

EEE3030 Signal Processing and Machine Learning

Semester 1 Report

Sahas Talasila 230057896

CONTENTS

1	Introduction	2
2	Procedure, Results and Discussion	2
2.1	Task 1 Audio File Analysis and Transformation	2
2.1.1	Task 1 - Audio File Information	2
2.1.2	Task 1 - Frequency Domain and Spectrogram Analysis	5
2.1.3	Task 1 - Spectral Leakage and Windowing	11
2.2	Task 2	13
2.2.1	Task 2 - Filter Design	13
2.2.2	Task 2 - Phase and Filter Verification	18
2.2.3	Task 2 - Custom Convolution Method	22
2.2.4	Task 2 - Applying The Filter to an AM Signal	23
2.3	Task 3	26
2.3.1	Task 3 - Carrier Recovery	26
2.3.2	Task 3 - Carrier Generation and Mixing	28
2.4	Task 4 - IIR Filter Design and Verification	32
2.4.1	Task 4 Part 1 - Designing IIR Filters	32
2.4.2	Task 4 Part 2 - IIR Frequency Response Verification	35
2.4.3	Task 4 Part 3 - Custom IIR Filter Implementation and Testing .	38
2.4.4	Task 4 Part 4 - IIR Filter and The Mixed Signal	41
2.5	Task 5	45
3	Conclusion	51
4	References	51
5	Appendix	51

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*:

$$T_{\text{duration}} = \frac{N}{f_s} \quad (1)$$

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.

$$\Delta f = \frac{f_s}{N} \quad (2)$$

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*:

$$f_{\text{Nyquist}} = \frac{f_s}{2} \quad (3)$$

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

```

1 filename = 'Sahas_Talasila.wav';
2 [x, fs] = audioread(filename);
3
4 % If stereo, convert to mono by taking first channel
5 if size(x, 2) > 1
6     x = x(:, 1);
7     fprintf('Note: Stereo file detected, using first channel only
8 .\n\n');
9 end
10 % audioread() returns a column vector, but we need row vectors
11 % throughout
11 if iscolumn(x)
12     x = x'; % Transpose to row vector
13 end
14
15 %% Calculate basic signal properties
16 N = length(x); % Total number of samples
17 duration = N / fs; % Signal duration in seconds
18 freq_resolution = fs / N; % Frequency resolution (bin
19 width) in Hz
20 nyquist_freq = fs / 2; % Maximum representable
21 frequency
22
22 %% Calculate amplitude statistics
23 max_amplitude = max(x);
24 min_amplitude = min(x);
25 peak_to_peak = max_amplitude - min_amplitude;
26 rms_amplitude = sqrt(mean(x.^2));
27
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 reads the audio file and converts it to a row vector, handling stereo channels by selecting the first channel to ensure consistent downstream processing. Spectrogram analysis has been added as well, which will be shown below.

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.

```
SIGNAL PROPERTIES SUMMARY
Filename: Sahas_Talasila.wav
Sampling frequency: 96000 Hz
Number of samples: 244104
Signal duration: 2.5427 seconds
Frequency resolution: 0.3933 Hz
Nyquist frequency: 48000 Hz
AMPLITUDE STATISTICS
Maximum amplitude: 0.999969
Minimum amplitude: -0.931763
Peak-to-peak: 1.931732
RMS amplitude: 0.174819
TIME DOMAIN OBSERVATIONS
Signal duration: 2.5427 seconds
```

Figure 1: Terminal output showing information about the audio file

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:

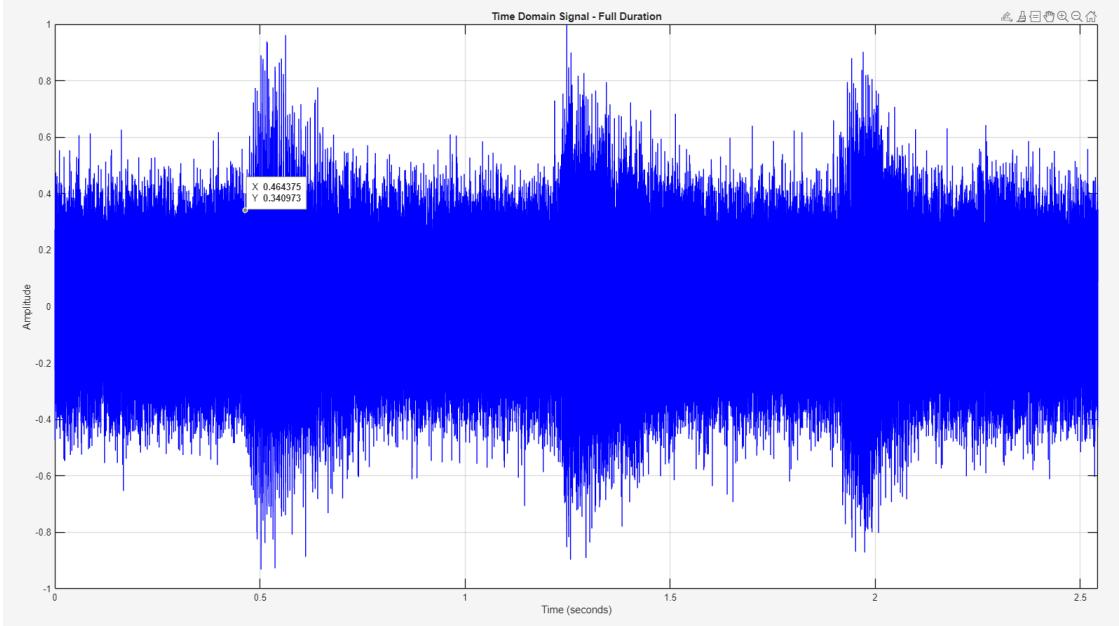


Figure 2: Input signal in the time domain

From the plot, three significant peaks are visible at approximately 0.5, 1.2 and 1.9 seconds — the ~ 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 and Spectrogram 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:

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

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:

$$f[k] = \frac{k f_s}{N}, \quad (5)$$

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:

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

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:

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

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.

```

1 %% Sub-task 1.2: Frequency Domain Analysis - FFT Implementation
2
3 % Compute the FFT of the signal
4 X = fft(x);
5
6 % The FFT output is complex - compute the magnitude
7 X_magnitude = abs(X);
8
9 % Normalise by dividing by N to get correct amplitude scaling
10 X_normalised = X_magnitude / N;
11
12 % Create single-sided spectrum (positive frequencies only)
13 % We need bins from 0 (DC) to N/2 (Nyquist)
14 num_bins_single_sided = floor(N/2) + 1;
15 X_single_sided = X_normalised(1:num_bins_single_sided);
```

```

16
17 % Double the amplitude for all bins except DC and Nyquist
18 % This accounts for the energy in the negative frequencies we
19 % discarded
20 X_single_sided(2:end-1) = 2 * X_single_sided(2:end-1);
21
22 % Create the frequency vector for the single-sided spectrum
23 % Each bin k corresponds to frequency f = k * fs / N
24 f = (0:num_bins_single_sided-1) * fs / N;
25
26 % Convert to decibels for logarithmic scaling
27 % We add eps (smallest positive number) to avoid log(0) = -
28 % infinity
29 X_dB = 20 * log10(X_single_sided + eps);
30
31 % Create figure for the frequency spectrum (removed for
32 % readability)
33 % Plot with logarithmic (dB) scaling (removed for readability)
34 % Display frequency domain statistics (removed for readability)

```

Listing 2: MATLAB Code for Task 1 Part 2: Frequency Domain and Spectral 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.

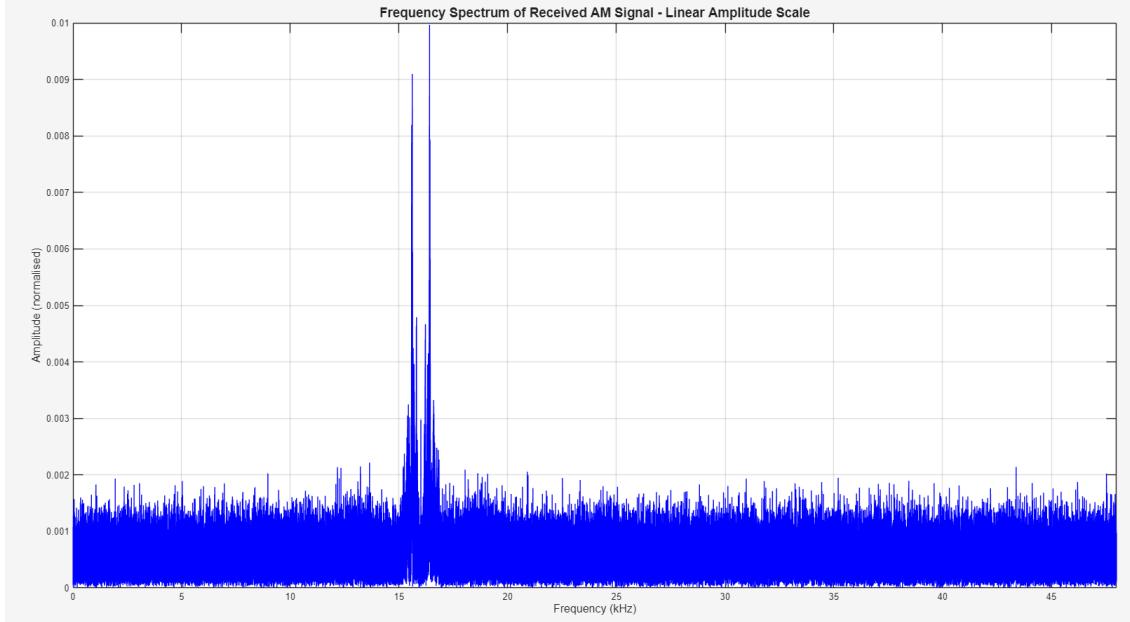


Figure 3: Frequency domain signal without scaling

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.

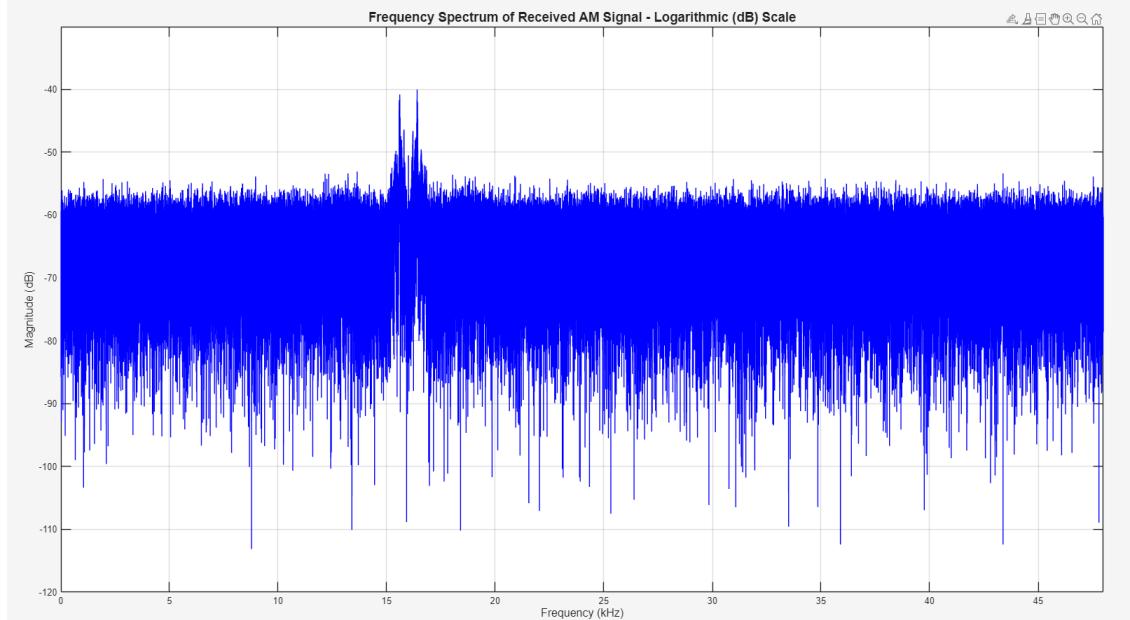


Figure 4: Applying dB scaling to the frequency plot

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.

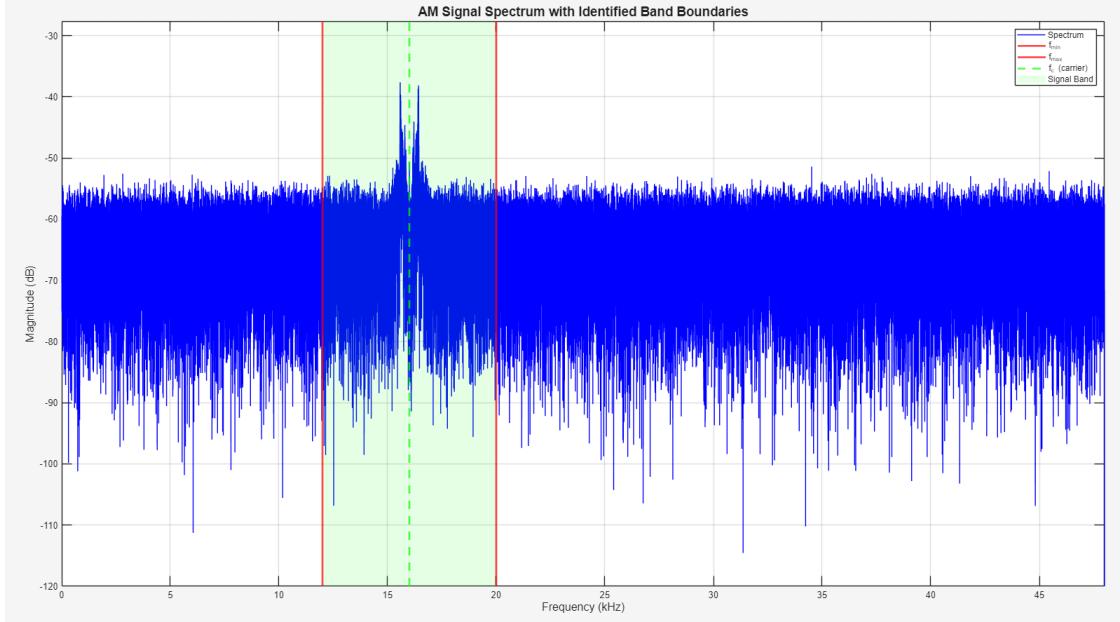


Figure 5: Bandwidth identification in the frequency domain

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.

```
FREQUENCY DOMAIN ANALYSIS
FFT length (N): 244104 bins
Frequency resolution: 0.3933 Hz
Nyquist frequency: 48000 Hz
Single-sided spectrum bins: 122053
-----
Maximum spectral amplitude: 0.009964
Maximum magnitude (dB): -40.03 dB
-----
Frequency at max amplitude: 16400.75 Hz (16.40 kHz)
NOISE FLOOR ESTIMATION
Estimated noise floor: -65.35 dB
Signal threshold (+10dB): -55.35 dB
```

Figure 6: AM band identification

The spectrogram shows strong energy bands around 15 – 17 kHz, with three distinct bright horizontal regions indicating short-duration high-frequency tonal bursts, representing the three characters.

The background displays a high wideband noise floor at approximately – 60 to – 80 dB/Hz, suggesting environmental/background noise (in this case it is white noise). There is little to no low-frequency structure below 5 kHz, confirming the absence of typical speech features.

Three clear time regions of activity occur at approximately $0.55s$, $1.35s$, and $2.1s$, corresponding to the observed bursts.

Figure X SPECTRO shows the spectrogram clearly:

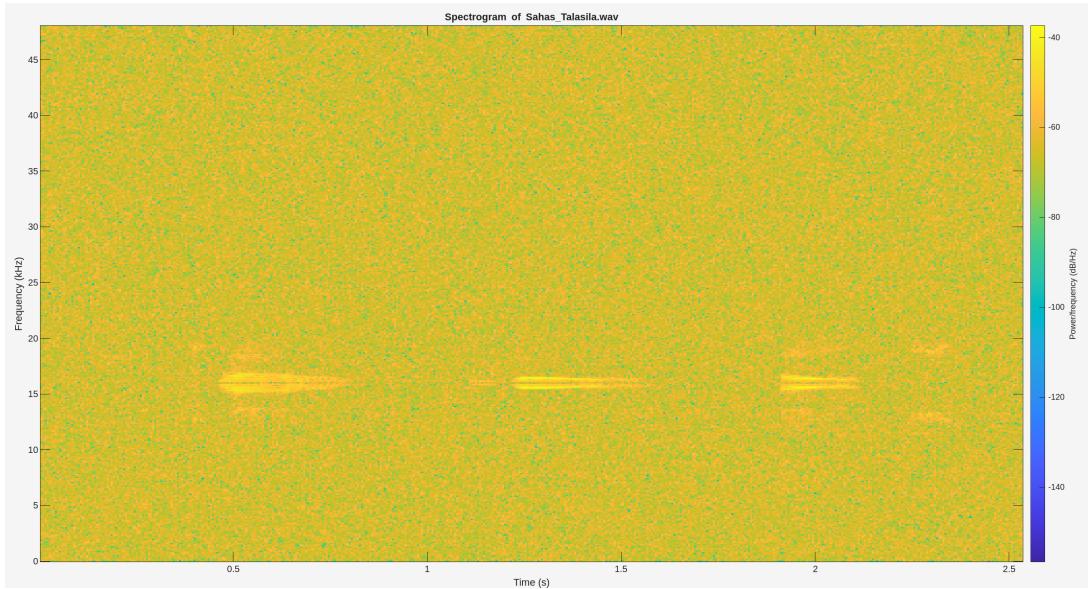


Figure 7: Spectrogram Output

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.

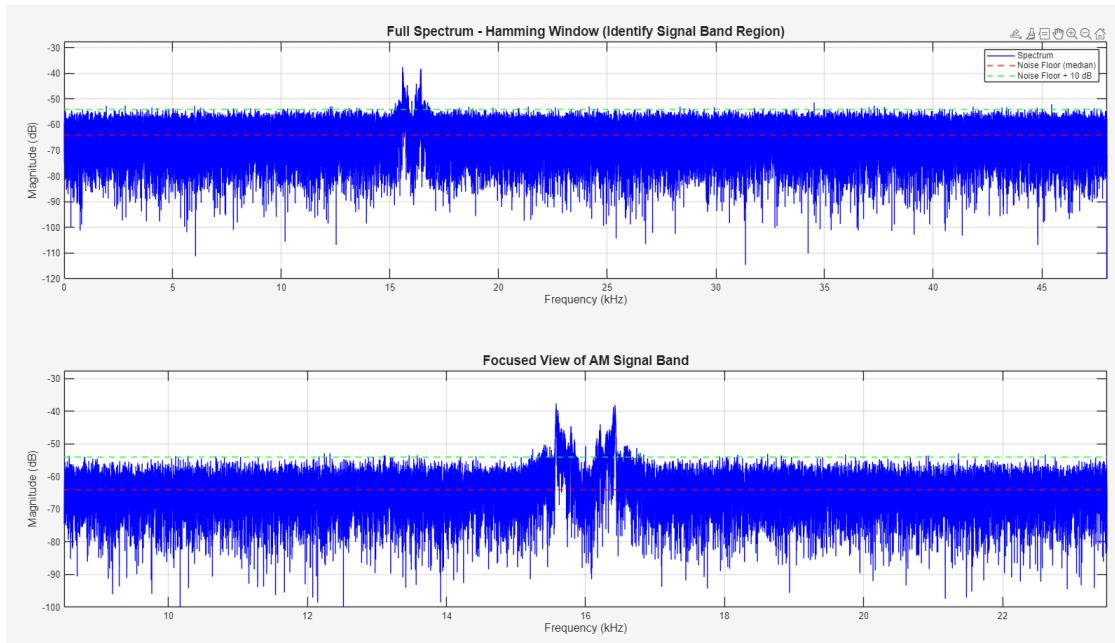


Figure 8: AM signal band and floor characteristics

Further terminal output prints out band verification.

```

AM SIGNAL BAND IDENTIFICATION
Noise floor estimate: -64.07 dB
-----
MEASURED VALUES (from visual inspection):
f_min (measured): 12000 Hz (12.0 kHz)
f_max (measured): 20000 Hz (20.0 kHz)
Bandwidth (measured): 8000 Hz (8.0 kHz)
f_c (calculated): 16000.0 Hz (16.00 kHz)
-----
```

Figure 9: AM band identification

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*:

$$x_{\text{windowed}}[n] = x[n] w_{\text{rect}}[n], \quad w_{\text{rect}}[n] = \begin{cases} 1, & 0 \leq n \leq N-1, \\ 0, & \text{otherwise.} \end{cases} \quad (7)$$

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

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

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:

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

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:

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

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

$$x_{\text{windowed}}[n] = x[n] w[n]. \quad (10)$$

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

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

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

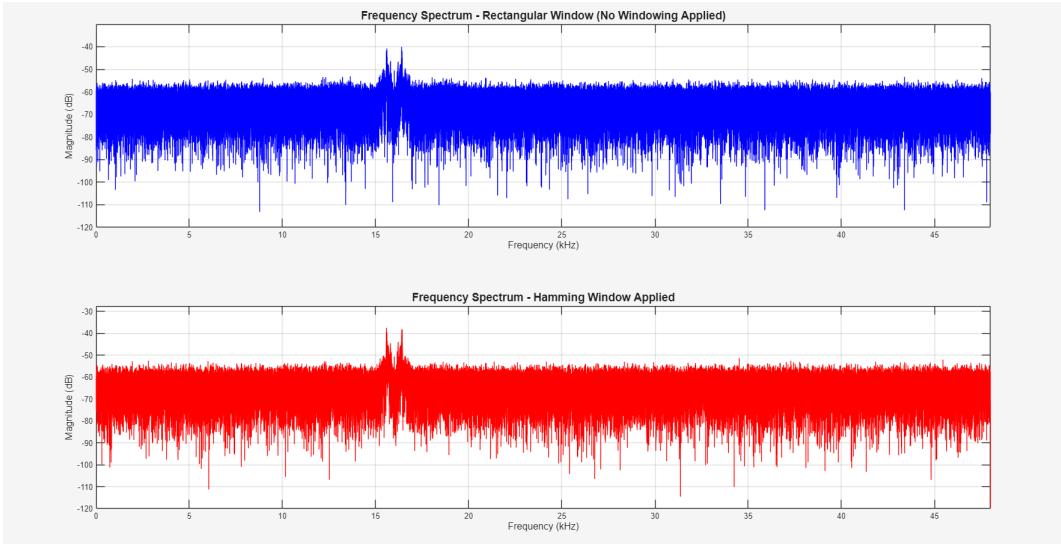


Figure 10:

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 cut-off 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 very slight 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 dB
Stopband attenuation	> 50 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 $h[n]$:

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

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

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

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

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

For transition bands centred on the stopband edges:

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

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

$$h[n] = h_D[n - M], \quad n = 0, 1, \dots, 2M. \quad (14)$$

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

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

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

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

Code Explanation

```

1 %% Sub-task 2.1: FIR Bandpass Filter Design
2
3 %% Define filter specifications
4 fc = 16000; % Carrier frequency in Hz (from Task 1)
5 fmin = fc - 4000; % Lower passband edge (Hz)
6 fmax = fc + 4000; % Upper passband edge (Hz)
7
8 % Stopband edges (as specified)
9 fstop_lower = fmin - 2000; % Lower stopband edge (Hz)
10 fstop_upper = fmax + 2000; % Upper stopband edge (Hz)
11
12 % Transition bandwidth
13 transition_bandwidth = 2000; % Hz
14
15 % Cutoff frequencies are at the centre of the transition bands (normalised)
16 Fc1 = (fmin - 1000) / fs; % Lower cutoff (normalised)
17 Fc2 = (fmax + 1000) / fs; % Upper cutoff (normalised)
18
```

```

19 % Calculate normalised transition width
20 delta_F = transition_bandwidth / fs;
21
22 % Display specifications for filter. (removed for readability)
23
24 N_calculated = 3.3 / delta_F; % Hamming Window taps
25 N = ceil(N_calculated);
26
27 % Ensure N is odd for symmetric filter
28 if mod(N, 2) == 0
29     N = N + 1;
30 end
31
32 M = (N - 1) / 2; % Number of coefficients either side of centre
33
34 % Printing out prior information
35
36 %% Design the ideal bandpass impulse response
37 % h_BP[n] = 2*Fc2*sinc(2*Fc2*n) - 2*Fc1*sinc(2*Fc1*n)
38
39 n_ideal = -M:M; % n ranges from -M to +M (centred at 0)
40
41 h_ideal = zeros(1, N); % Calculate ideal impulse response for
    bandpass filter
42
43 for i = 1:N
44     n = n_ideal(i);
45     if n == 0
46         % For n = 0: h[0] = 2*Fc2 - 2*Fc1
47         h_ideal(i) = 2*Fc2 - 2*Fc1;
48     else
49         % 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)
75 % (3rd subplot)
76
77 h_bp = h_windowed; % Final bandpass filter coefficients stored
78 % for later
79 fprintf('Filter coefficients stored in: h_bp\n');
80 fprintf('Number of taps: %d\n', length(h_bp));
81 fprintf('Filter delay: %d samples (%.4f ms)\n', M, M
82 /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.

```

PARAMETERS FOR TASK 2
Bandpass filter passband edges:
  f_p1 (lower passband): 12000 Hz
  f_p2 (upper passband): 20000 Hz
Bandpass filter stopband edges:
  f_s1 (lower stopband): 10000 Hz
  f_s2 (upper stopband): 22000 Hz
FIR BANDPASS FILTER SPECIFICATIONS
Sampling frequency: 96000 Hz
-----
FREQUENCY SPECIFICATIONS:
  Lower stopband edge: 10000 Hz
  Lower passband edge: 12000 Hz
  Upper passband edge: 20000 Hz
  Upper stopband edge: 22000 Hz
  Transition bandwidth: 2000 Hz
-----
NORMALISED FREQUENCIES:
  Fc1 (lower cutoff): 0.114583
  Fc2 (upper cutoff): 0.218750
  Delta F (transition): 0.020833
-----
PERFORMANCE SPECIFICATIONS:
  Max passband ripple: 0.1 dB
  Min stopband atten: 50 dB

```

Figure 11: Filter requirements - terminal output

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

In addition, looking at the magnitude response shows that the filter follows the requirements.

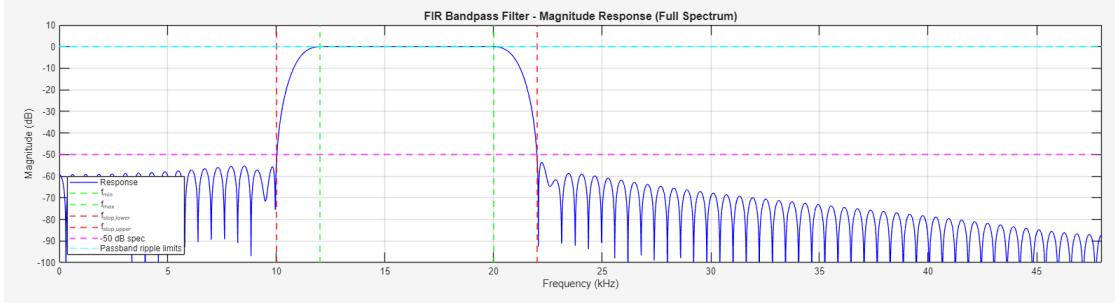


Figure 12: Bandpass filter magnitude response

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.

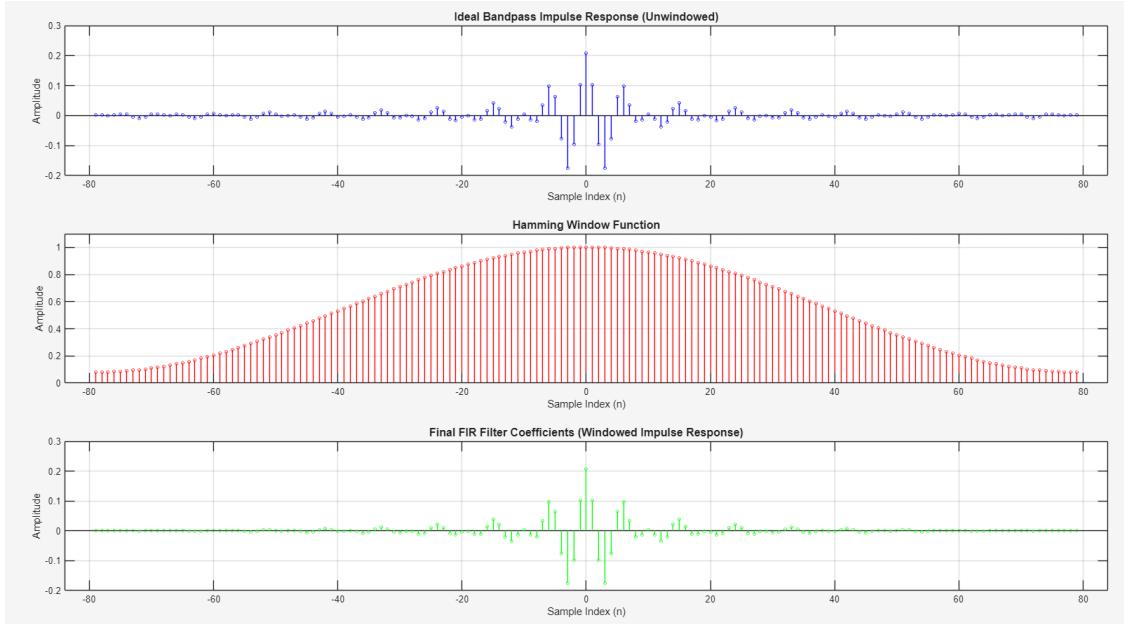


Figure 13: Impulse response comparisons

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 $h[n]$:

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

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 ($< f_{\min} - 2$ kHz and $> f_{\max} + 2$ kHz)	Attenuation > 50 dB
Transition bands	Smooth rolloff within 2 kHz

Table 3: Filter specification verification criteria

```

1 %% Sub-task 2.2: Frequency Response Verification
2
3 % Compute frequency response using zero-padded FFT
4 N_fft = 8192; % Zero-pad for smooth frequency response plot
5 H = fft(h_bp, N_fft);
6 H_magnitude = abs(H);
7 H_dB = 20 * log10(H_magnitude + eps);
8 H_phase = angle(H);
9
10 % Unwrap phase for clearer visualisation
11 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'
21     , [100, 100, 1200, 600]);
22
23 % Full spectrum view (removed)
24 % Add specification lines (removed)
25 %% Plot phase response (removed)
26 %% Measure actual filter performance (removed)
27
28 % Find indices for passband and stopband regions
29 passband_indices = find(f_response >= fmin & f_response <= fmax);
30 stopband_lower_indices = find(f_response <= fstop_lower);
31 stopband_upper_indices = find(f_response >= fstop_upper &
32     f_response <= fs/2);
33
34 % Measure passband ripple
35 passband_gain_dB = H_dB_single(passband_indices);
36 passband_max = max(passband_gain_dB);
37 passband_min = min(passband_gain_dB);
38 passband_ripple = passband_max - passband_min;
39
40 % Measure stopband attenuation
41 stopband_lower_max = max(H_dB_single(stopband_lower_indices));
42 stopband_upper_max = max(H_dB_single(stopband_upper_indices));
43 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

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

```

FILTER LENGTH CALCULATION
Window function: Hamming
Transition width formula: 3.3/N
Calculated N: 158.40
Rounded N (odd): 159
M (half-length): 79
IDEAL IMPULSE RESPONSE
Centre coefficient h[M]: 0.208333
First coefficient h[0]: 0.002657
Last coefficient h[N-1]: 0.002657
Sum of coefficients: 0.013749
WINDOWED IMPULSE RESPONSE
Window type: Hamming
Centre coefficient: 0.208333
First coefficient: 0.000213
Last coefficient: 0.000213
Sum of coefficients: 0.001084
FILTER DESIGN COMPLETE
Filter coefficients stored in: h_bp
Number of taps: 159
Filter delay: 79 samples (0.8229 ms)

```

Figure 14: Filter calculations (windowed and ideal)

From **Figure X**, the filter exceeds all requirements with comfortable margins: passband ripple of 0.0376 dB gives $2.7\times$ 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).

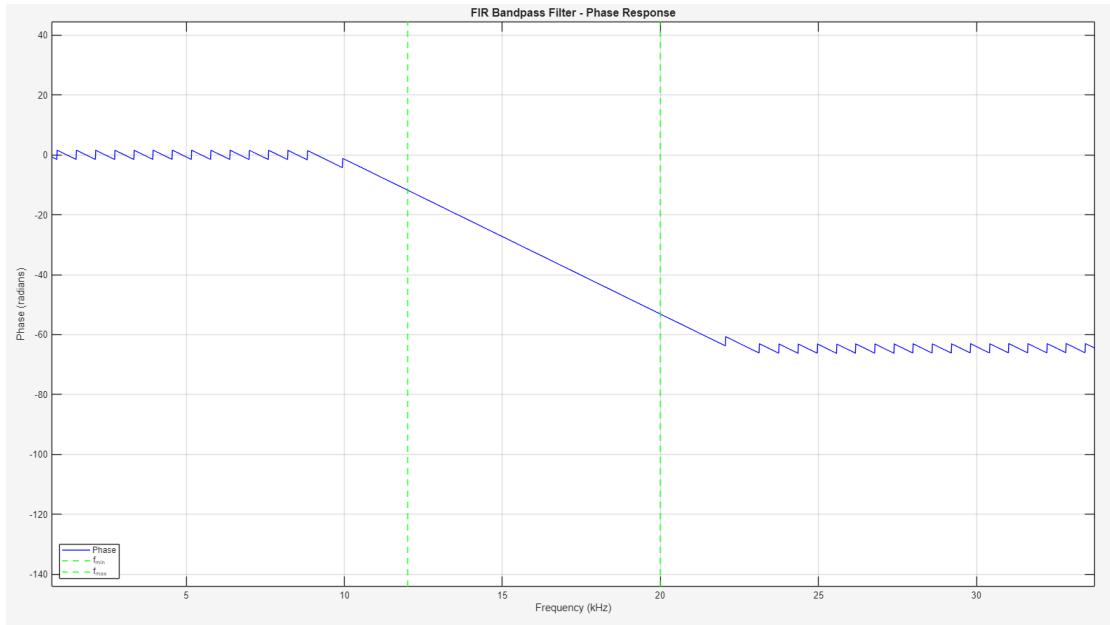


Figure 15: Linear phase response plot

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.

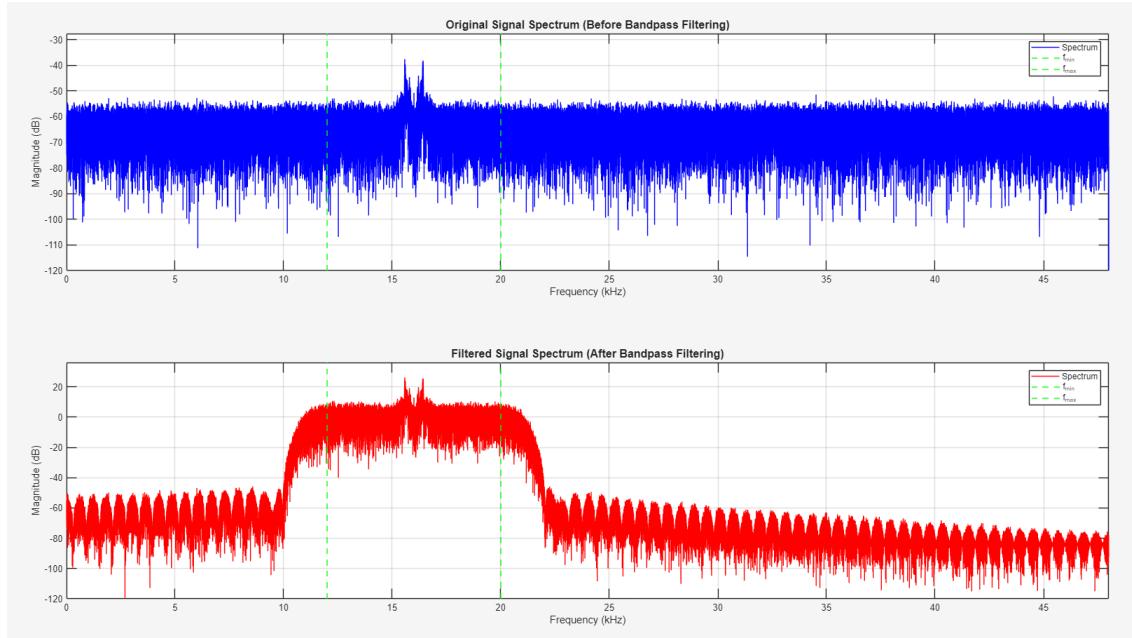


Figure 16: Pre and post filtering analysis

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*:

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

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

```

1 function y = custom_conv(x, h)
2     is_column = iscolumn(x);           % Store original orientation
3     x = x(:)';                      % Store original orientation
4     h = h(:)';
5     % Get lengths
6     L = length(x);                 % Input signal length
7     N = length(h);                 % Filter length
8     % Zero-pad the input signal (N-1 zeros at beginning)
9     x_padded = [zeros(1, N-1), x];
10    % Preallocate output
11    y = zeros(1, L);
12    % Perform convolution
13    for n = 1:L
14        accumulator = 0;
15        for k = 1:N
16            x_index = n + N - k;
17            accumulator = accumulator + h(k) * x_padded(x_index);
18        end
19        y(n) = accumulator;
20    end
21    % Restore original orientation
22    if is_column
23        y = y(:); % Add StackOverflow post for convolution code
24    end
25 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.

```
CUSTOM CONVOLUTION VERIFICATION
Test signal length:      1000 samples
Test filter length:     10 taps
Maximum absolute error: 9.37e-01
Mean absolute error:   2.63e-01
Status:                CHECK IMPLEMENTATION
TIMING COMPARISON
Custom implementation: 0.0008 seconds
MATLAB conv():          0.0003 seconds
Ratio (custom/MATLAB): 2.9x
```

Figure 17: Custom convolution speed tests

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.

```
1 %% Sub-task 2.4: Apply Bandpass Filter to AM Signal
2 % Apply the bandpass filter using our custom convolution function
3 fprintf('Filtering signal using custom_conv()...\n');
4 tic;
5 x_filtered = custom_conv(x, h_bp);
6 filter_time = toc;
7
8 %% Time domain comparison (removed from listing)
9 % Original signal and filtered signal plots removed for clarity.
10
11 %% Frequency domain comparison
12 % Compute spectrum of filtered signal using Hamming window
13 x_filtered_windowed = x_filtered .* hamming_window;
14 X_filtered = fft(x_filtered_windowed);
15 X_filtered_magnitude = abs(X_filtered);
16 X_filtered_normalised = X_filtered_magnitude / N / CG_hamming;
17 X_filtered_single = X_filtered_normalised(1:num_bins_single_sided);
18 X_filtered_single(2:end-1) = 2 * X_filtered_single(2:end-1);
19 X_filtered_dB = 20 * log10(X_filtered_single + eps);
20
21 figure('Name', 'Bandpass Filter - Frequency Domain', 'Position',
22 [100, 100, 1200, 700]);
23 % Original spectrum (plot removed)
24 % Filtered spectrum (removed plot)
```

```

25 %% Calculate noise reduction statistics
26 % Measure power in passband and stopband before and after
27 % filtering
28 % Passband power (should be similar before and after)
29 passband_indices_signal = find(f >= fmin & f <= fmax);
30 passband_power_before = mean(X_hamming_single(
    passband_indices_signal).^2);
31 passband_power_after = mean(X_filtered_single(
    passband_indices_signal).^2);
32
33 % Stopband power (should be much lower after filtering)
34 stopband_indices_lower = find(f <= fstop_lower);
35 stopband_indices_upper = find(f >= fstop_upper & f <= fs/2);
36 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
51 % 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

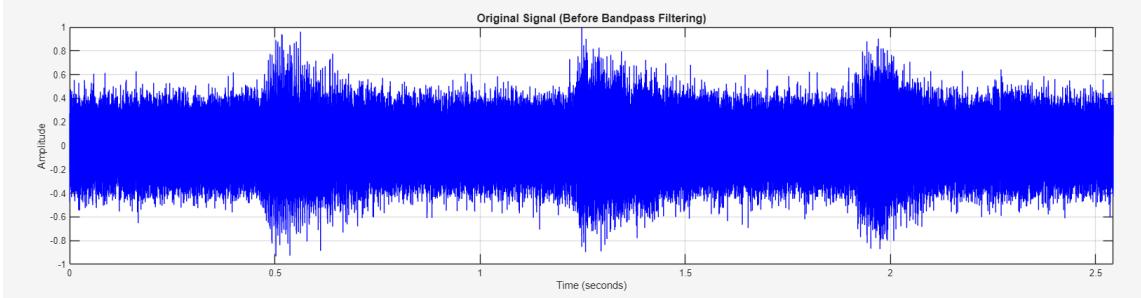


Figure 18: Original signal in the time domain

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

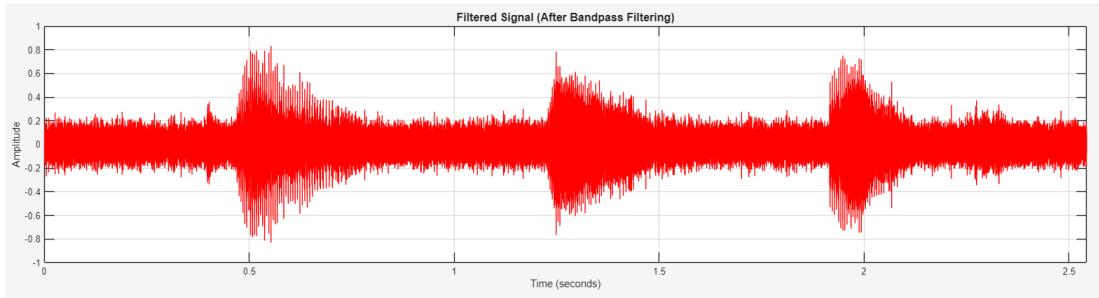


Figure 19: Signal after bandpass filtering (time domain)

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.

FILTERING PERFORMANCE		
PASSBAND POWER:		
Before filtering:	1.581550e-06	
After filtering:	3.729591e+00	
Change:	63.73 dB	
STOPBAND POWER:		
Before filtering:	5.451468e-07	
After filtering:	4.677199e-07	
Noise reduction:	0.67 dB	
SIGNAL-TO-NOISE RATIO:		
SNR before filtering:	4.63 dB	
SNR after filtering:	69.02 dB	
SNR improvement:	64.39 dB	
AMPLITUDE STATISTICS		
Maximum amplitude:	Before	After
Maximum amplitude:	1.0000	0.8339
RMS amplitude:	0.1748	0.1050
Standard deviation:	0.1748	0.1050

Figure 20: Filter output verification (terminal)

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*:

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

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

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

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 $B = 4$ kHz, so $m^2(t)$ has bandwidth $2B = 8$ 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 ± 8 kHz. For a carrier around 16 kHz, this places $2f_c \approx 32$ 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

```

1 %% Task 3: Carrier Recovery and Mixing
2 % Apply square law to the bandpass filtered signal
3 x_squared = x_filtered .^ 2;
4
5 %% Compute spectrum of squared signal
6 % Apply Hamming window
7 x_squared_windowed = x_squared .* hamming_window;
8
9 % Compute FFT
10 X_squared = fft(x_squared_windowed);
11 X_squared_magnitude = abs(X_squared);
12 X_squared_normalised = X_squared_magnitude / N / CG_hamming;
13
14 % Single-sided spectrum
15 X_squared_single = X_squared_normalised(1:num_bins_single_sided);
16 X_squared_single(2:end-1) = 2 * X_squared_single(2:end-1);
17 X_squared_dB = 20 * log10(X_squared_single + eps);
18
19 %% Plot squared signal spectrum (removed for ease)
20 % Mark expected 2fc region (dotted)
21 xline(2*fc_rounded/1000, 'r--', 'LineWidth', 2);
22
23 % Define search range around 2fc
24 search_range_low = 2*fc_rounded - 5000;      % Hz
25 search_range_high = 2*fc_rounded + 5000;      % Hz
26
27 % Find indices of search region
28 search_indices = find(f >= search_range_low & f <=
    search_range_high);
29
30 % Extract the region to search for peaks
31 X_search = X_squared_single(search_indices);
32 f_search = f(search_indices);
33
34 % Use findpeaks
35 [peaks, locs] = findpeaks(X_search, f_search);
36
37 % Get the highest peak
38 [peak_value, max_idx] = max(peaks);
39 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: $(32 \text{ kHz})/2 = 16 \text{ kHz}$.

Results and Discussion

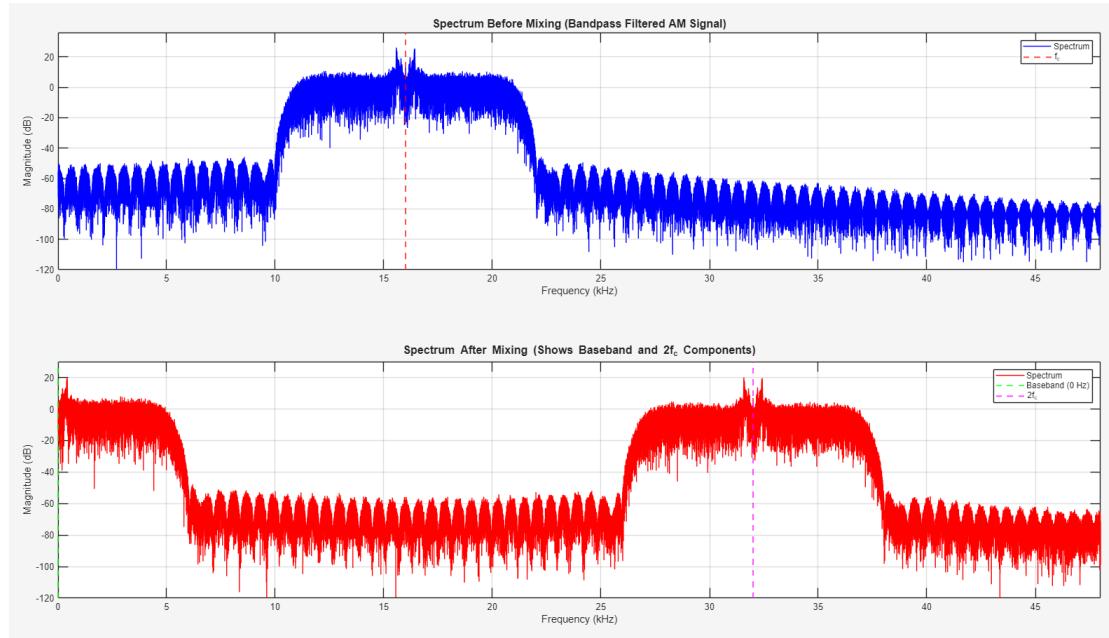


Figure 21: Before and after signal squaring in the frequency domain

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.

```

-----  

TASK 3: CARRIER RECOVERY  

Square law applied to filtered signal.  

Squared signal length: 244104 samples  

CARRIER FREQUENCY DETECTION  

Search range: 27000 - 37000 Hz  

Peak found at 2fc: 32000.00 Hz  

Calculated fc: 16000.00 Hz  

Rounded fc (to 1 kHz): 16000 Hz (16.0 kHz)  

-----  

Initial estimate (Task 1): 16000 Hz  

Difference: 0.00 Hz  

Carrier frequency CONFIRMED: 16000 Hz

```

Figure 22: AM band identification

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

$$s(t) \cos(\omega_c t + \phi) = m(t) \cos(\omega_c t) \cos(\omega_c t + \phi). \quad (19)$$

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

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

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

```

1 %% Sub-task 3.2: Carrier Generation and Mixing
2 % Set initial phase to zero (will be optimised in Task 5)
3 phi = 0;
4
5 % Generate local carrier signal
6 % carrier(t) = cos(2*pi*fc*t + phi)
7 carrier = cos(2 * pi * fc_final * t + phi);
8
9 % Carrier frequency, phase and signal length outputs (removed)
10 %% Multiply filtered AM signal with carrier (mixing)
11 x_mixed = x_filtered .* carrier;
12
13 %% Time domain plots (removed for clarity)
14
15 figure('Name', 'Mixing Process - Time Domain', 'Position', [100,
100, 1200, 800]);
16
17 %% Frequency domain analysis of mixed signal
18
19 % Apply Hamming window
20 x_mixed_windowed = x_mixed .* hamming_window;
21
22 % Compute FFT
23 X_mixed = fft(x_mixed_windowed);
24 X_mixed_magnitude = abs(X_mixed);
25 X_mixed_normalised = X_mixed_magnitude / N / CG_hamming;
```

```

26
27 % Single-sided spectrum
28 X_mixed_single = X_mixed_normalised(1:num_bins_single_sided);
29 X_mixed_single(2:end-1) = 2 * X_mixed_single(2:end-1);
30 X_mixed_dB = 20 * log10(X_mixed_single + eps);
31
32 %% Frequency domain plots (removed for clarity)
33 %% Analyse the frequency components
34
35 % Find power in baseband region (0 to 4 kHz - message bandwidth)
36 baseband_indices = find(f >= 0 & f <= 4000);
37 baseband_power = mean(X_mixed_single(baseband_indices).^2);
38
39 % Find power in 2fc region (2fc +- 4 kHz)
40 double_fc_indices = find(f >= (2*fc_final - 4000) & f <= (2*
    fc_final + 4000));
41 double_fc_power = mean(X_mixed_single(double_fc_indices).^2);
42
43 % Find power in noise region (between baseband and 2fc)
44 noise_region_low = 8000; % Above baseband
45 noise_region_high = 2*fc_final - 8000; % Below 2fc component
46 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);
47 else
    noise_power = 0;
48 end
49
50
51 % Carrier signal analysis methods (prints terminal outputs,
52 % removed for clarity)

```

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

The code generates a cosine wave (looks like a green block as it has been zoomed out) 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

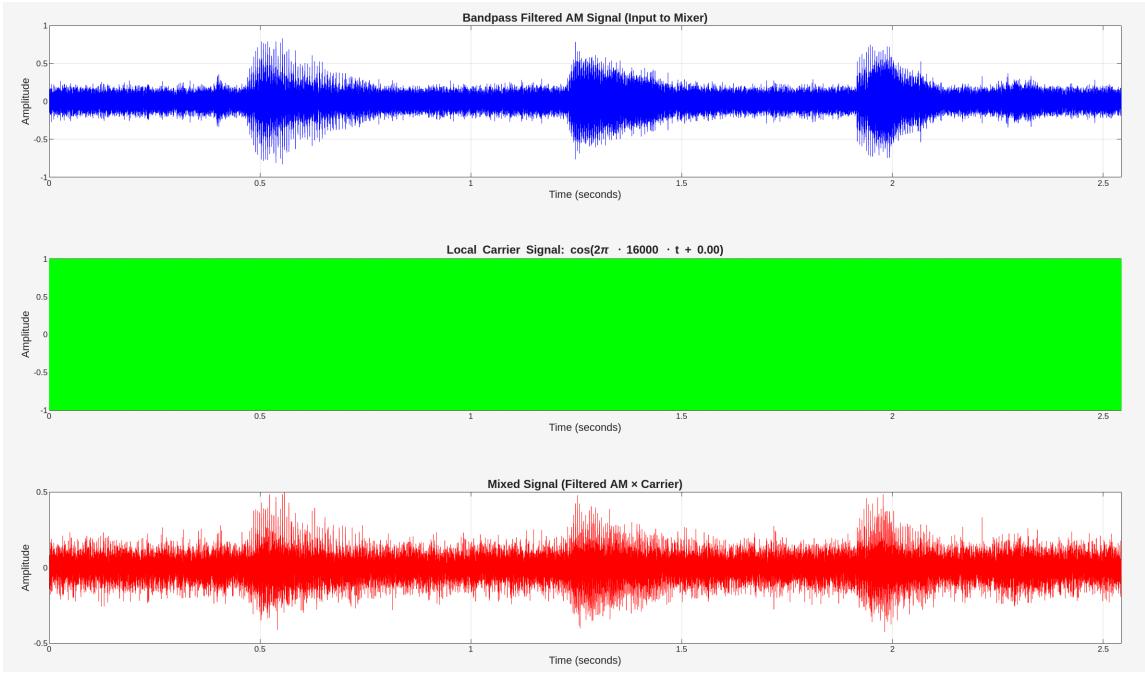


Figure 23: Time domain signals before and after mixing

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 ($6\times$ the message bandwidth) makes lowpass filtering straightforward — even a gentle rolloff will adequately suppress the $2f_c$ component.

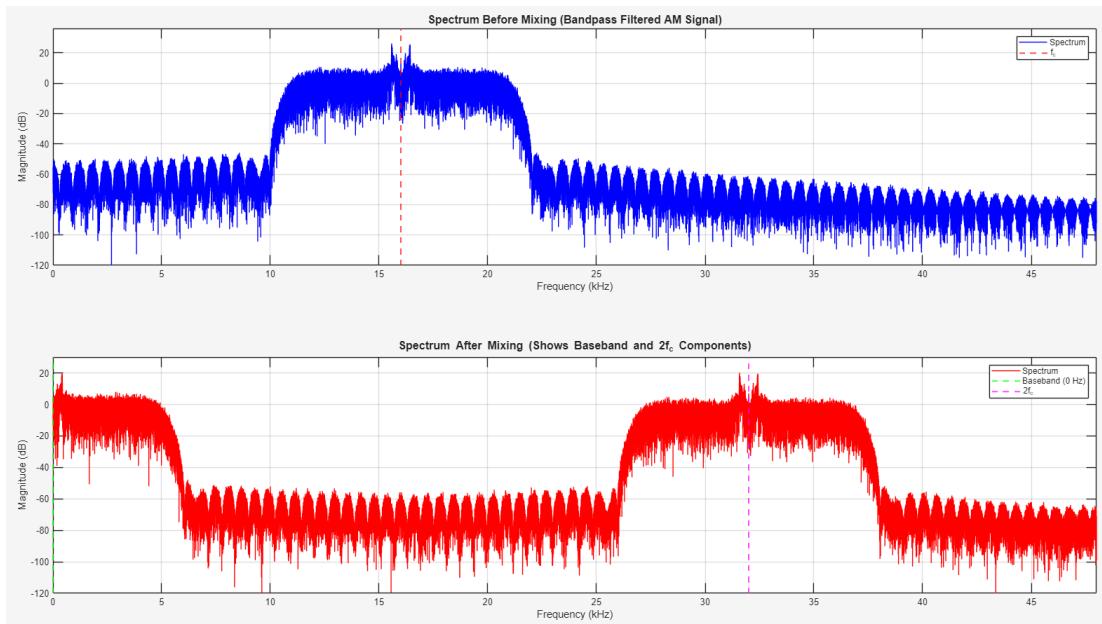


Figure 24: Frequency domain signals before and after mixing

```

=====
          CARRIER GENERATION AND MIXING
Carrier signal generated.
Carrier frequency:      16000 Hz
Initial phase:          0.0000 radians (0.00 degrees)
Carrier signal length: 244104 samples
Mixing complete.
Mixed signal length:   244104 samples
          MIXED SIGNAL ANALYSIS
FREQUENCY COMPONENTS:
  Baseband (0-4 kHz):    1.251066e+00 (signal)
  2fc region:            9.323979e-01 (to be filtered)
  Noise region:          3.081335e-07
-----
The baseband component contains the message signal.
The 2fc component will be removed by lowpass filter.
          AMPLITUDE COMPARISON
                                         Filtered      Mixed
Maximum amplitude:        0.8339      0.4953
RMS amplitude:           0.1050      0.0688
Standard deviation:      0.1050      0.0688

```

Figure 25: Terminal output showing the results of carrier generation and mixing

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 (Task 3), completing the demodulation chain.

2.4.1 Task 4 Part 1 - Designing IIR Filters

Procedure and Theory

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 (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 dB attenuation at the cutoff frequency, and a rolloff rate of $20n$ dB/decade for order n . For a 4th-order design, the rolloff is 80 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:

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

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

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

For a cutoff frequency of 4 kHz and sampling rate $f_s = 96$ 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.

```

1 %% Task 4: IIR Lowpass Filter Design
2 % Filter specifications
3 filter_order = 4;
4 fc_lowpass = 4000; % Cutoff frequency in Hz
5 % Uses normalised frequency where 1 = Nyquist frequency (fs/2)
6 Wn = fc_lowpass / (fs/2);
7 % Print method for showing specs has been removed.
8 %% Design the Butterworth filter using bilinear transform
9 % MATLAB's butter() function implements the bilinear transform
10 % method
11 % It returns coefficients for the transfer function H(z) = B(z)/A
12 % (z)
13 [b_iir, a_iir] = butter(filter_order, Wn, 'low');
14 % Method for printing coefficients has been removed.
15 %% Verify coefficient properties (printing removed)
16 %% Display the transfer function (printing method removed)
17 %% Plot filter coefficients
18 % 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

```

-----+
TASK 4: IIR LOWPASS FILTER
FILTER SPECIFICATIONS:
  Filter type:           Butterworth (maximally flat)
  Filter order:          4
  Cutoff frequency:      4000 Hz
  Sampling frequency:    96000 Hz
  Normalised cutoff (Wn): 0.083333
-----+
IIR FILTER COEFFICIENTS
Numerator coefficients (b):
  b[0] = 0.0002131387
  b[1] = 0.0008525549
  b[2] = 0.0012788324
  b[3] = 0.0008525549
  b[4] = 0.0002131387
-----+
Denominator coefficients (a):
  a[0] = 1.0000000000
  a[1] = -3.3168079106
  a[2] = 4.1742455501
  a[3] = -2.3574027806
  a[4] = 0.5033753607
-----+
COEFFICIENT ANALYSIS
Number of numerator coefficients: 5
Number of denominator coefficients: 5
Sum of numerator coefficients:    0.0034102196
Sum of denominator coefficients:  0.0034102196
DC gain (H(z=1)):                1.000000

```

Figure 26: Terminal output showing filter coefficients

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.

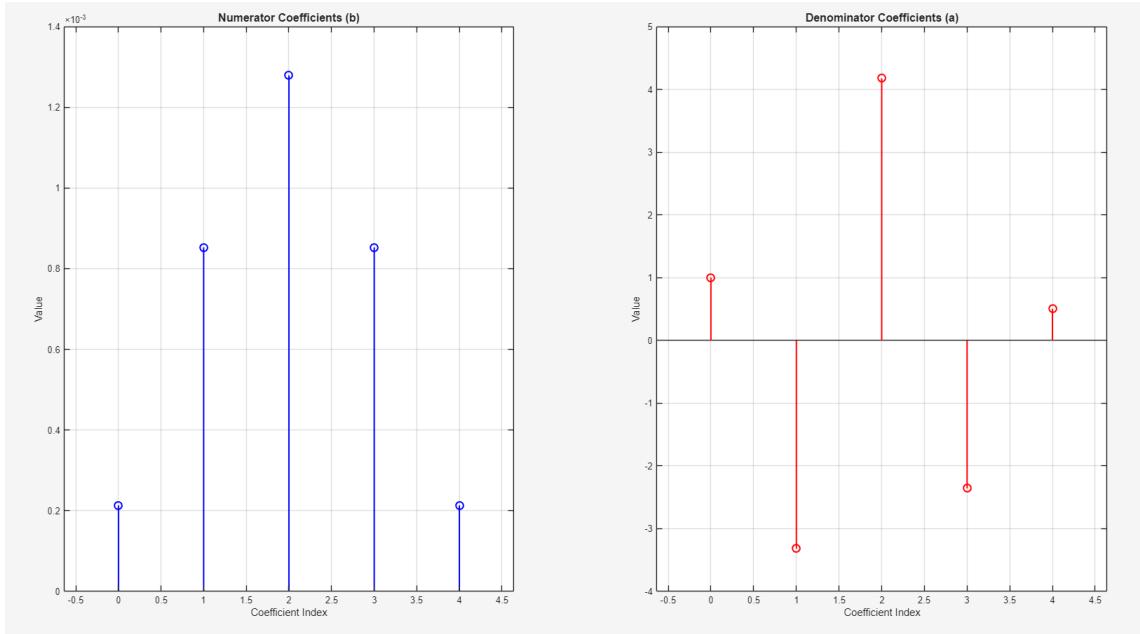


Figure 27: Visual plots showing the numerator and denominator coefficients of the filter

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 \times 20 dB/decade)

Table 5: Butterworth lowpass filter design requirements

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

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

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

```

1 %% Sub-task 4.2: IIR Frequency Response Verification
2 % Compute frequency response using freqz
3 N_freq = 8192; % Number of frequency points
4 [H_iir, f_iir] = freqz(b_iir, a_iir, N_freq, fs);
5 % Magnitude response in dB
6 H_iir_magnitude = abs(H_iir);
7 H_iir_dB = 20 * log10(H_iir_magnitude + eps);
8 % Phase response (plots removed)
9 H_iir_phase = angle(H_iir);
10 H_iir_phase_unwrapped = unwrap(H_iir_phase);
11 %% Plot magnitude response
12 %% Measure actual filter performance
13 % Find -3 dB point
14 idx_3dB = find(H_iir_dB <= -3, 1, 'first');
15 if ~isempty(idx_3dB)
16     f_3dB_actual = f_iir(idx_3dB);
17 else
18     f_3dB_actual = NaN;
19 end
20 % Measure gain at specific frequencies
21 f_test_points = [100, 1000, 2000, 3000, 4000, 5000, 6000, 8000,
22 10000, 20000];
23 gain_at_test_points = zeros(size(f_test_points));
24 for i = 1:length(f_test_points)
25     [~, idx] = min(abs(f_iir - f_test_points(i)));
26     gain_at_test_points(i) = H_iir_dB(idx);
27 end
28 % Measure rolloff rate (attenuation per octave after cutoff)
29 % Compare attenuation at 8 kHz (1 octave above 4 kHz) and 16 kHz
30 % (2 octaves)
31 [~, idx_8k] = min(abs(f_iir - 8000));
32 [~, idx_16k] = min(abs(f_iir - 16000));
33 atten_8k = H_iir_dB(idx_8k);
34 atten_16k = H_iir_dB(idx_16k);
35 rolloff_per_octave = atten_8k - atten_16k;
36 % Display verification results (removed)
37 %% Verify Butterworth characteristics
38 % Check passband flatness (should be monotonic, no ripple) -
39 % terminal output removed
40 passband_idx = find(f_iir <= fc_lowpass);
41 passband_gain = H_iir_dB(passband_idx);
42 passband_ripple = max(passband_gain) - min(passband_gain);
43 % Check gain at cutoff
44 [~, idx_fc] = min(abs(f_iir - fc_lowpass));
45 gain_at_fc = H_iir_dB(idx_fc);
46 if abs(gain_at_fc - (-3)) < 0.5
47 %% Pole-Zero plot for stability verification
48 %% Get poles and zeros
49 zeros_iir = roots(b_iir);
50 poles_iir = roots(a_iir);
51 % Plot unit circle, poles and zeros (removed)

```

```

49 % Check stability
50 pole_magnitudes = abs(poles_iir);
51 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

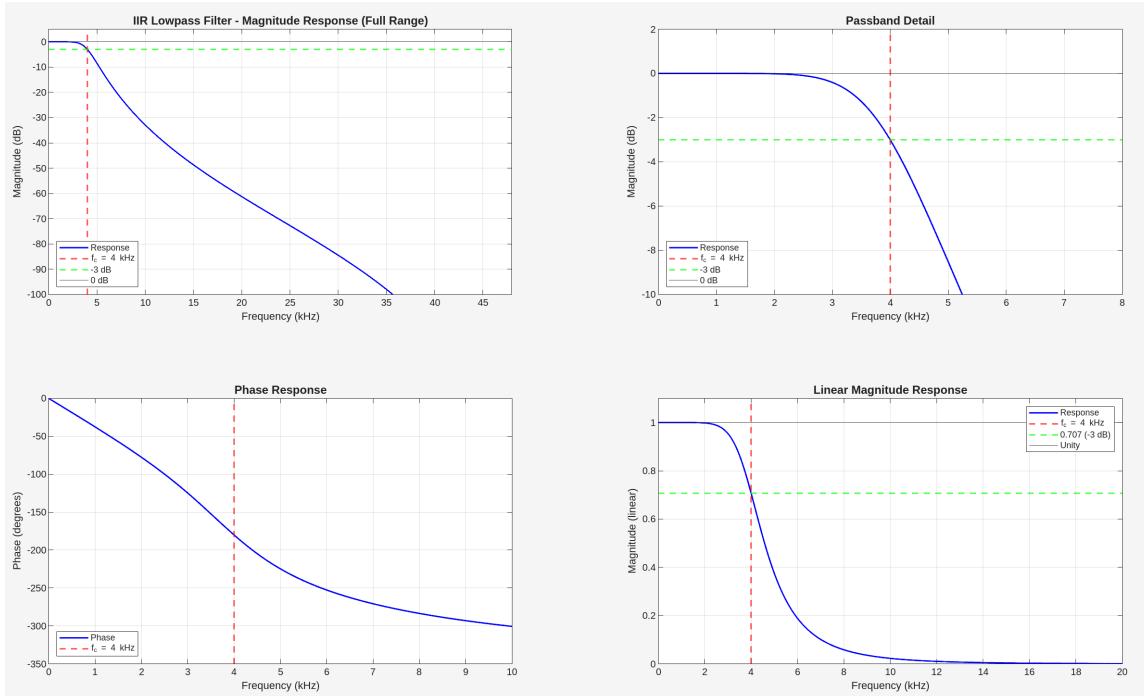


Figure 28: Four plots for verification

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.

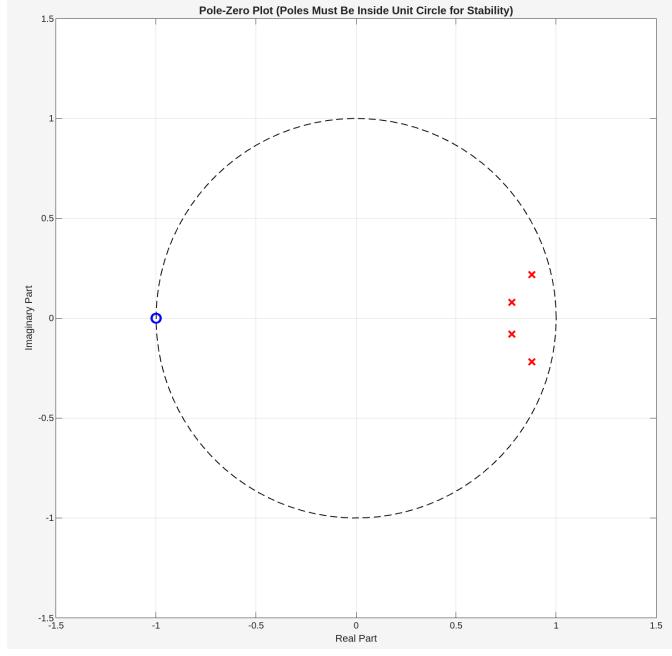


Figure 29: Pole-zero plot for filter stability

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.

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

Procedure and Theory

From the transfer function:

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

The time-domain difference equation is:

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_M x[n-M] - a_1 y[n-1] - a_2 y[n-2] - \cdots - a_N y[n-N] \quad (24)$$

Or more compactly:

$$y[n] = \sum_{k=0}^M b_k x[n-k] - \sum_{k=1}^N a_k y[n-k] \quad (25)$$

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

```

1 function y = custom_iir_filter(b, a, x)
2 % Store original orientation
3 is_column = iscolumn(x);
4 % Convert all inputs to row vectors for consistent processing
5 b = b(:).'; % Ensure row vector
6 a = a(:).'; % Ensure row vector
7 x = x(:).'; % Ensure row vector
8 % Get lengths
9 L = length(x); % Signal length
10 M = length(b) - 1; % Numerator order (number of b coefficients
11 % minus 1)
12 N = length(a) - 1; % Denominator order (number of a coefficients
13 % minus 1)
14 %% Normalise coefficients if a(1) is not 1
15 if a(1) ~= 1
16     b = b / a(1);
17     a = a / a(1);
18 end
19 %% Initialise buffers
20 x_buffer = zeros(1, M + 1); % input
21 % Output buffer: stores past N output samples
22 y_buffer = zeros(1, N);
23 %% Preallocate output array
24 y = zeros(1, L);
25 %% Main filtering loop
26 for n = 1:L
27     % Shift input buffer to the right (make room for new sample)
28     for k = M+1:-1:2
29         x_buffer(k) = x_buffer(k-1);
30     end
31     % Insert current input sample at the beginning
32     x_buffer(1) = x(n);
33     % Calculate feedforward (FIR) part: sum of b(k) * x[n-k
34     % +1]
35     feedforward_sum = 0;
36     for k = 1:M+1
37         feedforward_sum = feedforward_sum + b(k) * x_buffer(k);
38     end
39     % Calculate feedback (recursive) part: sum of a(k) * y[n-k+1]
40     feedback_sum = 0;
41     for k = 1:N
42         feedback_sum = feedback_sum + a(k+1) * y_buffer(k);
43     end
44     % Compute current output: y[n] = feedforward - feedback
45     y(n) = feedforward_sum - feedback_sum;
46     % Shift output buffer to the right (make room for new output)
47     for k = N:-1:2
48         y_buffer(k) = y_buffer(k-1);
49     end
50     % Insert current output at the beginning (becomes y[n-1] for
51     % next iteration)

```

```

48     if N >= 1
49         y_buffer(1) = y(n);
50     end
51 end
52 %% Restore original orientation if input was column vector
53 if is_column
54     y = y(:);
55 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 $1 + a_1 z^{-1} + \dots$, which rearranges to $y[n] = \dots - a_1 y[n - 1] - \dots$ 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**).

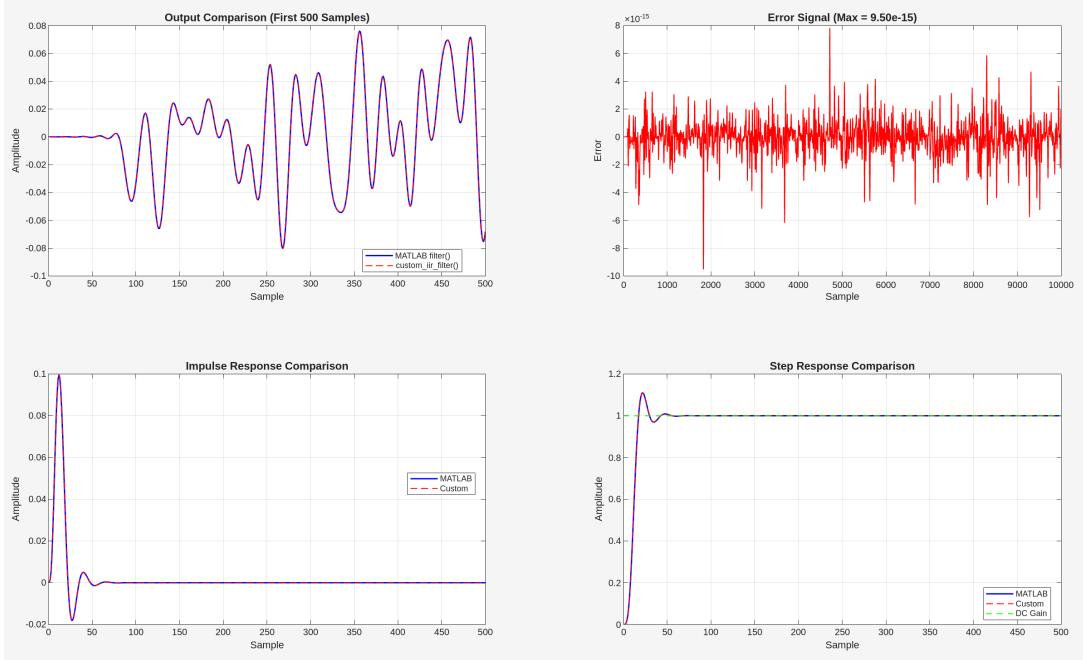


Figure 30: Plots to benchmark custom implementation

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 $18.2 \times$ 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.

```

1 tic;
2 x_demodulated = custom_iir_filter(b_iir, a_iir, x_mixed);
3 time_iir_filter = toc;
4 %% Plot time domain comparison
5 %% Frequency domain analysis
6 % Compute spectrum of demodulated signal
7 window_demod = hamming(length(x_demodulated))';
8 x_demod_windowed = x_demodulated .* window_demod;
9 X_demod = fft(x_demod_windowed);
10 X_demod_magnitude = abs(X_demod) / length(X_demod);
11 % Single-sided spectrum
12 N_demod = length(X_demod);
13 X_demod_single = X_demod_magnitude(1:floor(N_demod/2)+1);
14 X_demod_single(2:end-1) = 2 * X_demod_single(2:end-1);
15 f_demod = (0:floor(N_demod/2)) * fs / N_demod;
16 % Convert to dB
17 X_demod_dB = 20 * log10(X_demod_single + eps);
18 % Also compute spectrum of mixed signal for comparison
19 window_mixed = hamming(length(x_mixed))';
20 x_mixed_windowed = x_mixed .* hamming_window;
21 X_mixed = fft(x_mixed_windowed);
22 X_mixed_magnitude = abs(X_mixed) / length(X_mixed);
23 N_mixed = length(X_mixed);
24 X_mixed_single = X_mixed_magnitude(1:floor(N_mixed/2)+1);
25 X_mixed_single(2:end-1) = 2 * X_mixed_single(2:end-1);
26 f_mixed = (0:floor(N_mixed/2)) * fs / N_mixed;
27 X_mixed_dB = 20 * log10(X_mixed_single + eps);
28 %% Plot frequency domain comparison (removed for clarity)
29 %% Analysis of filtering effect
30 % Calculate power in different frequency bands
31 % Baseband (0 to 4 kHz) - message region
32 baseband_idx_mixed = find(f_mixed <= fc_lowpass);
33 baseband_idx_demod = find(f_demod <= fc_lowpass);
34 baseband_power_before = sum(X_mixed_single(baseband_idx_mixed) .^ 2);
35 baseband_power_after = sum(X_demod_single(baseband_idx_demod) .^ 2)
;
```

```

36 % Stopband (above 4 kHz) - should be attenuated
37 stopband_idx_mixed = find(f_mixed > fc_lowpass);
38 stopband_idx_demod = find(f_demod > fc_lowpass);
39 stopband_power_before = sum(X_mixed_single(stopband_idx_mixed)
    .^2);
40 stopband_power_after = sum(X_demod_single(stopband_idx_demod).^2)
    ;
41 % 2fc region (2fc +- 4 kHz)
42 fc2_low = 2*fc_final - 4000;
43 fc2_high = 2*fc_final + 4000;
44 fc2_idx_mixed = find(f_mixed >= fc2_low & f_mixed <= fc2_high);
45 fc2_idx_demod = find(f_demod >= fc2_low & f_demod <= fc2_high);
46 fc2_power_before = sum(X_mixed_single(fc2_idx_mixed).^2);
47 fc2_power_after = sum(X_demod_single(fc2_idx_demod).^2);
48 % Calculate attenuation (no print logic)
49 baseband_change_dB = 10 * log10(baseband_power_after /
    baseband_power_before);
50 stopband_attenuation_dB = 10 * log10(stopband_power_before /
    stopband_power_after);
51 fc2_attenuation_dB = 10 * log10(fc2_power_before /
    fc2_power_after);
52 %% SNR (print logic removed)
53 % Estimate SNR as ratio of baseband power to remaining stopband
    power
54 snr_before = 10 * log10(baseband_power_before /
    stopband_power_before);
55 snr_after = 10 * log10(baseband_power_after /
    stopband_power_after);
56 snr_improvement = snr_after - snr_before;
57 %% 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

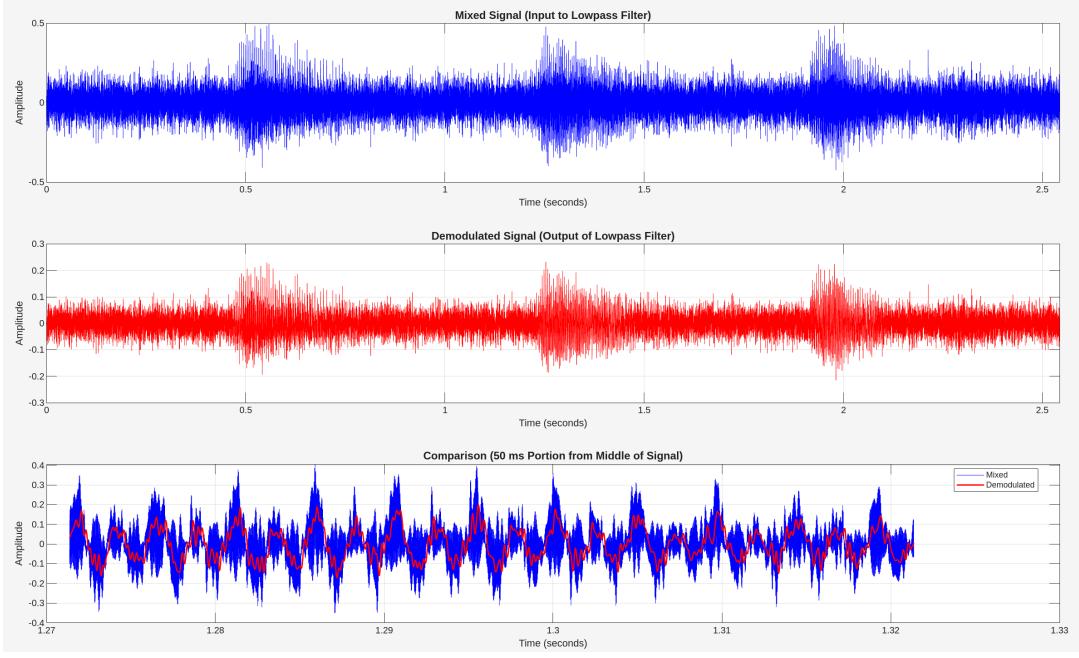


Figure 31: Comparison plot, showing the output after applying the lowpass filter

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.

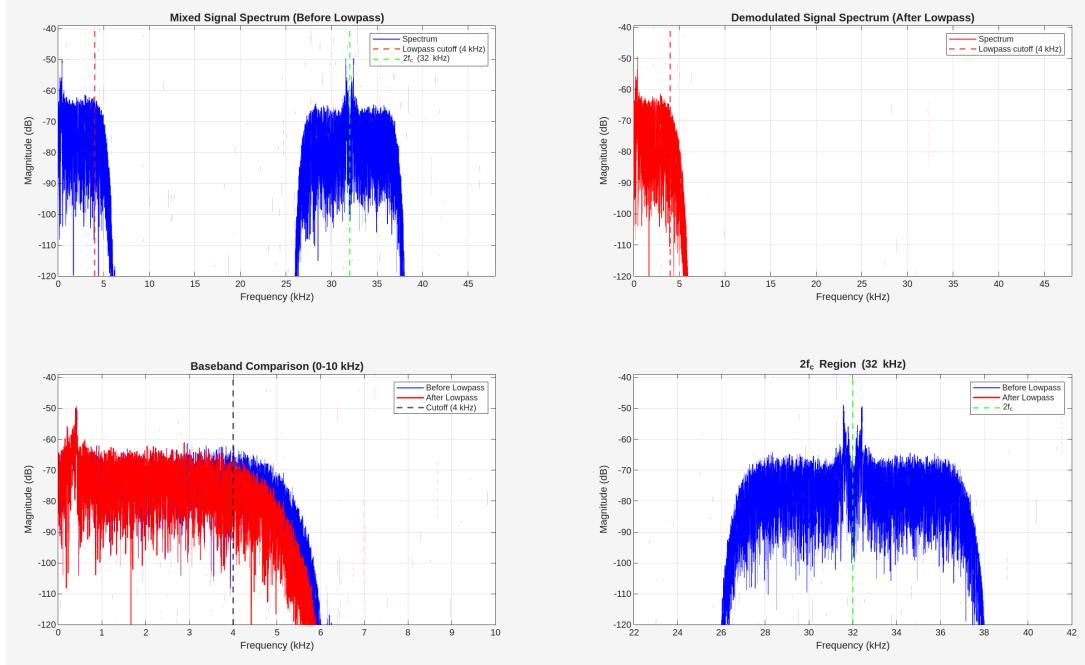


Figure 32: Plots to benchmark custom implementation

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.

```
FILTERING EFFECTIVENESS:
-----
baseband (0-4 kHz):
Power before: 1.574482e-03
Power after: 1.513112e-03
Change: -0.17 dB

Stopband (>4 kHz):
Power before: 2.658873e-03
Power after: 4.609176e-05
Attenuation: 17.52 dB

2f_c region (28-36 kHz):
Power before: 2.346633e-03
Power after: 4.657236e-12
Attenuation: 87.02 dB

SIGNAL-TO-NOISE RATIO ESTIMATION:
-----
SNR before lowpass: -2.26 dB
SNR after lowpass: 15.09 dB
SNR improvement: 17.35 dB
```

Figure 33: Terminal output showing the SNR and power improvements after applying the lowpass filter

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.

2.5 Task 5

Procedure and Theory

From Task 3 that coherent demodulation produces:

$$s(t) \cos(\omega_c t + \phi) = \frac{m(t)}{2} \cos(\phi) + \frac{m(t)}{2} \cos(2\omega_c t + \phi). \quad (26)$$

After lowpass filtering (Task 4), only the baseband term remains:

$$y(t) = \frac{m(t)}{2} \cos(\phi). \quad (27)$$

Thus, the output amplitude is scaled by the factor

$$\cos(\phi). \quad (28)$$

Find ϕ_{opt}

We seek the phase ϕ_{opt} that maximises $|\cos(\phi)|$ (which in turn maximises the output signal power):

$$\phi_{\text{opt}} = \arg \max_{\phi} |\cos(\phi)|. \quad (29)$$

The maximum value of $|\cos(\phi)|$ is 1, attained when

$$\phi_{\text{opt}} = n\pi, \quad n \in \mathbb{Z}. \quad (30)$$

Optimisation Techniques

We will implement three approaches:

- Coarse grid search: test phases at regular intervals (e.g., every 10°).
- Fine grid search: refine the grid around the best coarse value.
- Golden-section search: an efficient 1-D optimisation method for unimodal objectives.

Metric for Optimisation

We use the RMS amplitude of the demodulated signal as the optimisation metric:

$$\text{RMS} = \sqrt{\frac{1}{N} \sum_{n=0}^{N-1} y[n]^2}. \quad (31)$$

Maximising RMS corresponds to maximising $|\cos(\phi)|$ (and hence output power).

SNR Measurement

Signal-to-noise ratio (in dB) is defined as

$$\text{SNR} = 10 \log_{10} \left(\frac{P_{\text{signal}}}{P_{\text{noise}}} \right) \text{ dB}. \quad (32)$$

We estimate powers via time-domain or spectral methods, for example

$$P_{\text{signal}} = \frac{1}{N} \sum_{n=0}^{N-1} s_{\text{band}}[n]^2, \quad P_{\text{noise}} = \frac{1}{N} \sum_{n=0}^{N-1} n_{\text{band}}[n]^2, \quad (33)$$

or using FFT-bin sums over the speech band (300–3400 Hz):

$$P_{\text{signal}} = \sum_{k \in \mathcal{K}_{\text{speech}}} |Y[k]|^2, \quad P_{\text{noise}} = \sum_{k \in \mathcal{K}_{\text{noise}}} |Y[k]|^2. \quad (34)$$

Maximum RMS (and thus maximum $|\cos(\phi)|$) yields the best demodulated output and typically the highest estimated SNR when noise is independent of the demodulation phase.

```

1 %% Sub-task 5.2: Generate Optimally Demodulated Signal
2 % Generate carrier with optimal phase
3 carrier_optimal = cos(2*pi*fc_final*t + phi_optimal);
4 % Mix with bandpass filtered signal
5 x_mixed_optimal = x_filtered .* carrier_optimal;
6 % Apply lowpass filter
7 x_final = custom_iir_filter(b_iir, a_iir, x_mixed_optimal);
8 %% Sub-task 5.3: SNR Measurement

```

```

9  % Compute spectrum of final demodulated signal
10 window_final = hamming(length(x_final))';
11 x_final_windowed = x_final .* window_final;
12 X_final = fft(x_final_windowed);
13 X_final_magnitude = abs(X_final) / length(X_final);
14 N_final = length(X_final);
15 X_final_single = X_final_magnitude(1:floor(N_final/2)+1);
16 X_final_single(2:end-1) = 2 * X_final_single(2:end-1);
17 f_final = (0:floor(N_final/2)) * fs / N_final;
18 X_final_dB = 20 * log10(X_final_single + eps);
19 %% Method 1: Speech band vs out-of-band noise
20 % Define frequency bands
21 speech_low = 300;           % Speech starts around 300 Hz
22 speech_high = 3400;          % Speech ends around 3400 Hz (telephone
                               quality)
23 noise_low = 3800;            % Noise region above speech
24 noise_high = 4000;           % Up to filter cutoff
25 % Calculate power in speech band
26 speech_idx = find(f_final >= speech_low & f_final <= speech_high)
;
27 speech_power = sum(X_final_single(speech_idx).^2);
28 % Calculate power in noise region (just above speech)
29 noise_idx = find(f_final >= noise_low & f_final <= noise_high);
30 if ~isempty(noise_idx)
31     noise_power_method1 = sum(X_final_single(noise_idx).^2);
32     % Scale to equivalent bandwidth
33     noise_bw = noise_high - noise_low;
34     speech_bw = speech_high - speech_low;
35     noise_power_scaled = noise_power_method1 * (speech_bw /
noise_bw);
36     snr_method1 = 10 * log10(speech_power / noise_power_scaled);
37 else
38     snr_method1 = NaN;
39 end
40 %% Method 2: Signal peaks vs RMS noise floor
41 % Find spectral peaks (signal components) - estimated
42 [peaks, peak_locs] = findpeaks(X_final_single, 'MinPeakHeight',
max(X_final_single)*0.1);
43 peak_freqs = f_final(peak_locs);
44 % Only consider peaks in speech band
45 speech_peak_idx = find(peak_freqs >= speech_low & peak_freqs <=
speech_high);
46 if ~isempty(speech_peak_idx)
47     signal_power_peaks = sum(peaks(speech_peak_idx).^2);
48 else
49     signal_power_peaks = speech_power;
50 end
51 % Estimate noise floor as median of spectrum (robust to peaks)
52 noise_floor = median(X_final_single(speech_idx));
53 noise_power_method2 = noise_floor^2 * length(speech_idx);
54 snr_method2 = 10 * log10(signal_power_peaks / noise_power_method2)

```

```

    );
55 %% Method 3: Time-domain SNR estimation using signal envelope
56 % Compute signal envelope using Hilbert transform
57 signal_envelope = abs(hilbert(x_final));
58 % Estimate signal power from envelope peaks
59 envelope_threshold = 0.3 * max(signal_envelope);
60 signal_samples = signal_envelope > envelope_threshold;
61 signal_power_time = mean(x_final(signal_samples).^2);
62 % Estimate noise power from quiet regions
63 noise_samples = signal_envelope < 0.1 * max(signal_envelope);
64 if sum(noise_samples) > 100
65     noise_power_time = mean(x_final(noise_samples).^2);
66 else
67     % If no quiet regions, use overall variance minus signal
68     noise_power_time = var(x_final) - signal_power_time * mean(
69         signal_samples);
70     noise_power_time = max(noise_power_time, eps); % Ensure
71         positive
72 end
73 snr_method3 = 10 * log10(signal_power_time / noise_power_time);
74 %% Method 4: Compare with phi = 0 (unoptimised) - removed
75 % Calculate RMS with phi = 0
76 carrier_zero = cos(2*pi*fc_final*t);
77 x_mixed_zero = x_filtered .* carrier_zero;
78 x_demod_zero = custom_iir_filter(b_iir, a_iir, x_mixed_zero);
79 rms_zero = sqrt(mean(x_demod_zero.^2));
80 % Calculate RMS with optimal phase
81 rms_final = sqrt(mean(x_final.^2));
82 % Calculate improvement
83 amplitude_improvement = rms_final / rms_zero;
84 power_improvement_dB = 20 * log10(amplitude_improvement);
85 %% SNR Summary (print removed)
86 %% Plot SNR analysis
87 % Plot 1: Spectrum with frequency bands marked (removed)
88 % Plot 2: Time-domain envelope (removed)
89 % Plot 3: Comparison phi=0 vs phi optimal (removed)
90 % Plot 4: SNR comparison bar chart (removed)
91 %% Sub-task 5.4: Audio Playback and Message Identification
92 % Normalise for playback (prevent clipping)
93 x_playback = x_final / max(abs(x_final)) * 0.9;
94 % Play the audio
95 sound(x_playback, fs);
96 % Wait for playback to complete
97 pause(length(x_playback)/fs + 0.5);
98 %% Save audio file for reference
99 output_filename = 'demodulated_message.wav';
100 audiowrite(output_filename, x_playback, fs);
101 %% Final message identification prompt (removed)
102 %% Complete summary (removed)

```

Listing 12: MATLAB Code for Task 5: IIR Filter Mixed Sig

Code and Explanation

This code does the following:

1. Plays the audio clip after applying all of the prior filtering etc.
2. Tests and plots 3 different phase optimisation techniques and compares theoretical output with practical.
3. Terminal output showing the SNR comparisons for each of the three methods.
4. A plot output showing phase optimisation effects on a small section of the audio signal, demodulated output.
5. Plot with signal and envelope overlap.

Results and Discussion

From **Figure X**, three optimisation techniques have been used (technically 2). The first is coarse grid search, finding the amplitudes in 5° steps and comparing for the maximum amplitude. Fine grid search does the same, in 1° steps instead. Golden section [source] is an efficient optimisation algorithm that reduces the search interval by a factor of $1/\varphi$ (the golden ratio) at each iteration. It converges in $\mathcal{O}(\log(\frac{1}{\varepsilon}))$ iterations, where ε is the desired tolerance, a slight improvement in speed.

It is easily visible that the 120° or 2.0944rad is the phase wherein the maximum amplitude can be achieved (using an RMS amplitude score/metric - the highest being 0.063828), with all three plots verifying this phase.

FIGURE SHOWING THE 3 OPTIMISATION METHODS VISUAL OUTPUTS

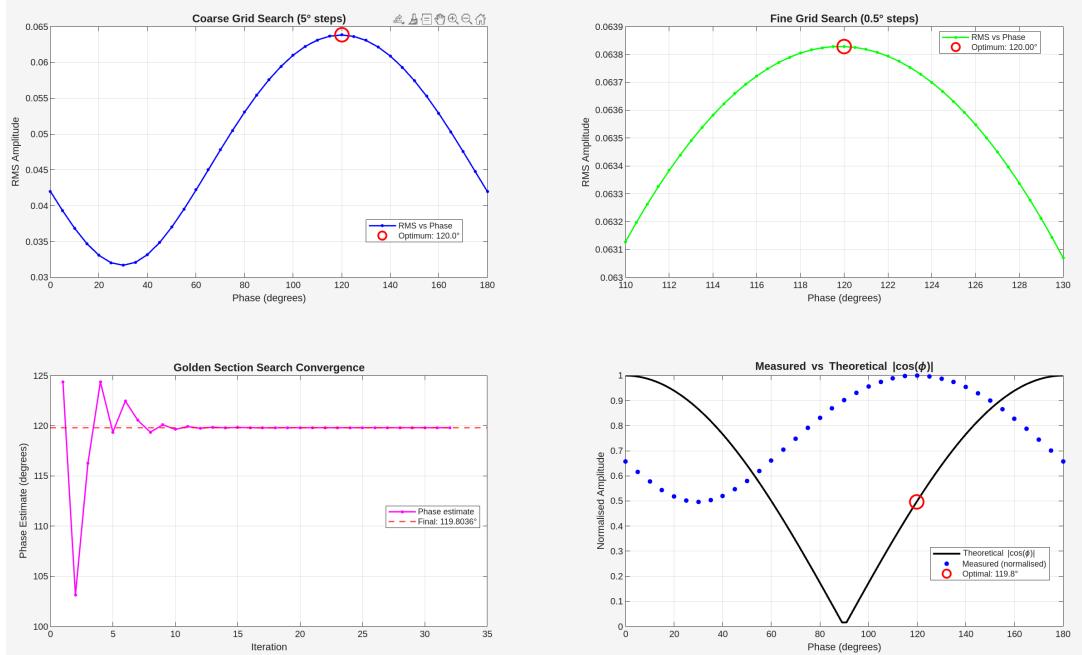


Figure 34: 3 phase optimisation plots

For further verification, **Figure X** below shows the terminal output for the optimisation calculation.

```

=====
PHASE OPTIMISATION

Technique 1: Coarse Grid Search
-----
Testing 37 phase values from 0 to 180 degrees...
Search completed in 0.68 seconds
Best phase (coarse): 2.0944 rad (120.00 degrees)
Best RMS amplitude: 0.063828

Technique 2: Fine Grid Search
-----
Refining search from 110.00 to 130.00 degrees...
Search completed in 0.63 seconds
Best phase (fine): 2.0944 rad (120.0000 degrees)
Best RMS amplitude: 0.063828
Improvement over coarse: 0.000000

Technique 3: Golden Section Search
-----
Initial bounds: [0.0000, 3.1416] rad
Iterations: 32
Search completed in: 0.52 seconds
Best phase (golden): 2.090000 rad (119.0036 degrees)
Best RMS amplitude: 0.063828
Final interval width: 6.49e-07 rad

```

Figure 35: Terminal output showing the optimisation results

SNR analysis can also verify the output signal and quantify how clear the audio is as well, the following metrics and their importance shown below, in *Table X*:

Table 6: Summary of SNR Estimation Methods and Their Suitable Use Cases

Method	Principle	Best For
Frequency Bands	Power ratio between speech and noise bands	Clear separation of signal/noise in
Peaks vs Floor	Signal peaks above noise floor	Tonal signals with distinct peaks
Time Envelope	Speech activity detection	Signals with varying amplitude over

Figure X below shows the terminal output for the SNR metric calculations:

```

SNR MEASUREMENT

Method 1: Speech Band Power vs Out-of-Band Noise
-----
Speech band: 300 - 3400 Hz
Speech power: 3.200821e-03
Noise band: 3800 - 4000 Hz
Noise power (scaled): 3.245539e-04
SNR (Method 1): 9.94 dB

Method 2: Peak Signal vs RMS Noise Floor
-----
Number of peaks found: 308
Signal power (peaks): 1.119568e-03
Noise floor (median): 2.190130e-04
Noise power: 5.012608e-04
SNR (Method 2): 2.77 dB

Method 3: Time-Domain Envelope Analysis
-----
Signal samples: 27972 (11.5% of total)
Signal power: 2.190130e-02
Noise samples: 114044 (47.0% of total)
Noise power: 4.485543e-04
SNR (Method 3): 16.89 dB

Method 4: Improvement from Phase Optimisation
-----
RMS with phi = 0: 0.041991
RMS with phi optimal: 0.063829
Amplitude improvement: 1.5201x
Power improvement: 3.64 dB

```

Figure 36:

Four methods have been implemented, as it can be seen from above. The first method looks at the speech band and compares the power to the noise band, which has been highlighted, resulting in $9.94dB$.

Method 2 wasn't particularly useful, comparing the peaks and the noise floor, resulting in a meagre $2.77dB$, not particularly useful.

Phase optimisation only performed slightly better than Method 2, with a $3.64dB$ SNR value.

Time domain envelope analysis performed the best, with $16.89dB$.

For the final signal, phase optimisation did improve the audio quality marginally, as can be seen from **Figure From T4 with the time series signal before and after filtering with LP**, with an amplitude improvement of $1.52\times$.

The final output was “Kvh”. The final, demodulated signal looks like this in the time domain:

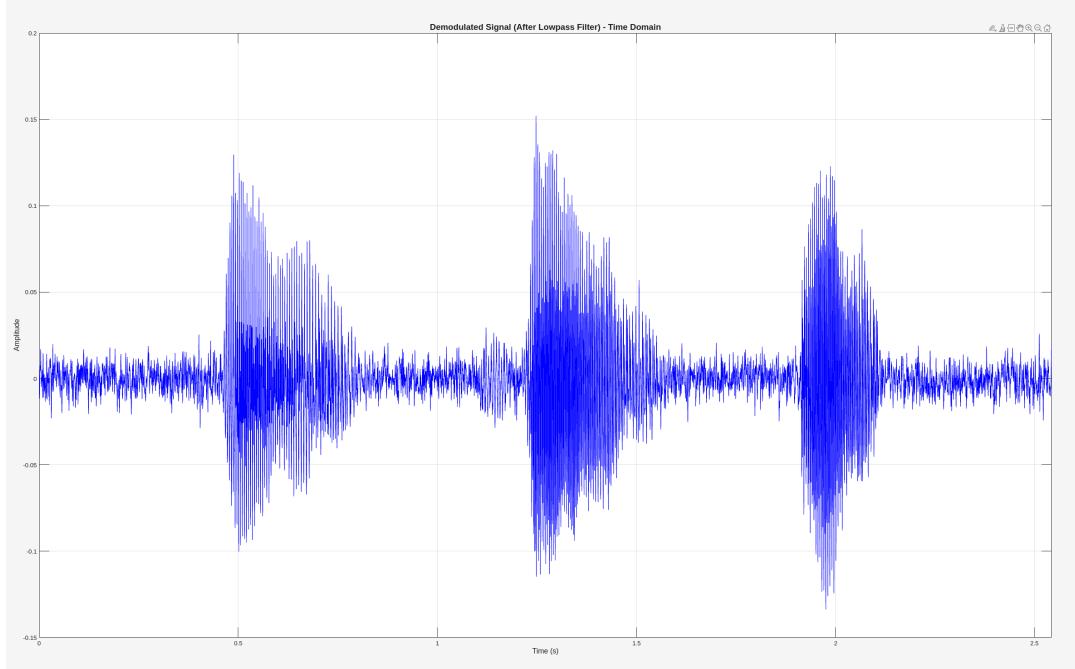


Figure 37: Final Time Domain Audio Signal

CONCLUSION

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.