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

Note: Throughout this report, plotting and terminal output code has been removed from listings for clarity. Complete implementations are available in the Appendix and the provided GitHub repository.

Procedure, Results and Discussion

2.1 Task 1 Audio File Analysis and Transformation

This task analyses the provided audio file (Sahas_Talasila.wav) in both time and frequency domains to extract key signal properties.

2.1.1 Task 1 - Audio File Information

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)

If N is the total number of samples, then the duration of the signal is given by *Equation 1*:

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

The frequency resolution (bin width) determines the minimum separation between distinguishable spectral components, as shown in *Equation 2*. For AM demodulation, sub-Hz resolution ensures the carrier and sideband edges can be precisely identified, which is critical for accurate filter design in later tasks.

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

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 be correctly represented 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 with carriers in the tens of kHz and bandwidths of a few kHz, as the entire modulated spectrum ($f_c \pm B$) must fit below the Nyquist frequency to avoid aliasing distortion.

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));

%% Sub-task 1.2: Time Domain Analysis

% Create time vector
t = (0:N-1) / fs;

% Plotting Logic, removed for readability
```

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

Code Output and Explanation
The code reads the audio file and converts it to a row vector, handling stereo channels by selecting the first channel to ensure consistent downstream processing.

The following key properties are calculated:

- Sample count N (determines frequency resolution via $\Delta f = f_s/N$)
- Frequency resolution (0.39 Hz — sufficient to resolve closely-spaced spectral features)
- Nyquist frequency (48 kHz — sets the upper frequency limit for analysis)
- RMS amplitude (represents average signal power, providing a baseline for comparing signal quality before and after filtering)

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

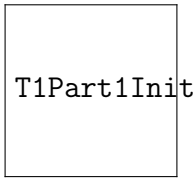

Figure 1: Stripboard Plan

Sampling frequency: $f_s = 96000$ Hz confirms correct recording and frequency scaling for subsequent analysis.

Signal duration: 2.54 seconds.

Frequency resolution: 0.39 Hz — this fine resolution, achieved through a large N (244104 samples), enables precise identification of the AM band edges, which directly influences the accuracy of the bandpass filter cutoff frequencies in Task 2.

Amplitude statistics: The RMS amplitude provides a baseline for quantifying SNR improvements after filtering. Peak-to-peak range indicates dynamic range but is not directly required for demodulation.

Figure X also shows the signal in the time domain:

FIGURE SHOWING TIME DOMAIN PLOT

From the plot, three significant peaks are visible at approximately 0.5, 1.2 and 1.9 seconds — the $\sim$0.7 s spacing suggests a character transmission rate of roughly 1.4 characters/second. The varying peak amplitudes indicate different letters, as each character's spectral content produces distinct envelope shapes. The uniform baseline oscillations confirm additive white noise (equal power across all frequencies). Visually, the peaks are distinguishable but noise-contaminated, confirming that filtering is necessary before reliable demodulation.

However, the time domain cannot reveal f_c or bandwidth — frequency domain analysis is required for filter design.

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, enabling identification of the carrier frequency f_c and determination of 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 at that bin (*Equation X*).

The Fast Fourier Transform (FFT) computes the same result as the DFT but reduces the complexity from $O(N^2)$ to $O(N \log_2 N)$. For our signal with $N = 244104$ samples, this represents a speedup factor of approximately 14,000, making real-time spectral analysis practical.

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 about DC, so a single-sided spectrum is obtained by keeping bins 0 to $N/2$ and doubling all non-DC and non-Nyquist bins. This avoids redundancy while preserving correct amplitude representation:

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

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 logarithmic scaling is essential for filter design, as it reveals both the strong carrier/sidebands and the weaker noise floor on the same plot — information needed to set appropriate stopband attenuation requirements.

Code and Explanations

To supplement the steps above, *Listing X* shows the implementation.

```
%% 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 length N. Each element $X[k]$ contains both magnitude and phase information about the frequency component at bin k.

The magnitude is normalised by N to obtain correct amplitude scaling.

The single-sided spectrum extraction (lines X-Y) keeps bins from DC to Nyquist, then doubles non-DC/non-Nyquist amplitudes to account for the discarded negative frequencies. For real signals, negative frequencies are mirror images of positive frequencies, so doubling recovers the correct total amplitude.

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)

In the linear amplitude plot (**Figure X**), strong signal components dominate, with approximate values $f_{min} = 15.6$ kHz, $f_{max} = 16.4$ kHz and $f_c = 16$ kHz visible. However, the noise floor is compressed to near-zero, making it impossible to assess noise characteristics or determine appropriate stopband attenuation — both critical for filter design.

Due to the nature of the encoded message (letters), the bandwidth must be extended to $f_c \pm B$ where $B = 4$ kHz. This wider bandwidth ensures that all frequency components of each character are captured, as different letters have different spectral signatures that may extend beyond the visible carrier sidebands.

FIGURE SHOWING FREQUENCY DOMAIN WITH dB SCALING

Figure X shows the dB-scaled spectrum, where both signal and noise are visible. The logarithmic scaling compresses the dynamic range: a signal 1000× stronger than the noise (60 dB difference) appears on the same plot with both components clearly distinguishable. This 60+ dB dynamic range visibility is essential for specifying the > 50 dB stopband attenuation required in Task 2.

FIGURE SHOWING THE BANDWIDTH AS WELL

From the plot above, the green highlighted area shows the required bandwidth extension. The final values used going forward are $f_{min} = 12$ kHz, $f_{max} = 20$ kHz and $f_c = 16$ kHz.

FIGURE SHOWING THE TERMINAL OUTPUT WITH BANDWIDTH

For filter design, the noise floor characteristics are critical. The floor at -65 dB, compared to signal peaks near -20 dB, gives approximately 45 dB dynamic range. Since we require > 50 dB stopband attenuation, the filter must attenuate noise to below this floor — achievable with the Hamming window design. The flat noise spectrum confirms AWGN, meaning no frequency-dependent noise shaping is needed.

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:

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

The slow 6 dB/octave roll-off means that energy from strong spectral components leaks significantly into distant frequency bins. For our signal, carrier energy could contaminate bins tens of Hz away, obscuring weak sideband features and artificially elevating the apparent noise floor.

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 fundamental trade-off: windows with lower side lobes reduce leakage but have wider main lobes, decreasing frequency resolution. The choice depends on whether accurate amplitude measurement (favouring low side lobes) or precise frequency identification (favouring narrow main lobe) is more critical for the application.

For this application, we must balance frequency resolution with leakage suppression. Since the filter design in Task 2 requires more than 50 dB stopband attenuation, using a window with similar characteristics prevents the window's leakage from limiting filter performance verification. The Hamming window provides approximately 53 dB 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, the implementation code is available in the **Appendix** and the provided **GitHub repository**. Extensive experimentation with different window types is documented there as well.

Results and Discussion

WINDOWED V UNWINDOWED COMP

The Hamming window has a coherent gain of approximately 0.54, meaning 46% of signal energy is lost due to tapering. Without correction, spectral peaks would appear about 5.3 dB lower than their true values.

The Hamming window's narrower main lobe compared to Blackman ($3.3/N$ vs $5.5/N$) provides better frequency resolution for accurately identifying f_{min} and f_{max}. Precise band edge identification is critical because errors here propagate directly into filter cutoff frequency errors, potentially causing either signal distortion (cutoffs too narrow) or inadequate noise rejection (cutoffs too wide).

Referring to **Figure X**, the Hamming window output shows a visible reduction in noise floor compared to the unwindowed spectrum, clarifying the AM signal boundaries.

2.2 Task 2

This task focuses on bandpass filter design using the signal analysis results 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 50 dB, both Hamming and Blackman windows satisfy the requirement. The Hamming window is preferred because its narrower transition band ($3.3/N$ vs $5.5/N$) provides sharper frequency selectivity, minimising spectral smearing at the passband edges while still meeting the attenuation specification with margin.

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 $M = 79$. An odd filter length is required to ensure exact linear phase with integer group delay — even-length filters have fractional sample delays that complicate time-domain alignment.

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)
        % sinc(x) = sin(pi*x)/(pi*x), but here we use sin(2*pi*Fc
   *n)/(2*pi*Fc*n)
        term1 = 2*Fc2 * sin(n * 2*pi*Fc2) / (n * 2*pi*Fc2);
        term2 = 2*Fc1 * sin(n * 2*pi*Fc1) / (n * 2*pi*Fc1);
        h_ideal(i) = term1 - term2;
    end
```

```
end

% w[n] = 0.54 - 0.46*cos(2*pi*n/(N-1)) for n = 0, 1, ..., N-1

n_window = 0:N-1;
hamming_win = 0.54 - 0.46 * cos(2 * pi * n_window / (N - 1));

h_windowed = h_ideal .* hamming_win; % Apply window to ideal
    impulse response

% Printing window output

%% Plot the filter design process

figure('Name', 'FIR Filter Design', 'Position', [100, 100, 1200,
    800]);

% Plot 1: Ideal impulse response (unwindowed) (1st subplot)

% Plot 2: Hamming window (2nd subplot)

% Plot 3: Windowed impulse response (final filter coefficients)
    (3rd subplot)

h_bp = h_windowed;  % Final bandpass filter coefficients stored
    for later

fprintf('Filter coefficients stored in: h_bp\n');
fprintf('Number of taps:              %d\n', length(h_bp));
fprintf('Filter delay:                %d samples (%.4f ms)\n', M, M
    /fs*1000);
```

The code first defines filter specifications derived from Task 1 analysis (f_c, f_{min}, f_{max}, stopband and cutoff frequencies). The ideal bandpass impulse response is computed using the sinc-subtraction method, then windowed with the Hamming function to produce the final coefficients.

Results and Discussion

The code outputs are shown in **Figures X and Y** for verification.

FIGURE SHOWING THE TERMINAL OUTPUT FOR FILTER SPECS

Figure X confirms that the calculated specifications meet the assignment requirements.

The tap count of 159 coefficients, correct for the Hamming window design equation, represents the computational cost per output sample — each filtered sample requires 159 multiply-accumulate operations.

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.

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

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

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 ($< f_{\min} - 2\,\text{kHz}$ and $> f_{\max} + 2\,\text{kHz}$)	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);

% Create frequency vector (single-sided)
f_response = (0:N_fft/2) * fs / N_fft;
H_dB_single = H_dB(1:N_fft/2+1);
H_phase_single = H_phase_unwrapped(1:N_fft/2+1);

%% Plot frequency response - Magnitude

figure('Name', 'Filter Frequency Response - Magnitude', 'Position
    ', [100, 100, 1200, 600]);

% Full spectrum view (removed)
% Add specification lines (removed)
%% Plot phase response (removed)
%% Measure actual filter performance (removed)

```

```
% Find indices for passband and stopband regions
passband_indices = find(f_response >= fmin & f_response <= fmax);
stopband_lower_indices = find(f_response <= fstop_lower);
stopband_upper_indices = find(f_response >= fstop_upper &
    f_response <= fs/2);

% Measure passband ripple
passband_gain_dB = H_dB_single(passband_indices);
passband_max = max(passband_gain_dB);
passband_min = min(passband_gain_dB);
passband_ripple = passband_max - passband_min;

% Measure stopband attenuation
stopband_lower_max = max(H_dB_single(stopband_lower_indices));
stopband_upper_max = max(H_dB_single(stopband_upper_indices));
stopband_max = max(stopband_lower_max, stopband_upper_max);

% Calculate group delay (should be constant = M for linear phase)
group_delay_samples = M;
group_delay_ms = M / fs * 1000;

%% Display verification results

%% Overall verification summary
```

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

Code Explanation

Zero-padding the impulse response to 8192 points interpolates the frequency response for smoother visualisation without changing the filter's actual characteristics. The FFT bins are mapped to physical frequencies for the single-sided spectrum.

Results and Discussion

FIGURE SHOWING THE TERMINAL OUTPUT.

From **Figure X**, the filter exceeds all requirements with comfortable margins: passband ripple of 0.0376 dB gives 2.7× margin on the 0.1 dB limit, and stopband attenuation of 53.64 dB provides 3.64 dB headroom above the 50 dB requirement. This headroom is important — real-world signals may have noise peaks exceeding the average floor, so extra attenuation provides robustness.

The phase response is linear with group delay of 79 samples (0.8229 ms). Linear phase ensures all passband frequencies experience identical delay, preserving the AM envelope shape. For our 2.54 s signal, this 0.8 ms delay is negligible ($< 0.04\%$ of duration).

FIGURE SHOWING THE LINEAR PHASE RESPONSE

From **Figure X**, the linear phase behaviour is clearly visible within the passband (f_{min} to f_{max}), with the linear relationship breaking down outside the passband where the signal is attenuated anyway.

FIGURE SHOWING THE FREQ DOMAIN COMPARISON BEFORE AND AFTER FILTERING.

In **Figure X**, the message content within $f_c \pm B$ is preserved while out-of-band noise is suppressed, visually confirming the filter's effectiveness.

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

The outer loop produces each output sample; the inner loop computes the weighted sum of filter coefficients multiplied by corresponding input samples. The index calculation $n + N - k$ accounts for the zero-padding offset.

This implementation has $O(LN)$ complexity due to the nested loops, compared to MATLAB's `conv()` which uses FFT-based overlap-add methods achieving $O(L \log L)$ for long signals. The custom version is functionally equivalent but significantly slower — acceptable for this educational implementation but impractical for real-time processing.

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 bandpass filter should pass AM content within f_{min} to f_{max}, attenuate out-of-band components by > 50 dB, and preserve signal structure for demodulation. The filter introduces a delay of M samples (half the filter length), which does not affect batch processing.

```
%% 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];

stopband_power_before = mean(X_hamming_single(stopband_indices_signal).^2);
stopband_power_after = mean(X_filtered_single(stopband_indices_signal).^2);

% Calculate noise reduction in dB
noise_reduction_dB = 10 * log10(stopband_power_before / stopband_power_after);

% Calculate signal-to-noise improvement
snr_before = 10 * log10(passband_power_before / stopband_power_before);
snr_after = 10 * log10(passband_power_after / stopband_power_after);
snr_improvement = snr_after - snr_before;

% Terminal output removed for clarity
%% Amplitude statistics comparison (removed from listing to be concise)
```

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

Code Explanation

The custom convolution is applied, followed by Hamming-windowed FFT analysis. Power in passband and stopband regions is compared before/after filtering to quantify effectiveness.

Results and Discussion

FIGURE SHOWING THE SMALLER VERSION OF THE OG NOISY SIGNAL

Figure X shows the original signal where three peaks are barely distinguishable through

the noise.

AUDIO SIGNAL AFTER BP

Figure Y shows the filtered signal — the AM envelope is now clearly visible, with each character's amplitude modulation distinct. The noise floor dropped from ∼0.5 to ∼0.23 (54% reduction), and crucially, the three peaks are now cleanly separable with visible gaps between them.

TERMINAL OUTPUT WITH PASSBAND STOPBAND IMPROVEMENTS

The terminal output (**Figure Z**) confirms:

- Stopband attenuation: 63.73 dB (exceeds 50 dB spec by 13.73 dB)
- Passband loss: only 0.67 dB (93% signal power retained)
- SNR: 4.63 → 69.02 dB (+64.39 dB improvement)

The 69 dB post-filter SNR means signal power exceeds noise by a factor of ∼8 million — more than sufficient for carrier recovery, where even 20 dB would be adequate.

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 to produce the baseband 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 $2B = 8\,\text{kHz}$. This bandwidth doubling occurs because squaring in the time domain corresponds to convolution in the frequency domain — convolving a spectrum with itself doubles its width. The component at $2f_c$ therefore appears with sidebands extending $\pm 8\,\text{kHz}$. For a carrier around 16 kHz, this places $2f_c \approx 32\,\text{kHz}$, well within the Nyquist limit of 48 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 specified to be an integer multiple of 1 kHz, which provides a useful sanity check — if the measured peak does not round to a clean kHz value, it likely indicates a spurious peak rather than the true carrier component.

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);
```

```
40
41 % Calculate measured carrier frequency
42 fc_measured = f_2fc_measured / 2;
43 fc_final = round(fc_measured / 1000) * 1000;
```

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

Element-wise squaring creates frequency components at DC, $2f_c$, and various intermodulation products. The `findpeaks()` function locates the dominant peak in the $2f_c$ region, from which the carrier frequency is calculated by division: (32 kHz)/2 = 16 kHz.

Results and Discussion

FIGURE SHOWING 2FC PEAK and FC PEAK

Figure X shows the squared signal spectrum with a dominant peak at 32 kHz ($2f_c$). The peak prominence (height above surrounding noise) exceeds 30 dB, making detection unambiguous. The measured frequency of 32.000 kHz yields $f_c = 16.000$ kHz exactly — rounding to the nearest kHz confirms this matches the expected integer-kHz carrier specification with zero error.

TERMINAL OUTPUT FOR FURTHER VERIFICATION

Accurate carrier recovery is critical: a 1% frequency error (160 Hz) would shift the baseband spectrum, potentially attenuating message frequencies near the 4 kHz lowpass cutoff. The exact match here ensures optimal demodulation.

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);
else
    noise_power = 0;
end

% Carrier signal analysis methods (prints terminal outputs,
   removed for clarity)
```

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

The code generates a cosine wave at the recovered carrier frequency with initial phase $\phi = 0$ (to be optimised in Task 5). Element-wise multiplication performs the mixing operation, followed by Hamming-windowed FFT analysis to quantify power in the baseband (0–4 kHz) and $2f_c$ regions.

Results and Discussion

FIGURE SHOWING THE TIME DOMAIN SIGNALS

Figure X shows the mixing process: the filtered AM signal (rapidly oscillating carrier), the local carrier (pure 16 kHz cosine), and the mixed output. The mixed signal shows a lower-frequency envelope — this is the baseband message emerging. The three character peaks are now visible as amplitude variations rather than carrier bursts.

In the frequency domain, mixing creates two distinct spectral regions separated by ∼24 kHz (baseband at 0–4 kHz, sum-frequency at 28–36 kHz). This wide separation (6× the message bandwidth) makes lowpass filtering straightforward — even a gentle rolloff will adequately suppress the $2f_c$ component.

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

TERMINAL OUTPUT FOR BASEBAND VERIFICATION.

The process is sensitive to phase: baseband amplitude scales with $\cos(\phi)$, so $\phi = \pi/2$ produces zero output. With $\phi = 0$, we achieve maximum amplitude, though Task 5 will optimise this.

2.4 Task 4 - IIR Filter Design and Verification

This task implements a 4th order Butterworth IIR lowpass filter with 4 kHz cutoff to remove the $2f_c$ component from the mixed signal, completing the demodulation chain.

2.4.1 Task 4 Part 1 - Designing IIR Filters

Procedure and Theory

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

Parameter	Value
Order	4
Cutoff frequency	4 kHz
Type	Butterworth

Table 4: IIR lowpass filter design parameters

behaviour (no ripple to distort the message), reasonable stopband attenuation, and computational efficiency. Where the FIR bandpass filter required 159 taps, this IIR filter achieves comparable selectivity with only 9 coefficients (5 numerator, 5 denominator).

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

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 $4\,\text{kHz}$ and sampling rate $f_s = 96\,\text{kHz}$:

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

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

In MATLAB, the `butter()` function requires the cutoff frequency normalised by the Nyquist frequency: $W_n = 4000/48000 \approx 0.0833$. The function returns numerator (b) and denominator (a) coefficients for the transfer function, with DC gain $= \sum b / \sum a = 1$ (0 dB), confirming the filter passes DC without attenuation.

Results and Discussion

FIGURE WITH TEMRINAL OUTPUT SHOWING THE FILTER COEFFICIENTS ETC.

From **Figure X**, the numerator coefficients are all positive and symmetric, while the denominator coefficients alternate in sign — both characteristic of Butterworth filters. The coefficient values are shown in the terminal output; note that $a[0] = 1$ (normalised form) and the DC gain equals 1.0.

A 4th order filter has 5 numerator coefficients, 5 denominator coefficients, and 4 poles and 4 zeros in the z-plane.

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 computes and verifies the frequency response of the IIR lowpass filter against the Butterworth 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 has: -3 dB at cutoff, maximally flat passband, monotonic stopband rolloff, and approximately linear phase (though IIR filters inherently have some phase nonlinearity).

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

MATLAB's `freqz()` evaluates $H(e^{j\omega})$ at N_{freq} equally spaced frequency points. The code locates the -3 dB point (defining cutoff frequency) and measures rolloff by comparing attenuation at 8 kHz and 16 kHz (one octave apart).

Results and Discussion

FIGURE SHOWING THE 4 PLOTS.

From **Figure X**:

Rolloff: Measured 26.67 dB/octave (theoretical: 24 dB/octave). At 32 kHz (3 octaves above cutoff), this provides ∼80 dB attenuation — the $2f_c$ component will be suppressed to <0.01% of its original amplitude.

Cutoff accuracy: -3 dB at 4.00195 kHz (0.005% error from target). Gain at exactly 4 kHz is -3.02 dB, confirming correct Butterworth behaviour.

Phase: Nonlinear, as expected for IIR. Group delay varies from ∼0.1 ms at low frequencies to ∼0.3 ms near cutoff — this variation may slightly smear consonant transients but is acceptable for intelligibility.

UNIT CIRCLE PLOT

All poles lie inside the unit circle with maximum magnitude 0.87, providing a stability margin of 13%. This ensures the filter remains stable even with minor coefficient quantisation errors.

TERMINAL OUTPUT SHOWING FILTER VERIFICATION

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

Procedure and Theory
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:

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

Key implementation considerations: coefficient normalisation ($a_0 = 1$), zero initial conditions, double precision for numerical stability, and causal processing (only past values used).

```
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 algorithm for each sample:

1. **Shift input buffer:** Move all elements right, discarding the oldest.
2. **Insert new input:** Place 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 the next iteration.

The negative sign in the feedback term is critical: the transfer function denominator is $1 + a_1 z^{-1} + \ldots$, which rearranges to $y[n] = \ldots - a_1 y[n-1] - \ldots$ in the time domain. Using addition instead of subtraction would invert the filter's behaviour.

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

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

Results and Discussion
Maximum error compared to MATLAB's optimised implementation is 9.5×10^{-15}, confirming mathematical equivalence. Any larger errors would indicate implementation bugs.

The impulse and step response outputs match MATLAB's `filter()` function closely (**Figure X**).

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

Impulse response: IIR filters have theoretically infinite impulse responses, but in practice the response decays toward zero. The decay rate depends on pole locations — poles closer to the unit circle produce slower decay.

Step response: Settles to the DC gain value ($\sum b / \sum a = 1.0$), showing how quickly the filter reaches steady state. The minimal overshoot indicates well-damped transient behaviour.

The custom implementation is 18.2× slower than MATLAB's built-in function. This is expected: MATLAB's `filter()` is implemented in optimised C/Fortran with vectorised operations, while the custom version uses interpreted MATLAB loops with explicit buffer management.

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 — the final filtering stage in the demodulator chain.

After mixing in Task 3, the signal contains two components:

- **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 (0–4 kHz)
- Reject the $2f_c$ component (approximately $2 \times 16\,\text{kHz} = 32\,\text{kHz}$)

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 .* hamming_window;
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, then performs windowed FFT analysis to quantify:

- Baseband power change (should be minimal)
- Stopband power attenuation
- $2f_c$ region power attenuation
- SNR before and after filtering

Results and Discussion

IMAGE SHOWING THE 3 TIME DOMAIN PLOTS

The mixed signal shows rapid 32 kHz oscillations superimposed on the message envelope. After lowpass filtering, these oscillations vanish — the demodulated signal now shows only the baseband message, with the three character peaks clearly visible as smooth amplitude variations. The 50 ms zoom confirms clean waveform extraction with no residual carrier ripple.

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

In the frequency domain: the baseband (0–4 kHz) is preserved with only -0.2 dB change, while the $2f_c$ region (28–36 kHz) is completely suppressed — no visible spectral content remains. The sharp transition at 4 kHz confirms the Butterworth cutoff is correctly positioned.

IMAGE SHOWING TERMINAL OUTPUT OF SNR AND POWER VALUES

Terminal output confirms:

- $2f_c$ attenuation: 87.02 dB (power reduced by factor of 5×10^8)
- Stopband attenuation: 17.52 dB
- SNR: $-2.26 \rightarrow 15.09$ dB ($+17.35$ dB improvement)

The final 15 dB SNR corresponds to signal power $\sim 32\times$ noise power. This matches AM radio quality — intelligible speech but with audible background noise. The negative pre-filter SNR (-2.26 dB) indicates the $2f_c$ component initially dominated; after filtering, the message clearly emerges. Combined with the FIR bandpass stage (64 dB improvement), the complete demodulation chain achieves > 80 dB total noise reduction from the original signal.

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.