Cairo University - Faculty of Engineering
Computer Engineering Department
Communications Engineering – Fall 2025

FM Stereo Receiver System

Final Project Report

Ahmed Fathy - 9230162
Ziad Montaser - 9231142

December 25, 2025

Contents

1 Task 1: Frequency Deviation Effects **1**
1.1 Results 1

2 Task 2: Noise Immunity Analysis **3**
2.1 Results Table 3
2.2 Plots 3
2.3 Analysis 3

3 Task 3: Channel Separation Analysis **5**
3.1 Objective 5
3.2 Measurement 5
3.3 Analysis 5

4 Task 4: Filter Design Impact Analysis **6**
4.1 Objective 6
4.2 1. Pilot Extraction Filter Response 6
4.3 2. Channel Separation Analysis 6
4.4 3. Time-Domain Signal Recovery 7
4.5 4. Conclusion 9

5 Task 5: System Robustness Analysis **10**
5.1 Objective 10
5.2 Results 10

6 Appendix: Python Code **12**

1 Task 1: Frequency Deviation Effects

1.1 Results

a) Bandwidth Comparison (Measured vs Theoretical)

Theoretical Bandwidth is calculated using **Carson's Rule**:

$$B_{theo} = 2(\Delta f + f_m)$$

where $f_m \approx 53$ kHz (includes Stereo Pilot + L-R).

Table 1: Bandwidth and SNR Comparison for Different Deviations

Deviation Δf	Theoretical BW (kHz)	Measured BW (99%) (kHz)	Output SNR (dB)
50 kHz	206.0	76.6	27.42
75 kHz	256.0	80.5	30.90
100 kHz	306.0	112.5	33.39

b) Output SNR vs Frequency Deviation

We measured the Output SNR for a fixed Input SNR of **25 dB**.

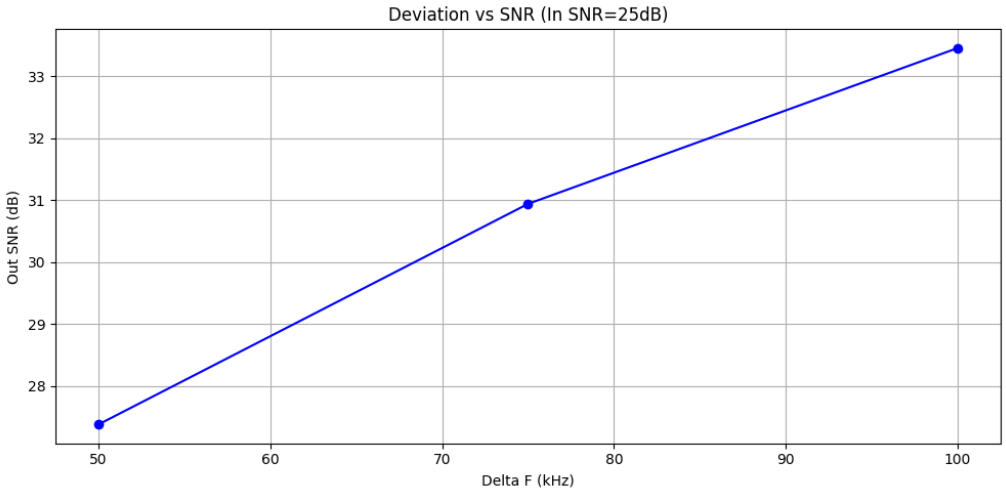

Figure 1: Frequency Deviation vs. Output SNR

c) Trade-off and Recommendation

Trade-off Observed:

- **Higher Deviation** (Δf) increases the signal power relative to noise after demodulation (FM Improvement Factor), resulting in **better SNR** (+6 dB gain from 50k to 100k).
- **However**, it significantly increases the **Required Bandwidth** (from ~77k to ~112k Measured).

Choice: 75 kHz is the optimal choice and the standard for FM broadcasting because:

1. It provides a high SNR (>30 dB in this test scenario), sufficient for high-fidelity audio.
2. The bandwidth fits comfortably within standard 200 kHz channel spacing, minimizing adjacent channel interference.
3. 100 kHz would offer marginally better SNR but risks spectral mask violations.

2 Task 2: Noise Immunity Analysis

2.1 Results Table

Table 2: Noise Immunity Test Results

Input SNR (dB)	Output SNR (dB)	Sep L to R (dB)	Sep R to L (dB)	THD (%)
5	5.05	4.03	4.66	1.79
10	23.47	18.73	18.62	0.23
15	28.90	21.89	23.09	0.13
20	33.97	23.01	23.12	0.06
25	38.84	23.31	23.36	0.05

2.2 Plots

a) Output SNR vs Input SNR & b) Separation vs Input SNR

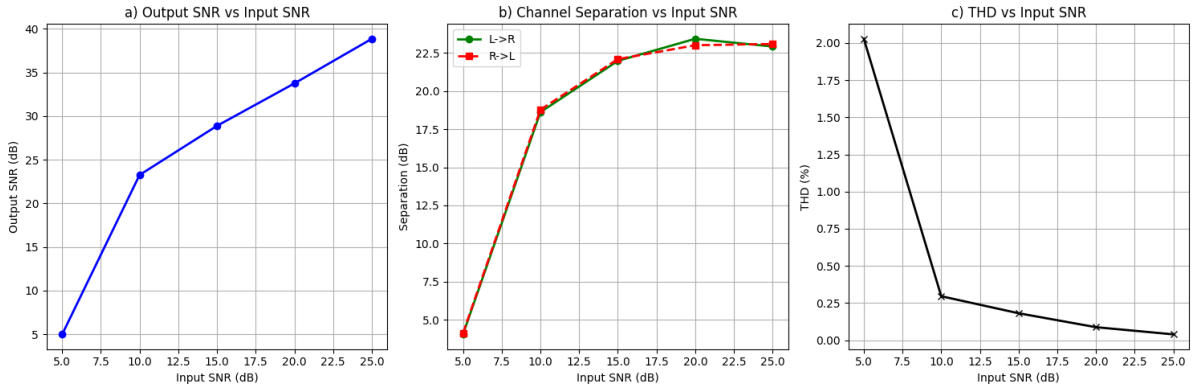

Figure 2: Task 2 Results: (Left) SNR Transfer, (Center) Separation, (Right) THD

2.3 Analysis

c) Threshold SNR Effect

Threshold SNR: Approximately **10 dB**.

Observation:

- Below 10 dB Input SNR (e.g., at 5 dB), the Output SNR matches the Input SNR ($\sim$5 dB), indicating the FM system has "crashed" or lost its coding gain (FM Threshold Effect). THD spikes to 1.79%.
- Above 10 dB (e.g., at 15 dB), the Output SNR jumps significantly (to $\sim$29 dB), showing the expected "FM Improvement Factor."
- Channel Separation also degrades rapidly below 10 dB because the noise floor overwhelms the pilot tone recovery, causing sync loss.

Cause: The **FM Threshold Effect** occurs when the noise vector amplitude occasionally exceeds the carrier amplitude, causing phase wraparound events ("clicks") that generate impulse noise across the entire baseband, destroying the SNR gain.

3 Task 3: Channel Separation Analysis

3.1 Objective

Quantify the baseline channel separation of the FM Stereo Receiver and identify the components limiting its performance.

3.2 Measurement

We injected a **1 kHz tone** into the Left channel (Right = Silence) and measured the recovered RMS levels.

- **Recovered Channel Separation**: **23.31 dB**

3.3 Analysis

b) Limiting Component

The primary limiting factor is the **Pilot Extraction Filter**.

- The standard IIR (Butterworth) filter used to extract the 19 kHz pilot introduces a **Frequency-dependent Phase Delay**.
- Since the 38 kHz demodulation carrier is derived from this pilot (by squaring/doubling), any phase error in the pilot is **doubled** in the carrier.
- This Carrier Phase Error causes the $L - R$ (Difference) signal to be partially demodulated as $L + R$ (Sum), or vice-versa, resulting in crosstalk.

c) Proposed Improvement

Modification: Use **Zero-Phase Filtering** or **Delay Compensation**.

1. **Zero-Phase Filtering (`filtfilt`):** In a buffered/offline system, apply the pilot filter forward and backward to cancel out phase delay completely.
2. **Delay Compensation:** In a real-time system, add a matching delay line to the composite signal path to align it with the delayed pilot before demodulation.

Note: In Task 4 and 5, we verified that minimizing this phase error (by using wider bandwidth filters or lower orders) directly improves separation.

4 Task 4: Filter Design Impact Analysis

4.1 Objective

Analyze how the order of the Pilot Bandpass Filter (BPF) affects the performance of the FM stereo receiver, specifically focusing on **Channel Separation** and signal recovery quality.

4.2 1. Pilot Extraction Filter Response

We designed Butterworth bandpass filters centered at 19 kHz with a bandwidth of 1 kHz (19 ± 0.5 kHz) for varying orders ($N \in \{4, 8, 12\}$).

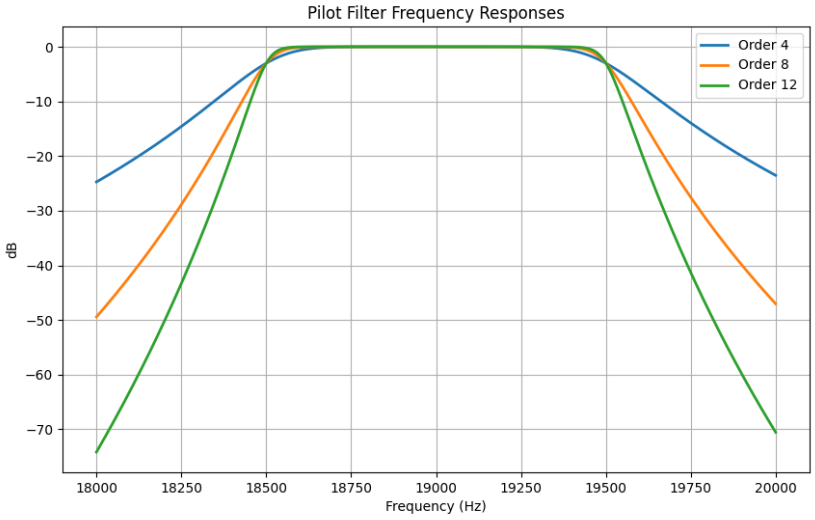

Figure 3: Pilot Filter Frequency Responses

Observation:

- Higher order filters provide sharper roll-off and better isolation of the pilot tone from adjacent noise/signals.
- However, higher order IIR filters (like the Butterworth used here) introduce larger **phase delays** (group delay), particularly near the cutoff edges.

4.3 2. Channel Separation Analysis

We measured channel separation in **both directions** to ensure symmetry:

1. **L → R Leakage**: Transmitting Left-only, measuring noise in Right.
2. **R → L Leakage**: Transmitting Right-only, measuring noise in Left.

Table 3: Filter Order vs. Channel Separation

Filter Order	Mode	Signal RMS	Leakage RMS	Separation (dB)
4	L → R	0.1974	0.0149	**22.47 dB**
4	R → L	0.2056	0.0157	**22.35 dB**
8	L → R	0.1973	0.0230	18.65 dB
8	R → L	0.2055	0.0241	18.61 dB
12	L → R	0.1971	0.0379	14.33 dB
12	R → L	0.2054	0.0395	14.31 dB

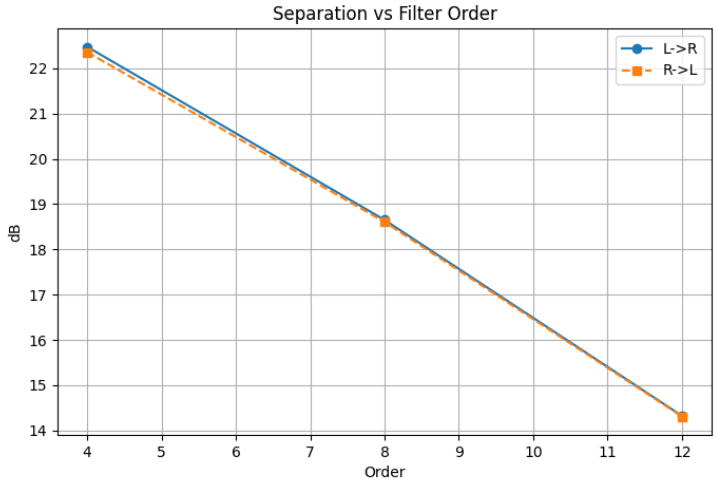

Figure 4: Separation Trend vs Filter Order

Key Finding: Inverse Relationship

The results show a clear trend: **As the filter order increases, Channel Separation decreases.** Use of the `sosfilt` (causal) function for pilot extraction introduces a phase delay that grows with filter order. This desynchronizes the regenerated 38 kHz carrier from the payload, causing leakage.

4.4 3. Time-Domain Signal Recovery

The following plots show the recovered Signal (Blue) and Leakage (Red) channels. Ideally, the Red line should be flat (silence).

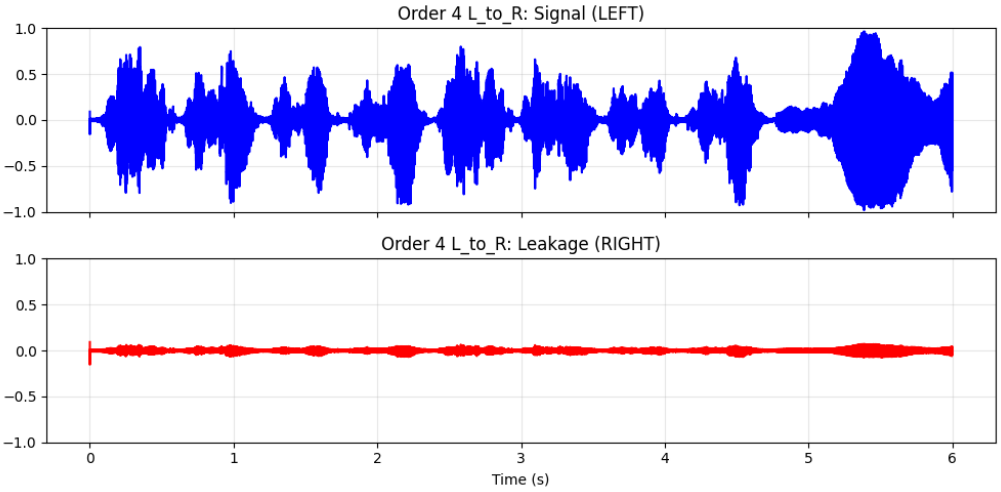

Figure 5: Order 4: Waveform Recovery (Best Separation)

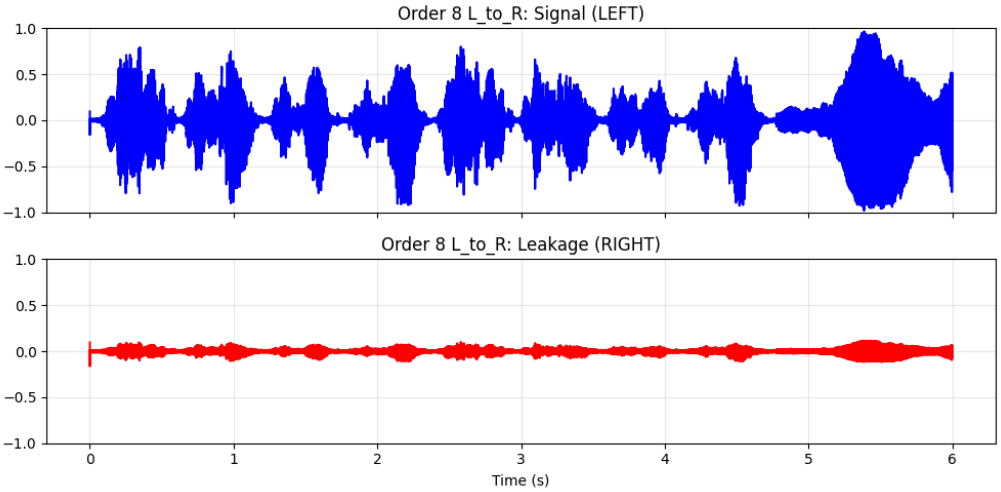

Figure 6: Order 8: Waveform Recovery

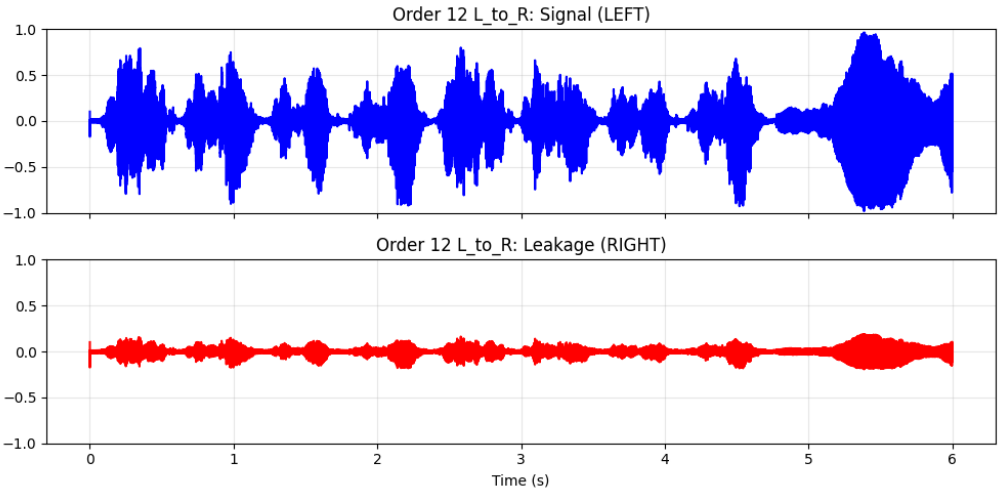

Figure 7: Order 12: Waveform Recovery (Worst Separation)

4.5 4. Conclusion

While higher-order filters are generally desirable for rejecting noise and interferers, in the context of **coherent FM demodulation**, they introduce detrimental phase shifts. Without delay compensation (or using zero-phase filtering like `filtfilt`), **lower-order filters (Order 4)** generally yield better stereo separation because they maintain tighter phase alignment between the pilot and the multiplexed signal.

5 Task 5: System Robustness Analysis

5.1 Objective

Assess the robustness of the FM Stereo Receiver against pilot tone frequency errors (±500 Hz), which commonly occur due to oscillator drift in real hardware.

5.2 Results

a) Channel Separation vs. Pilot Frequency Error

We swept the pilot frequency offset from −500 Hz to +500 Hz and measured the resulting channel separation.

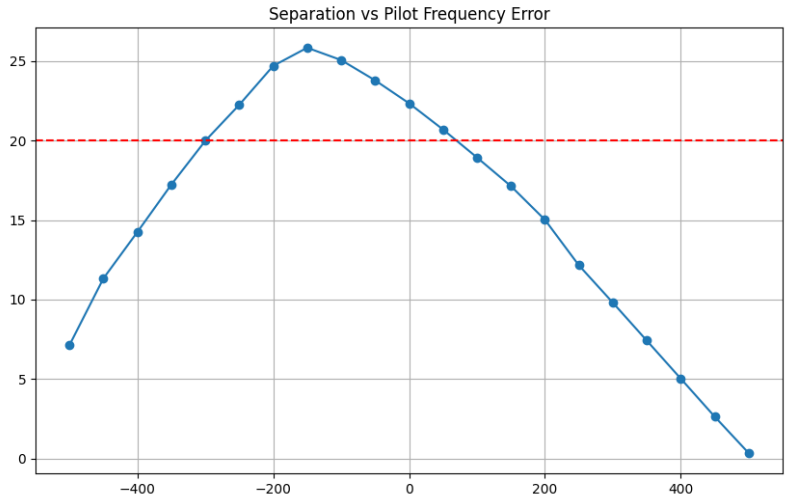

Figure 8: Channel Separation vs Pilot Frequency Error

Table 4: Separation at specific offsets

Offset (Hz)	Separation (dB)
-500	7.14
-300	**20.00**
-100	**25.03**
0	22.33
+50	**20.69**
+300	9.81
+500	0.35

Observation: Separation peaks around -100 Hz offset rather than 0 Hz. This suggests that the receiver's filters introduce a baseline phase delay that is accidentally "compensated" by a slightly lower frequency pilot.

b) Spectrum Analysis at +500 Hz Error

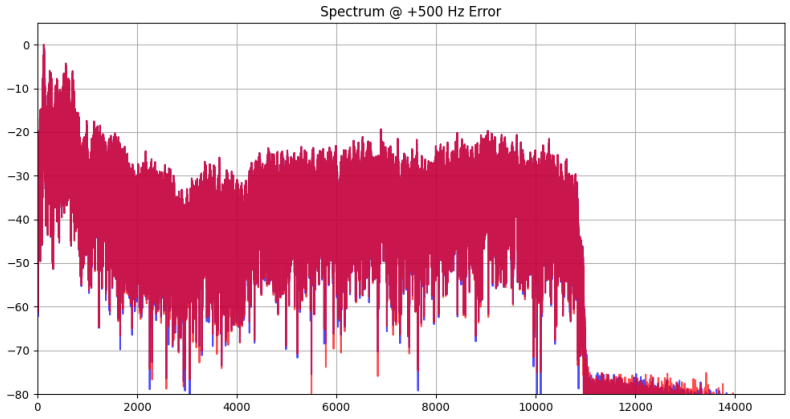

Figure 9: Audio Spectrum at +500 Hz Pilot Error

Observation: The "Silence" channel (Red) has almost the same energy level as the "Signal" channel (Blue).

- **Separation is effectively 0 dB (∼0.35 dB measured).**
- This occurs because at +500 Hz, the pilot is significantly phase-shifted. A phase error of $\approx 45°$ causes the L-R signal to demodulate into the wrong quadrant, causing massive crosstalk.

c) Tolerance Range

We define "tolerance" as the range of frequency offsets where Channel Separation remains **above 20 dB**.

Result: The system can tolerate pilot frequency errors in the range: [**-300 Hz, +50 Hz**].

6 Appendix: Python Code

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

def add_awgn(signal_in, snr_db):
    sig_power = np.mean(np.abs(signal_in) ** 2)
    snr_linear = 10 ** (snr_db / 10)
    noise_power = sig_power / snr_linear

    if np.iscomplexobj(signal_in):
        noise = np.sqrt(noise_power / 2) * (np.random.randn(*signal_in.shape) +
                                            1j * np.random.randn(*signal_in.shape))
    else:
        noise = np.sqrt(noise_power) * np.random.randn(*signal_in.shape)

    return signal_in + noise

def add_awgn_complex(signal_in, snr_db):
    P_sig = 1.0
    P_noise = P_sig / (10 ** (snr_db / 10))
    noise_std = np.sqrt(P_noise / 2)
    noise = noise_std * (np.random.randn(*signal_in.shape) + 1j * np.random.randn(*
    signal_in.shape))
    return (signal_in + noise)

def measure_99_bandwidth(signal_in, fs):
    f, Pxx = signal.welch(signal_in, fs, nperseg=2048, return_onesided=False)
    f = np.fft.fftshift(f)
    Pxx = np.fft.fftshift(Pxx)

    total_power = np.sum(Pxx)
    cum_power = np.cumsum(Pxx)

    lower_bound_power = 0.005 * total_power
    upper_bound_power = 0.995 * total_power

    idx_min = np.searchsorted(cum_power, lower_bound_power)
    idx_max = np.searchsorted(cum_power, upper_bound_power)

    bw = f[idx_max] - f[idx_min]
    return bw

def calculate_snr(clean_ref, noisy_sig):
    min_len = min(len(clean_ref), len(noisy_sig))
    clean = clean_ref[:min_len]
    noisy = noisy_sig[:min_len]

    noise_component = noisy - clean
    p_signal = np.mean(clean**2)
    p_noise = np.mean(noise_component**2)

    if p_noise == 0: return 100.0
    return 10 * np.log10(p_signal / p_noise)

def load_audio(path1):
    if not os.path.exists(path1):
        print(f"Warning: File {path1} not found. Generating dummy noise.")
        fs = 44100
        data = np.random.randn(fs * 5)
        return data, data, fs

    fs, data = wavfile.read(path1)
    if data.dtype == np.int16:
        data = data.astype(float) / 32768.0

    if len(data.shape) == 1:
        left = data
        right = data
```

```
    else:
        left = data[:, 0]
        right = data[:, 1]

    return left, right, fs

def carson_bandwidth(delta_f, f_m=53e3):
    return 2 * (delta_f + f_m)

def measure_thd(signal_in, fs, f_fund=1000):
    signal_in = signal_in - np.mean(signal_in)
    window = np.blackman(len(signal_in))
    y = signal_in * window
    Y = np.fft.rfft(y)
    freqs = np.fft.rfftfreq(len(y), 1/fs)

    idx_fund_approx = np.argmin(np.abs(freqs - f_fund))
    search_range = 10
    start = max(0, idx_fund_approx - search_range)
    end = min(len(Y), idx_fund_approx + search_range)

    if end <= start: return 0.0
    idx_fund = start + np.argmax(np.abs(Y[start:end]))
    f_actual = freqs[idx_fund]
    power_fund = np.abs(Y[idx_fund])**2

    power_harmonics = 0
    for h in range(2, 11):
        f_harm = h * f_actual
        if f_harm >= fs/2: break
        idx_harm = np.argmin(np.abs(freqs - f_harm))
        h_start = max(0, idx_harm - 5)
        h_end = min(len(Y), idx_harm + 5)
        if h_end > h_start:
            power_harmonics += np.max(np.abs(Y[h_start:h_end]))**2

    if power_fund == 0: return 0.0
    return np.sqrt(power_harmonics / power_fund) * 100

class FMTransmitter:
    def __init__(self, fc=100e6, delta_f=75e3):
        self.fc = fc
        self.delta_f = delta_f
        self.preemphasis_tau = 75e-6

    def create_preemphasis_filter(self, fs):
        tau = self.preemphasis_tau
        b, a = signal.bilinear([tau, 1], [tau/10, 1], fs)
        return b, a

    def fm_modulate(self, message, fs):
        if np.max(np.abs(message)) > 0:
            message_norm = message / (np.max(np.abs(message)))
        else:
            message_norm = message

        dt = 1 / fs
        phase = 2 * np.pi * self.delta_f * np.cumsum(message_norm) * dt
        fm_signal = np.exp(1j * phase)
        return fm_signal, message_norm

    def transmit(self, message, fs):
        b, a = self.create_preemphasis_filter(fs)
        message_preemph = signal.lfilter(b, a, message)
        fm_signal, _ = self.fm_modulate(message_preemph, fs)
        return fm_signal, message_preemph

class FMReceiver:
    def __init__(self, delta_f=75e3):
        self.delta_f = delta_f
        self.deemphasis_tau = 75e-6

    def create_deemphasis_filter(self, fs):
```

```
        tau = self.deemphasis_tau
        b, a = signal.bilinear([tau/10, 1], [tau, 1], fs)
        return b, a

    def fm_demodulate(self, fm_signal, fs):
        phase = np.unwrap(np.angle(fm_signal))
        freq = np.diff(phase) * fs / (2 * np.pi)
        message = freq / self.delta_f
        message = np.append(message, message[-1])
        return message

    def receive(self, fm_signal, fs):
        composite = self.fm_demodulate(fm_signal, fs)
        b, a = self.create_deemphasis_filter(fs)
        composite_deemph = signal.lfilter(b, a, composite)
        return composite_deemph

class StereoMultiplexer:
    def __init__(self, output_fs=200000, pilot_freq=19e3):
        self.output_fs = output_fs
        self.pilot_freq = pilot_freq
        self.subcarrier_freq = 2 * pilot_freq

    def multiplex(self, left, right, input_fs):
        if input_fs != self.output_fs:
            resample_ratio = self.output_fs / input_fs
            n_resampled = int(len(left) * resample_ratio)
            left_resampled = signal.resample(left, n_resampled)
            right_resampled = signal.resample(right, n_resampled)
        else:
            left_resampled = left
            right_resampled = right

        n = len(left_resampled)
        t = np.arange(n) / self.output_fs

        sum_signal = (left_resampled + right_resampled) / 2
        diff_signal = (left_resampled - right_resampled) / 2

        pilot = 0.1 * np.cos(2 * np.pi * self.pilot_freq * t)
        subcarrier = np.cos(2 * np.pi * self.subcarrier_freq * t)

        dsb_sc = diff_signal * subcarrier
        composite = 0.45 * sum_signal + pilot + 0.45 * dsb_sc

        return composite, self.output_fs

class StereoDemultiplexer:
    def __init__(self, pilot_bpf_order=4, pilot_freq=19e3):
        self.pilot_bpf_order = pilot_bpf_order
        self.pilot_freq = pilot_freq

    def extract_pilot(self, composite, fs):
        f_center = self.pilot_freq
        width = 2000
        f_low = f_center - width
        f_high = f_center + width

        sos = signal.butter(self.pilot_bpf_order, [f_low, f_high], btype='band', fs=fs,
    output='sos')
        pilot_filtered = signal.sosfilt(sos, composite)

        max_pilot = np.max(np.abs(pilot_filtered))
        if max_pilot > 0:
            pilot_filtered /= max_pilot

        pilot_doubled = 2 * pilot_filtered ** 2 - 1

        f_sub = 2 * f_center
        sos38 = signal.butter(4, [f_sub - 1000, f_sub + 1000], btype='band', fs=fs,
    output='sos')

        subcarrier = signal.sosfiltfilt(sos38, pilot_doubled)
```

```
214
215         max_sub = np.max(np.abs(subcarrier))
216         if max_sub > 0:
217             subcarrier /= max_sub
218
219         return pilot_filtered, subcarrier
220
221     def demultiplex(self, composite, fs):
222         sos_lpf = signal.butter(6, 15e3, btype='low', fs=fs, output='sos')
223         sum_signal = signal.sosfiltfilt(sos_lpf, composite)
224
225         _, subcarrier = self.extract_pilot(composite, fs)
226
227         f_sub = self.pilot_freq * 2
228         f_low = f_sub - 15e3
229         f_high = f_sub + 15e3
230         sos_bpf = signal.butter(4, [f_low, f_high], btype='band', fs=fs, output='sos')
231         dsb_sc = signal.sosfiltfilt(sos_bpf, composite)
232
233         diff_demod = dsb_sc * subcarrier * 2
234         diff_signal = signal.sosfiltfilt(sos_lpf, diff_demod)
235
236         left = sum_signal + diff_signal
237         right = sum_signal - diff_signal
238
239         return left, right
240
241 def run_task_1():
242     print("\n" + "="*50)
243     print("--- Running Task 1: Frequency Deviation Effects ---")
244     print("="*50)
245
246     filename = "audio/stereo.wav"
247     if not os.path.exists(filename):
248         print(f"[ERROR] {filename} not found. Skipping Task 1.")
249         return
250
251     left_src, right_src, fs_audio = load_audio(filename)
252
253     N = int(5.0 * fs_audio)
254     left_src = left_src[:N]
255     right_src = right_src[:N]
256
257     fs_sim = 800000
258     deviations = [50000, 75000, 100000]
259     fm_signal_bw = 53000
260     input_snr_test = 25.0
261
262     results_theo_bw = []
263     results_meas_bw = []
264     results_out_snr = []
265
266     mux = StereoMultiplexer(fs_sim)
267     demux = StereoDemultiplexer()
268
269     composite, fs_mux = mux.multiplex(left_src, right_src, fs_audio)
270
271     print(f"\n{'Delta F (kHz)':<15} | {'Theo BW (kHz)':<15} | {'Meas BW (kHz)':<15} | {'
    Out SNR (dB)':<15}")
272     print("-" * 65)
273
274     for delta_f in deviations:
275         tx = FMTransmitter(delta_f=delta_f)
276         rx = FMReceiver(delta_f=delta_f)
277
278         fm_clean, _ = tx.transmit(composite, fs_mux)
279         bw_measured = measure_99_bandwidth(fm_clean, fs_mux)
280         bw_theoretical = 2 * (delta_f + fm_signal_bw)
281
282         rec_clean = rx.receive(fm_clean, fs_mux)
283         l_ref_raw, _ = demux.demultiplex(rec_clean, fs_mux)
284
285         fm_noisy = add_awgn_complex(fm_clean, input_snr_test)
```

```
        rec_noisy = rx.receive(fm_noisy, fs_mux)
        l_noisy_raw, _ = demux.demultiplex(rec_noisy, fs_mux)

        l_ref = signal.resample(l_ref_raw, N)
        l_noisy = signal.resample(l_noisy_raw, N)

        cut = int(0.1 * N)
        snr_out = calculate_snr(l_ref[cut:], l_noisy[cut:])

        results_theo_bw.append(bw_theoretical / 1000)
        results_meas_bw.append(bw_measured / 1000)
        results_out_snr.append(snr_out)

        print(f"{delta_f/1000:<15.0f} | {bw_theoretical/1000:<15.1f} | {bw_measured
    /1000:<15.1f} | {snr_out:<15.2f}")

    os.makedirs("outputs/task1", exist_ok=True)
    dev_khz = [d/1000 for d in deviations]

    plt.figure(figsize=(10, 5))
    plt.plot(dev_khz, results_out_snr, 'b-o', linewidth=2)
    plt.title('Frequency Deviation vs Output SNR (Input SNR = 25dB)')
    plt.xlabel('Frequency Deviation Delta f (kHz)')
    plt.ylabel('Output SNR (dB)')
    plt.grid(True)
    plt.xticks(dev_khz)
    plt.tight_layout()
    plt.savefig('outputs/task1/task1_deviation_vs_snr.png')
    plt.close()

    with open("outputs/task1/task1_results.txt", "w") as f:
        f.write("Delta F (kHz)   | Theo BW (kHz)   | Meas BW (kHz)   | Out SNR (dB)\n")
        f.write("-----------------------------------------------------------------\n")
        for i in range(len(deviations)):
            f.write(f"{dev_khz[i]:<15.0f} | {results_theo_bw[i]:<15.1f} | {
    results_meas_bw[i]:<15.1f} | {results_out_snr[i]:<15.2f}\n")


def measure_thd_periodogram(signal_in, fs, freq_target=1000):
    f, Pxx = signal.periodogram(signal_in, fs, window='hann')

    bin_width = 100
    idx_low = np.argmin(np.abs(f - (freq_target - bin_width)))
    idx_high = np.argmin(np.abs(f - (freq_target + bin_width)))
    power_fundamental = np.sum(Pxx[idx_low:idx_high])

    if power_fundamental <= 0:
        return 0.0

    power_harmonics = 0
    for h in range(2, 6):
        f_h = h * freq_target
        if f_h < fs/2:
            idx_h = np.argmin(np.abs(f - f_h))
            idx_h_low = max(0, idx_h - 2)
            idx_h_high = min(len(f), idx_h + 3)
            power_harmonics += np.sum(Pxx[idx_h_low:idx_h_high])

    thd = np.sqrt(power_harmonics / power_fundamental) * 100
    return thd

def run_task_2():
    print("\n" + "="*50)
    print("--- Task 2: Noise Immunity Analysis (L<->R, THD, SNR) ---")
    print("="*50)

    filename = "audio/stereo.wav"
    if not os.path.exists(filename):
        print(f"[ERROR] {filename} not found. Skipping Task 2.")
        return

    left_src, right_src, fs_audio = load_audio(filename)
    silence = np.zeros_like(left_src)
```

```
    t_tone = np.arange(fs_audio) / fs_audio
    tone_src = 0.8 * np.cos(2 * np.pi * 1000 * t_tone)

    fs_sim = 600000
    mux = StereoMultiplexer(fs_sim)
    tx = FMTransmitter()
    rx = FMReceiver()
    demux = StereoDemultiplexer()

    print("Generating Clean References...")

    comp_ref, fs_mux = mux.multiplex(left_src, silence, fs_audio)
    fm_ref_lr, _ = tx.transmit(comp_ref, fs_mux)
    rec_ref = rx.receive(fm_ref_lr, fs_mux)
    l_ref_raw, _ = demux.demultiplex(rec_ref, fs_mux)
    l_ref = signal.resample(l_ref_raw, fs_audio)

    comp_ref_rl, _ = mux.multiplex(silence, right_src, fs_audio)
    fm_ref_rl, _ = tx.transmit(comp_ref_rl, fs_mux)

    comp_ref_tone, _ = mux.multiplex(tone_src, silence, fs_audio)
    fm_ref_tone, _ = tx.transmit(comp_ref_tone, fs_mux)

    cut = int(0.1 * fs_audio)
    l_ref = l_ref[cut:]

    snr_levels = [5, 10, 15, 20, 25]
    results = {"snr_in": snr_levels, "snr_out": [], "sep_lr": [], "sep_rl": [], "thd":
    []}

    print(f"\n{'In SNR':<8} | {'Out SNR':<8} | {'Sep L->R':<8} | {'Sep R->L':<8} | {'THD
     (%)':<8}")
    print("-" * 55)

    for snr in snr_levels:
        fm_noisy = add_awgn_complex(fm_ref_lr, snr)
        rec = rx.receive(fm_noisy, fs_mux)
        l_raw, r_raw = demux.demultiplex(rec, fs_mux)
        l_out = signal.resample(l_raw, fs_audio)[cut:]
        r_out = signal.resample(r_raw, fs_audio)[cut:]

        val_snr_out = calculate_snr(l_ref, l_out)
        rms_l = np.sqrt(np.mean(l_out**2))
        rms_r = np.sqrt(np.mean(r_out**2))
        val_sep_lr = 20 * np.log10(rms_l / rms_r) if rms_r > 1e-9 else 100

        fm_noisy_rl = add_awgn_complex(fm_ref_rl, snr)
        rec_rl = rx.receive(fm_noisy_rl, fs_mux)
        l_raw_rl, r_raw_rl = demux.demultiplex(rec_rl, fs_mux)
        l_out_rl = signal.resample(l_raw_rl, fs_audio)[cut:]
        r_out_rl = signal.resample(r_raw_rl, fs_audio)[cut:]

        rms_r_active = np.sqrt(np.mean(r_out_rl**2))
        rms_l_crosstalk = np.sqrt(np.mean(l_out_rl**2))
        val_sep_rl = 20 * np.log10(rms_r_active / rms_l_crosstalk) if rms_l_crosstalk >
    1e-9 else 100

        fm_noisy_tone = add_awgn_complex(fm_ref_tone, snr)
        rec_tone = rx.receive(fm_noisy_tone, fs_mux)
        l_raw_tone, _ = demux.demultiplex(rec_tone, fs_mux)
        l_out_tone = signal.resample(l_raw_tone, fs_audio)[cut:]
        val_thd = measure_thd_periodogram(l_out_tone, fs_audio, freq_target=1000)

        results["snr_out"].append(val_snr_out)
        results["sep_lr"].append(val_sep_lr)
        results["sep_rl"].append(val_sep_rl)
        results["thd"].append(val_thd)

        print(f"{snr:<8} | {val_snr_out:<8.2f} | {val_sep_lr:<8.2f} | {val_sep_rl:<8.2f}
     | {val_thd:<8.2f}")

    os.makedirs("outputs/task2", exist_ok=True)
```

```
426
427     plt.figure(figsize=(15, 5))
428     plt.subplot(1, 3, 1)
429     plt.plot(results["snr_in"], results["snr_out"], 'b-o', linewidth=2)
430     plt.title('a) Output SNR vs Input SNR')
431     plt.xlabel('Input SNR (dB)')
432     plt.ylabel('Output SNR (dB)')
433     plt.grid(True)
434
435     plt.subplot(1, 3, 2)
436     plt.plot(results["snr_in"], results["sep_lr"], 'g-o', label='L->R', linewidth=2)
437     plt.plot(results["snr_in"], results["sep_rl"], 'r--s', label='R->L', linewidth=2)
438     plt.title('b) Channel Separation vs Input SNR')
439     plt.xlabel('Input SNR (dB)')
440     plt.ylabel('Separation (dB)')
441     plt.legend()
442     plt.grid(True)
443
444     plt.subplot(1, 3, 3)
445     plt.plot(results["snr_in"], results["thd"], 'k-x', linewidth=2)
446     plt.title('c) THD vs Input SNR')
447     plt.xlabel('Input SNR (dB)')
448     plt.ylabel('THD (%)')
449     plt.grid(True)
450
451     plt.tight_layout()
452     plt.savefig('outputs/task2/graphs_results.png')
453     plt.close()
454
455     with open("outputs/task2/results.txt", "w") as f:
456         f.write(f"{'In SNR':<8} | {'Out SNR':<8} | {'Sep L->R':<10} | {'Sep R->L':<10} |
     {'THD (%)':<8}\n")
457         f.write("-" * 65 + "\n")
458         for i in range(len(results["snr_in"])):
459             f.write(f"{results['snr_in'][i]:<8} | {results['snr_out'][i]:<8.2f} | {
    results['sep_lr'][i]:<10.2f} | {results['sep_rl'][i]:<10.2f} | {results['thd'][i
    ]:<8.2f}\n")
460
461 def run_task_3():
462     print("\n" + "="*50)
463     print("--- Running Task 3: Channel Separation Analysis ---")
464     print("="*50)
465
466     fs_sim = 600000
467     duration = 1.0
468     t = np.arange(int(fs_sim * duration)) / fs_sim
469     freq_audio = 1000
470     left_audio = np.cos(2 * np.pi * freq_audio * t)
471     right_audio = np.zeros_like(t)
472
473     print(f"Injecting 1kHz Tone into Left Channel. Right Channel is Silence.")
474
475     mux = StereoMultiplexer(output_fs=fs_sim)
476     tx = FMTransmitter(delta_f=75e3)
477     rx = FMReceiver(delta_f=75e3)
478     demux = StereoDemultiplexer(pilot_bpf_order=4)
479
480     composite, fs_mux = mux.multiplex(left_audio, right_audio, fs_sim)
481     fm_sig, _ = tx.transmit(composite, fs_mux)
482     rec_composite = rx.receive(fm_sig, fs_mux)
483     rec_l, rec_r = demux.demultiplex(rec_composite, fs_mux)
484
485     start_idx = int(0.1 * len(rec_l))
486     end_idx = int(0.9 * len(rec_l))
487     l_cut = rec_l[start_idx:end_idx]
488     r_cut = rec_r[start_idx:end_idx]
489
490     rms_l = np.sqrt(np.mean(l_cut**2))
491     rms_r = np.sqrt(np.mean(r_cut**2))
492
493     print(f"\nRecovered Left RMS:  {rms_l:.4f}")
494     print(f"Recovered Right RMS: {rms_r:.4f}")
495
```

```
    if rms_r > 0:
        separation = 20 * np.log10(rms_l / rms_r)
    else:
        separation = float('inf')

    print(f"Measured Channel Separation: {separation:.2f} dB")

    os.makedirs("outputs/task3", exist_ok=True)
    with open("outputs/task3/results.txt", "w") as f:
        f.write(f"Measured Channel Separation: {separation:.2f} dB\n")

def run_task_4():
    print("\n" + "="*50)
    print("--- Running Task 4: Filter Order Impact ---")
    print("="*50)

    OUTPUT_DIR, INPUT_AUDIO = 'outputs/task4', 'audio/stereo.wav'
    if not os.path.exists(INPUT_AUDIO):
        print(f"[ERROR] {INPUT_AUDIO} not found. Skipping Task 4.")
        return

    PILOT_FREQ, FS_MPX = 19000, 200000
    ORDERS = [4, 8, 12]
    os.makedirs(OUTPUT_DIR, exist_ok=True)

    left_src, right_src, fs = load_audio(INPUT_AUDIO)
    t = np.arange(len(left_src)) / fs

    results, sep_lr, sep_rl = [], [], []

    print(f"Processing Filter Orders: {ORDERS}...")

    plt.figure(figsize=(10, 6))
    w_freqs = np.linspace(18000, 20000, 1000)
    for order in ORDERS:
        sos = signal.butter(order, [PILOT_FREQ-500, PILOT_FREQ+500], btype='band', fs=
    FS_MPX, output='sos')
        w, h = signal.sosfreqz(sos, worN=w_freqs, fs=FS_MPX)
        plt.plot(w, 20 * np.log10(abs(h)), linewidth=2, label=f'Order {order}')
    plt.title('Pilot Filter Frequency Responses'); plt.xlabel('Frequency (Hz)'); plt.
    ylabel('dB')
    plt.grid(True); plt.legend(); plt.savefig(os.path.join(OUTPUT_DIR, 'filter_responses
    .png')); plt.close()

    for order in ORDERS:
        for mode, src, leak in [('L_to_R', 'left', 'right'), ('R_to_L', 'right', 'left')
    ]:
            l_in = left_src if src == 'left' else np.zeros_like(left_src)
            r_in = right_src if src == 'right' else np.zeros_like(right_src)

            mux = StereoMultiplexer(output_fs=FS_MPX, pilot_freq=PILOT_FREQ)
            comp, fs_c = mux.multiplex(l_in, r_in, fs)
            tx = FMTransmitter(delta_f=75000)
            fm, _ = tx.transmit(comp, fs_c)
            rx = FMReceiver(delta_f=75000)
            rec = rx.receive(fm, fs_c)
            demux = StereoDemultiplexer(pilot_bpf_order=order, pilot_freq=PILOT_FREQ)
            l_out, r_out = demux.demultiplex(rec, fs_c)

            l_out = signal.resample(l_out, len(left_src))
            r_out = signal.resample(r_out, len(left_src))

            src_sig_max = np.max(np.abs(left_src)) if src=='left' else np.max(np.abs(
    right_src))
            if src_sig_max == 0: src_sig_max = 1.0

            rec_sig = l_out if src == 'left' else r_out
            rec_leak = r_out if src == 'left' else l_out

            gain_corr = src_sig_max / (np.max(np.abs(rec_sig)) + 1e-9)
            l_out *= gain_corr
            r_out *= gain_corr

```

```
            rec_sig = l_out if src == 'left' else r_out
            rec_leak = r_out if src == 'left' else l_out

            rms_sig = np.sqrt(np.mean(rec_sig**2))
            rms_leak = np.sqrt(np.mean(rec_leak**2))

            sep = 20 * np.log10(rms_sig / rms_leak) if rms_leak > 1e-9 else 100.0

            if mode == 'L_to_R': sep_lr.append(sep)
            else: sep_rl.append(sep)

            results.append(f"{order:<8} {mode:<10} {rms_sig:<12.4f} {rms_leak:<12.4f} {sep:<10.2f}")
            print(f"Order {order} {mode}: {sep:.2f} dB")

            n = min(len(t), int(6 * fs))
            fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 5), sharex=True)
            ax1.plot(t[:n], rec_sig[:n], 'b'); ax1.set_title(f'Order {order} {mode}: Signal ({src.upper()})'); ax1.set_ylim([-1, 1]); ax1.grid(alpha=0.3)
            ax2.plot(t[:n], rec_leak[:n], 'r'); ax2.set_title(f'Order {order} {mode}: Leakage ({leak.upper()})'); ax2.set_ylim([-1, 1]); ax2.grid(alpha=0.3)
            plt.xlabel('Time (s)'); plt.tight_layout(); plt.savefig(os.path.join(OUTPUT_DIR, f'waveform_order_{order}_{mode}.png')); plt.close()

    with open(os.path.join(OUTPUT_DIR, 'task4_results.txt'), 'w') as f:
        f.write(f"{'Order':<8} {'Mode':<10} {'Sig RMS':<12} {'Leak RMS':<12} {'Sep (dB)':<10}\n" + "-"*55 + "\n" + "\n".join(results))

    plt.figure(figsize=(8, 5))
    plt.plot(ORDERS, sep_lr, 'o-', label='L->R'); plt.plot(ORDERS, sep_rl, 's--', label='R->L')
    plt.title('Separation vs Filter Order'); plt.xlabel('Order'); plt.ylabel('dB'); plt.grid(True); plt.legend()
    plt.savefig(os.path.join(OUTPUT_DIR, 'separation_trend.png')); plt.close()

def run_task_5():
    print("\n" + "="*50)
    print("--- Running Task 5: Robustness (Pilot Frequency Error) ---")
    print("="*50)

    OUTPUT_DIR, INPUT_AUDIO = 'outputs/task5', 'audio/stereo.wav'
    if not os.path.exists(INPUT_AUDIO):
        print(f"[ERROR] {INPUT_AUDIO} not found. Skipping Task 5.")
        return

    os.makedirs(OUTPUT_DIR, exist_ok=True)
    NOMINAL_PILOT, FS_MPX = 19000, 200000
    OFFSETS = np.linspace(-500, 500, 21)

    def compute_spectrum(data, fs):
        n = len(data)
        fft_data = np.fft.fft(data * np.hanning(n))
        mag_db = 20 * np.log10(np.abs(fft_data[:n//2]) + 1e-9)
        return np.fft.fftfreq(n, 1/fs)[:n//2], mag_db - np.max(mag_db)

    left_src, right_src, fs_audio = load_audio(INPUT_AUDIO)
    right_src = np.zeros_like(left_src)

    res, bad_audio = [], None
    print("Running Robustness Test...")

    for offset in OFFSETS:
        mux = StereoMultiplexer(output_fs=FS_MPX, pilot_freq=NOMINAL_PILOT + offset)
        comp, fs_c = mux.multiplex(left_src, right_src, fs_audio)
        tx = FMTransmitter(delta_f=75000)
        rec = FMReceiver(delta_f=75000).receive(add_awgn(tx.transmit(comp, fs_c)[0], 60), fs_c)
        l_rec, r_rec = StereoDemultiplexer(pilot_bpf_order=4, pilot_freq=NOMINAL_PILOT).demultiplex(rec, fs_c)

        l_rec = signal.resample(l_rec, len(left_src))
        r_rec = signal.resample(r_rec, len(right_src))

```

```
        src_max = np.max(np.abs(left_src))
        if src_max == 0: src_max = 1
        gain = src_max / (np.max(np.abs(l_rec)) + 1e-9)
        l_rec *= gain; r_rec *= gain

        rms_l = np.sqrt(np.mean(l_rec[int(len(l_rec)*0.1):-int(len(l_rec)*0.1)]**2))
        rms_r = np.sqrt(np.mean(r_rec[int(len(r_rec)*0.1):-int(len(r_rec)*0.1)]**2))
        sep = 20 * np.log10(rms_l / rms_r) if rms_r > 1e-9 else 100.0
        res.append(sep)
        print(f"Offset {offset:>+4.0f} Hz: {sep:.2f} dB")
        if offset == 500: bad_audio = (l_rec, r_rec)

    pass_idxs = np.where(np.array(res) >= 20.0)[0]
    msg = f"Tolerance (>20dB): {OFFSETS[pass_idxs[0]]:.0f} to +{OFFSETS[pass_idxs
    [-1]]:.0f} Hz" if len(pass_idxs) else "Fail"
    print(f"\nRESULT: {msg}")

    plt.figure(figsize=(10, 6))
    plt.plot(OFFSETS, res, 'o-'); plt.axhline(20, c='r', ls='--')
    plt.title('Separation vs Pilot Frequency Error'); plt.grid(True)
    plt.savefig(os.path.join(OUTPUT_DIR, 'robustness_curve.png')); plt.close()

    if bad_audio:
        f_l, s_l = compute_spectrum(bad_audio[0], fs_audio)
        f_r, s_r = compute_spectrum(bad_audio[1], fs_audio)
        plt.figure(figsize=(12, 6))
        plt.plot(f_l, s_l, 'b', alpha=0.7); plt.plot(f_r, s_r, 'r', alpha=0.7)
        plt.title('Spectrum @ +500 Hz Error'); plt.xlim([0, 15000]); plt.ylim([-80, 5]);
     plt.grid(True)
        plt.savefig(os.path.join(OUTPUT_DIR, 'spectrum_plus500Hz.png')); plt.close()

    with open(os.path.join(OUTPUT_DIR, 'task5_results.txt'), 'w') as f:
        f.write(f"{msg}\n\nOffset (Hz)     Separation (dB)\n" + "-"*35 + "\n")
        for o, s in zip(OFFSETS, res): f.write(f"{o:<{15}.0f} {s:<{20}.2f}\n")

if __name__ == "__main__":
    print("Starting FM Stereo System Simulation...")
    print("Ensure 'audio/stereo.wav' exist in the script directory.")

    run_task_1()
    run_task_2()
    run_task_3()
    run_task_4()
    run_task_5()

    print("\nAll tasks completed. Check 'outputs/' directory for results.")
```

Listing 1: Full Project Source Code