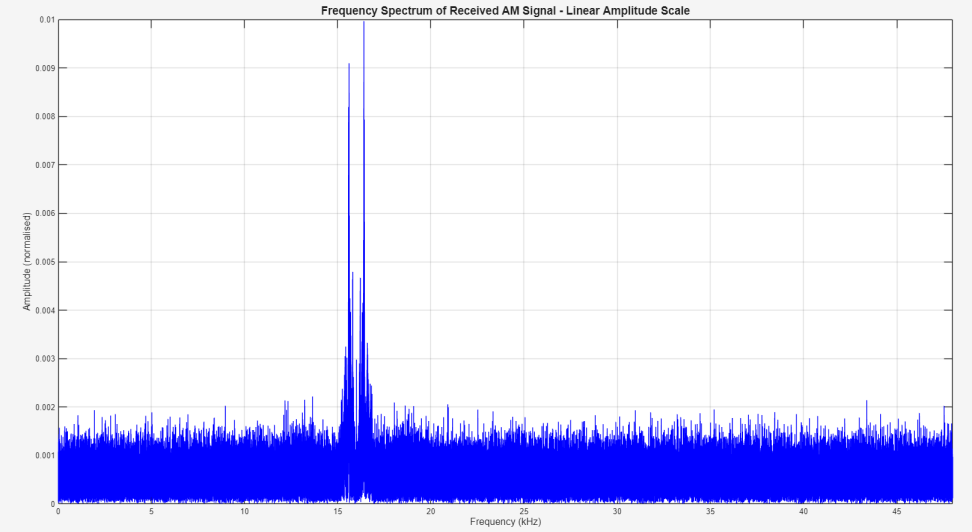

Figure 2: Stripboard Plan

In the linear amplitude plot above (**FIgure X**), the strong signal components dominate the display (can approximate $f_{min} = 15.6kHz$, $f_{max} = 16.4kHz$ and $f_c = 16kHz$), and the noise floor is likely compressed to a thin line near zero. Initially, it might seem that all of our parameters have been identified, but due to the nature of the signal, the bandwidth has to be extended to $f_c \pm B$, where $B = 4kHz$. This makes it difficult to assess the noise characteristics or identify the exact boundaries of the signal band. This might seem very useful at first, but it is important that both the signal and noise components can be represented so that precise filter design is possible.

FIGURE SHOWING FREQUENCY DOMAIN WITH dB SCALING

Figure X above now shows the full image, showing the noise and the stronger signal components. In the dB-scaled plot, both the signal and noise are visible because the logarithmic scaling compresses the dynamic range. A signal that is 1000X stronger than the noise (60 dB difference) can be displayed on the same plot with both components clearly visible. Also, due to the nature of the signal (letters), a slight alteration is needed for the visual output, which can be seen below, in **Figure X**:

FIGURE SHOWING THE BANDWIDTH AS WELL

From the plot above, it is easy to see from the green highlighted area that we need a larger range. This is because some characters can be obscured or altered if the bandwidth is too low.

The terminal output can be used for easier verification of the above. So now it is easier to see the new values that will be used going forward; with $f_{min} = 12kHz$, $f_{max} = 20kHz$ and $f_c = 16kHz$, now accounting for the wider bandwidth needed for various letters.

FIGURE SHOWING THE TERMINAL OUTPUT WITH BANDWIDTH

For future tasks including filter design, the noise floor needs to be analysed as well. The shape of the noise (from **Figure X**) can show that a uniform, white noise is the problem. From **Figure Below**, the floor is at $-65dB$, a useful baseline for filter design, and there is a very small amount of elevation, characteristic of spectral leakage and shows the need

for windowing, which will be explained next.

IMAGE WITH NOISE FLOOR

2.1.3 Task 1 - Spectral Leakage and Windowing

Procedure and Theory

Spectral leakage occurs because we analyse only a finite-length segment of a signal. This is equivalent to multiplying the infinite-duration signal by a rectangular window *Equation X*:

$$x_{\text{windowed}}[n] = x[n]\, w_{\text{rect}}[n], \qquad w_{\text{rect}}[n] = \begin{cases} 1, & 0 \le n \le N-1, \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

Multiplication in the time domain corresponds to convolution in the frequency domain (*Equation X*):

$$X_{\text{windowed}}(f) = X(f) * W_{\text{rect}}(f), \tag{8}$$

where $W_{\text{rect}}(f)$ is the Fourier transform of the rectangular window.

The spectrum of the rectangular window is a sinc function with a narrow main lobe and high side lobes. The characteristics are:

$$\text{Main-lobe width} \approx \frac{0.9}{N}, \qquad \text{1st side lobe} \approx -13 \text{ dB}, \qquad \text{Roll-off} \approx 6 \text{ dB per octave.}$$

Convolution with this sinc causes energy from each frequency to spread into neighbouring bins, resulting in: broadening of spectral peaks, masking of weak components by side lobes, inaccurate amplitude estimates, and an elevated noise floor.

Leakage is worst when a sinusoid does not lie exactly on a DFT bin centre; only bin-centred sinusoids align with the sinc nulls and avoid leakage. Because the AM carrier frequency is arbitrary relative to the bin spacing, leakage is generally unavoidable.

To reduce leakage, alternative window functions with lower side lobes can be applied. This introduces a trade-off: windows with lower side lobes reduce leakage but have wider main lobes, decreasing frequency resolution.

For this application, we must balance frequency resolution with leakage suppression. Since the filter design in Task 2 requires more than $50dB$ stopband attenuation, it is appropriate to use a window with similar characteristics. The Hamming window provides approximately $53dB$ attenuation, a reasonable main-lobe width ($3.3/N$), and strong leakage reduction.

The Hamming window is defined as:

$$w_{\text{Hamming}}[n] = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N-1}\right), \qquad n = 0, 1, \ldots, N-1. \tag{9}$$

Applying the window simply multiplies the signal sample-by-sample:

$$x_{\text{windowed}}[n] = x[n]\, w[n]. \tag{10}$$

Windowing reduces the signal energy, so amplitude correction is required. The coherent gain of a window is shown:

$$CG = \frac{1}{N} \sum_{n=0}^{N-1} w[n], \tag{11}$$

which for the Hamming window is approximately 0.54. To restore correct spectral amplitudes, we divide by this gain:

$$|X[k]|_{\text{corrected}} = \frac{|X[k]|}{N \cdot CG}.$$

After windowing, side lobes around strong components are greatly reduced, the noise floor becomes cleaner, and the AM band edges (f_{min}, f_{max}) are more clearly visible. The main-lobe width increases slightly, but for large N this effect is negligible in absolute frequency.

Note: Due to the length of the code listing, and for sake of brevity, the implementation code will not be shown here, but is available from the provided **github link** and from the **appendix** as well. Extensive experimentation with different window types is available here as well.

Results and Discussion

WINDOWED V UNWINDOWED COMP

In the **Procedure and Theory** part, the Hamming window was discussed as the ideal window due to the specifications discussed in the rubric (needs to be as close to the $50dB$ requirement as possible).

The Hamming window has a coherent gain (showing how much the amplitude has decreased by) of approximately 0.54, meaning we lose about 46% of the signal energy due to the tapering. Without correction, spectral peaks would appear about 5.3 dB lower than their true values.

In addition, the Hamming window has a narrower main lobe than Blackman ($3.3/N$ vs $5.5/N$), providing better frequency resolution for identifying f_{min} and f_{max} accurately. For our signal with a large N, the main lobe width of $3.3/N$ translates to excellent absolute frequency resolution.

Referring to **FIgure X**, the output of the Hamming window can be seen and compared to the unwindowed application. Visually, a slight reduction in the noise floor clarifies the components of the audio file.

2.2 Task 2

This task mainly focuses on bandpass filter design, which will be used with the signal work/analysis from Task 1

2.2.1 Task 2 - Filter Design

The task is to design a bandpass FIR filter with the following specifications:

Parameter	Value
Passband edges	$f_{\min},\ f_{\max}$
Stopband edges	$f_{\min} - 2\,\text{kHz},\ f_{\max} + 2\,\text{kHz}$
Max passband ripple	$0.1\,\text{dB}$
Stopband attenuation	$> 50\,\text{dB}$

Table 1: Bandpass FIR filter specifications.

The design uses the impulse response truncation (IRT) method, which relies on the Fourier transform relationship between the frequency response $H(\Omega)$ and impulse response $h[n]$:

$$H(\Omega) = \sum_{n=-\infty}^{\infty} h[n]\, e^{-j\Omega n}, \qquad h[n] = \frac{1}{2\pi} \int_{-\pi}^{\pi} H(\Omega) e^{j\Omega n}\, d\Omega. \tag{12}$$

For an ideal lowpass filter with normalised cutoff frequency $F_c = f_c/f_s$, the impulse response is

$$h_D[n] = 2F_c \frac{\sin(2\pi F_c n)}{2\pi F_c n} = 2F_c\, \text{sinc}(2F_c n), \tag{13}$$

with

$$h_D[0] = 2F_c.$$

A bandpass filter is obtained by subtracting two ideal lowpass filters with cutoff frequencies $F_2 > F_1$:

$$h_{BP}[n] = 2F_2\, \frac{\sin(2\pi F_2 n)}{2\pi F_2 n} - 2F_1\, \frac{\sin(2\pi F_1 n)}{2\pi F_1 n}.$$

At $n = 0$,

$$h_{BP}[0] = 2(F_2 - F_1).$$

The normalised frequencies are defined as

$$F = \frac{f}{f_s}, \qquad F_1 = \frac{f_{\min}}{f_s}, \qquad F_2 = \frac{f_{\max}}{f_s}.$$

For transition bands centred on the stopband edges:

$$F_{c1} = \frac{f_{\min} - 1000}{f_s}, \qquad F_{c2} = \frac{f_{\max} + 1000}{f_s}.$$

The ideal impulse response is infinite, so it is truncated to $N = 2M + 1$ samples:

$$h[n] = h_D[n - M], \qquad n = 0, 1, \ldots, 2M. \tag{14}$$

This centres the impulse response and produces a causal filter.

Different window functions give different transition widths and stopband attenuations:

Window	**Transition Width**	**Stopband Attenuation**
Rectangular	$0.9/N$	21 dB
Hanning	$3.1/N$	44 dB
Hamming	$3.3/N$	53 dB
Blackman	$5.5/N$	74 dB

Table 2: Comparison of window functions for FIR filter design.

Because the required stopband attenuation is greater than $50dB$, the Hamming and Blackman windows both satisfy the requirement; the Hamming window gives the narrower transition band.

For the Hamming window, the transition width is approximately

$$\Delta F = \frac{3.3}{N}.$$

The transition band is 2 kHz wide, so

$$\Delta F = \frac{2000}{96000} = 0.02083, \qquad N = \frac{3.3}{0.02083} \approx 158.4.$$

Choosing the nearest odd length gives $N = 159$ and $M = 79$.

The final filter coefficients are computed by applying the chosen window:

$$h[n] = w[n]\, h_D[n - M].$$

The Hamming window is defined as

$$w[n] = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N - 1}\right), \qquad n = 0, 1, \ldots, N - 1.$$

Code Explanation

```
%% Sub-task 2.1: FIR Bandpass Filter Design

%% Define filter specifications
fc = 16000;               % Carrier frequency in Hz (from Task 1)
fmin = fc - 4000;         % Lower passband edge (Hz)
fmax = fc + 4000;         % Upper passband edge (Hz)

% Stopband edges (as specified)
fstop_lower = fmin - 2000;    % Lower stopband edge (Hz)
fstop_upper = fmax + 2000;    % Upper stopband edge (Hz)

% Transition bandwidth
transition_bandwidth = 2000;  % Hz

% Cutoff frequencies are at the centre of the transition bands (
   normalised)
Fc1 = (fmin - 1000) / fs;     % Lower cutoff (normalised)
Fc2 = (fmax + 1000) / fs;     % Upper cutoff (normalised)

% Calculate normalised transition width
delta_F = transition_bandwidth / fs;

% Display specifications for filter. (removed for readability)

N_calculated = 3.3 / delta_F; % Hamming Window taps
N = ceil(N_calculated);

% Ensure N is odd for symmetric filter
if mod(N, 2) == 0
    N = N + 1;
end

M = (N - 1) / 2;  % Number of coefficients either side of centre

% Printing out prior information

%% Design the ideal bandpass impulse response
% h_BP[n] = 2*Fc2*sinc(2*Fc2*n) - 2*Fc1*sinc(2*Fc1*n)

n_ideal = -M:M; % n ranges from -M to +M (centred at 0)

h_ideal = zeros(1, N); % Calculate ideal impulse response for
   bandpass filter

for i = 1:N
    n = n_ideal(i);
    if n == 0
        % For n = 0: h[0] = 2*Fc2 - 2*Fc1
        h_ideal(i) = 2*Fc2 - 2*Fc1;
    else
        % For n != 0: h[n] = 2*Fc2*sinc(2*Fc2*n) - 2*Fc1*sinc(2*
```

```
   Fc1*n)
50         % sinc(x) = sin(pi*x)/(pi*x), but here we use sin(2*pi*Fc
   *n)/(2*pi*Fc*n)
51         term1 = 2*Fc2 * sin(n * 2*pi*Fc2) / (n * 2*pi*Fc2);
52         term2 = 2*Fc1 * sin(n * 2*pi*Fc1) / (n * 2*pi*Fc1);
53         h_ideal(i) = term1 - term2;
54     end
55 end
56
57 % w[n] = 0.54 - 0.46*cos(2*pi*n/(N-1)) for n = 0, 1, ..., N-1
58
59 n_window = 0:N-1;
60 hamming_win = 0.54 - 0.46 * cos(2 * pi * n_window / (N - 1));
61
62 h_windowed = h_ideal .* hamming_win; % Apply window to ideal
   impulse response
63
64 % Printing window output
65
66 %% Plot the filter design process
67
68 figure('Name', 'FIR Filter Design', 'Position', [100, 100, 1200,
   800]);
69
70 % Plot 1: Ideal impulse response (unwindowed) (1st subplot)
71
72 % Plot 2: Hamming window (2nd subplot)
73
74 % Plot 3: Windowed impulse response (final filter coefficients)
   (3rd subplot)
75
76 h_bp = h_windowed;  % Final bandpass filter coefficients stored
   for later
77
78 fprintf('Filter coefficients stored in: h_bp\n');
79 fprintf('Number of taps:              %d\n', length(h_bp));
80 fprintf('Filter delay:                %d samples (%.4f ms)\n', M, M
   /fs*1000);
```

From the listing above, several implementations can be seen. The first part of the listing shows the specifications for the band-pass filter, from f_c, f_{min}, f_{max} to the stopband and cut-off frequencies.

Afterwards, a quick print block to verify the specifications and then creating the Hamming window for the filter response. Then, ideal impulse response output has been calculated. Further verification continues, for the coefficients of the impulse response.

Finally, plots to show the 1. The unwindowed impulse response, 2. Hamming window response and 3. windowed impulse response.

Results and Discussion

The code from *listing X* gives us the following outputs (**Figure X and Y**), which we can show as verification. Firstly, it is important to analyse the terminal output, verifying the calculations

FIGURE SHOWING THE TERMINAL OUTPUT FOR FILTER SPECS

Figure X clearly shows that our calculations meet the requirements demanded by the initial assignment file.

More importantly, **Figure Y** shows that our filter, firstly, adheres to the specification. Starting of with the tap count (159 coefficients), which is correct for the Hamming window (referring to *Equation X*).

FIGURE SHOWING THE 3 IMPULSE RESPONSES

2.2.2 Task 2 - Phase and Filter Verification

Procedure and Theory
Before applying the filter to the AM signal, we verify that it satisfies the required specifications by computing its frequency response across the full frequency range. Using the prior information (*Equations X, Y and Z*) and the impulse response output can be used for this part of the task.

The frequency response $H(f)$ of an FIR filter is given by the Fourier transform of its impulse response $h[n]$:

$$H(f) = \sum_{n=0}^{N-1} h[n]\, e^{-j2\pi f n/f_s}. \tag{15}$$

In practice, the response is obtained using the FFT, typically with zero-padding to improve frequency resolution and produce a smoother spectral estimate.

Table X shows the requirements that need to be verified.

Parameter	Requirement
Passband (f_{min} to f_{max})	Gain ≈ 0 dB, ripple < 0.1 dB
Stopband ($< f_{\text{min}} - 2\,\text{kHz}$ and $> f_{\text{max}} + 2\,\text{kHz}$)	Attenuation > 50 dB
Transition bands	Smooth rolloff within 2 kHz

Table 3: Filter specification verification criteria

```
%% Sub-task 2.2: Frequency Response Verification

% Compute frequency response using zero-padded FFT
N_fft = 8192;  % Zero-pad for smooth frequency response plot
H = fft(h_bp, N_fft);
H_magnitude = abs(H);
H_dB = 20 * log10(H_magnitude + eps);
H_phase = angle(H);

% Unwrap phase for clearer visualisation
H_phase_unwrapped = unwrap(H_phase);
```

```
12
13 % Create frequency vector (single-sided)
14 f_response = (0:N_fft/2) * fs / N_fft;
15 H_dB_single = H_dB(1:N_fft/2+1);
16 H_phase_single = H_phase_unwrapped(1:N_fft/2+1);
17
18 %% Plot frequency response - Magnitude
19
20 figure('Name', 'Filter Frequency Response - Magnitude', 'Position
      ', [100, 100, 1200, 600]);
21
22 % Full spectrum view (removed)
23 % Add specification lines (removed)
24 %% Plot phase response (removed)
25 %% Measure actual filter performance (removed)
26
27 % Find indices for passband and stopband regions
28 passband_indices = find(f_response >= fmin & f_response <= fmax);
29 stopband_lower_indices = find(f_response <= fstop_lower);
30 stopband_upper_indices = find(f_response >= fstop_upper &
      f_response <= fs/2);
31
32 % Measure passband ripple
33 passband_gain_dB = H_dB_single(passband_indices);
34 passband_max = max(passband_gain_dB);
35 passband_min = min(passband_gain_dB);
36 passband_ripple = passband_max - passband_min;
37
38 % Measure stopband attenuation
39 stopband_lower_max = max(H_dB_single(stopband_lower_indices));
40 stopband_upper_max = max(H_dB_single(stopband_upper_indices));
41 stopband_max = max(stopband_lower_max, stopband_upper_max);
42
43 % Calculate group delay (should be constant = M for linear phase)
44 group_delay_samples = M;
45 group_delay_ms = M / fs * 1000;
46
47 %% Display verification results
48
49 %% Overall verification summary
```

Listing 3: MATLAB Code for Task 2 Part 2: Impulse Response Verification

Code Explanation

For the sake of clarity, the logic for plotting and printing the terminal output has been removed. The plots and output will be shown directly instead.

Looking at the first few lines of *Listing X*, zero-padding interpolates the frequency response for a smoother plot without changing the filter characteristics. The FFT bins are then mapped to the physical frequencies accordingly, for the single sided spectrum. Afterwards, other information is shown, including the linear phase plot and the bandpass

view (zoomed and wider views). Terminal output can be shown as well, which can be used for visual verification.

Results and Discussion

As it has been previously stated, the logic for the plotting is available on Github and in the appendix. Regarding Task 2, **Figure X** (depicted below), shows the terminal output for verification purposes.

FIGURE SHOWING THE TERMINAL OUTPUT.

From the figure, it is clearly visible that our filter meets the requirements. Firstly, the peak to peak ripple is $0.0376dB$, (lower than the needed $0.1dB$). Regarding the stopband, the attenuation needs to be greater than $50dB$, and, in the worst case, $53.64dB$. The phase response is linear as well, with a very small delay (79 samples or $0.8229ms$), which shows that the time delay experienced by the amplitude envelope of a signal is very low.

FIGURE SHOWING THE LINEAR PHASE RESPONSE

From **Figure X**, the linear response is clearly visible and fits the requirements (highlighted f_{min} and f_{max}) and the linear behaviour stops after the passband frequencies as well ($10kHz$ and $22kHz$). This ensures that there is a strictly linear response and no phase distortion as well, resulting in a clearer audio output.

For further verification, magnitude response and frequency domain spectrum comparisons can be made as well, showcasing that the filter has worked. In **Figure X**, the message (the three characters), lie between $f_c \pm B$, and then adding in the bandpass filter specifications shows that the needed output has been 'elevated' above the noise, compared to the initial frequency spectrum.

FIGURE SHOWING THE FREQ DOMAIN COMPARISON BEFORE AND AFTER FILTERING.

In addition, **Figure Y** reveals the magnitude response, solidifying the verification described above.

2.2.3 Task 2 - Custom Convolution Method

Procedure and Theory
The filtering stage must be implemented using custom convolution code rather than MATLAB's built-in `filter()` or `conv()`. This requires computing the FIR convolution directly.

For an input signal $x[n]$ and FIR coefficients $h[n]$, the output is shown in *Equation X*:

$$y[n] = \sum_{k=0}^{N-1} h[k]\, x[n-k], \tag{16}$$

where N is the number of filter taps. Each output sample is obtained by taking the most recent N input samples $x[n], x[n-1], \ldots, x[n-(N-1)]$, multiplying them by the corresponding coefficients $h[0], h[1], \ldots, h[N-1]$, and summing the products.

At the beginning of the signal, where past samples do not exist, zero-padding is used so that $x[n] = 0$ for $n < 0$. This ensures that the output length matches the input length.

Code and Explanation

```
function y = custom_conv(x, h)
    is_column = iscolumn(x);      % Store original orientation
    x = x(:)';      % Store original orientation
    h = h(:)';
    % Get lengths
    L = length(x);  % Input signal length
    N = length(h);  % Filter length
    % Zero-pad the input signal (N-1 zeros at beginning)
    x_padded = [zeros(1, N-1), x];
    % Preallocate output
    y = zeros(1, L);
    % Perform convolution
    for n = 1:L
        accumulator = 0;
        for k = 1:N
            x_index = n + N - k;
            accumulator = accumulator + h(k) * x_padded(x_index);
        end
        y(n) = accumulator;
    end
    % Restore original orientation
    if is_column
        y = y(:); % Add StackOverflow post for convolution code
    end
end
```

Listing 4: MATLAB Code for Task 2 Part 3: Custom Convolution Operation

Listing X above shows the code for applying the convolution function (requiring an input signal and FIR coefficients).

From lines (X - Y) outer loop produces each output sample. Inner loop computes the weighted sum of filter coefficients multiplied by corresponding input samples. The index $n + N - k$ accounts for zero-padding offset (mentioned earlier). For the sake of clarity, code has been added for terminal output to show that the convolution operation has occurred. This implementation is a lot slower compared MATLAB's `conv()` function, but similar results will be obtained in the next tasks.

TERMINAL OUTPUT SHOWING THE SPEED OF THE OUTPUT

2.2.4 Task 2 - Applying The Filter to an AM Signal

Procedure and Theory

With the custom convolution function and the FIR bandpass filter designed, the filter is applied to the AM signal to suppress out-of-band noise.

The expected behaviour of the bandpass filter is as follows:

- It passes the AM content within the range $f_{\min}$ to $f_{\max}$.
- It attenuates components below $f_{\min} - 2\,\text{kHz}$ and above $f_{\max} + 2\,\text{kHz}$ by more than 50 dB.
- It preserves the signal structure required for subsequent demodulation.

The FIR filter introduces a delay of M samples, corresponding to half the filter length. This delay can be compensated for visualization, although it does not affect the demodulation process since the entire signal is processed.

```
%% Sub-task 2.4: Apply Bandpass Filter to AM Signal
% Apply the bandpass filter using our custom convolution function
fprintf('Filtering signal using custom_conv()...\n');
tic;
x_filtered = custom_conv(x, h_bp);
filter_time = toc;

%% Time domain comparison (removed from listing)
% Original signal and filtered signal plots removed for clarity.

%% Frequency domain comparison
% Compute spectrum of filtered signal using Hamming window
x_filtered_windowed = x_filtered .* hamming_window;
X_filtered = fft(x_filtered_windowed);
X_filtered_magnitude = abs(X_filtered);
X_filtered_normalised = X_filtered_magnitude / N / CG_hamming;
X_filtered_single = X_filtered_normalised(1:num_bins_single_sided
    );
X_filtered_single(2:end-1) = 2 * X_filtered_single(2:end-1);
X_filtered_dB = 20 * log10(X_filtered_single + eps);

figure('Name', 'Bandpass Filter - Frequency Domain', 'Position',
    [100, 100, 1200, 700]);

% Original spectrum (plot removed)
% Filtered spectrum (removed plot)
%% Calculate noise reduction statistics
% Measure power in passband and stopband before and after
    filtering

% Passband power (should be similar before and after)
passband_indices_signal = find(f >= fmin & f <= fmax);
passband_power_before = mean(X_hamming_single(
    passband_indices_signal).^2);
passband_power_after = mean(X_filtered_single(
    passband_indices_signal).^2);

% Stopband power (should be much lower after filtering)
stopband_indices_lower = find(f <= fstop_lower);
stopband_indices_upper = find(f >= fstop_upper & f <= fs/2);
stopband_indices_signal = [stopband_indices_lower,
    stopband_indices_upper];
```

```
37
38 stopband_power_before = mean(X_hamming_single(
       stopband_indices_signal).^2);
39 stopband_power_after = mean(X_filtered_single(
       stopband_indices_signal).^2);
40
41 % Calculate noise reduction in dB
42 noise_reduction_dB = 10 * log10(stopband_power_before /
       stopband_power_after);
43
44 % Calculate signal-to-noise improvement
45 snr_before = 10 * log10(passband_power_before /
       stopband_power_before);
46 snr_after = 10 * log10(passband_power_after /
       stopband_power_after);
47 snr_improvement = snr_after - snr_before;
48
49 % Terminal output removed for clarity
50 %% Amplitude statistics comparison (removed from listing to be
       concise)
```

Listing 5: MATLAB Code for Task 2 Part 3: Signal Filter Application

Code Explanation

Line X implements the convolution function to the input signal (output then matches input length).

Lines X to Y apply the same Hamming window (and normalisation etc) as before to ensure consistent spectral analysis and fair comparison with the original signal spectrum.

Lines X and Y Are used for power calculations. Power is proportional to the mean of squared magnitudes. Comparing passband and stopband power before/after filtering quantifies the filter's effectiveness.

Lines X and Y Shows the signal-to-noise ratio compares the power in the signal band to the power in the noise (stopband) region. The improvement indicates how much better the signal quality is after filtering.

Results and Discussion

The first image **Figure X** shows the original, noisy signal in the time domain. Initially, 3 peaks are visible, but there is a lot of white noise (uniform).

FIGURE SHOWING THE SMALLER VERSION OF THE OG NOISY SIGNAL

The second, **Figure Y** shows the audio signal again (in time domain) after applying the bandpass filter. Now, the amplitude of the noise has decreased (roughly at 0.23), leaving behind 3 much clearer peaks, preserving the initial signal structure.

AUDIO SIGNAL AFTER BP

For the frequency response, it is simpler to reuse the discussion regarding **Figure X**, with a clear visual response showing the working bandpass filter.

As some of the code has been removed (plotting and terminal output logic) - **Figure Z** shows the terminal output, comparing a few key metrics, before and after filtering:

TERMINAL OUTPUT WITH PASSBAND STOPBAND IMPROVEMENTS

- Stopband power ($63.73dB$ increase), consistent with the requirement of $> 50dB$
- Passband power (slight decrease, $0.67dB$), which is acceptable.
- SNR before ($4.63dB$) and after ($69.02dB$) with an improvement of ($64.39dB$).

As out-of-band noise and the AM signal have been preserved properly, carrier recovery can commence.

2.3 Task 3

This task involves applying the square law to the filtered signal, identifying and computing the carrier frequency, generating the local carrier signal, then mixing, for the final output.

2.3.1 Task 3 - Carrier Recovery

Procedure
The carrier frequency f_c is present within the AM signal but may not be directly observable, especially in DSB-SC signals where the carrier is suppressed. To recover it, we apply a square-law operation.

Squaring exploits the identity from *Equation X*:

$$\cos^2(\omega_c t) = \frac{1}{2}\left(1 + \cos(2\omega_c t)\right), \tag{17}$$

so that for a DSB-SC signal $s(t) = m(t)\cos(\omega_c t)$,

$$s^2(t) = m^2(t)\cos^2(\omega_c t) = \frac{m^2(t)}{2}\left(1 + \cos(2\omega_c t)\right). \tag{18}$$

This generates a strong spectral component at $2f_c$, even when the original carrier at f_c is suppressed.

The message $m(t)$ has bandwidth $B = 4\,\text{kHz}$, so $m^2(t)$ has bandwidth $2B = 8\,\text{kHz}$. The component at $2f_c$ therefore appears with sidebands extending $\pm 8\,\text{kHz}$. For a carrier around $20\,\text{kHz}$, this places $2f_c \approx 40\,\text{kHz}$, well within the Nyquist limit of $48\,\text{kHz}$.

From the spectrum of the squared signal, the carrier is recovered by identifying the peak near $2f_c$ and computing

$$f_c = \frac{f_{\text{peak}}}{2}.$$

The carrier frequency is an integer multiple of $1\,\text{kHz}$, which provides an additional consistency check.

Code and Explanation

```
%% Task 3: Carrier Recovery and Mixing
% Apply square law to the bandpass filtered signal
x_squared = x_filtered .^ 2;

%% Compute spectrum of squared signal
% Apply Hamming window
x_squared_windowed = x_squared .* hamming_window;

% Compute FFT
X_squared = fft(x_squared_windowed);
X_squared_magnitude = abs(X_squared);
X_squared_normalised = X_squared_magnitude / N / CG_hamming;

% Single-sided spectrum
X_squared_single = X_squared_normalised(1:num_bins_single_sided);
X_squared_single(2:end-1) = 2 * X_squared_single(2:end-1);
X_squared_dB = 20 * log10(X_squared_single + eps);

%% Plot squared signal spectrum (removed for ease)
% Mark expected 2fc region (dotted)
xline(2*fc_rounded/1000, 'r--', 'LineWidth', 2);

% Define search range around 2fc
search_range_low = 2*fc_rounded - 5000;    % Hz
search_range_high = 2*fc_rounded + 5000;   % Hz

% Find indices of search region
search_indices = find(f >= search_range_low & f <=
   search_range_high);

% Extract the region to search for peaks
X_search = X_squared_single(search_indices);
f_search = f(search_indices);

% Use findpeaks
[peaks, locs] = findpeaks(X_search, f_search);

% Get the highest peak
[peak_value, max_idx] = max(peaks);
f_2fc_measured = locs(max_idx);

% Calculate measured carrier frequency
fc_measured = f_2fc_measured / 2;
fc_final = round(fc_measured / 1000) * 1000;
```

Listing 6: MATLAB Code for Task 3 Part 1: Carrier Recovery

At *Line X* element-wise squaring of the filtered signal occurs. This nonlinear operation creates frequency components at 0 (DC), $2f_c$, and various intermodulation products. The required region is then zoned, and the function `findpeaks()` is used to find the required $2f_c$, which can then be plotted. (Logic has been removed for clarity).

Carrier frequency is calculated by simply dividing the peak frequency ($2f_c$) by 2, utilising the square law rule (**Equation X**), so $(32kHz)/2 = 16kHz$, which is the correct f_c value.

Results and Discussion
Firstly, there is a clear peak at the $2f_c$ frequency and also at f_c. This shift in frequency can be seen as well. Sidebands have been preserved accordingly, correctly following what is to be expected from a bandpass filter. In addition, the new carrier frequency is an exact multiple of the initial frequency (double).

This can be proven by simply looking at **Figure X**

FIGURE SHOWING 2FC PEAK and FC PEAK

And for further verification, terminal output with error calculations, shown by **Figure Y**, exactly matching with the requirements.

TERMINAL OUTPUT FOR FURTHER VERIFICATION

It is paramount that this sub-task has been verified properly as the frequency will be used for the demodulation aspect of the task.

2.3.2 Task 3 - Carrier Generation and Mixing

With the carrier frequency f_c determined, a local carrier is generated and multiplied with the filtered AM signal as part of coherent demodulation.

For a DSB-SC signal $s(t) = m(t)\cos(\omega_c t)$, mixing with a local carrier $\cos(\omega_c t + \phi)$ gives

$$s(t)\cos(\omega_c t + \phi) = m(t)\cos(\omega_c t)\cos(\omega_c t + \phi). \tag{19}$$

Using the identity $\cos A \cos B = \frac{1}{2}[\cos(A - B) + \cos(A + B)]$,

$$s(t)\cos(\omega_c t + \phi) = \frac{m(t)}{2}\left[\cos(\phi) + \cos(2\omega_c t + \phi)\right].$$

This produces a baseband term $\frac{1}{2}m(t)\cos(\phi)$ and a high-frequency term at $2f_c$.

The phase ϕ determines the amplitude and polarity of the recovered signal:

$$\phi = 0 \Rightarrow \cos(\phi) = 1, \text{ maximum amplitude,}$$

$$\phi = \frac{\pi}{2} \Rightarrow \cos(\phi) = 0, \text{ no output,}$$

$$\phi = \pi \Rightarrow \cos(\phi) = -1, \text{ inverted output.}$$

For the present stage, $\phi = 0$ is used.

In the frequency domain, mixing shifts the spectrum such that the AM content around f_c appears both at 0 Hz (baseband) and at $2f_c$. The baseband component contains the message, while the component at $2f_c$ will be removed by the lowpass filter in the next stage.

Code and Explanation

```
%% Sub-task 3.2: Carrier Generation and Mixing
% Set initial phase to zero (will be optimised in Task 5)
phi = 0;

% Generate local carrier signal
% carrier(t) = cos(2*pi*fc*t + phi)
carrier = cos(2 * pi * fc_final * t + phi);

% Carrier frequency, phase and signal length outputs (removed)
%% Multiply filtered AM signal with carrier (mixing)
x_mixed = x_filtered .* carrier;

%% Time domain plots (removed for clarity)

figure('Name', 'Mixing Process - Time Domain', 'Position', [100,
    100, 1200, 800]);

%% Frequency domain analysis of mixed signal

% Apply Hamming window
x_mixed_windowed = x_mixed .* hamming_window;

% Compute FFT
X_mixed = fft(x_mixed_windowed);
X_mixed_magnitude = abs(X_mixed);
X_mixed_normalised = X_mixed_magnitude / N / CG_hamming;

% Single-sided spectrum
X_mixed_single = X_mixed_normalised(1:num_bins_single_sided);
X_mixed_single(2:end-1) = 2 * X_mixed_single(2:end-1);
X_mixed_dB = 20 * log10(X_mixed_single + eps);

%% Frequency domain plots (removed for clarity)
%% Analyse the frequency components

% Find power in baseband region (0 to 4 kHz - message bandwidth)
baseband_indices = find(f >= 0 & f <= 4000);
baseband_power = mean(X_mixed_single(baseband_indices).^2);

% Find power in 2fc region (2fc +- 4 kHz)
double_fc_indices = find(f >= (2*fc_final - 4000) & f <= (2*
    fc_final + 4000));
double_fc_power = mean(X_mixed_single(double_fc_indices).^2);

% Find power in noise region (between baseband and 2fc)
noise_region_low = 8000;  % Above baseband
noise_region_high = 2*fc_final - 8000;  % Below 2fc component
if noise_region_high > noise_region_low
    noise_indices = find(f >= noise_region_low & f <=
    noise_region_high);
    noise_power = mean(X_mixed_single(noise_indices).^2);
```

```
49 else
50     noise_power = 0;
51 end
52
53 % Carrier signal analysis methods (prints terminal outputs,
     removed for clarity)
```

Listing 7: MATLAB Code for Task 3 Part 2: Carrier Gen and Mixing

Listing X has terminal output and plotting logic removed for clarity. It showcases the techniques for:

1. Generating the cosine waves using the recovered carrier frequency (initial phase (ϕ) is set to 0 for future optimisation)
2. *Line X - Listing X* shows the mixing operation using element-wise multiplication.
3. *Lines X - Y, Listing X* applies the Hamming window and FFT. Afterwards, information regarding the baseband is calculated.

Results and Discussion

Firstly, for visual confirmation, the time domain plots before and after recovery + mixing shows the final output for the code from Task 3. The filtered AM signal shows the modulated carrier and the local carrier is a pure cosine at frequency f_c. The mixed signal also shows a decrease in noise, making the peaks clearer. This can be seen in **Figure X**.

FIGURE SHOWING THE TIME DOMAIN SIGNALS

In the frequency domain, the spectra before and after mixing can be compared. Before mixing, the signal energy is concentrated around f_c. After mixing, two components appear: a baseband term near 0 Hz, corresponding to the downconverted message, and a high-frequency term around $2f_c$, corresponding to the sum-frequency component.

The multiplication shifts the spectrum both downward and upward. The baseband component occupies the range 0 to 4 kHz, equal to the message bandwidth (**Figure X** below), while the component at $2f_c$ occupies $2f_c \pm 4\,\text{kHz}$. Little energy appears between these regions.

A lowpass filter is required because the mixed signal contains both the desired baseband message and the undesired $2f_c$ term. The filter will pass the range 0–4 kHz while rejecting frequencies around $2f_c$.

The process is sensitive to the phase ϕ. The baseband amplitude is scaled by $\cos(\phi)$; an incorrect phase therefore attenuates the recovered signal. At $\phi = \pi/2$, the output vanishes entirely. A later stage (Task 5 code) will determine the optimal phase.

FIGURE 16 WITH MIXING. NEED TO RECHECK WHAT IMAGES I NEED TO USE FOR THE PREVIOUS IMAGES.

TERMINAL OUTPUT FOR BASEBAND VERIFICATION.

2.4 Task 4 - IIR Filter Design and Verification

This task requires a full implementation of an IIR lowpass filter, taking the output of the FIR filter and filDesign a 4th order Butterworth lowpass filter with 4 kHz cutoff Verify the frequency response of the designed filter Implement your own IIR filter code (not MATLAB's filter() function) Apply the filter to the mixed signal Plot and describe output in time and frequency domainstering again (making it multistage), for clearer output.

2.4.1 Task 4 Part 1 - Designing IIR Filters

Procedure and Theory

It is important to understand the requirements for the filter's design. *Table X* shows the requirements:

Parameter	Value
Order	4
Cutoff frequency	4 kHz
Type	Butterworth

Table 4: IIR lowpass filter design parameters

IIR filters provide sharp cutoff characteristics with far fewer coefficients than FIR filters. A 4th-order Butterworth lowpass filter offers a good balance of monotonic passband behaviour, reasonable stopband attenuation, and computational efficiency.

A Butterworth filter is characterised by a maximally flat passband, monotonic magnitude response, $-3\,\mathrm{dB}$ attenuation at the cutoff frequency, and a rolloff rate of $20n\,\mathrm{dB/decade}$ for order n. For a 4th-order design, the rolloff is $80\,\mathrm{dB/decade}$.

Digital IIR filters are commonly designed by starting from an analogue prototype and applying the bilinear transform, which maps the s-plane to the z-plane:

$$s = \frac{2}{T_s}\frac{1 - z^{-1}}{1 + z^{-1}}. \tag{20}$$

Because the bilinear transform introduces frequency warping, pre-warping is used to preserve the desired cutoff frequency:

$$\omega_{\text{analog}} = \frac{2}{T_s}\tan\left(\frac{\omega_{\text{digital}} T_s}{2}\right). \tag{21}$$

For a cutoff frequency of $4\,\mathrm{kHz}$ and sampling rate $f_s = 96\,\mathrm{kHz}$:

$$\Omega_c = 2f_s \tan\left(\frac{\pi f_c}{f_s}\right) = 2 \cdot 96000 \tan\left(\frac{\pi \cdot 4000}{96000}\right).$$

The resulting IIR filter has transfer function:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_M z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}},$$

with corresponding difference equation:

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_M x[n-M] - a_1 y[n-1] - a_2 y[n-2] - \cdots - a_N y[n-N].$$

This highlights the recursive nature of IIR filters: each output sample depends on both past inputs and past outputs.

```
%% Task 4: IIR Lowpass Filter Design
% Filter specifications
filter_order = 4;
fc_lowpass = 4000;  % Cutoff frequency in Hz
% Uses normalised frequency where 1 = Nyquist frequency (fs/2)
Wn = fc_lowpass / (fs/2);
% Print method for showing specs has been removed.
%% Design the Butterworth filter using bilinear transform
% MATLAB's butter() function implements the bilinear transform
   method
% It returns coefficients for the transfer function H(z) = B(z)/A
   (z)
[b_iir, a_iir] = butter(filter_order, Wn, 'low');

% Method for printing coefficients has been removed.
%% Verify coefficient properties (printing removed)
%% Display the transfer function (printing method removed)
%% Plot filter coefficients
% Plotting logic for coefficients has been removed
```

Listing 8: MATLAB Code for Task 4 part 1: IIR Lowpass Filter Construction

For *Listing X*, the design of a 4th order Butterworth filter has been described. In MATLAB, the `butter()` function requires the cutoff frequency to be normalised by the Nyquist frequency $f_s/2$. For a cutoff of $f_c = 4\,\text{kHz}$ and sampling rate $f_s = 96\,\text{kHz}$,

$$W_n = \frac{f_c}{f_s/2} = \frac{4000}{48000} \approx 0.0833.$$

Lines X-Y - Listing X returns numerator coefficients (b) and denominator coefficients (a) for the transfer function. The 'low' parameter specifies a lowpass filter. At DC (z = 1), the transfer function reduces to $sum(b)/sum(a)$. Then, for a properly designed lowpass filter, this should equal 1 ($0dB$ gain at DC).

Results and Discussion

From **Figure X**, the numerator coefficients (b) are all positive and symmetric. The denominator coefficients (a) alternate in sign (characteristic of Butterworth) $a[0] = 1$ (normalised form). The DC gain should be 1.0 (0 dB), confirming the filter passes DC and low frequencies without attenuation.

A 4th order filter has:

- 5 numerator coefficients ($b[0] to b[4]$)
- 5 denominator coefficients ($a[0] to a[4]$)
- 4 poles and 4 zeros in the z-plane

The terminal output for filter behaviour verification can be seen here:

FIGURE WITH TEMRINAL OUTPUT SHOWING THE FILTER COEFFICIENTS ETC.

The numerator coefficients show:

1. $b[0] = 0.0002131387$
2. $b[1] = 0.0008525549$
3. $b[2] = 0.0012788324$
4. $b[3] = 0.0008525549$
5. $b[4] = 0.0002131387$

And the denominator coefficients are:

1. $a[0] = 1$
2. $a[1] = -3.3168079106$
3. $a[2] = 4.1742455501$
4. $a[3] = -2.3574027806$
5. $a[4] = 0.5033753607$

With a gain of 1 (DC), and a numerator and denominator sums equal 0.0034102196.

A visual output can also show the numerator and denominator coefficients, verifying the terminal output from above (**Figure X** below)

FIGURE SHOWING THE DENOMINATOR AND NUMERATORS FOR THE COEFFICIENT

2.4.2 Task 4 Part 2 - IIR Frequency Response Verification

Procedure and Theory
This part focuses on computing and verifying the frequency response of the IIR lowpass filter. To refresh, **Table X** shows the Butterworth filter specifications:

Parameter	Requirement
Cutoff frequency (-3 dB point)	4 kHz
Passband (0 to 4 kHz)	Monotonically flat
Rolloff rate	80 dB/decade (4th order × 20 dB/decade)

Table 5: Butterworth lowpass filter design requirements

The frequency response of an IIR filter is given by *Equation X*:

$$H(e^{j\omega}) = \frac{\sum_{k=0}^{M} b_k \, e^{-j\omega k}}{\sum_{k=0}^{N} a_k \, e^{-j\omega k}}. \tag{22}$$

In practice, this is evaluated by taking the FFT of the zero-padded numerator and denominator coefficient sequences and computing their ratio.

A Butterworth filter exhibits the following properties:

- -3 dB attenuation at the cutoff frequency,
- maximally flat passband without ripple,
- monotonic rolloff in the stopband,
- approximately linear phase (IIR filters generally have nonlinear phase).

Code and Explanation

```
%% Sub-task 4.2: IIR Frequency Response Verification
% Compute frequency response using freqz
N_freq = 8192;  % Number of frequency points
[H_iir, f_iir] = freqz(b_iir, a_iir, N_freq, fs);
% Magnitude response in dB
H_iir_magnitude = abs(H_iir);
H_iir_dB = 20 * log10(H_iir_magnitude + eps);
% Phase response (plots removed)
H_iir_phase = angle(H_iir);
H_iir_phase_unwrapped = unwrap(H_iir_phase);
%% Plot magnitude response
%% Measure actual filter performance
% Find -3 dB point
idx_3dB = find(H_iir_dB <= -3, 1, 'first');
if ~isempty(idx_3dB)
    f_3dB_actual = f_iir(idx_3dB);
else
    f_3dB_actual = NaN;
end
% Measure gain at specific frequencies
f_test_points = [100, 1000, 2000, 3000, 4000, 5000, 6000, 8000,
   10000, 20000];
gain_at_test_points = zeros(size(f_test_points));
for i = 1:length(f_test_points)
    [~, idx] = min(abs(f_iir - f_test_points(i)));
    gain_at_test_points(i) = H_iir_dB(idx);
end
% Measure rolloff rate (attenuation per octave after cutoff)
% Compare attenuation at 8 kHz (1 octave above 4 kHz) and 16 kHz
   (2 octaves)
[~, idx_8k] = min(abs(f_iir - 8000));
```

```
[~, idx_16k] = min(abs(f_iir - 16000));
atten_8k = H_iir_dB(idx_8k);
atten_16k = H_iir_dB(idx_16k);
rolloff_per_octave = atten_8k - atten_16k;
%% Display verification results (removed)
%% Verify Butterworth characteristics
% Check passband flatness (should be monotonic, no ripple) -
   terminal output removed
passband_idx = find(f_iir <= fc_lowpass);
passband_gain = H_iir_dB(passband_idx);
passband_ripple = max(passband_gain) - min(passband_gain);
% Check gain at cutoff
[~, idx_fc] = min(abs(f_iir - fc_lowpass));
gain_at_fc = H_iir_dB(idx_fc);
if abs(gain_at_fc - (-3)) < 0.5
%% Pole-Zero plot for stability verification
% Get poles and zeros
zeros_iir = roots(b_iir);
poles_iir = roots(a_iir);
% Plot unit circle, poles and zeros (removed)
% Check stability
pole_magnitudes = abs(poles_iir);
max_pole_magnitude = max(pole_magnitudes);
```

Listing 9: MATLAB Code for Task 4 part 1: IIR Lowpass Filter Construction

From *Listing X*, MATLAB's `freqz()` function computes the frequency response by evaluating $H(e^{j\omega})$ at N_{freq} equally spaced frequency points. The fourth argument specifies the sampling frequency, ensuring that the frequency axis is correctly scaled in hertz. *Line X-Y - Listing X*, locates where the magnitude first drops to $-3dB$, which defines the cutoff frequency for Butterworth filters, and then compares attenuation at 8 kHz and 16 kHz (one octave apart) to verify the expected 24 dB/octave rolloff (4th order $\times$ 6 dB/octave per pole).

Further verification has been added (code not shown for clarity), including a plot of the unit circle, highlighting the poles and zeros, showing that the filter response is expected and stable.

Results and Discussion

There are several factors to consider:

1. Behaviour at the cutoff frequency (at $-3dB$)
2. Passband ripple
3. Roll-off rate compared to the expected value.
4. Cut-off gain
5. Phase response

FIGURE SHOWING THE 4 PLOTS.

From **Figure X**, looking at the top left subplot, after $4kHz$, there is a roll-off, showing that the filter is removing higher frequencies (expected from a low-pass filter). The expected roll off rate is $24dB$ per octave, but this implementation achieves $26.67dB$ per octave.

For the cutoff frequency ($4kHz$), the gain should be $-3dB$. For this implementation, it happens at $4.00195kHz$, which is very close to the required frequency. If instead, the comparison is looking at the gain from the frequency, then the gain is $-3.02dB$. All of this can be seen clearly from plots on the right hand side.

As IIR filters have a non-linear phase response, the output from the bottom left plot is correct, but the phase change is not sharp enough, this can introduce phase distortion, decreasing signal quality. This could be resolved by looking at higher order filters or even a multi-stage lowpass filter as well.

For extra confirmation of the filter's effectiveness, **Figure X** shows the poles and zeros map, showing that the poles lie within the unit circle. As this output agrees with the theory, this can be considered successful, ensuring stability.

UNIT CIRCLE PLOT

Figure X has the terminal output as well, showing the comparison (expected vs actual output) as well the mapping of the poles and zeros.

TERMINAL OUTPUT SHOWING FILTER VERIFICATION

2.4.3 Task 4 Part 3 - Custom IIR Filter Implementation and Testing

Procedure and Theory
From *Equation X*, using the transfer function and time-difference equations to get a simpler representation of the IIR filter.

From the transfer function:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_M z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}} \tag{23}$$

The time-domain difference equation is:

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_M x[n-M] - a_1 y[n-1] - a_2 y[n-2] - \cdots - a_N y[n-N] \tag{24}$$

Or more compactly:

$$y[n] = \sum_{k=0}^{M} b_k \, x[n-k] \; - \; \sum_{k=1}^{N} a_k \, y[n-k] \tag{25}$$

A few considerations for implementation:

- Coefficient normalisation: The denominator coefficient a_0 should equal 1. If not, divide all coefficients by a_0.

- Initial conditions: We assume zero initial conditions (all past inputs and outputs are zero at the start).
- Numerical precision: IIR filters can accumulate numerical errors due to feedback. Double precision (MATLAB default) is sufficient for the application.
- Causality: We can only use past values of $y[n]$, not future values.

```
function y = custom_iir_filter(b, a, x)
% Store original orientation
is_column = iscolumn(x);
% Convert all inputs to row vectors for consistent processing
b = b(:).';  % Ensure row vector
a = a(:).';  % Ensure row vector
x = x(:).';  % Ensure row vector
% Get lengths
L = length(x);       % Signal length
M = length(b) - 1;   % Numerator order (number of b coefficients
   minus 1)
N = length(a) - 1;   % Denominator order (number of a coefficients
    minus 1)
%% Normalise coefficients if a(1) is not 1
if a(1) ~= 1
    b = b / a(1);
    a = a / a(1);
end
%% Initialise buffers
x_buffer = zeros(1, M + 1); % input
% Output buffer: stores past N output samples
y_buffer = zeros(1, N);
%% Preallocate output array
y = zeros(1, L);
%% Main filtering loop
for n = 1:L
    % Shift input buffer to the right (make room for new sample)
    for k = M+1:-1:2
        x_buffer(k) = x_buffer(k-1);
    end
    % Insert current input sample at the beginning
    x_buffer(1) = x(n);
        % Calculate feedforward (FIR) part: sum of b(k) * x[n-k
   +1]
    feedforward_sum = 0;
    for k = 1:M+1
        feedforward_sum = feedforward_sum + b(k) * x_buffer(k);
    end
    % Calculate feedback (recursive) part: sum of a(k) * y[n-k+1]
    feedback_sum = 0;
    for k = 1:N
        feedback_sum = feedback_sum + a(k+1) * y_buffer(k);
    end
    % Compute current output: y[n] = feedforward - feedback
    y(n) = feedforward_sum - feedback_sum;
```

```
    % Shift output buffer to the right (make room for new output)
    for k = N:-1:2
        y_buffer(k) = y_buffer(k-1);
    end
    % Insert current output at the beginning (becomes y[n-1] for
   next iteration)
    if N >= 1
        y_buffer(1) = y(n);
    end
end
%% Restore original orientation if input was column vector
if is_column
    y = y(:);
end
```

Listing 10: MATLAB Code for Task 4 part 3: IIR Filter Custom

The list below describes how the IIR filter works,

1. **Shift input buffer:** Move all elements to the right, discarding the oldest.
2. **Insert new input:** Place the current sample $x[n]$ at position 1.
3. **Feedforward sum:** Compute
$$\sum_{k=0}^{M} b_k \, x[n-k]$$
4. **Feedback sum:** Compute
$$\sum_{k=1}^{N} a_k \, y[n-k]$$
5. **Output:**
$$y[n] = \text{feedforward} - \text{feedback}$$
6. **Shift output buffer:** Make room for the new output value.
7. **Store output:** Insert $y[n]$ into the output buffer for use in the next iteration.

Feedback needs to be subtracted as well. The difference equation of the IIR filter is:

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

The negative sign in the feedback terms ensures the filter implements the correct transfer function:

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

which requires subtracting the feedback contributions in the time-domain equation.

Results and Discussion

Maximum error is extremely small, with a maximum error value of $9.5e \times 10^{-15}$, which can

be seen in the top right plot. This confirms mathematical equivalence with MATLAB's optimised implementation. Any larger errors would indicate a bug in the implementation. The impulse and step response outputs from the custom method also follow the inbuilt MATLAB function's behaviour very closely, with a minimal difference (**Figure X** below).

Observing the impulse response shape, IIR filters have theoretically infinite impulse responses, but in practice, the response decays towards zero The decay rate depends on pole locations.

For the step response, it should settle to the DC gain value $sum(b)/sum(a) = 1.0\%$ for the filter. This shows how quickly the filter reaches steady state Any oscillation or overshoot (minimal) indicates the filter's transient behaviour.

IMAGE SHOWING THE 4 PLOTS WITH THE STEP, IMPULSE ETC.

Regarding the timing, the custom implementation is written in MATLAB and has a higher time complexity due to the structure of the code, as well as MATLAB being a higher level language. From **Figure X**, the custom implementation is 18.2× slower. Error is also quite low, and for the sake of speed, only the first 500 samples have been used, ($2.33e \times 10^{-15}$ for impulse response and $1.73e \times 10^{-13}$ for the step response).

2.4.4 Task 4 Part 4 - IIR Filter and The Mixed Signal

Procedure and Theory
Now applying the verified IIR lowpass filter to the mixed signal from Task 3.This is the final filtering stage in the demodulator chain.

After mixing in Task-3, the signal contains two components, which is where the lowpass filter comes in:

- **Baseband (0–4 kHz):** The desired message signal

$$\frac{m(t)}{2}\cos(\phi)$$

- **High-frequency (around $2f_c$):** An unwanted component

$$\frac{m(t)}{2}\cos(2\omega_c t + \phi)$$

The 4 kHz lowpass filter will:

- Pass the baseband message (0–$4kHz$)
- Reject the $2f_c$ component (approximately $2 \times 20\,\text{kHz} = 40\,\text{kHz}$ for a typical carrier)

After lowpass filtering the expected outcome is:

- Only the message signal remains
- The output should resemble speech (though possibly noisy or phase-dependent)
- The amplitude will be scaled by

$$\frac{1}{2}\cos(\phi)$$

  where ϕ is the carrier phase.

```
tic;
x_demodulated = custom_iir_filter(b_iir, a_iir, x_mixed);
time_iir_filter = toc;
%% Plot time domain comparison
%% Frequency domain analysis
% Compute spectrum of demodulated signal
window_demod = hamming(length(x_demodulated))';
x_demod_windowed = x_demodulated .* window_demod;
X_demod = fft(x_demod_windowed);
X_demod_magnitude = abs(X_demod) / length(X_demod);
% Single-sided spectrum
N_demod = length(X_demod);
X_demod_single = X_demod_magnitude(1:floor(N_demod/2)+1);
X_demod_single(2:end-1) = 2 * X_demod_single(2:end-1);
f_demod = (0:floor(N_demod/2)) * fs / N_demod;
% Convert to dB
X_demod_dB = 20 * log10(X_demod_single + eps);
% Also compute spectrum of mixed signal for comparison
window_mixed = hamming(length(x_mixed))';
x_mixed_windowed = x_mixed .* window_mixed;
X_mixed = fft(x_mixed_windowed);
X_mixed_magnitude = abs(X_mixed) / length(X_mixed);
N_mixed = length(X_mixed);
X_mixed_single = X_mixed_magnitude(1:floor(N_mixed/2)+1);
X_mixed_single(2:end-1) = 2 * X_mixed_single(2:end-1);
f_mixed = (0:floor(N_mixed/2)) * fs / N_mixed;
X_mixed_dB = 20 * log10(X_mixed_single + eps);
%% Plot frequency domain comparison (removed for clarity)
%% Analysis of filtering effect
% Calculate power in different frequency bands
% Baseband (0 to 4 kHz) - message region
baseband_idx_mixed = find(f_mixed <= fc_lowpass);
baseband_idx_demod = find(f_demod <= fc_lowpass);
baseband_power_before = sum(X_mixed_single(baseband_idx_mixed)
    .^2);
baseband_power_after = sum(X_demod_single(baseband_idx_demod).^2)
    ;
% Stopband (above 4 kHz) - should be attenuated
stopband_idx_mixed = find(f_mixed > fc_lowpass);
stopband_idx_demod = find(f_demod > fc_lowpass);
stopband_power_before = sum(X_mixed_single(stopband_idx_mixed)
    .^2);
stopband_power_after = sum(X_demod_single(stopband_idx_demod).^2)
    ;
% 2fc region (2fc +- 4 kHz)
fc2_low = 2*fc_final - 4000;
fc2_high = 2*fc_final + 4000;
fc2_idx_mixed = find(f_mixed >= fc2_low & f_mixed <= fc2_high);
fc2_idx_demod = find(f_demod >= fc2_low & f_demod <= fc2_high);
fc2_power_before = sum(X_mixed_single(fc2_idx_mixed).^2);
fc2_power_after = sum(X_demod_single(fc2_idx_demod).^2);
```

```
% Calculate attenuation (no print logic)
baseband_change_dB = 10 * log10(baseband_power_after /
    baseband_power_before);
stopband_attenuation_dB = 10 * log10(stopband_power_before /
    stopband_power_after);
fc2_attenuation_dB = 10 * log10(fc2_power_before /
    fc2_power_after);
%% SNR (print logic removed)
% Estimate SNR as ratio of baseband power to remaining stopband
    power
snr_before = 10 * log10(baseband_power_before /
    stopband_power_before);
snr_after = 10 * log10(baseband_power_after /
    stopband_power_after);
snr_improvement = snr_after - snr_before;
%% Summary print (not shown)
```

Listing 11: MATLAB Code for Task 4 part 4: IIR Filter Mixed Sig

This code applies the IIR filter to the mixed signal and then applies windowing, etc, then finds various information about the signal in both the time and frequency domains:

- Baseband power change
- Stopband power change and attenuation
- $2f_c$ region power changes and attenuation
- SNR before and after the lowpass filter has been applied.

Finally, 2 plots are computed, the first showing the signals before and after demodulation, along with a zoomed in view (time domain), then another window with 4 plots, showing the baseband comparison, signal spectra before and after filtering, and the $2f_c$ region changes.

Results and Discussion
The mixed signal contains rapid oscillations (from the $2f_c$ component), but the demodulated signal is smoother, with lower frequency content. The $50ms$ overlay shows how the lowpass filter removes rogue high-frequency components, resulting in a slightly clearer output.

IMAGE SHOWING THE 3 TIME DOMAIN PLOTS

For the frequency domain, there are 4 plots to talk about. The top 2 plots show the before and after when applying the demodulation techniques, showing the higher frequency component being removed correctly.

In the baseband, there is a very minimal change, resulting in a slightly sharper roll off in the stopband, but nothing significant. The transition at $4kHz$ is clearly visible in the baseband comparison plot.

In the bottom right plot, looking at the $2f_c$ region after applying the filter, there aren't any red lines, indicating that the high frequency component has been rmeoved. Further verification (**Figure X - terminal output**) shows the improvement.

IMAGE SHOWING THE 4 PLOTS DEMOD, POST FILTER, 2FC etc

The terminal output below (**FIgure X**), shows that the baseband change is minimal (as expected), but there is a sizeable increase in the stopband power (attenuation of $17.52dB$). When considering the $2f_c$ region, the attenuation shows $87.02dB$, which correctly implies a near-total removal of that component. Final SNR improvement shows an increase from $-2.26dB$ to $15.09dB$, an acceptable increase in SNR, indicating a clearer audio output.

IMAGE SHOWING TERMINAL OUTPUT OF SNR AND POWER VALUES

References

References that I have used in the report. (articles, MATLAB documentation, textbooks etc.)

Appendix

Include some flowcharts for code design if possible. Include entire code listings if possible or split for the tasks. (code snippet for task 1, task 2 etc.) Include conv.m and iir filter design code.