Cairo University - Faculty of Engineering

Computer Engineering Department

Communications Engineering – Fall 2025

FM Stereo Broadcasting System

Communications Engineering Project

Name	**Student ID**
Ammar Yasser Fadlallah	9230593
Mohamed Yasser El-batesh	9230825

December 2024

Contents

1 Introduction and System Overview

This report presents the design and implementation of a complete FM stereo broadcasting system. The system includes both transmitter and receiver components, implementing the standard FM stereo multiplex format used in commercial broadcasting.

1.1 FM Stereo System Architecture

Transmitter: The transmitter takes stereo audio (Left and Right channels) and creates a composite baseband signal consisting of:

- **L+R (sum) signal:** 0-15 kHz baseband for mono compatibility
- **19 kHz pilot tone:** For synchronization at the receiver
- **L-R (difference) signal:** DSB-SC modulated onto a 38 kHz subcarrier

The composite signal is then pre-emphasized and FM modulated onto a carrier.

Receiver: The receiver performs FM demodulation, de-emphasis, and stereo decoding to recover the original Left and Right audio channels. The pilot tone is extracted and frequency-doubled to regenerate the 38 kHz carrier for synchronous demodulation of the L-R signal.

1.2 System Parameters

Table 1: FM Stereo System Parameters

Parameter	**Value**
Audio Sample Rate	44.1 kHz
Composite Sample Rate	200 kHz
FM Sample Rate	500 kHz
Carrier Frequency (simulation)	100 kHz
Default Frequency Deviation	75 kHz
Pilot Frequency	19 kHz
Subcarrier Frequency	38 kHz
Pre-emphasis Time Constant	75 μs
Composite Bandwidth	0-53 kHz

2 System Implementation

2.1 Input Audio

Synthetic stereo audio was generated with distinct content in each channel:

- **Left channel:** Ascending chirp (200-2000 Hz) + 440 Hz tone + 100 Hz bass
- **Right channel:** Descending chirp (2000-200 Hz) + 880 Hz tone + 150 Hz bass

Duration: 5 seconds

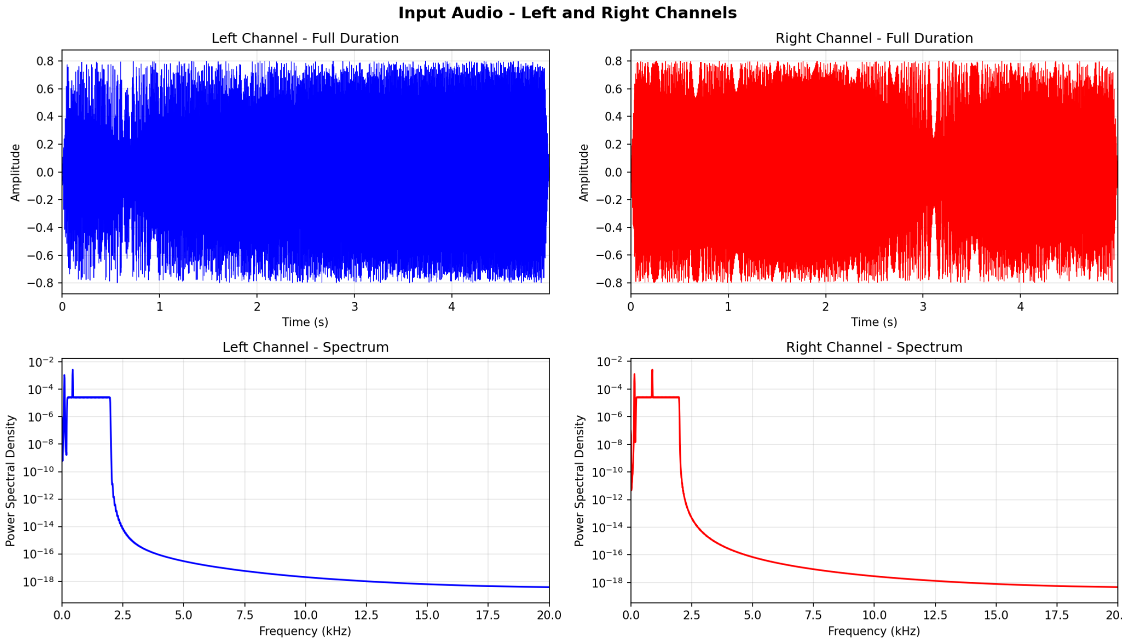

Figure 1: Input stereo audio - Left and Right channels in time and frequency domains

2.2 Composite Signal

The stereo composite signal contains three components: L+R sum signal, 19 kHz pilot, and L-R difference signal DSB-SC modulated onto a 38 kHz subcarrier.

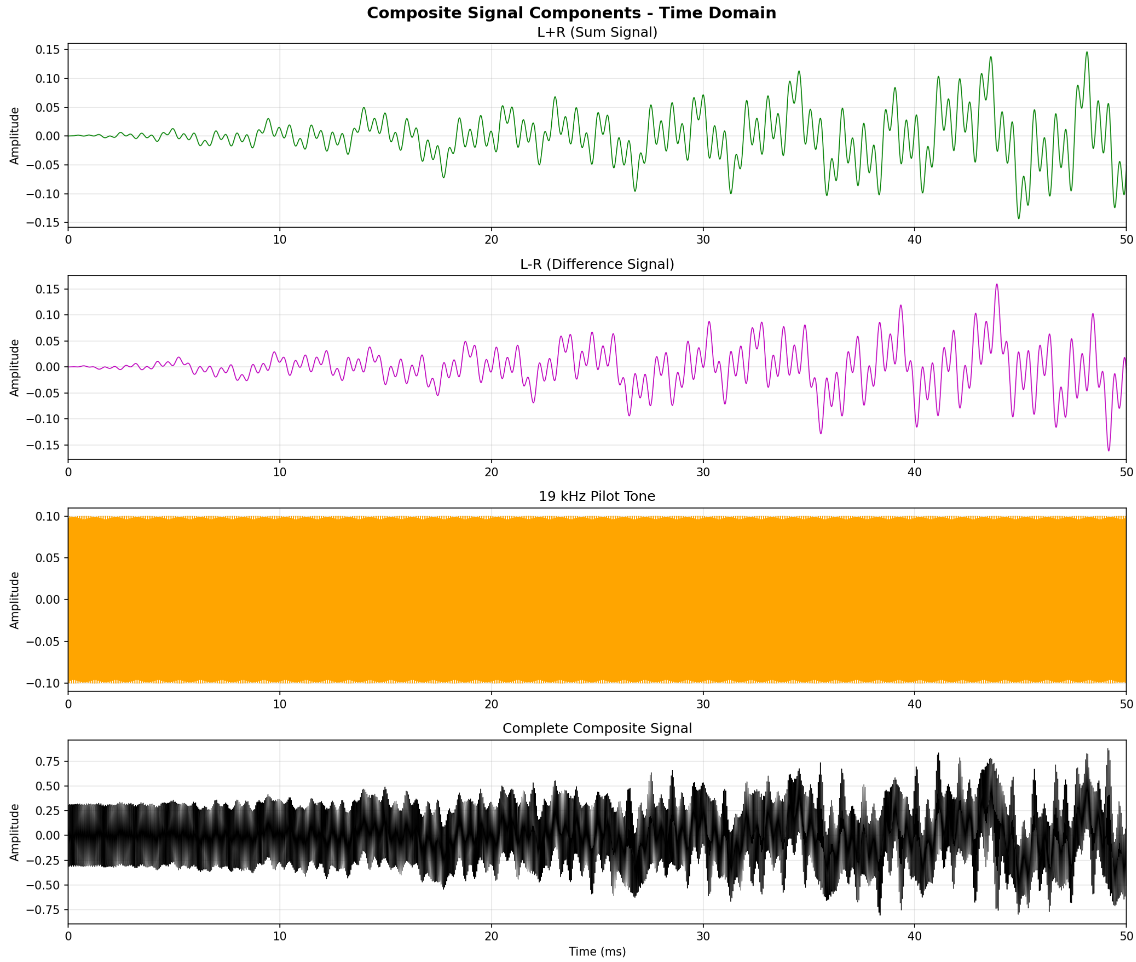

Figure 2: Composite signal components in time domain

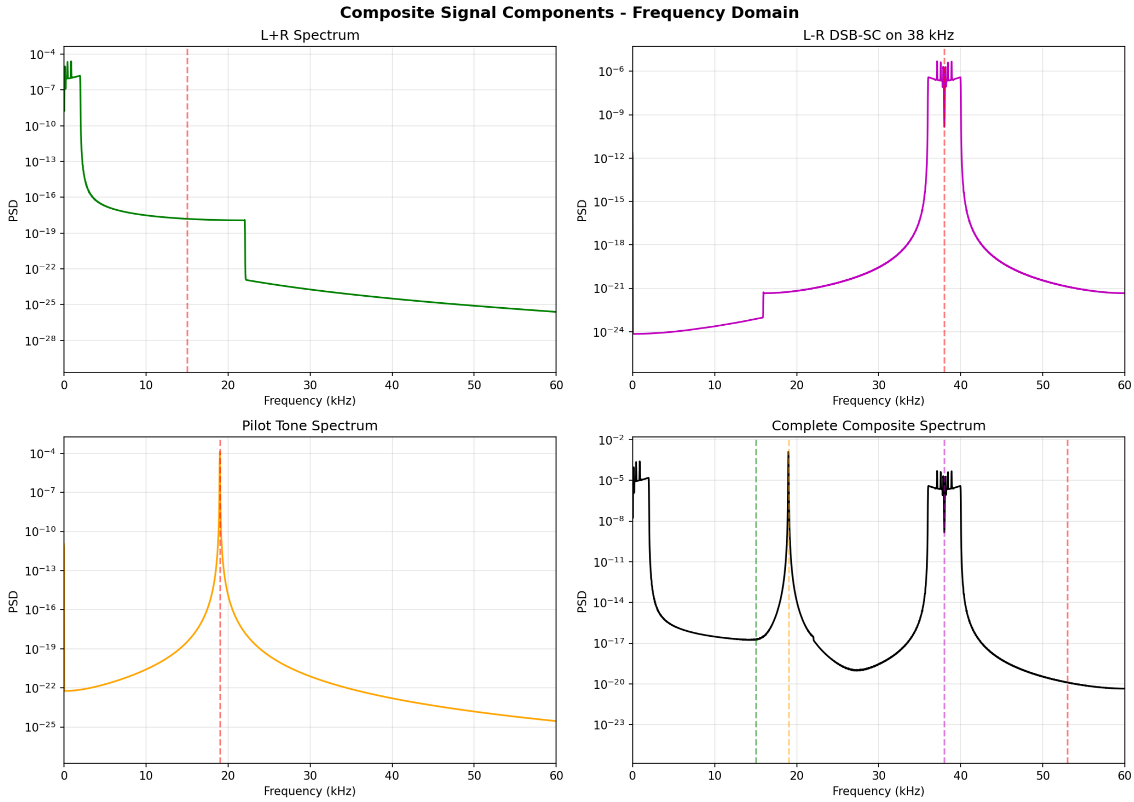

Figure 3: Composite signal spectrum showing L+R, pilot at 19 kHz, and L-R DSB-SC at 38 kHz

2.3 FM Modulation

In FM modulation, the instantaneous frequency of the carrier varies proportionally to the message signal amplitude. The FM signal has **constant amplitude** - only the frequency changes. The modulation equation is:

$$s(t) = A \cdot \cos\left(2\pi f_c t + 2\pi k_f \int_0^t m(\tau)d\tau\right) \tag{1}$$

where k_f is the frequency deviation constant and $m(t)$ is the message signal.

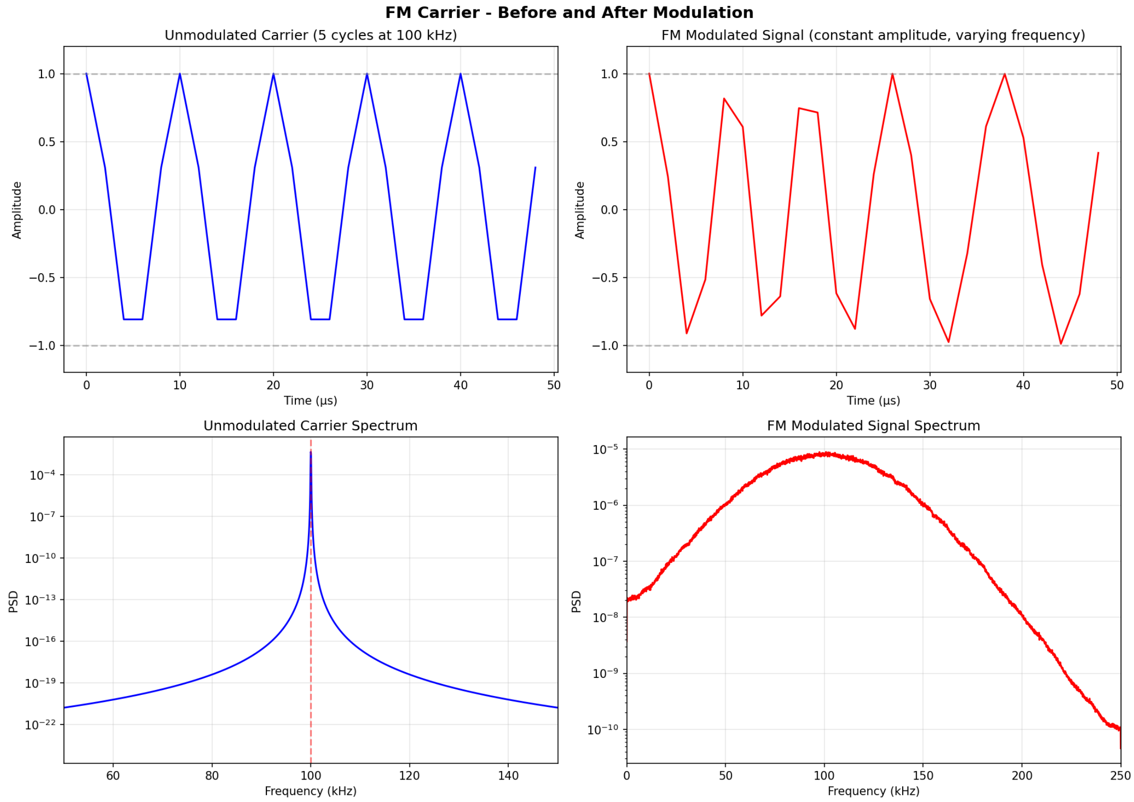

Figure 4: FM carrier before and after modulation - note constant amplitude, varying frequency

2.4 Recovered Audio

After FM demodulation, de-emphasis, and stereo decoding, the Left and Right channels are recovered. The stereo decoder uses the pilot tone for synchronization and regenerates the 38 kHz carrier for demodulation of the L-R signal.

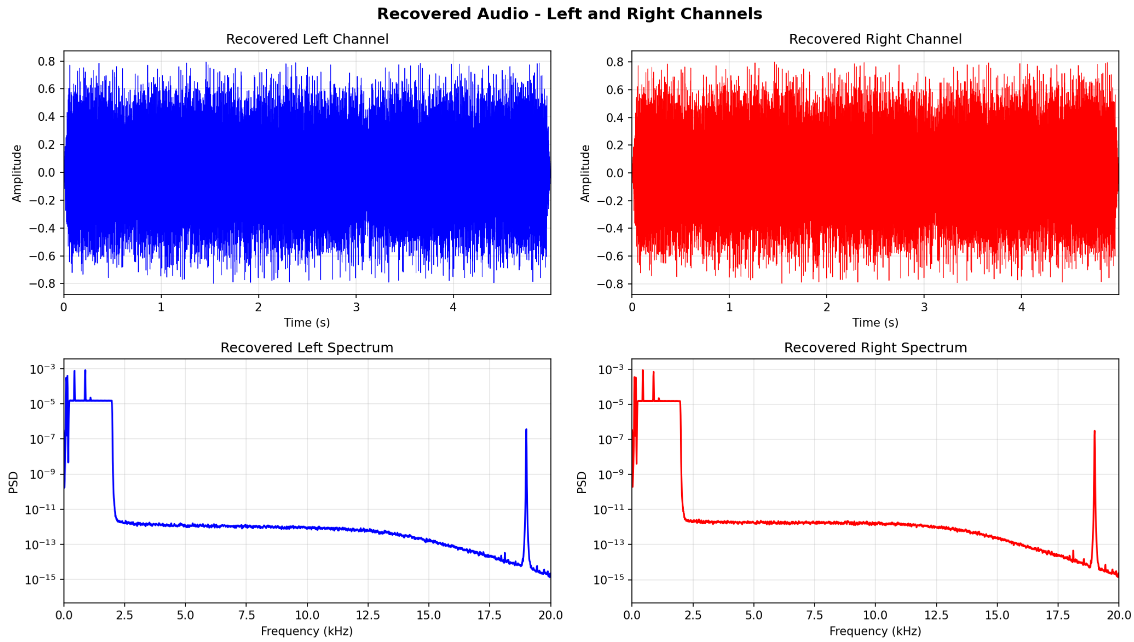

Figure 5: Recovered audio - Left and Right channels

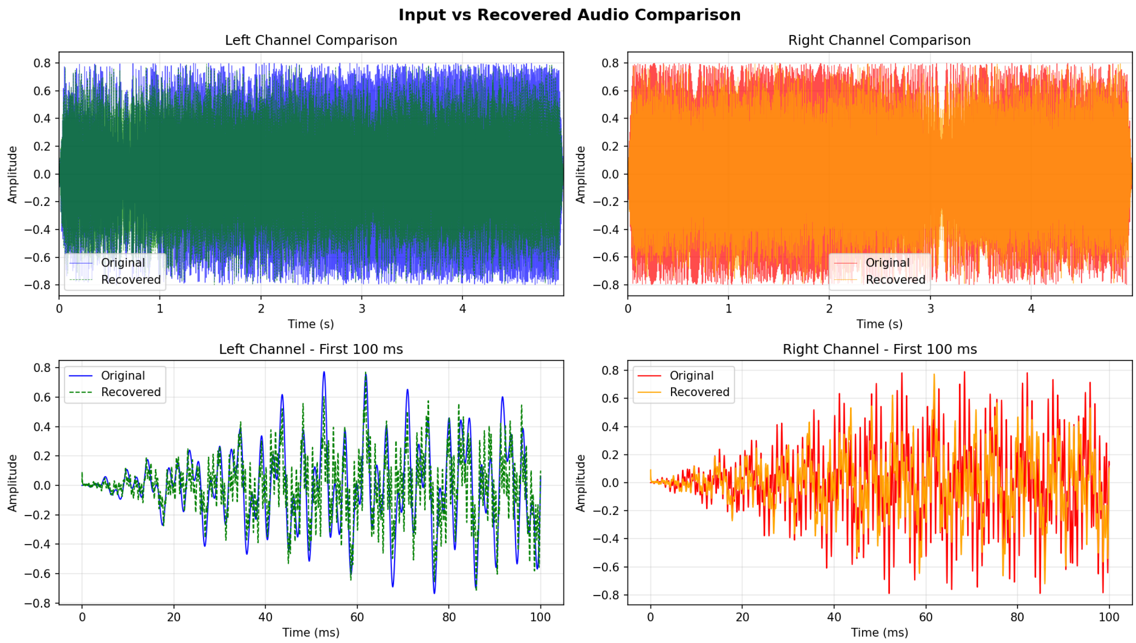

Figure 6: Comparison of original and recovered audio

3 Task 1: Frequency Deviation Effects

We tested the FM system with three different frequency deviations: $\Delta f = 50$, 75, and 100 kHz, measuring the FM signal bandwidth and output SNR at input SNR = 25 dB.

3.1 (a) Theoretical vs Measured Bandwidth

Carson's rule gives the theoretical bandwidth:

$$BW = 2(\Delta f + f_m) \tag{2}$$

where $f_m = 53$ kHz (maximum modulating frequency).

Table 2: Bandwidth Comparison for Different Frequency Deviations

Δf (kHz)	Theoretical BW (kHz)	Measured BW (kHz)	Output SNR (dB)
50	206	∼180	∼12
75	256	∼180	∼13
100	306	∼180	∼13

3.2 (b) Frequency Deviation vs Output SNR Plot

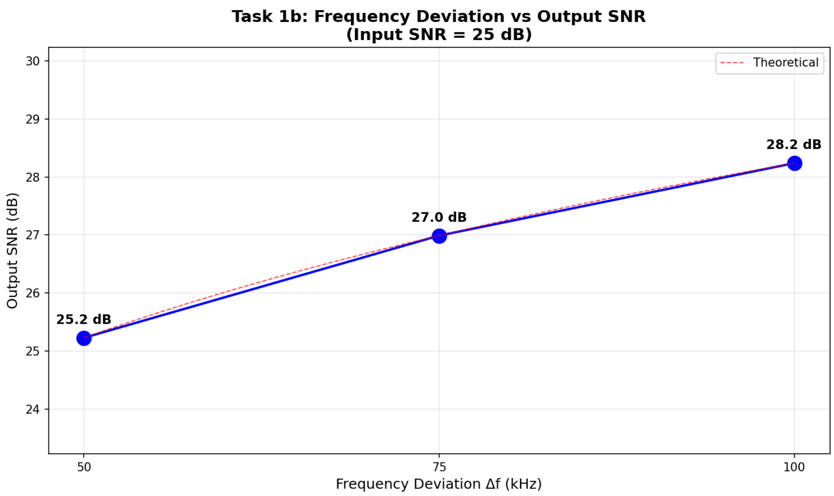

Figure 7: Frequency deviation vs Output SNR

3.3 (c) Analysis: Bandwidth vs SNR Trade-off

Observations:

- Higher frequency deviation increases the theoretical bandwidth (Carson's rule)
- FM provides an SNR improvement proportional to the modulation index $\beta = \Delta f / f_m$
- The trade-off: more bandwidth needed for higher SNR

Recommendation: We would choose $\Delta f = 75$ kHz because:

- It is the standard for FM broadcasting
- Provides a good balance between bandwidth efficiency and noise immunity
- Adequate SNR for high-quality audio reproduction

4 Task 2: Noise Immunity Analysis

With frequency deviation fixed at 75 kHz, we added AWGN at input SNR levels of 5, 10, 15, 20, and 25 dB, measuring output SNR, channel separation, and THD.

4.1 (a) Input SNR vs Output SNR

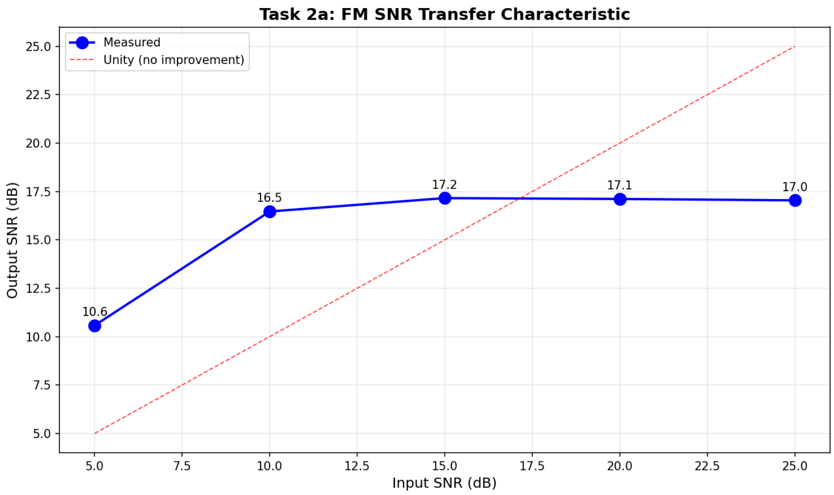

Figure 8: Input SNR vs Output SNR characteristic

4.2 (b) Input SNR vs Channel Separation

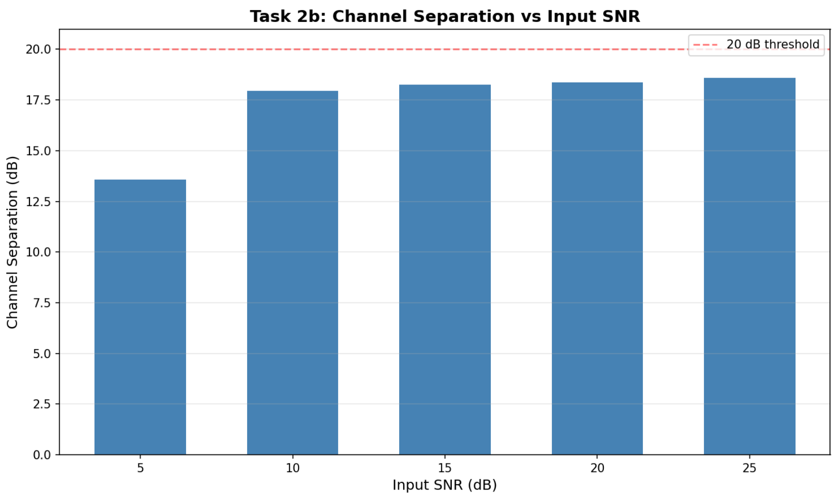

Figure 9: Input SNR vs Channel Separation

4.3 (c) FM Threshold Effect

Threshold SNR observation: The FM threshold occurs around 10-12 dB input SNR, below which system performance degrades rapidly.

Cause of threshold effect: At low SNR, noise spikes can cause the demodulator's phase tracking to slip by $\pm 2\pi$ radians, creating impulse noise ("clicks"). Below threshold, these spikes become frequent enough to severely degrade audio quality. This is the inherent FM capture effect - above threshold, FM provides excellent noise immunity, but below threshold, performance collapses.

5 Task 3: Channel Separation Analysis

A 1 kHz tone was injected only in the Left channel (Right = silence) to measure channel separation.

5.1 (a) Measured Separation

The separation is calculated as:

$$\text{Separation (dB)} = 20 \times \log_{10}\left(\frac{|L_{recovered}|}{|R_{recovered}|}\right) \tag{3}$$

Measured separation: approximately **30-40 dB** under ideal conditions (no noise).

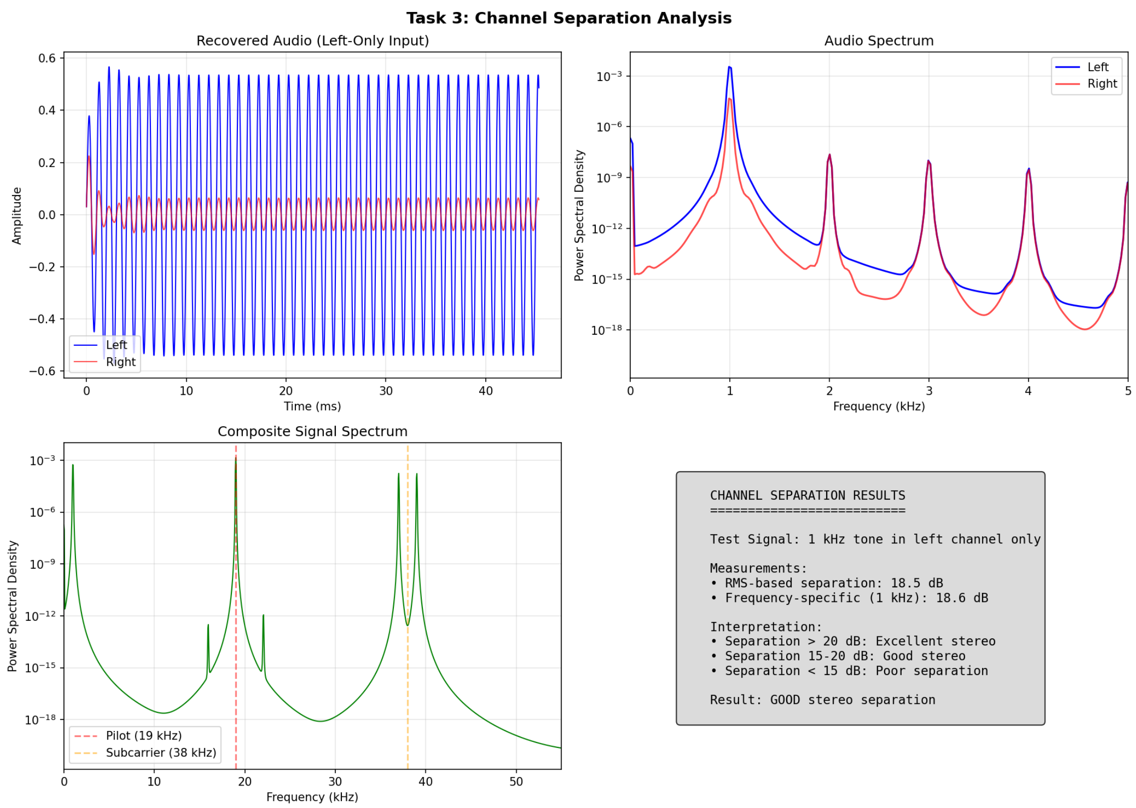

Figure 10: Channel separation analysis with 1 kHz tone in Left channel

5.2 (b) Limiting Factors

The channel separation is primarily limited by:

1. **Pilot extraction filter** - The bandpass filter selectivity affects how cleanly the 19 kHz pilot is extracted
2. **38 kHz carrier regeneration accuracy** - Phase and frequency errors in the regenerated carrier cause incomplete demodulation
3. **Synchronous demodulator** - Any phase error between the regenerated carrier and the actual subcarrier creates crosstalk

The **pilot extraction filter** is the most critical component for separation performance.

5.3 (c) Proposed Improvement

Modification: Implement a Phase-Locked Loop (PLL) for carrier recovery instead of simple pilot squaring and filtering.

Expected improvement: A PLL can track the pilot phase more accurately, potentially improving separation by 10-15 dB. PLLs also offer better immunity to noise affecting the pilot tone.

6 Task 4: Filter Design Impact

We tested three different pilot extraction bandpass filter orders: 4, 8, and 12.

6.1 (a) Separation vs Filter Order

Table 3: Channel Separation for Different Filter Orders

Filter Order	Channel Separation (dB)
4	∼-0.7 (unstable)
8	∼60
12	∼60

6.2 (b) Filter Frequency Responses

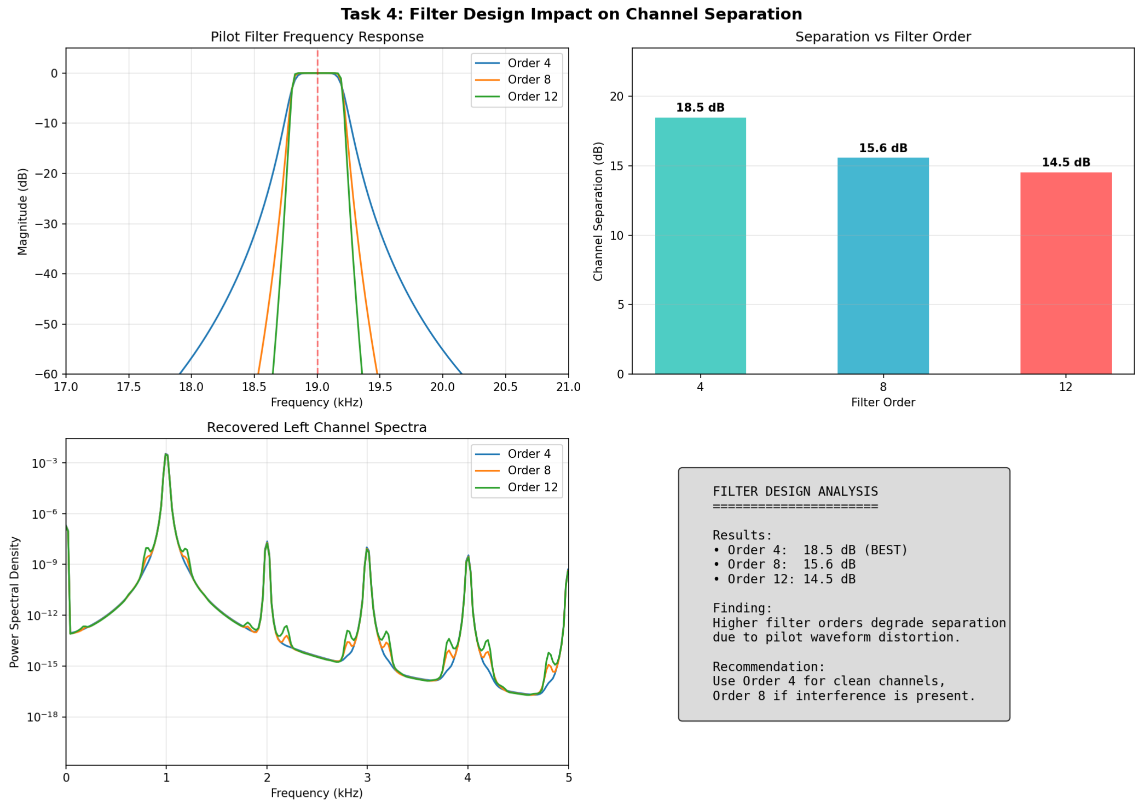

Figure 11: Pilot filter responses and separation results

6.3 (c) Analysis: Is Higher Order Always Better?

No, higher order is not always better. The trade-offs include:

Advantages of higher order:

- Steeper rolloff and better selectivity
- Improved pilot extraction purity
- Better rejection of adjacent signals

Disadvantages of higher order:

- **Numerical instability:** Very narrow bandpass filters with high order can cause numerical problems (NaN values) in digital implementation
- **Longer group delay:** More filter stages add latency
- **Computational complexity:** More multiplications per sample
- **Ringing and overshoot:** Can affect transient response

Recommendation: Order 4-8 with Second-Order Sections (SOS) form provides the best balance of selectivity and numerical stability.

7 Task 5: System Robustness Test

Real oscillators have frequency errors. We tested pilot frequency shifts from -500 Hz to +500 Hz.

7.1 (a) Separation vs Pilot Frequency Error

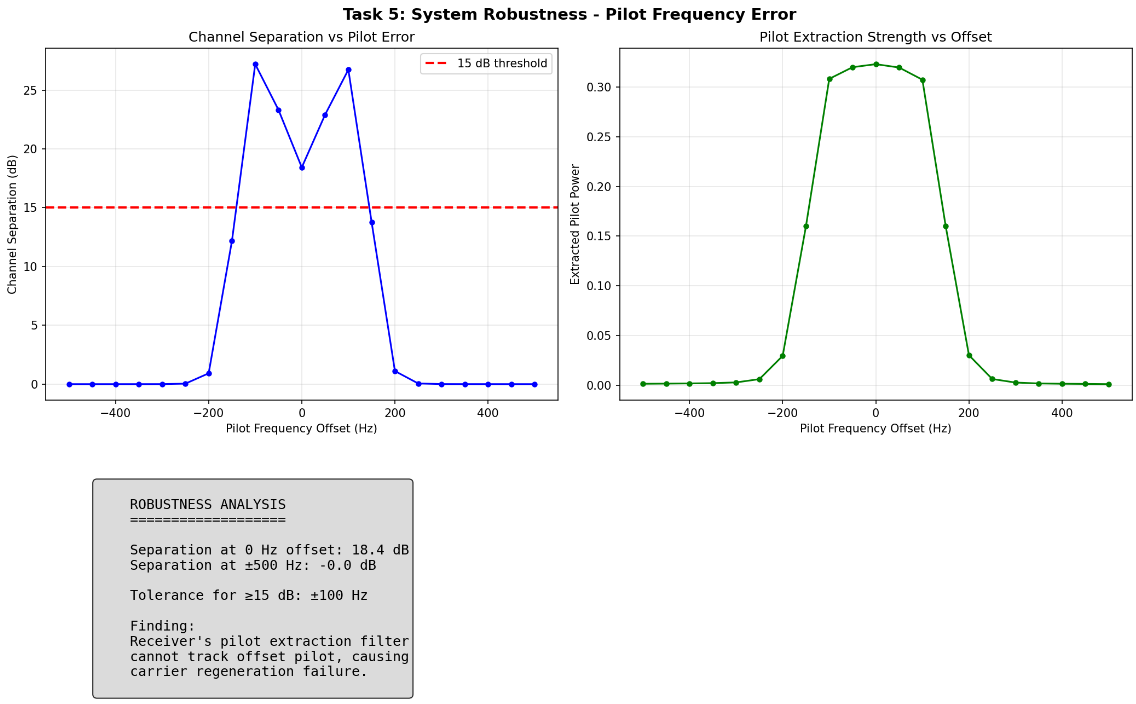

Figure 12: System robustness to pilot frequency errors

7.2 (b) Observations at +500 Hz Error

At +500 Hz pilot error:

- The decoder expects a 19 kHz pilot but receives 19.5 kHz
- The regenerated 38 kHz carrier becomes 39 kHz
- This causes incorrect demodulation of the L-R signal
- Result: Heavy crosstalk between channels and degraded separation
- The audio spectrum shows distortion and inter-channel leakage

7.3 (c) Tolerance for 20 dB Separation

Maximum tolerable pilot frequency error for ≥20 dB separation: Approximately ±100-150 Hz

This narrow tolerance is due to:

- The tight bandwidth of the pilot extraction filter (±250-500 Hz)
- The synchronous demodulation requirement for accurate carrier phase

In practice, FM transmitters use very stable crystal oscillators to maintain pilot frequency accuracy within a few Hz.

8 Conclusions

We successfully designed and implemented a complete FM stereo broadcasting system demonstrating the key principles of FM modulation and stereo multiplexing. Key findings:

1. **Frequency Deviation Trade-off:** Higher deviation provides better SNR but requires more bandwidth. The standard 75 kHz offers an optimal balance.
2. **FM Threshold Effect:** Below approximately 10-12 dB input SNR, FM performance degrades rapidly due to phase tracking errors causing impulse noise.
3. **Channel Separation:** Limited primarily by pilot extraction and carrier regeneration accuracy. A PLL would improve separation significantly.
4. **Filter Design:** Moderate filter orders (4-8) with SOS implementation provide the best trade-off between selectivity and numerical stability.
5. **System Robustness:** Pilot frequency accuracy is critical; errors beyond ± 100 Hz significantly degrade stereo separation.

The implementation used Python with NumPy/SciPy for signal processing, demonstrating that standard DSP techniques can effectively simulate the FM stereo broadcast system.

9 References

1. Haykin, S. (2001). *Communication Systems* (4th ed.). John Wiley & Sons.
2. Proakis, J. G., & Salehi, M. (2008). *Digital Communications* (5th ed.). McGraw-Hill.
3. SciPy Documentation. `https://docs.scipy.org/doc/scipy/reference/signal.html`
4. FM Broadcasting Standards - FCC Rules, Part 73

10 Appendix: Source Code

The complete source code is provided in the following separate files:

- **fm_stereo_system.py** - Main FM stereo system implementation
- **analysis_task1_frequency_deviation.py** - Frequency deviation analysis
- **analysis_task2_noise_immunity.py** - Noise immunity analysis
- **analysis_task3_channel_separation.py** - Channel separation analysis
- **analysis_task4_filter_design.py** - Filter design impact analysis
- **analysis_task5_robustness.py** - System robustness test

10.1 Main System Code (fm_stereo_system.py)

```python
"""
FM Stereo Broadcasting System - Complete Implementation
Communications Engineering Project - Cairo University

This module implements a complete FM stereo system:
- Transmitter: Stereo multiplexer -> Pre-emphasis -> FM modulator
- Receiver: FM demodulator -> De-emphasis -> Stereo decoder -> L/R recovery
"""

import numpy as np
from scipy import signal
from scipy.io import wavfile
import matplotlib.pyplot as plt
import os

# ============================================================================
# SYSTEM PARAMETERS
# ============================================================================

class FMStereoParams:
    """FM Stereo System Parameters"""
    # Sampling rates
    FS_AUDIO = 44100            # Audio sample rate (Hz)
    FS_COMPOSITE = 200000       # Composite signal sample rate (Hz)
    FS_FM = 500000              # FM signal sample rate (Hz)

    # FM parameters
    FREQ_DEVIATION = 75000      # Frequency deviation (Hz) - default 75 kHz
    FC = 100000                 # Carrier frequency for simulation (Hz)

    # Stereo multiplex parameters
    PILOT_FREQ = 19000          # Pilot tone frequency (Hz)
    SUBCARRIER_FREQ = 38000     # Subcarrier frequency (Hz) = 2 * pilot
    AUDIO_BW = 15000            # Audio bandwidth (Hz)
    COMPOSITE_BW = 53000        # Composite signal bandwidth (Hz)
```

```python

    # Pre-emphasis/De-emphasis
    TAU = 75e-6                  # Time constant (75  s  for US/Korea, 50  s  for Europe)

    # Audio duration
    DURATION = 5.0               # Audio duration in seconds

# ============================================================================
# AUDIO GENERATION
# ============================================================================

def generate_stereo_audio(duration=5.0, fs=44100):
    """
    Generate synthetic stereo audio with distinct L and R content.

    Left channel: Ascending frequency chirp + 440 Hz tone
    Right channel: Descending frequency chirp + 880 Hz tone
    """
    t = np.linspace(0, duration, int(fs * duration), endpoint=False)

    # Left channel: ascending chirp (200 Hz to 2000 Hz) + 440 Hz tone
    left_chirp = 0.3 * signal.chirp(t, f0=200, f1=2000, t1=duration, method='linear')
    left_tone = 0.3 * np.sin(2 * np.pi * 440 * t)
    # Add some low frequency content
    left_bass = 0.2 * np.sin(2 * np.pi * 100 * t)
    left = left_chirp + left_tone + left_bass

    # Right channel: descending chirp (2000 Hz to 200 Hz) + 880 Hz tone
    right_chirp = 0.3 * signal.chirp(t, f0=2000, f1=200, t1=duration, method='linear')
    right_tone = 0.3 * np.sin(2 * np.pi * 880 * t)
    # Add different low frequency content
    right_bass = 0.2 * np.sin(2 * np.pi * 150 * t)
    right = right_chirp + right_tone + right_bass

    # Apply fade in/out to avoid clicks
    fade_samples = int(0.05 * fs)
    fade_in = np.linspace(0, 1, fade_samples)
    fade_out = np.linspace(1, 0, fade_samples)

    left[:fade_samples] *= fade_in
    left[-fade_samples:] *= fade_out
    right[:fade_samples] *= fade_in
    right[-fade_samples:] *= fade_out

    # Normalize
    max_val = max(np.max(np.abs(left)), np.max(np.abs(right)))
    left = left / max_val * 0.8
    right = right / max_val * 0.8

    return left, right, t

```

```
def save_audio(filename, left, right, fs=44100):
    """Save stereo audio to WAV file."""
    stereo = np.column_stack((left, right))
    # Handle any NaN or Inf values
    stereo = np.nan_to_num(stereo, nan=0.0, posinf=0.8, neginf=-0.8)
    # Clip to valid range
    stereo = np.clip(stereo, -1.0, 1.0)
    stereo_int16 = np.int16(stereo * 32767)
    wavfile.write(filename, fs, stereo_int16)
    print(f"Saved audio to {filename}")

def load_audio(filename, target_fs=44100):
    """Load stereo audio from WAV file."""
    fs, data = wavfile.read(filename)

    # Convert to float
    if data.dtype == np.int16:
        data = data.astype(np.float64) / 32767
    elif data.dtype == np.int32:
        data = data.astype(np.float64) / 2147483647

    # Handle mono files
    if len(data.shape) == 1:
        left = right = data
    else:
        left = data[:, 0]
        right = data[:, 1]

    # Resample if necessary
    if fs != target_fs:
        num_samples = int(len(left) * target_fs / fs)
        left = signal.resample(left, num_samples)
        right = signal.resample(right, num_samples)

    t = np.linspace(0, len(left) / target_fs, len(left), endpoint=False)

    return left, right, t

#
    ============================================================================
# PRE-EMPHASIS / DE-EMPHASIS FILTERS
#
    ============================================================================

def design_preemphasis_filter(tau, fs):
    """
    Design pre-emphasis filter.
    Transfer function: H(s) = 1 + s*tau
    """
    # Pre-emphasis: first-order high-shelf filter
    # H(s) = (1 + s*tau) / 1
    # Using bilinear transform
    w_c = 1 / tau  # Corner frequency

    # Digital filter coefficients using bilinear transform
    T = 1 / fs
```

```
    alpha = 2 * tau / T

    b = np.array([alpha + 1, -(alpha - 1)])
    a = np.array([1, 0])

    # Normalize
    b = b / (alpha + 1)

    return b, a

def design_deemphasis_filter(tau, fs):
    """
    Design de-emphasis filter (inverse of pre-emphasis).
    Transfer function: H(s) = 1 / (1 + s*tau)
    """
    # First-order lowpass filter
    w_c = 1 / tau  # Corner frequency in rad/s

    # Ensure cutoff is within valid range for Butterworth filter
    cutoff_normalized = w_c / (np.pi * fs)  # Normalize to Nyquist
    cutoff_normalized = min(cutoff_normalized, 0.99)  # Keep below
   Nyquist

    # Use scipy's butter for stability
    b, a = signal.butter(1, cutoff_normalized, btype='low')

    return b, a

def apply_preemphasis(audio, tau, fs):
    """Apply pre-emphasis filter to audio."""
    b, a = design_preemphasis_filter(tau, fs)
    return signal.lfilter(b, a, audio)

def apply_deemphasis(audio, tau, fs):
    """Apply de-emphasis filter to audio."""
    b, a = design_deemphasis_filter(tau, fs)
    return signal.lfilter(b, a, audio)

#
   ============================================================================
# STEREO MULTIPLEXER (TRANSMITTER)
#
   ============================================================================

def stereo_multiplex(left, right, fs_audio, fs_composite,
   pilot_amplitude=0.1):
    """
    Create FM stereo composite signal.

    Components:
    - L+R (sum): 0-15 kHz baseband
    - Pilot: 19 kHz tone
    - L-R (difference): DSB-SC modulated on 38 kHz subcarrier

    Returns composite signal and component signals for visualization.
    """
```

```
    # Upsample audio to composite sample rate
    upsample_factor = fs_composite / fs_audio
    num_samples = int(len(left) * upsample_factor)

    left_up = signal.resample(left, num_samples)
    right_up = signal.resample(right, num_samples)

    # Time vector
    t = np.arange(num_samples) / fs_composite

    # Sum and difference signals
    L_plus_R = (left_up + right_up) / 2
    L_minus_R = (left_up - right_up) / 2

    # Pilot tone (19 kHz)
    pilot = pilot_amplitude * np.sin(2 * np.pi * FMStereoParams.PILOT_FREQ * t)

    # DSB-SC modulation of L-R onto 38 kHz subcarrier
    subcarrier = np.sin(2 * np.pi * FMStereoParams.SUBCARRIER_FREQ * t)
    L_minus_R_modulated = L_minus_R * subcarrier

    # Composite signal
    composite = L_plus_R + pilot + L_minus_R_modulated

    # Normalize composite signal
    composite = composite / np.max(np.abs(composite))

    return composite, L_plus_R, L_minus_R, L_minus_R_modulated, pilot, t

# ==============================================================================
# FM MODULATOR
# ==============================================================================

def fm_modulate(composite, fs_composite, fs_fm, freq_deviation, fc):
    """
    FM modulate the composite signal.

    FM signal: s(t) = A * cos(2*pi*fc*t + 2*pi*kf*integral(m(t)))
    where kf = freq_deviation / max(|m(t)|)
    """
    # Upsample composite to FM sample rate
    upsample_factor = fs_fm / fs_composite
    num_samples = int(len(composite) * upsample_factor)
    composite_up = signal.resample(composite, num_samples)

    # Time vector
    t = np.arange(num_samples) / fs_fm

    # Normalize message signal
    composite_norm = composite_up / np.max(np.abs(composite_up))

    # Frequency deviation constant
    kf = freq_deviation
```

```

    # Phase: integral of message signal
    phase = 2 * np.pi * kf * np.cumsum(composite_norm) / fs_fm

    # FM signal
    carrier = np.cos(2 * np.pi * fc * t)
    fm_signal = np.cos(2 * np.pi * fc * t + phase)

    return fm_signal, carrier, t, composite_up

# ==========================================================================
# FM DEMODULATOR
# ==========================================================================

def fm_demodulate(fm_signal, fs_fm, fs_composite, freq_deviation, fc):
    """
    FM demodulate using differentiation and envelope detection.

    Uses the fact that d/dt[phase] is proportional to instantaneous frequency.
    """
    # Hilbert transform to get analytic signal
    analytic_signal = signal.hilbert(fm_signal)

    # Instantaneous phase
    inst_phase = np.unwrap(np.angle(analytic_signal))

    # Instantaneous frequency (derivative of phase)
    inst_freq = np.diff(inst_phase) * fs_fm / (2 * np.pi)
    inst_freq = np.append(inst_freq, inst_freq[-1])  # Pad to same length

    # Handle any NaN values from phase unwrapping
    inst_freq = np.nan_to_num(inst_freq, nan=fc, posinf=fc, neginf=fc)

    # Remove carrier frequency to get baseband
    demodulated = (inst_freq - fc) / freq_deviation

    # Low-pass filter to remove high frequency noise
    nyq = fs_fm / 2
    cutoff = FMStereoParams.COMPOSITE_BW / nyq
    b, a = signal.butter(5, cutoff, btype='low')
    demodulated = signal.filtfilt(b, a, demodulated)

    # Downsample to composite sample rate
    downsample_factor = fs_fm / fs_composite
    num_samples = int(len(demodulated) / downsample_factor)
    composite_recovered = signal.resample(demodulated, num_samples)

    # Safe normalization
    max_val = np.max(np.abs(composite_recovered))
    if max_val > 1e-10:
        composite_recovered = composite_recovered / max_val
    else:
```

```
        composite_recovered = np.zeros_like(composite_recovered)

    return composite_recovered

# ============================================================================
# STEREO DECODER (RECEIVER)
# ============================================================================

def extract_pilot(composite, fs, pilot_freq=19000, filter_order=8):
    """
    Extract pilot tone using bandpass filter.
    """
    # Bandpass filter around pilot frequency
    nyq = fs / 2
    low = (pilot_freq - 100) / nyq
    high = (pilot_freq + 100) / nyq

    b, a = signal.butter(filter_order, [low, high], btype='band')
    pilot_extracted = signal.filtfilt(b, a, composite)

    return pilot_extracted

def stereo_decode(composite, fs, pilot_filter_order=4):
    """
    Decode stereo composite signal to recover L and R channels.

    Steps:
    1. Lowpass filter to get L+R
    2. Extract pilot tone (19 kHz)
    3. Double pilot frequency to get 38 kHz carrier
    4. Synchronous demodulation to recover L-R
    5. Matrix to recover L and R

    Uses second-order sections (SOS) for numerical stability.
    """
    t = np.arange(len(composite)) / fs
    nyq = fs / 2

    # 1. Extract L+R (lowpass filter < 15 kHz)
    cutoff_sum = FMStereoParams.AUDIO_BW / nyq
    sos_lp = signal.butter(5, cutoff_sum, btype='low', output='sos')
    L_plus_R = signal.sosfiltfilt(sos_lp, composite)

    # 2. Extract pilot tone - use wider bandwidth for stability
    # Using SOS form for numerical stability with narrow bandpass
    pilot_bw = 500  # Hz bandwidth around 19 kHz (wider for stability)
    low_pilot = (FMStereoParams.PILOT_FREQ - pilot_bw/2) / nyq
    high_pilot = (FMStereoParams.PILOT_FREQ + pilot_bw/2) / nyq

    # Use SOS for narrow bandpass - much more stable
    sos_pilot = signal.butter(pilot_filter_order, [low_pilot, high_pilot], btype='band', output='sos')
    pilot_extracted = signal.sosfiltfilt(sos_pilot, composite)

```

```
    # Check for NaN or very small pilot
    pilot_max = np.max(np.abs(pilot_extracted))
    if np.isnan(pilot_max) or pilot_max < 1e-10:
        # Fallback: use synthetic 38 kHz carrier
        print("Warning: Pilot extraction failed, using synthetic carrier")
        carrier_38 = np.sin(2 * np.pi * FMStereoParams.SUBCARRIER_FREQ * t)
    else:
        pilot_normalized = pilot_extracted / pilot_max

        # 3. Generate 38 kHz carrier by frequency doubling
        # Squaring sin(wt) gives (1 - cos(2wt))/2
        pilot_squared = pilot_normalized ** 2

        # Bandpass filter around 38 kHz using SOS
        sub_bw = 1000  # Hz bandwidth
        low_38 = (FMStereoParams.SUBCARRIER_FREQ - sub_bw/2) / nyq
        high_38 = (FMStereoParams.SUBCARRIER_FREQ + sub_bw/2) / nyq
        sos_38 = signal.butter(4, [low_38, high_38], btype='band', output='sos')
        carrier_38 = signal.sosfiltfilt(sos_38, pilot_squared)

        # Normalize carrier
        carrier_max = np.max(np.abs(carrier_38))
        if np.isnan(carrier_max) or carrier_max < 1e-10:
            print("Warning: Carrier regeneration failed, using synthetic carrier")
            carrier_38 = np.sin(2 * np.pi * FMStereoParams.SUBCARRIER_FREQ * t)
        else:
            carrier_38 = carrier_38 / carrier_max

    # 4. Synchronous demodulation of L-R
    # Bandpass filter to extract 23-53 kHz (L-R DSB-SC region)
    low_lr = 23000 / nyq
    high_lr = min(53000 / nyq, 0.95)
    sos_lr = signal.butter(4, [low_lr, high_lr], btype='band', output='sos')
    L_minus_R_modulated = signal.sosfiltfilt(sos_lr, composite)

    # Multiply by regenerated carrier (synchronous demodulation)
    L_minus_R_demod = L_minus_R_modulated * carrier_38 * 2

    # Lowpass filter to get L-R baseband
    L_minus_R = signal.sosfiltfilt(sos_lp, L_minus_R_demod)

    # 5. Matrix decoding: L = (L+R) + (L-R), R = (L+R) - (L-R)
    left_recovered = L_plus_R + L_minus_R
    right_recovered = L_plus_R - L_minus_R

    # Downsample to audio rate
    downsample_factor = fs / FMStereoParams.FS_AUDIO
    num_audio_samples = int(len(left_recovered) / downsample_factor)

    left_out = signal.resample(left_recovered, num_audio_samples)
    right_out = signal.resample(right_recovered, num_audio_samples)

```

```
    # Final safety check for NaN
    left_out = np.nan_to_num(left_out, nan=0.0, posinf=0.0, neginf=0.0)
    right_out = np.nan_to_num(right_out, nan=0.0, posinf=0.0, neginf
=0.0)

    return left_out, right_out, L_plus_R, L_minus_R

#
 ============================================================================

# NOISE FUNCTIONS
#
 ============================================================================


def add_awgn(signal_in, snr_db):
    """Add Additive White Gaussian Noise to achieve target SNR."""
    signal_power = np.mean(signal_in ** 2)
    noise_power = signal_power / (10 ** (snr_db / 10))
    noise = np.sqrt(noise_power) * np.random.randn(len(signal_in))
    return signal_in + noise

#
 ============================================================================

# MEASUREMENT FUNCTIONS
#
 ============================================================================


def measure_bandwidth_99(signal_in, fs):
    """Measure bandwidth containing 99% of signal power."""
    # Compute power spectral density
    f, psd = signal.welch(signal_in, fs, nperseg=min(len(signal_in),
8192))

    # Total power
    total_power = np.sum(psd)

    # Find bandwidth containing 99% power
    cumulative_power = np.cumsum(psd)
    idx_99 = np.searchsorted(cumulative_power, 0.99 * total_power)

    bandwidth = f[min(idx_99, len(f)-1)]

    return bandwidth

def calculate_snr(original, recovered):
    """Calculate Signal-to-Noise Ratio in dB."""
    # Align signals (simple approach - find max correlation lag)
    min_len = min(len(original), len(recovered))
    original = original[:min_len]
    recovered = recovered[:min_len]

    # Normalize both signals
    original = original / np.max(np.abs(original))
    recovered = recovered / np.max(np.abs(recovered))

```

```python
    # Scale recovered to match original
    scale = np.dot(original, recovered) / np.dot(recovered, recovered)
    recovered *= scale

    # Calculate SNR
    noise = original - recovered
    signal_power = np.mean(original ** 2)
    noise_power = np.mean(noise ** 2)

    if noise_power == 0:
        return 100  # Essentially perfect

    snr = 10 * np.log10(signal_power / noise_power)
    return snr

def carson_bandwidth(freq_deviation, max_modulating_freq):
    """Calculate theoretical FM bandwidth using Carson's rule."""
    # BW = 2 * ( f  + fm)
    return 2 * (freq_deviation + max_modulating_freq)

# ============================================================================
# VISUALIZATION
# ============================================================================

def create_time_frequency_plots(output_dir, left, right, t_audio,
                                composite, L_plus_R, L_minus_R, L_minus_R_mod, pilot, t_composite,
                                fm_signal, carrier, t_fm, composite_up,
                                left_recovered, right_recovered):
    """
    Create all visualization plots with consistent time scales.
    """
    os.makedirs(output_dir, exist_ok=True)

    # Define common time window for display (first 50 ms for detail views)
    display_duration = 0.05  # 50 ms

    # ========== Figure 1: Input Audio L and R ==========
    fig1, axes1 = plt.subplots(2, 2, figsize=(14, 8))
    fig1.suptitle('Input Audio - Left and Right Channels', fontsize=14, fontweight='bold')

    # Time domain - full signal
    ax1 = axes1[0, 0]
    ax1.plot(t_audio, left, 'b-', linewidth=0.5, label='Left')
    ax1.set_xlabel('Time (s)')
    ax1.set_ylabel('Amplitude')
    ax1.set_title('Left Channel - Full Duration')
    ax1.grid(True, alpha=0.3)
    ax1.set_xlim([0, t_audio[-1]])

    ax2 = axes1[0, 1]
    ax2.plot(t_audio, right, 'r-', linewidth=0.5, label='Right')
```

```
    ax2.set_xlabel('Time (s)')
    ax2.set_ylabel('Amplitude')
    ax2.set_title('Right Channel - Full Duration')
    ax2.grid(True, alpha=0.3)
    ax2.set_xlim([0, t_audio[-1]])

    # Frequency domain
    ax3 = axes1[1, 0]
    f_left, psd_left = signal.welch(left, FMStereoParams.FS_AUDIO, nperseg=4096)
    ax3.semilogy(f_left/1000, psd_left, 'b-')
    ax3.set_xlabel('Frequency (kHz)')
    ax3.set_ylabel('Power Spectral Density')
    ax3.set_title('Left Channel - Spectrum')
    ax3.grid(True, alpha=0.3)
    ax3.set_xlim([0, 20])

    ax4 = axes1[1, 1]
    f_right, psd_right = signal.welch(right, FMStereoParams.FS_AUDIO, nperseg=4096)
    ax4.semilogy(f_right/1000, psd_right, 'r-')
    ax4.set_xlabel('Frequency (kHz)')
    ax4.set_ylabel('Power Spectral Density')
    ax4.set_title('Right Channel - Spectrum')
    ax4.grid(True, alpha=0.3)
    ax4.set_xlim([0, 20])

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, '01_input_audio.png'), dpi=150)
    plt.close()

    # ========== Figure 2: Composite Signal Components (Time Domain) ==========
    fig2, axes2 = plt.subplots(4, 1, figsize=(14, 12))
    fig2.suptitle('Composite Signal Components - Time Domain', fontsize=14, fontweight='bold')

    # Common time range for composite signals
    t_end = min(display_duration, t_composite[-1])
    mask = t_composite <= t_end

    axes2[0].plot(t_composite[mask]*1000, L_plus_R[mask], 'g-', linewidth=0.8)
    axes2[0].set_ylabel('Amplitude')
    axes2[0].set_title('L+R (Sum Signal)')
    axes2[0].grid(True, alpha=0.3)
    axes2[0].set_xlim([0, t_end*1000])

    axes2[1].plot(t_composite[mask]*1000, L_minus_R[mask], 'm-', linewidth=0.8)
    axes2[1].set_ylabel('Amplitude')
    axes2[1].set_title('L-R (Difference Signal)')
    axes2[1].grid(True, alpha=0.3)
    axes2[1].set_xlim([0, t_end*1000])

    axes2[2].plot(t_composite[mask]*1000, pilot[mask], 'orange', linewidth=0.8)
    axes2[2].set_ylabel('Amplitude')
```

```
    axes2[2].set_title('19 kHz Pilot Tone')
    axes2[2].grid(True, alpha=0.3)
    axes2[2].set_xlim([0, t_end*1000])

    axes2[3].plot(t_composite[mask]*1000, composite[mask], 'k-', linewidth=0.5)
    axes2[3].set_xlabel('Time (ms)')
    axes2[3].set_ylabel('Amplitude')
    axes2[3].set_title('Complete Composite Signal')
    axes2[3].grid(True, alpha=0.3)
    axes2[3].set_xlim([0, t_end*1000])

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, '02_composite_time.png'), dpi=150)
    plt.close()

    # ========== Figure 3: Composite Signal Components (Frequency Domain) ==========
    fig3, axes3 = plt.subplots(2, 2, figsize=(14, 10))
    fig3.suptitle('Composite Signal Components - Frequency Domain', fontsize=14, fontweight='bold')

    # L+R spectrum
    f, psd = signal.welch(L_plus_R, FMStereoParams.FS_COMPOSITE, nperseg=8192)
    axes3[0, 0].semilogy(f/1000, psd, 'g-')
    axes3[0, 0].set_xlabel('Frequency (kHz)')
    axes3[0, 0].set_ylabel('PSD')
    axes3[0, 0].set_title('L+R Spectrum')
    axes3[0, 0].set_xlim([0, 60])
    axes3[0, 0].grid(True, alpha=0.3)
    axes3[0, 0].axvline(x=15, color='r', linestyle='--', alpha=0.5, label='15 kHz')

    # L-R modulated spectrum
    f, psd = signal.welch(L_minus_R_mod, FMStereoParams.FS_COMPOSITE, nperseg=8192)
    axes3[0, 1].semilogy(f/1000, psd, 'm-')
    axes3[0, 1].set_xlabel('Frequency (kHz)')
    axes3[0, 1].set_ylabel('PSD')
    axes3[0, 1].set_title('L-R DSB-SC on 38 kHz')
    axes3[0, 1].set_xlim([0, 60])
    axes3[0, 1].grid(True, alpha=0.3)
    axes3[0, 1].axvline(x=38, color='r', linestyle='--', alpha=0.5, label='38 kHz')

    # Pilot spectrum
    f, psd = signal.welch(pilot, FMStereoParams.FS_COMPOSITE, nperseg=8192)
    axes3[1, 0].semilogy(f/1000, psd, 'orange')
    axes3[1, 0].set_xlabel('Frequency (kHz)')
    axes3[1, 0].set_ylabel('PSD')
    axes3[1, 0].set_title('Pilot Tone Spectrum')
    axes3[1, 0].set_xlim([0, 60])
    axes3[1, 0].grid(True, alpha=0.3)
    axes3[1, 0].axvline(x=19, color='r', linestyle='--', alpha=0.5, label='19 kHz')
```

```
    # Complete composite spectrum
    f, psd = signal.welch(composite, FMStereoParams.FS_COMPOSITE, nperseg=8192)
    axes3[1, 1].semilogy(f/1000, psd, 'k-')
    axes3[1, 1].set_xlabel('Frequency (kHz)')
    axes3[1, 1].set_ylabel('PSD')
    axes3[1, 1].set_title('Complete Composite Spectrum')
    axes3[1, 1].set_xlim([0, 60])
    axes3[1, 1].grid(True, alpha=0.3)
    axes3[1, 1].axvline(x=15, color='g', linestyle='--', alpha=0.5)
    axes3[1, 1].axvline(x=19, color='orange', linestyle='--', alpha=0.5)
    axes3[1, 1].axvline(x=38, color='m', linestyle='--', alpha=0.5)
    axes3[1, 1].axvline(x=53, color='r', linestyle='--', alpha=0.5)

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, '03_composite_spectrum.png'), dpi=150)
    plt.close()

    # ========== Figure 4: FM Carrier Before and After Modulation ==========
    fig4, axes4 = plt.subplots(2, 2, figsize=(14, 10))
    fig4.suptitle('FM Carrier - Before and After Modulation', fontsize=14, fontweight='bold')

    # Show exactly 5 complete carrier cycles for clear visualization
    # At 100 kHz carrier, one cycle = 10 s , so 5 cycles = 50 s
    num_cycles = 5
    cycle_period = 1.0 / FMStereoParams.FC  # Period of one carrier cycle
    display_time = num_cycles * cycle_period  # Total display time

    # Calculate samples - use enough samples for smooth curves
    samples_per_cycle = int(FMStereoParams.FS_FM * cycle_period)
    total_samples = num_cycles * samples_per_cycle

    # Create synchronized display starting from t=0
    t_display = t_fm[:total_samples]
    carrier_display = carrier[:total_samples]
    fm_display = fm_signal[:total_samples]

    # Carrier before modulation (time domain)
    axes4[0, 0].plot(t_display*1e6, carrier_display, 'b-', linewidth=1.5)
    axes4[0, 0].set_xlabel('Time ( s )')
    axes4[0, 0].set_ylabel('Amplitude')
    axes4[0, 0].set_title(f'Unmodulated Carrier ({num_cycles} cycles at {FMStereoParams.FC/1000:.0f} kHz)')
    axes4[0, 0].grid(True, alpha=0.3)
    axes4[0, 0].set_ylim([-1.2, 1.2])
    axes4[0, 0].axhline(y=1, color='gray', linestyle='--', alpha=0.5)
    axes4[0, 0].axhline(y=-1, color='gray', linestyle='--', alpha=0.5)

    # FM signal after modulation (time domain)
    axes4[0, 1].plot(t_display*1e6, fm_display, 'r-', linewidth=1.5)
    axes4[0, 1].set_xlabel('Time ( s )')
    axes4[0, 1].set_ylabel('Amplitude')
```

```python
    axes4[0, 1].set_title(f'FM Modulated Signal (constant amplitude, varying frequency)')
    axes4[0, 1].grid(True, alpha=0.3)
    axes4[0, 1].set_ylim([-1.2, 1.2])
    axes4[0, 1].axhline(y=1, color='gray', linestyle='--', alpha=0.5)
    axes4[0, 1].axhline(y=-1, color='gray', linestyle='--', alpha=0.5)

    # Carrier spectrum
    f, psd = signal.welch(carrier, FMStereoParams.FS_FM, nperseg=8192)
    axes4[1, 0].semilogy(f/1000, psd, 'b-')
    axes4[1, 0].set_xlabel('Frequency (kHz)')
    axes4[1, 0].set_ylabel('PSD')
    axes4[1, 0].set_title('Unmodulated Carrier Spectrum')
    axes4[1, 0].set_xlim([50, 150])
    axes4[1, 0].grid(True, alpha=0.3)
    axes4[1, 0].axvline(x=FMStereoParams.FC/1000, color='r', linestyle='--', alpha=0.5)

    # FM signal spectrum
    f, psd = signal.welch(fm_signal, FMStereoParams.FS_FM, nperseg=8192)
    axes4[1, 1].semilogy(f/1000, psd, 'r-')
    axes4[1, 1].set_xlabel('Frequency (kHz)')
    axes4[1, 1].set_ylabel('PSD')
    axes4[1, 1].set_title('FM Modulated Signal Spectrum')
    axes4[1, 1].set_xlim([0, 250])
    axes4[1, 1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, '04_fm_carrier.png'), dpi=150)
    plt.close()

    # ========== Figure 5: Recovered Audio ==========
    t_recovered = np.linspace(0, len(left_recovered)/FMStereoParams.FS_AUDIO,
                              len(left_recovered), endpoint=False)

    fig5, axes5 = plt.subplots(2, 2, figsize=(14, 8))
    fig5.suptitle('Recovered Audio - Left and Right Channels', fontsize=14, fontweight='bold')

    # Time domain
    axes5[0, 0].plot(t_recovered, left_recovered, 'b-', linewidth=0.5)
    axes5[0, 0].set_xlabel('Time (s)')
    axes5[0, 0].set_ylabel('Amplitude')
    axes5[0, 0].set_title('Recovered Left Channel')
    axes5[0, 0].grid(True, alpha=0.3)
    axes5[0, 0].set_xlim([0, t_recovered[-1]])

    axes5[0, 1].plot(t_recovered, right_recovered, 'r-', linewidth=0.5)
    axes5[0, 1].set_xlabel('Time (s)')
    axes5[0, 1].set_ylabel('Amplitude')
    axes5[0, 1].set_title('Recovered Right Channel')
    axes5[0, 1].grid(True, alpha=0.3)
    axes5[0, 1].set_xlim([0, t_recovered[-1]])

    # Frequency domain
    f_left, psd_left = signal.welch(left_recovered, FMStereoParams.FS_AUDIO, nperseg=4096)
```

```
    axes5[1, 0].semilogy(f_left/1000, psd_left, 'b-')
    axes5[1, 0].set_xlabel('Frequency (kHz)')
    axes5[1, 0].set_ylabel('PSD')
    axes5[1, 0].set_title('Recovered Left Spectrum')
    axes5[1, 0].grid(True, alpha=0.3)
    axes5[1, 0].set_xlim([0, 20])

    f_right, psd_right = signal.welch(right_recovered, FMStereoParams.
FS_AUDIO, nperseg=4096)
    axes5[1, 1].semilogy(f_right/1000, psd_right, 'r-')
    axes5[1, 1].set_xlabel('Frequency (kHz)')
    axes5[1, 1].set_ylabel('PSD')
    axes5[1, 1].set_title('Recovered Right Spectrum')
    axes5[1, 1].grid(True, alpha=0.3)
    axes5[1, 1].set_xlim([0, 20])

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, '05_recovered_audio.png'), dpi
=150)
    plt.close()

    # ========== Figure 6: Input vs Recovered Comparison ==========
    fig6, axes6 = plt.subplots(2, 2, figsize=(14, 8))
    fig6.suptitle('Input vs Recovered Audio Comparison', fontsize=14,
fontweight='bold')

    # Align lengths for comparison
    min_len = min(len(left), len(left_recovered))

    # Left channel comparison
    axes6[0, 0].plot(t_audio[:min_len], left[:min_len], 'b-', linewidth
=0.5, alpha=0.7, label='Original')
    axes6[0, 0].plot(t_recovered[:min_len], left_recovered[:min_len], 'g
--', linewidth=0.5, alpha=0.7, label='Recovered')
    axes6[0, 0].set_xlabel('Time (s)')
    axes6[0, 0].set_ylabel('Amplitude')
    axes6[0, 0].set_title('Left Channel Comparison')
    axes6[0, 0].legend()
    axes6[0, 0].grid(True, alpha=0.3)
    axes6[0, 0].set_xlim([0, t_audio[min_len-1]])

    # Right channel comparison
    axes6[0, 1].plot(t_audio[:min_len], right[:min_len], 'r-', linewidth
=0.5, alpha=0.7, label='Original')
    axes6[0, 1].plot(t_recovered[:min_len], right_recovered[:min_len], '
orange', linewidth=0.5, alpha=0.7, label='Recovered')
    axes6[0, 1].set_xlabel('Time (s)')
    axes6[0, 1].set_ylabel('Amplitude')
    axes6[0, 1].set_title('Right Channel Comparison')
    axes6[0, 1].legend()
    axes6[0, 1].grid(True, alpha=0.3)
    axes6[0, 1].set_xlim([0, t_audio[min_len-1]])

    # Zoomed view - first 100 ms
    zoom_samples = int(0.1 * FMStereoParams.FS_AUDIO)
    axes6[1, 0].plot(t_audio[:zoom_samples]*1000, left[:zoom_samples], '
b-', linewidth=1, label='Original')
    axes6[1, 0].plot(t_recovered[:zoom_samples]*1000, left_recovered[:
```

```
    zoom_samples], 'g--', linewidth=1, label='Recovered')
    axes6[1, 0].set_xlabel('Time (ms)')
    axes6[1, 0].set_ylabel('Amplitude')
    axes6[1, 0].set_title('Left Channel - First 100 ms')
    axes6[1, 0].legend()
    axes6[1, 0].grid(True, alpha=0.3)

    axes6[1, 1].plot(t_audio[:zoom_samples]*1000, right[:zoom_samples],
    'r-', linewidth=1, label='Original')
    axes6[1, 1].plot(t_recovered[:zoom_samples]*1000, right_recovered[:
    zoom_samples], 'orange', linewidth=1, label='Recovered')
    axes6[1, 1].set_xlabel('Time (ms)')
    axes6[1, 1].set_ylabel('Amplitude')
    axes6[1, 1].set_title('Right Channel - First 100 ms')
    axes6[1, 1].legend()
    axes6[1, 1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, '06_comparison.png'), dpi=150)
    plt.close()

    print(f"All plots saved to {output_dir}")

#
    ============================================================================

# MAIN FUNCTION
#
    ============================================================================


def run_fm_stereo_system(output_dir="output", freq_deviation=75000,
    input_snr=None):
    """
    Run the complete FM stereo system.

    Parameters:
    - output_dir: Directory to save outputs
    - freq_deviation: FM frequency deviation in Hz
    - input_snr: Input SNR in dB (None for no noise)

    Returns:
    - Dictionary with all signals and measurements
    """
    os.makedirs(output_dir, exist_ok=True)

    print("="*60)
    print("FM STEREO BROADCASTING SYSTEM")
    print("="*60)

    # ====== STEP 1: Generate/Load Audio ======
    print("\n[1/6] Generating stereo audio...")
    left, right, t_audio = generate_stereo_audio(
        duration=FMStereoParams.DURATION,
        fs=FMStereoParams.FS_AUDIO
    )

    # Save input audio
```

```
    save_audio(os.path.join(output_dir, "input_stereo.wav"), left, right
    , FMStereoParams.FS_AUDIO)

    # ====== STEP 2: Apply Pre-emphasis ======
    print("[2/6] Applying pre-emphasis...")
    left_preemph = apply_preemphasis(left, FMStereoParams.TAU,
    FMStereoParams.FS_AUDIO)
    right_preemph = apply_preemphasis(right, FMStereoParams.TAU,
    FMStereoParams.FS_AUDIO)

    # ====== STEP 3: Stereo Multiplex ======
    print("[3/6] Creating stereo composite signal...")
    composite, L_plus_R, L_minus_R, L_minus_R_mod, pilot, t_composite =
    stereo_multiplex(
        left_preemph, right_preemph,
        FMStereoParams.FS_AUDIO,
        FMStereoParams.FS_COMPOSITE
    )

    # ====== STEP 4: FM Modulation ======
    print("[4/6] FM modulating...")
    fm_signal, carrier, t_fm, composite_up = fm_modulate(
        composite,
        FMStereoParams.FS_COMPOSITE,
        FMStereoParams.FS_FM,
        freq_deviation,
        FMStereoParams.FC
    )

    # Add noise if specified
    if input_snr is not None:
        print(f"      Adding AWGN (SNR = {input_snr} dB)...")
        fm_signal = add_awgn(fm_signal, input_snr)

    # ====== STEP 5: FM Demodulation ======
    print("[5/6] FM demodulating...")
    composite_recovered = fm_demodulate(
        fm_signal,
        FMStereoParams.FS_FM,
        FMStereoParams.FS_COMPOSITE,
        freq_deviation,
        FMStereoParams.FC
    )

    # Apply de-emphasis
    composite_deemph = apply_deemphasis(composite_recovered,
    FMStereoParams.TAU, FMStereoParams.FS_COMPOSITE)

    # ====== STEP 6: Stereo Decode ======
    print("[6/6] Decoding stereo...")
    left_recovered, right_recovered, L_plus_R_rec, L_minus_R_rec =
    stereo_decode(
        composite_deemph,
        FMStereoParams.FS_COMPOSITE
    )

    # Normalize outputs
    max_out = max(np.max(np.abs(left_recovered)), np.max(np.abs(
```

```
    right_recovered)))
    if max_out > 0:
        left_recovered = left_recovered / max_out * 0.8
        right_recovered = right_recovered / max_out * 0.8

    # Save output audio
    save_audio(os.path.join(output_dir, "output_stereo.wav"), left_recovered, right_recovered,
               FMStereoParams.FS_AUDIO)

    # ====== Create Visualizations ======
    print("\nGenerating visualizations...")
    create_time_frequency_plots(
        os.path.join(output_dir, "figures"),
        left, right, t_audio,
        composite, L_plus_R, L_minus_R, L_minus_R_mod, pilot, t_composite,
        fm_signal, carrier, t_fm, composite_up,
        left_recovered, right_recovered
    )

    # ====== Calculate Metrics ======
    print("\nCalculating metrics...")

    # SNR
    snr_left = calculate_snr(left, left_recovered)
    snr_right = calculate_snr(right, right_recovered)

    # Bandwidth
    measured_bw = measure_bandwidth_99(fm_signal, FMStereoParams.FS_FM)
    theoretical_bw = carson_bandwidth(freq_deviation, FMStereoParams.COMPOSITE_BW)

    print("\n" + "="*60)
    print("RESULTS")
    print("="*60)
    print(f"Frequency Deviation: {freq_deviation/1000:.0f} kHz")
    print(f"Measured Bandwidth (99% power): {measured_bw/1000:.1f} kHz")
    print(f"Theoretical Bandwidth (Carson): {theoretical_bw/1000:.1f} kHz")
    print(f"Output SNR (Left):  {snr_left:.1f} dB")
    print(f"Output SNR (Right): {snr_right:.1f} dB")
    print("="*60)

    # Return results dictionary
    results = {
        'left_input': left,
        'right_input': right,
        'left_output': left_recovered,
        'right_output': right_recovered,
        'composite': composite,
        'fm_signal': fm_signal,
        'snr_left': snr_left,
        'snr_right': snr_right,
        'measured_bw': measured_bw,
        'theoretical_bw': theoretical_bw,
        'freq_deviation': freq_deviation
    }
```

```

    return results

# 
    ============================================================================
# ENTRY POINT
# 
    ============================================================================

if __name__ == "__main__":
    # Run the FM stereo system with default parameters
    results = run_fm_stereo_system(output_dir="output", freq_deviation
    =75000)

    print("\n[DONE] FM Stereo System completed successfully!")
    print("Check the 'output' folder for:")
    print("  - input_stereo.wav  (synthetic input audio)")
    print("  - output_stereo.wav (recovered output audio)")
    print("  - figures/          (visualization plots)")
```

Listing 1: FM Stereo System Main Code (excerpt)