



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	Measurement	5
3.2	Analysis	5
4	Task 4: Filter Design Impact Analysis	6
4.1	1. Pilot Extraction Filter Response	6
4.2	2. Channel Separation Analysis	6
4.3	3. Time-Domain Signal Recovery	7
4.4	4. Conclusion	9
5	Task 5: System Robustness Analysis	10
5.1	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**.

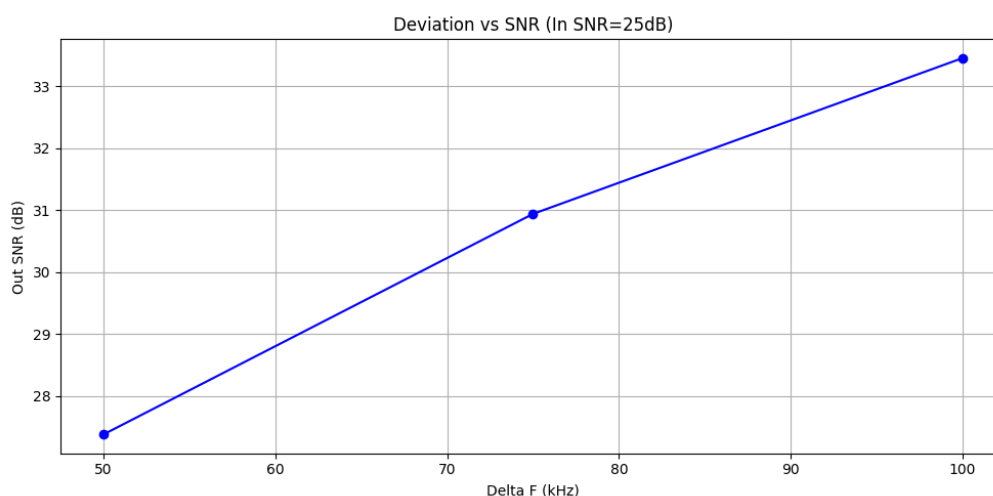


Figure 1: Frequency Deviation vs. Output SNR

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 ~ 77 k to ~ 112 k 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

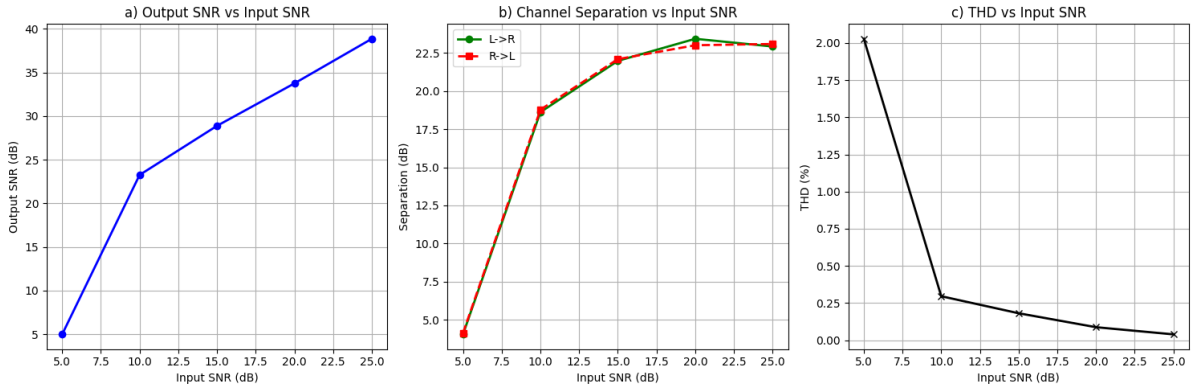


Figure 2: Task 2 Results: (Left) SNR Transfer, (Center) Separation, (Right) THD

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 (~ 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 ~ 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 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**

3.2 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 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 ($N \in \{4, 8, 12\}$).

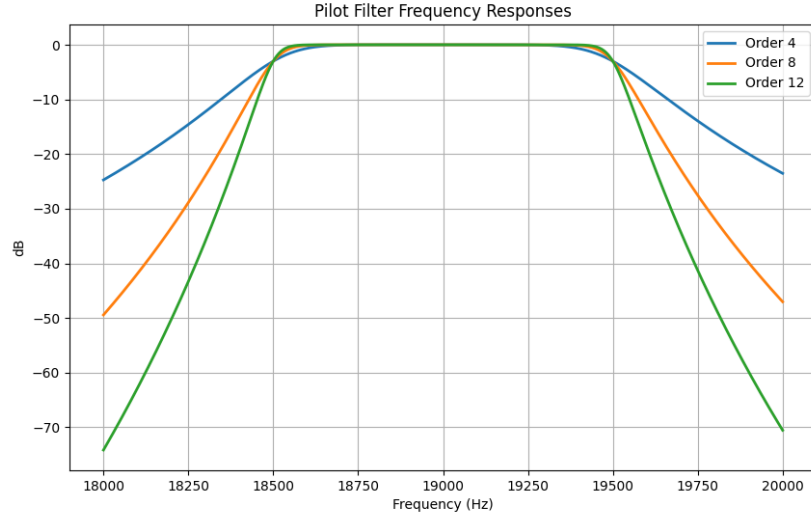


Figure 3: Pilot Filter Frequency Responses

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.

4.2 2. Channel Separation Analysis

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

1. **L \rightarrow R Leakage**: Transmitting Left-only, measuring noise in Right.
2. **R \rightarrow 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 \rightarrow R	0.1974	0.0149	22.47 dB
4	R \rightarrow L	0.2056	0.0157	22.35 dB
8	L \rightarrow R	0.1973	0.0230	18.65 dB
8	R \rightarrow L	0.2055	0.0241	18.61 dB
12	L \rightarrow R	0.1971	0.0379	14.33 dB
12	R \rightarrow L	0.2054	0.0395	14.31 dB

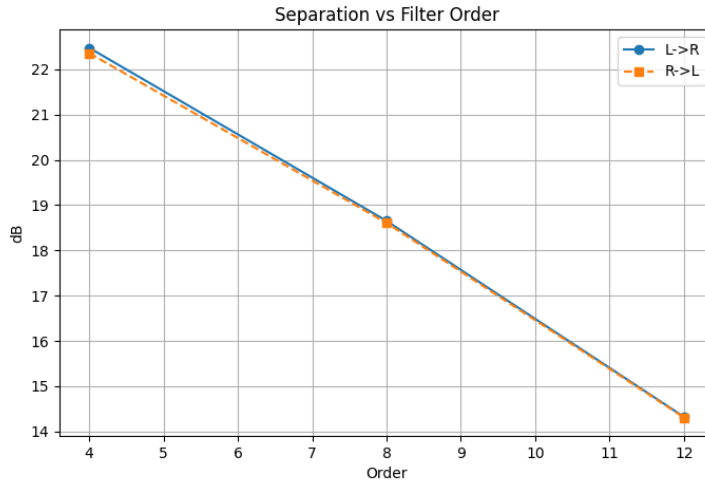


Figure 4: Separation Trend vs Filter Order

Key Finding: Inverse Relationship

The results show a clear trend: **As the filter order increases, Channel Separation decreases.** Use of the `sofilt` (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.

4.3 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).

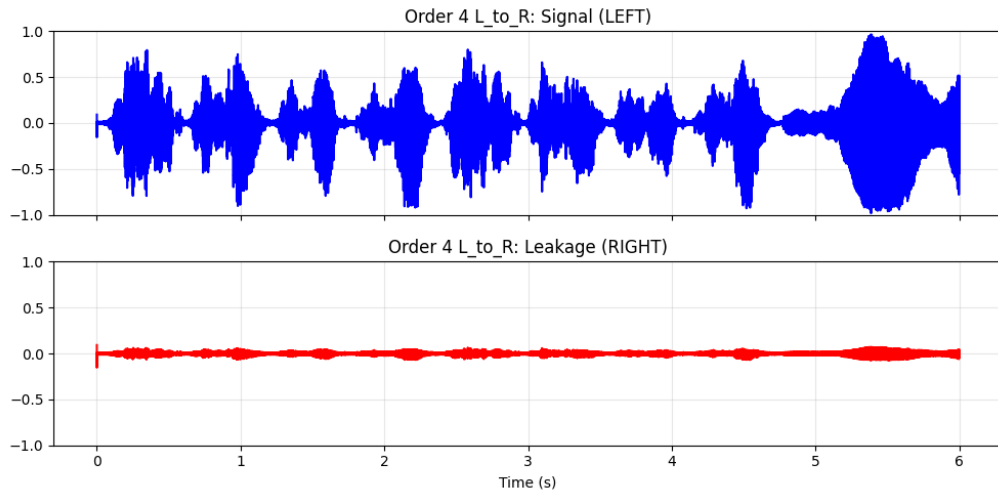


Figure 5: Order 4: Waveform Recovery (Best Separation)

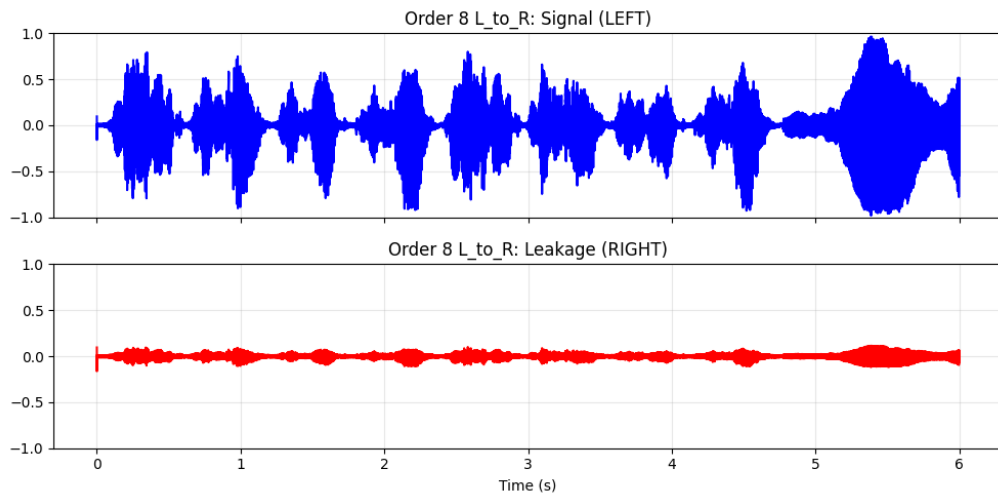


Figure 6: Order 8: Waveform Recovery

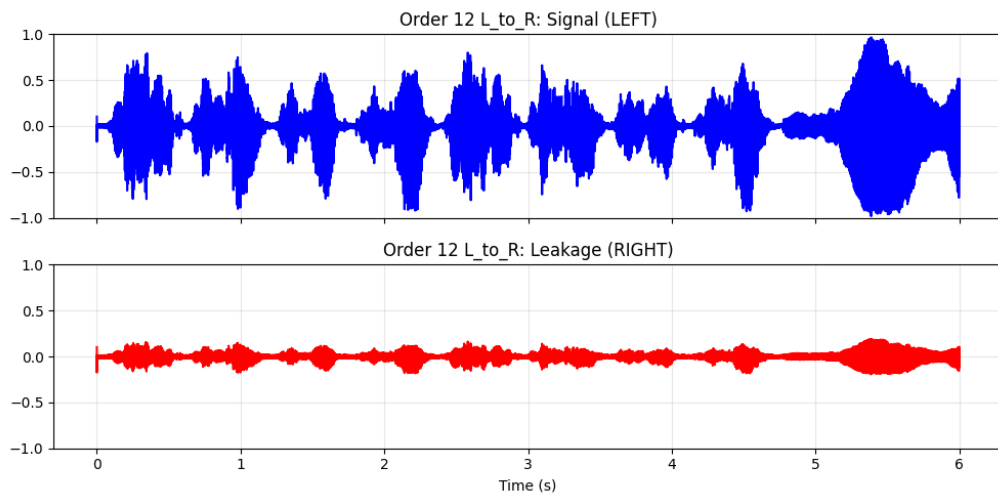


Figure 7: Order 12: Waveform Recovery (Worst Separation)

4.4 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 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.

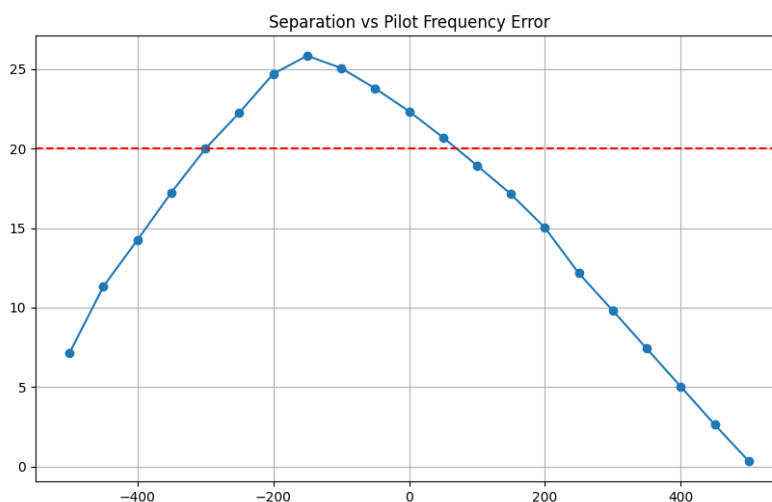


Figure 8: Channel Separation vs Pilot Frequency Error

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

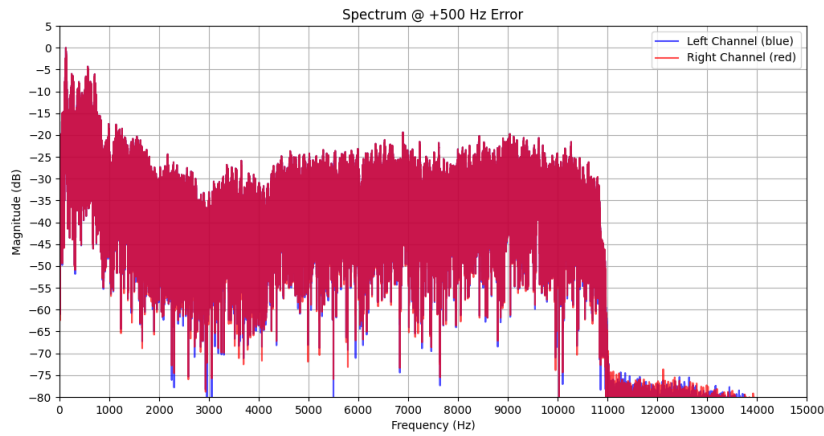


Figure 9: Audio Spectrum at +500 Hz Pilot Error

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 $\approx 45^\circ$ 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

```
1 import numpy as np
2 import scipy.signal as signal
3 from scipy.io import wavfile
4 import matplotlib.pyplot as plt
5 import os
6
7 def add_awgn(signal_in, snr_db):
8     sig_power = np.mean(np.abs(signal_in) ** 2)
9     snr_linear = 10 ** (snr_db / 10)
10    noise_power = sig_power / snr_linear
11
12    if np.iscomplexobj(signal_in):
13        noise = np.sqrt(noise_power / 2) * (np.random.randn(*signal_in.shape) +
14                                             1j * np.random.randn(*signal_in.shape))
15    else:
16        noise = np.sqrt(noise_power) * np.random.randn(*signal_in.shape)
17
18    return signal_in + noise
19
20 def add_awgn_complex(signal_in, snr_db):
21     P_sig = 1.0
22     P_noise = P_sig / (10 ** (snr_db / 10))
23     noise_std = np.sqrt(P_noise / 2)
24     noise = noise_std * (np.random.randn(*signal_in.shape) + 1j * np.random.randn(*
25 signal_in.shape))
26     return (signal_in + noise)
27
28 def measure_99_bandwidth(signal_in, fs):
29     f, Pxx = signal.welch(signal_in, fs, nperseg=2048, return_onesided=False)
30     f = np.fft.fftshift(f)
31     Pxx = np.fft.fftshift(Pxx)
32
33     total_power = np.sum(Pxx)
34     cum_power = np.cumsum(Pxx)
35
36     lower_bound_power = 0.005 * total_power
37     upper_bound_power = 0.995 * total_power
38
39     idx_min = np.searchsorted(cum_power, lower_bound_power)
40     idx_max = np.searchsorted(cum_power, upper_bound_power)
41
42     bw = f[idx_max] - f[idx_min]
43     return bw
44
45 def calculate_snr(clean_ref, noisy_sig):
46     min_len = min(len(clean_ref), len(noisy_sig))
47     clean = clean_ref[:min_len]
48     noisy = noisy_sig[:min_len]
49
50     noise_component = noisy - clean
51     p_signal = np.mean(clean**2)
52     p_noise = np.mean(noise_component**2)
53
54     if p_noise == 0: return 100.0
55     return 10 * np.log10(p_signal / p_noise)
56
57 def load_audio(path1):
58     if not os.path.exists(path1):
59         print(f"Warning: File {path1} not found. Generating dummy noise.")
60         fs = 44100
61         data = np.random.randn(fs * 5)
62         return data, data, fs
63
64     fs, data = wavfile.read(path1)
65     if data.dtype == np.int16:
66         data = data.astype(float) / 32768.0
67
68     if len(data.shape) == 1:
69         left = data
70         right = data
```

```

70     else:
71         left = data[:, 0]
72         right = data[:, 1]
73
74     return left, right, fs
75
76 def carson_bandwidth(delta_f, f_m=53e3):
77     return 2 * (delta_f + f_m)
78
79 def measure_thd(signal_in, fs, f_fund=1000):
80     signal_in = signal_in - np.mean(signal_in)
81     window = np.blackman(len(signal_in))
82     y = signal_in * window
83     Y = np.fft.rfft(y)
84     freqs = np.fft.rfftfreq(len(y), 1/fs)
85
86     idx_fund_approx = np.argmin(np.abs(freqs - f_fund))
87     search_range = 10
88     start = max(0, idx_fund_approx - search_range)
89     end = min(len(Y), idx_fund_approx + search_range)
90
91     if end <= start: return 0.0
92     idx_fund = start + np.argmax(np.abs(Y[start:end]))
93     f_actual = freqs[idx_fund]
94     power_fund = np.abs(Y[idx_fund])**2
95
96     power_harmonics = 0
97     for h in range(2, 11):
98         f_harm = h * f_actual
99         if f_harm >= fs/2: break
100        idx_harm = np.argmin(np.abs(freqs - f_harm))
101        h_start = max(0, idx_harm - 5)
102        h_end = min(len(Y), idx_harm + 5)
103        if h_end > h_start:
104            power_harmonics += np.max(np.abs(Y[h_start:h_end]))**2
105
106    if power_fund == 0: return 0.0
107    return np.sqrt(power_harmonics / power_fund) * 100
108
109 class FMTransmitter:
110     def __init__(self, fc=100e6, delta_f=75e3):
111         self.fc = fc
112         self.delta_f = delta_f
113         self.preemphasis_tau = 75e-6
114
115     def create_preemphasis_filter(self, fs):
116         tau = self.preemphasis_tau
117         b, a = signal.bilinear([tau, 1], [tau/10, 1], fs)
118         return b, a
119
120     def fm_modulate(self, message, fs):
121         if np.max(np.abs(message)) > 0:
122             message_norm = message / (np.max(np.abs(message)))
123         else:
124             message_norm = message
125
126         dt = 1 / fs
127         phase = 2 * np.pi * self.delta_f * np.cumsum(message_norm) * dt
128         fm_signal = np.exp(1j * phase)
129         return fm_signal, message_norm
130
131     def transmit(self, message, fs):
132         b, a = self.create_preemphasis_filter(fs)
133         message_preemph = signal.lfilter(b, a, message)
134         fm_signal, _ = self.fm_modulate(message_preemph, fs)
135         return fm_signal, message_preemph
136
137 class FMReceiver:
138     def __init__(self, delta_f=75e3):
139         self.delta_f = delta_f
140         self.deemphasis_tau = 75e-6
141
142     def create_deemphasis_filter(self, fs):

```

```

143     tau = self.deemphasis_tau
144     b, a = signal.bilinear([tau/10, 1], [tau, 1], fs)
145     return b, a
146
147     def fm_demodulate(self, fm_signal, fs):
148         phase = np.unwrap(np.angle(fm_signal))
149         freq = np.diff(phase) * fs / (2 * np.pi)
150         message = freq / self.delta_f
151         message = np.append(message, message[-1])
152         return message
153
154     def receive(self, fm_signal, fs):
155         composite = self.fm_demodulate(fm_signal, fs)
156         b, a = self.create_deemphasis_filter(fs)
157         composite_deemph = signal.lfilter(b, a, composite)
158         return composite_deemph
159
160 class StereoMultiplexer:
161     def __init__(self, output_fs=200000, pilot_freq=19e3):
162         self.output_fs = output_fs
163         self.pilot_freq = pilot_freq
164         self.subcarrier_freq = 2 * pilot_freq
165
166     def multiplex(self, left, right, input_fs):
167         if input_fs != self.output_fs:
168             resample_ratio = self.output_fs / input_fs
169             n_resampled = int(len(left) * resample_ratio)
170             left_resampled = signal.resample(left, n_resampled)
171             right_resampled = signal.resample(right, n_resampled)
172         else:
173             left_resampled = left
174             right_resampled = right
175
176         n = len(left_resampled)
177         t = np.arange(n) / self.output_fs
178
179         sum_signal = (left_resampled + right_resampled) / 2
180         diff_signal = (left_resampled - right_resampled) / 2
181
182         pilot = 0.1 * np.cos(2 * np.pi * self.pilot_freq * t)
183         subcarrier = np.cos(2 * np.pi * self.subcarrier_freq * t)
184
185         dsb_sc = diff_signal * subcarrier
186         composite = 0.45 * sum_signal + pilot + 0.45 * dsb_sc
187
188         return composite, self.output_fs
189
190 class StereoDemultiplexer:
191     def __init__(self, pilot_bpf_order=4, pilot_freq=19e3):
192         self.pilot_bpf_order = pilot_bpf_order
193         self.pilot_freq = pilot_freq
194
195     def extract_pilot(self, composite, fs):
196         f_center = self.pilot_freq
197         width = 2000
198         f_low = f_center - width
199         f_high = f_center + width
200
201         sos = signal.butter(self.pilot_bpf_order, [f_low, f_high], btype='band', fs=fs,
202                             output='sos')
203         pilot_filtered = signal.sosfilt(sos, composite)
204
205         max_pilot = np.max(np.abs(pilot_filtered))
206         if max_pilot > 0:
207             pilot_filtered /= max_pilot
208
209         pilot_doubled = 2 * pilot_filtered ** 2 - 1
210
211         f_sub = 2 * f_center
212         sos38 = signal.butter(4, [f_sub - 1000, f_sub + 1000], btype='band', fs=fs,
213                               output='sos')
214         subcarrier = signal.sosfiltfilt(sos38, pilot_doubled)

```



```

214     max_sub = np.max(np.abs(subcarrier))
215     if max_sub > 0:
216         subcarrier /= max_sub
217
218     return pilot_filtered, subcarrier
219
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)

```

```

286     rec_noisy = rx.receive(fm_noisy, fs_mux)
287     l_noisy_raw, _ = demux.demultiplex(rec_noisy, fs_mux)
288
289     l_ref = signal.resample(l_ref_raw, N)
290     l_noisy = signal.resample(l_noisy_raw, N)
291
292     cut = int(0.1 * N)
293     snr_out = calculate_snr(l_ref[cut:], l_noisy[cut:])
294
295     results_theo_bw.append(bw_theoretical / 1000)
296     results_meas_bw.append(bw_measured / 1000)
297     results_out_snr.append(snr_out)
298
299     print(f"{delta_f/1000:<15.0f} | {bw_theoretical/1000:<15.1f} | {bw_measured/1000:<15.1f} | {snr_out:<15.2f}")
300
301     os.makedirs("outputs/task1", exist_ok=True)
302     dev_khz = [d/1000 for d in deviations]
303
304     plt.figure(figsize=(10, 5))
305     plt.plot(dev_khz, results_out_snr, 'b-o', linewidth=2)
306     plt.title('Frequency Deviation vs Output SNR (Input SNR = 25dB)')
307     plt.xlabel('Frequency Deviation Delta f (kHz)')
308     plt.ylabel('Output SNR (dB)')
309     plt.grid(True)
310     plt.xticks(dev_khz)
311     plt.tight_layout()
312     plt.savefig('outputs/task1/task1_deviation_vs_snr.png')
313     plt.close()
314
315     with open("outputs/task1/task1_results.txt", "w") as f:
316         f.write("Delta F (kHz) | Theo BW (kHz) | Meas BW (kHz) | Out SNR (dB)\n")
317         f.write("-----\n")
318         for i in range(len(deviations)):
319             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")
320
321
322 def measure_thd_periodogram(signal_in, fs, freq_target=1000):
323     f, Pxx = signal.periodogram(signal_in, fs, window='hann')
324
325     bin_width = 100
326     idx_low = np.argmin(np.abs(f - (freq_target - bin_width)))
327     idx_high = np.argmin(np.abs(f - (freq_target + bin_width)))
328     power_fundamental = np.sum(Pxx[idx_low:idx_high])
329
330     if power_fundamental <= 0:
331         return 0.0
332
333     power_harmonics = 0
334     for h in range(2, 6):
335         f_h = h * freq_target
336         if f_h < fs/2:
337             idx_h = np.argmin(np.abs(f - f_h))
338             idx_h_low = max(0, idx_h - 2)
339             idx_h_high = min(len(f), idx_h + 3)
340             power_harmonics += np.sum(Pxx[idx_h_low:idx_h_high])
341
342     thd = np.sqrt(power_harmonics / power_fundamental) * 100
343     return thd
344
345 def run_task_2():
346     print("\n" + "="*50)
347     print("--- Task 2: Noise Immunity Analysis (L<->R, THD, SNR) ---")
348     print("="*50)
349
350     filename = "audio/stereo.wav"
351     if not os.path.exists(filename):
352         print(f"[ERROR] {filename} not found. Skipping Task 2.")
353         return
354
355     left_src, right_src, fs_audio = load_audio(filename)
356     silence = np.zeros_like(left_src)

```

```

357 t_tone = np.arange(fs_audio) / fs_audio
358 tone_src = 0.8 * np.cos(2 * np.pi * 1000 * t_tone)
359
360
361 fs_sim = 600000
362 mux = StereoMultiplexer(fs_sim)
363 tx = FMTransmitter()
364 rx = FMReceiver()
365 demux = StereoDemultiplexer()
366
367 print("Generating Clean References...")
368
369 comp_ref, fs_mux = mux.multiplex(left_src, silence, fs_audio)
370 fm_ref_lr, _ = tx.transmit(comp_ref, fs_mux)
371 rec_ref = rx.receive(fm_ref_lr, fs_mux)
372 l_ref_raw, _ = demux.demultiplex(rec_ref, fs_mux)
373 l_ref = signal.resample(l_ref_raw, fs_audio)
374
375 comp_ref_rl, _ = mux.multiplex(silence, right_src, fs_audio)
376 fm_ref_rl, _ = tx.transmit(comp_ref_rl, fs_mux)
377
378 comp_ref_tone, _ = mux.multiplex(tone_src, silence, fs_audio)
379 fm_ref_tone, _ = tx.transmit(comp_ref_tone, fs_mux)
380
381 cut = int(0.1 * fs_audio)
382 l_ref = l_ref[cut:]
383
384 snr_levels = [5, 10, 15, 20, 25]
385 results = {"snr_in": snr_levels, "snr_out": [], "sep_lr": [], "sep_rl": [], "thd":
386 []}
387
388 print(f"\n{'In SNR':<8} | {'Out SNR':<8} | {'Sep L->R':<8} | {'Sep R->L':<8} | {'THD
389 (%)':<8}")
390 print("-" * 55)
391
392 for snr in snr_levels:
393     fm_noisy = add_awgn_complex(fm_ref_lr, snr)
394     rec = rx.receive(fm_noisy, fs_mux)
395     l_raw, r_raw = demux.demultiplex(rec, fs_mux)
396     l_out = signal.resample(l_raw, fs_audio)[cut:]
397     r_out = signal.resample(r_raw, fs_audio)[cut:]
398
399     val_snr_out = calculate_snr(l_ref, l_out)
400     rms_l = np.sqrt(np.mean(l_out**2))
401     rms_r = np.sqrt(np.mean(r_out**2))
402     val_sep_lr = 20 * np.log10(rms_l / rms_r) if rms_r > 1e-9 else 100
403
404     fm_noisy_rl = add_awgn_complex(fm_ref_rl, snr)
405     rec_rl = rx.receive(fm_noisy_rl, fs_mux)
406     l_raw_rl, r_raw_rl = demux.demultiplex(rec_rl, fs_mux)
407     l_out_rl = signal.resample(l_raw_rl, fs_audio)[cut:]
408     r_out_rl = signal.resample(r_raw_rl, fs_audio)[cut:]
409
410     rms_r_active = np.sqrt(np.mean(r_out_rl**2))
411     rms_l_crosstalk = np.sqrt(np.mean(l_out_rl**2))
412     val_sep_rl = 20 * np.log10(rms_r_active / rms_l_crosstalk) if rms_l_crosstalk >
413 1e-9 else 100
414
415     fm_noisy_tone = add_awgn_complex(fm_ref_tone, snr)
416     rec_tone = rx.receive(fm_noisy_tone, fs_mux)
417     l_raw_tone, _ = demux.demultiplex(rec_tone, fs_mux)
418     l_out_tone = signal.resample(l_raw_tone, fs_audio)[cut:]
419     val_thd = measure_thd_periodogram(l_out_tone, fs_audio, freq_target=1000)
420
421     results["snr_out"].append(val_snr_out)
422     results["sep_lr"].append(val_sep_lr)
423     results["sep_rl"].append(val_sep_rl)
424     results["thd"].append(val_thd)
425
426     print(f"{snr:<8} | {val_snr_out:<8.2f} | {val_sep_lr:<8.2f} | {val_sep_rl:<8.2f}
427 | {val_thd:<8.2f}")
428
429 os.makedirs("outputs/task2", exist_ok=True)

```

```

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

```

```

496     if rms_r > 0:
497         separation = 20 * np.log10(rms_l / rms_r)
498     else:
499         separation = float('inf')
500
501     print(f"Measured Channel Separation: {separation:.2f} dB")
502
503     os.makedirs("outputs/task3", exist_ok=True)
504     with open("outputs/task3/results.txt", "w") as f:
505         f.write(f"Measured Channel Separation: {separation:.2f} dB\n")
506
507 def run_task_4():
508     print("\n" + "="*50)
509     print("--- Running Task 4: Filter Order Impact ---")
510     print("="*50)
511
512     OUTPUT_DIR, INPUT_AUDIO = 'outputs/task4', 'audio/stereo.wav'
513     if not os.path.exists(INPUT_AUDIO):
514         print(f"[ERROR] {INPUT_AUDIO} not found. Skipping Task 4.")
515         return
516
517     PILOT_FREQ, FS_MPX = 19000, 200000
518     ORDERS = [4, 8, 12]
519     os.makedirs(OUTPUT_DIR, exist_ok=True)
520
521     left_src, right_src, fs = load_audio(INPUT_AUDIO)
522     t = np.arange(len(left_src)) / fs
523
524     results, sep_lr, sep_rl = [], [], []
525
526     print(f"Processing Filter Orders: {ORDERS}...")
527
528     plt.figure(figsize=(10, 6))
529     w_freqs = np.linspace(18000, 20000, 1000)
530     for order in ORDERS:
531         sos = signal.butter(order, [PILOT_FREQ-500, PILOT_FREQ+500], btype='band', fs=
532         FS_MPX, output='sos')
533         w, h = signal.sosfreqz(sos, worN=w_freqs, fs=FS_MPX)
534         plt.plot(w, 20 * np.log10(abs(h)), linewidth=2, label=f'Order {order}')
535         plt.title('Pilot Filter Frequency Responses'); plt.xlabel('Frequency (Hz)'); plt.
536         ylabel('dB')
537         plt.grid(True); plt.legend(); plt.savefig(os.path.join(OUTPUT_DIR, 'filter_responses
538         .png')); plt.close()
539
540     for order in ORDERS:
541         for mode, src, leak in [('L_to_R', 'left', 'right'), ('R_to_L', 'right', 'left')
542         ]:
543             l_in = left_src if src == 'left' else np.zeros_like(left_src)
544             r_in = right_src if src == 'right' else np.zeros_like(right_src)
545
546             mux = StereoMultiplexer(output_fs=FS_MPX, pilot_freq=PILOT_FREQ)
547             comp, fs_c = mux.multiplex(l_in, r_in, fs)
548             tx = FMTransmitter(delta_f=75000)
549             fm, _ = tx.transmit(comp, fs_c)
550             rx = FMReceiver(delta_f=75000)
551             rec = rx.receive(fm, fs_c)
552             demux = StereoDemultiplexer(pilot_bpf_order=order, pilot_freq=PILOT_FREQ)
553             l_out, r_out = demux.demultiplex(rec, fs_c)
554
555             l_out = signal.resample(l_out, len(left_src))
556             r_out = signal.resample(r_out, len(left_src))
557
558             src_sig_max = np.max(np.abs(left_src)) if src=='left' else np.max(np.abs(
559             right_src))
560             if src_sig_max == 0: src_sig_max = 1.0
561
562             rec_sig = l_out if src == 'left' else r_out
563             rec_leak = r_out if src == 'left' else l_out

```

```

564     rec_sig = l_out if src == 'left' else r_out
565     rec_leak = r_out if src == 'left' else l_out
566
567     rms_sig = np.sqrt(np.mean(rec_sig**2))
568     rms_leak = np.sqrt(np.mean(rec_leak**2))
569
570     sep = 20 * np.log10(rms_sig / rms_leak) if rms_leak > 1e-9 else 100.0
571
572     if mode == 'L_to_R': sep_lr.append(sep)
573     else: sep_rl.append(sep)
574
575     results.append(f"{order:<8} {mode:<10} {rms_sig:<12.4f} {rms_leak:<12.4f} {
sep:<10.2f}")
576     print(f"Order {order} {mode}: {sep:.2f} dB")
577
578     n = min(len(t), int(6 * fs))
579     fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 5), sharex=True)
580     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)
581     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)
582     plt.xlabel('Time (s)'); plt.tight_layout(); plt.savefig(os.path.join(
OUTPUT_DIR, f'waveform_order_{order}_{mode}.png')); plt.close()
583
584     with open(os.path.join(OUTPUT_DIR, 'task4_results.txt'), 'w') as f:
585         f.write(f"{order:<8} {mode:<10} {rms_sig:<12.4f} {rms_leak:<12.4f} {
sep:<10.2f}\n" + "-"*55 + "\n" + "\n".join(results))
586
587     plt.figure(figsize=(8, 5))
588     plt.plot(ORDERS, sep_lr, 'o-', label='L->R'); plt.plot(ORDERS, sep_rl, 's--', label=
'R->L')
589     plt.title('Separation vs Filter Order'); plt.xlabel('Order'); plt.ylabel('dB'); plt.
grid(True); plt.legend()
590     plt.savefig(os.path.join(OUTPUT_DIR, 'separation_trend.png')); plt.close()
591
592 def run_task_5():
593     print("\n" + "="*50)
594     print("--- Running Task 5: Robustness (Pilot Frequency Error) ---")
595     print("="*50)
596
597     OUTPUT_DIR, INPUT_AUDIO = 'outputs/task5', 'audio/stereo.wav'
598     if not os.path.exists(INPUT_AUDIO):
599         print(f"[ERROR] {INPUT_AUDIO} not found. Skipping Task 5.")
600         return
601
602     os.makedirs(OUTPUT_DIR, exist_ok=True)
603     NOMINAL_PILOT, FS_MPX = 19000, 200000
604     OFFSETS = np.linspace(-500, 500, 21)
605
606     def compute_spectrum(data, fs):
607         n = len(data)
608         fft_data = np.fft.fft(data * np.hanning(n))
609         mag_db = 20 * np.log10(np.abs(fft_data[:n//2]) + 1e-9)
610         return np.fft.fftfreq(n, 1/fs)[:n//2], mag_db - np.max(mag_db)
611
612     left_src, right_src, fs_audio = load_audio(INPUT_AUDIO)
613     right_src = np.zeros_like(left_src)
614
615     res, bad_audio = [], None
616     print("Running Robustness Test...")
617
618     for offset in OFFSETS:
619         mux = StereoMultiplexer(output_fs=FS_MPX, pilot_freq=NOMINAL_PILOT + offset)
620         comp, fs_c = mux.multiplex(left_src, right_src, fs_audio)
621         tx = FMTransmitter(delta_f=75000)
622         rec = FMReceiver(delta_f=75000).receive(add_awgn(tx.transmit(comp, fs_c)[0], 60)
, fs_c)
623         l_rec, r_rec = StereoDemultiplexer(pilot_bpf_order=4, pilot_freq=NOMINAL_PILOT).
demultiplex(rec, fs_c)
624
625         l_rec = signal.resample(l_rec, len(left_src))
626         r_rec = signal.resample(r_rec, len(right_src))
627

```

```

628     src_max = np.max(np.abs(left_src))
629     if src_max == 0: src_max = 1
630     gain = src_max / (np.max(np.abs(l_rec)) + 1e-9)
631     l_rec *= gain; r_rec *= gain
632
633     rms_l = np.sqrt(np.mean(l_rec[int(len(l_rec)*0.1):-int(len(l_rec)*0.1)]**2))
634     rms_r = np.sqrt(np.mean(r_rec[int(len(r_rec)*0.1):-int(len(r_rec)*0.1)]**2))
635     sep = 20 * np.log10(rms_l / rms_r) if rms_r > 1e-9 else 100.0
636     res.append(sep)
637     print(f"Offset {offset:>4.0f} Hz: {sep:.2f} dB")
638     if offset == 500: bad_audio = (l_rec, r_rec)
639
640     pass_idx = np.where(np.array(res) >= 20.0)[0]
641     msg = f"Tolerance (>20dB): {OFFSETS[pass_idx[0]]:.0f} to {OFFSETS[pass_idx[-1]]:.0f} Hz" if len(pass_idx) else "Fail"
642     print(f"\nRESULT: {msg}")
643
644     plt.figure(figsize=(10, 6))
645     plt.plot(OFFSETS, res, 'o-'); plt.axhline(20, c='r', ls='--')
646     plt.title('Separation vs Pilot Frequency Error'); plt.grid(True)
647     plt.savefig(os.path.join(OUTPUT_DIR, 'robustness_curve.png')); plt.close()
648
649     if bad_audio:
650         f_l, s_l = compute_spectrum(bad_audio[0], fs_audio)
651         f_r, s_r = compute_spectrum(bad_audio[1], fs_audio)
652         plt.figure(figsize=(12, 6))
653         plt.plot(f_l, s_l, 'b', alpha=0.7); plt.plot(f_r, s_r, 'r', alpha=0.7)
654         plt.title('Spectrum @ +500 Hz Error'); plt.xlim([0, 15000]); plt.ylim([-80, 5]);
655         plt.grid(True)
656         plt.savefig(os.path.join(OUTPUT_DIR, 'spectrum_plus500Hz.png')); plt.close()
657
658     with open(os.path.join(OUTPUT_DIR, 'task5_results.txt'), 'w') as f:
659         f.write(f"{msg}\n\nOffset (Hz)      Separation (dB)\n" + "-"*35 + "\n")
660         for o, s in zip(OFFSETS, res): f.write(f"{o:<{15}.0f} {s:<{20}.2f}\n")
661
662 if __name__ == "__main__":
663     print("Starting FM Stereo System Simulation...")
664     print("Ensure 'audio/stereo.wav' exist in the script directory.")
665
666     run_task_1()
667     run_task_2()
668     run_task_3()
669     run_task_4()
670     run_task_5()
671
672     print("\nAll tasks completed. Check 'outputs/' directory for results.")

```

Listing 1: Full Project Source Code