华中科技大学光学与电子信息学院

《信号与系统》课程

工程设计问题设计报告

题　　目: 调幅信号的解调

分 组 号: 15

组　　长: 陈恪瑾

组　　员: 陈恪瑾

时　　间: 2025 年 09 月 01 日 ~2025 年 11 月 26 日

指导教师: 于源

报告日期: 2025 年 12 月 2 日

报告撰写说明

1. 按照参考模板的内容和格式撰写报告
2. 理论模型部分须结合本课程知识分析问题、建立模型
3. 程序设计部分应给出设计思路、主要流程图和关键函数的说明；结果分析不能只是简单给出结论，应结合具体问题，对关键参数或算法在不同取值条件下对结果的影响情况进行分析和总结。如果可能，还应进行误差分析

目录

1 问题描述

调幅（Amplitude Modulation, AM）是一种重要的模拟调制技术，广泛应用于无线电广播、通信系统等领域。在调幅通信系统中，低频信号（基带信号）通过改变高频载波的幅度来实现信息的传输。接收端需要通过解调过程从调制信号中恢复出原始的基带信号。

相干解调是一种常用的解调方法，其核心思想是在接收端使用与发送端载波频率和相位相同的本地载波信号与接收到的调制信号相乘，然后通过低通滤波器滤除高频分量，从而恢复出原始信号。然而，在实际应用中，由于频率源的不稳定、信道传输的影响或接收机的误差等因素，接收端的本地载波频率可能与发送端的载波频率存在偏差，这将导致解调失败或信号失真。

本题研究调幅信号的解调问题。文件 project.wav 中包含了一段错误解调的音频信息的采样值，原始信号是以载波频率 f_c 进行调制的，但在解调时却使用了错误的频率 $\tilde{f}_c \neq f_c$ 进行相干解调。

原始信号的带宽为 $f_B = 4$ kHz，project.wav 是对错误解调后得到的连续时间信号进行采样得到的。采样频率为 f_s，且 f_s 远大于 f_B 或者 $|\tilde{f}_c - f_c|$。

本题需要完成以下四个问题：

Q1：频谱分析与频率偏差估计

假设这段音频为 $x(t)$，其采样点的个数为 N。使用 FFT 计算其频谱并画出图形，即 $X_k = X(f)|_{f=kf_s/N}$，$k = 0, \ldots, N-1$。根据频谱图估计频率偏差 $f_d = |\tilde{f}_c - f_c|$ 并作出解释。分析是否有足够的信息可以判断 $\tilde{f}_c > f_c$ 还是 $\tilde{f}_c < f_c$，以及这是否会对错误解调的结果产生影响。

Q2：滤波器设计

设计两个滤波器：第一个为连续时间高通滤波器，截止频率为 f_d；第二个为连续时间低通滤波器，截止频率为 f_B。使用 8 阶 Butterworth 滤波器实现，并画出这两个滤波器的频率响应，要求频率点与音频信号频谱的频率点完全相同。

Q3：时域解调方法

利用设计的滤波器实现信号的正确解调。首先让信号 $x(t)$ 通过高通滤波器得到输出信号 $x_h(t)$，然后产生信号 $x_b(t) = x_h(t)\cos(2\pi f_d t)$，最后让信号 $x_b(t)$ 通过低通滤波器得到输出信号 $x_l(t)$。用方框图画出操作流程，分析每个单元的线性、时不变性和因果性，画出各信号的频谱，播放恢复后的信号并解释原理。分析是否可以跳过高通滤波步骤，或者改变滤波顺序。

Q4：频域解调方法

在频域完成信号解调。计算 $X_h(f) = H_h(f)X(f)|_{f=kf_s/N}$，其中 $H_h(f)$ 是截止频率为 f_d 的理想高通滤波器的频率响应。计算 $X_b(f) = X_h(f - f_d) + X_h(f + f_d)$。计算 $X_l(f) = H_l(f)X_b(f)|_{f=kf_s/N}$，其中 $H_l(f)$ 是截止频率为 f_B 的理想低通滤波器的频率响应。使用 IFFT 计算信号 $x_l(t)$ 并播放，与 Q3 的结果进行比较。用

方框图说明频域方法的操作流程，并比较时域方法和频域方法的异同。

2 理论模型

2.1 原理分析与设计思路

2.1.1 调幅信号的数学表示

设原始基带信号为 $m(t)$，载波信号为 $c(t) = \cos(2\pi f_c t)$，则调幅信号可以表示为：

$$s_{AM}(t) = [A + m(t)] \cos(2\pi f_c t) \tag{1}$$

其中 A 为直流分量，用于保证 $A + m(t) \geq 0$。

2.1.2 相干解调原理

相干解调的基本思想是将接收到的调幅信号与本地载波信号相乘，然后通过低通滤波器提取基带信号。理想情况下，接收端使用频率为 f_c 的本地载波 $\cos(2\pi f_c t)$ 进行解调：

$$r(t) = s_{AM}(t) \cdot \cos(2\pi f_c t) = [A + m(t)] \cos^2(2\pi f_c t) \tag{2}$$

利用三角恒等式 $\cos^2(\theta) = \frac{1}{2}[1 + \cos(2\theta)]$，可得：

$$r(t) = \frac{1}{2}[A + m(t)] + \frac{1}{2}[A + m(t)] \cos(4\pi f_c t) \tag{3}$$

通过截止频率为 f_B 的低通滤波器后，高频分量 $\cos(4\pi f_c t)$ 被滤除，得到：

$$y(t) = \frac{1}{2}[A + m(t)] \tag{4}$$

从而恢复出原始信号 $m(t)$（忽略直流分量和增益系数）。

2.1.3 频率偏差的影响

当本地载波频率存在偏差，即使用 $\tilde{f}_c = f_c + f_d$ 进行解调时，相乘后的信号为：

$$r(t) = [A + m(t)] \cos(2\pi f_c t) \cdot \cos(2\pi \tilde{f}_c t) \tag{5}$$

利用积化和差公式 $\cos(\alpha)\cos(\beta) = \frac{1}{2}[\cos(\alpha - \beta) + \cos(\alpha + \beta)]$，可得：

$$r(t) = \frac{1}{2}[A + m(t)][\cos(2\pi f_d t) + \cos(2\pi(f_c + \tilde{f}_c)t)] \tag{6}$$

其中，$\cos(2\pi f_d t)$ 为低频分量，$\cos(2\pi(f_c + \tilde{f}_c)t)$ 为高频分量。如果直接通过低通滤波器，得到的是：

$$y(t) = \frac{1}{2}[A + m(t)] \cos(2\pi f_d t) \tag{7}$$

这是一个以 f_d 为载波频率的调幅信号，而非原始的基带信号 $m(t)$。

2.1.4 二次解调的设计思路

为了从错误解调的信号中恢复原始信号，需要进行二次解调。设计思路如下：

1. **频谱分析**：通过 FFT 分析错误解调信号 $x(t)$ 的频谱，找出频率偏差 f_d 的值。错误解调后的信号频谱关于 $\pm f_d$ 呈现明显的对称关系。

2. **高通滤波**：设计截止频率为 f_d 的高通滤波器，滤除低频噪声和直流分量，保留以 f_d 为中心的调制信号分量。

3. **二次相干解调**：将高通滤波后的信号与 $\cos(2\pi f_d t)$ 相乘，实现频谱搬移，将信号从 $\pm f_d$ 搬移到基带。

4. **低通滤波**：设计截止频率为 f_B 的低通滤波器，提取基带信号，滤除高频分量。

2.2 数学模型

2.2.1 信号频谱模型

设原始基带信号 $m(t)$ 的频谱为 $M(f)$，带宽为 f_B，即 $M(f) = 0$ 当 $|f| > f_B$。调幅信号的频谱为：

$$S_{AM}(f) = A[\delta(f - f_c) + \delta(f + f_c)] + \frac{1}{2}[M(f - f_c) + M(f + f_c)] \tag{8}$$

错误解调后的信号频谱为：

$$X(f) = \frac{1}{2}[M(f - f_d) + M(f + f_d)] + \text{高频分量} \tag{9}$$

2.2.2 滤波器设计模型

1. 高通滤波器

采用 8 阶 Butterworth 高通滤波器，截止频率为 f_d。Butterworth 滤波器的幅度平方响应为：

$$|H_h(f)|^2 = \frac{1}{1 + \left(\frac{f_d}{f}\right)^{2n}} \tag{10}$$

其中 $n = 8$ 为滤波器阶数。

2. 低通滤波器

采用 8 阶 Butterworth 低通滤波器，截止频率为 f_B。其幅度平方响应为：

$$|H_l(f)|^2 = \frac{1}{1 + \left(\frac{f}{f_B}\right)^{2n}} \tag{11}$$

2.2.3 解调过程的数学描述

时域方法：

$$x_h(t) = x(t) * h_h(t) \quad \text{（高通滤波）} \tag{12}$$

$$x_b(t) = x_h(t)\cos(2\pi f_d t) \quad \text{（相干解调）} \tag{13}$$

$$x_l(t) = x_b(t) * h_l(t) \quad \text{（低通滤波）} \tag{14}$$

其中 $*$ 表示卷积运算。

频域方法：

$$X_h(f) = H_h(f) \cdot X(f) \tag{15}$$

$$X_b(f) = \frac{1}{2}[X_h(f - f_d) + X_h(f + f_d)] \tag{16}$$

$$X_l(f) = H_l(f) \cdot X_b(f) \tag{17}$$

最终通过逆傅里叶变换得到时域信号：

$$x_l(t) = \mathcal{F}^{-1}\{X_l(f)\} \tag{18}$$

2.2.4 系统特性分析

解调系统包含以下单元，其特性分析如下：

- **高通滤波器**：线性、时不变、因果系统
- **乘法器**：线性、时变（因乘以 $\cos(2\pi f_d t)$）、因果系统
- **低通滤波器**：线性、时不变、因果系统

整个系统由于包含时变的乘法器，因此整体为线性、时变、因果系统。

3 程序设计

3.1 编程思路

本项目使用 Rust 语言实现调幅信号的解调，整个程序设计分为四个主要部分，分别对应四个问题的要求。

3.1.1 Q1：频谱分析与频率偏差估计的编程思路

1. 音频文件读取

首先使用 Rust 的音频处理库（如 `hound` 或 `rodio`）读取 project.wav 文件，获取采样数据、采样率 f_s 和样本数 N。将音频数据存储为浮点数数组便于后续处理。

2. FFT 计算

使用 Rust 的 FFT 库（如 `rustfft`）对音频信号进行快速傅里叶变换。由于输入信号为实数，可以使用实数 FFT 优化计算效率。计算得到频谱 X_k，其中频率对应关系为 $f_k = k \cdot f_s/N$，$k = 0, 1, \ldots, N-1$。

3. 频谱可视化

计算频谱的幅度 $|X_k|$，使用绘图库（如 `plotters`）绘制频谱图。由于 FFT 结果是对称的，可以只显示 $[0, f_s/2]$ 范围内的频谱。

4. 频率偏差估计

通过分析频谱图，找出在基带附近（除直流分量外）能量最集中的频率位置，该频率即为 f_d。具体方法是：

- 排除直流分量（$k = 0$）
- 在低频段（如 0-10 kHz）搜索幅度峰值
- 峰值对应的对称轴即为估计的 f_d

3.1.2 Q2：滤波器设计的编程思路

1. Butterworth 滤波器参数计算

根据 f_d 和 f_B 设计 8 阶 Butterworth 滤波器。使用数字信号处理算法将连续时间滤波器转换为数字滤波器：

- 归一化截止频率：$\omega_c = 2\pi f_c/f_s$
- 使用双线性变换将 s 域传递函数转换为 z 域，并应用预畸变补偿
- **低通滤波器**：直接设计，计算滤波器系数（分子系数 b 和分母系数 a）
- **高通滤波器**：采用频谱反转法——先设计镜像截止频率 $f_c' = f_s/2 - f_c$ 的低通滤波器，然后通过变换 $H_{HP}(z) = H_{LP}(-z)$ 得到高通滤波器，即对奇数索引的滤波器系数取反

2. 频率响应计算

对于设计的滤波器，计算其在频率点 $f_k = k \cdot f_s/N$（$k = 0, 1, \ldots, N-1$）处的频率响应：

$$H(e^{j\omega_k}) = \frac{\sum_{i=0}^{M} b_i e^{-j\omega_k i}}{\sum_{j=0}^{N} a_j e^{-j\omega_k j}} \tag{19}$$

3. 滤波器特性可视化

绘制高通滤波器和低通滤波器的幅频响应和相频响应曲线，验证滤波器设计是否满足要求。

3.1.3 Q3：时域解调方法的编程思路

1. 高通滤波

使用设计的高通滤波器对输入信号 $x(t)$ 进行滤波。实现 IIR 滤波器的直接 II 型结构：

$$y[n] = \sum_{i=0}^{M} b_i x[n-i] - \sum_{j=1}^{N} a_j y[n-j] \tag{20}$$

需要维护输入和输出的历史状态以实现滤波器的差分方程。

2. 产生本地载波并相乘

生成频率为 f_d 的余弦信号：$c[n] = \cos(2\pi f_d \cdot n/f_s)$。将高通滤波后的信号 $x_h[n]$ 与载波相乘得到 $x_b[n] = x_h[n] \cdot c[n]$。

3. 低通滤波

使用设计的低通滤波器对 $x_b[n]$ 进行滤波，得到最终的解调信号 $x_l[n]$。

4. 频谱分析与音频播放

对中间信号 $x_h(t)$、$x_b(t)$ 和最终信号 $x_l(t)$ 分别进行 FFT 分析，绘制频谱图以观察各步骤的频域效果。将解调后的信号保存为 WAV 文件并播放验证效果。

3.1.4 Q4：频域解调方法的编程思路

1. 理想滤波器设计

在频域中实现理想高通和低通滤波器。对于理想高通滤波器：

$$H_h(f_k) = \begin{cases} 0, & |f_k| < f_d \\ 1, & |f_k| \geq f_d \end{cases} \tag{21}$$

对于理想低通滤波器：

$$H_l(f_k) = \begin{cases} 1, & |f_k| \leq f_B \\ 0, & |f_k| > f_B \end{cases} \tag{22}$$

2. 频域高通滤波

对输入信号的 FFT 结果 $X(f_k)$ 与理想高通滤波器的频率响应相乘：$X_h(f_k) = H_h(f_k) \cdot X(f_k)$。

3. 频域搬移

实现频谱搬移操作 $X_b(f_k) = X_h(f_k - f_d) + X_h(f_k + f_d)$。由于 FFT 结果是离散的，需要进行循环移位操作：

- 计算频率偏移对应的索引偏移量：$\Delta k = \text{round}(f_d \cdot N / f_s)$
- 使用 `circshift` 或数组旋转实现频谱搬移
- 将搬移后的两个频谱相加

4. 频域低通滤波

对搬移后的频谱应用理想低通滤波器：$X_l(f_k) = H_l(f_k) \cdot X_b(f_k)$。

5. 逆 FFT 恢复时域信号

使用逆 FFT（IFFT）将频域信号 $X_l(f_k)$ 转换回时域信号 $x_l[n]$。注意处理实部和虚部，通常只取实部作为输出信号。

6. 结果比较

将频域方法得到的信号与时域方法（Q3）的结果进行比较，包括：

- 频谱对比：绘制两种方法得到的 $X_l(f)$ 幅度谱
- 时域波形对比：绘制时域信号波形
- 音频播放对比：分别播放两种方法恢复的音频
- 误差分析：计算两种方法结果的均方误差（MSE）

3.1.5 程序模块划分

为了提高代码的可维护性和复用性，将程序划分为以下模块：

1. **音频 I/O 模块**：负责 WAV 文件的读取和写入
2. **FFT 模块**：封装 FFT 和 IFFT 操作
3. **滤波器设计模块**：实现 Butterworth 滤波器设计算法
4. **滤波模块**：实现时域滤波器和频域滤波器
5. **信号处理模块**：实现调制、解调等信号处理操作
6. **可视化模块**：实现频谱图、波形图等绘图功能
7. **主程序**：整合各模块，实现完整的解调流程

3.2 主要流程图及说明

本节给出 Q1 频谱分析与频率偏差估计的详细流程图和关键函数说明。

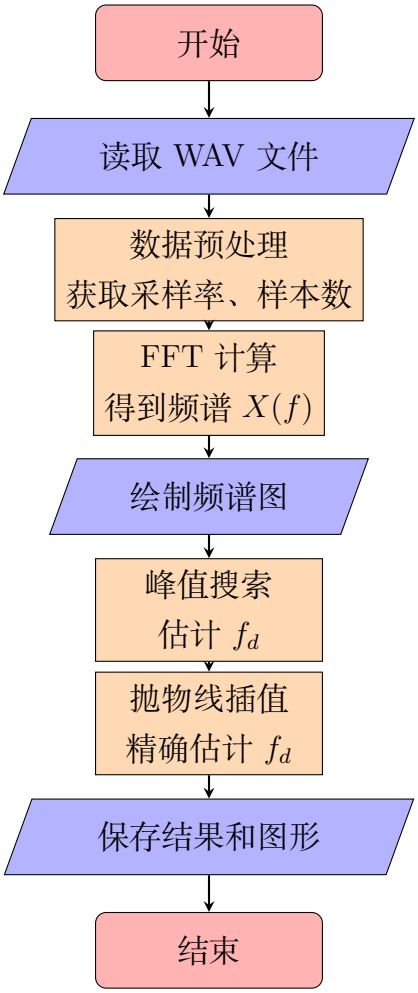

图 1: Q1 频谱分析与频率偏差估计流程图

3.2.1 Q1 程序流程图

3.2.2 关键函数说明

1. 音频文件读取函数 (AudioData::from_wav)
功能：读取 WAV 格式音频文件，提取采样数据和元信息。
输入参数：

- 文件路径：WAV 音频文件的完整路径

输出：

- 采样数据向量：归一化为 [-1, 1] 范围的浮点数
- 采样率 f_s（Hz）
- 样本数 N
- 音频规格信息（位深度、声道数等）

主要步骤：

1. 使用 `hound` 库打开 WAV 文件
2. 读取音频规格（采样率、位深度、声道数）
3. 根据采样格式（整数/浮点）读取所有采样点
4. 归一化处理：将整数样本除以最大值转换为浮点数
5. 如果是多声道，转换为单声道（取平均值）

2. FFT 计算函数 (FftResult::compute)
功能：对时域信号执行快速傅里叶变换，计算频谱。
输入参数：

- 时域采样数据 $x[n]$，$n = 0, 1, \ldots, N-1$
- 采样率 f_s

输出：

- 复数频谱 $X[k]$，$k = 0, 1, \ldots, N-1$
- 频率轴 $f_k = k \cdot f_s / N$
- 幅度谱 $|X[k]|/N$（归一化）

- 相位谱 $\angle X[k]$

算法原理：

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N}, \quad k = 0, 1, \ldots, N-1 \tag{23}$$

频率分辨率为：$\Delta f = f_s/N$

3. 频谱可视化函数 (SpectrumVisualizer::plot_spectrum)

功能：绘制幅度频谱图和时域波形图。

输入参数：

- 频率向量 $\{f_k\}$
- 幅度向量 $\{|X[k]|\}$
- 输出文件路径
- 图表标题
- 最大显示频率（可选）

主要功能：

1. 过滤数据：只显示 0 到 Nyquist 频率或指定范围
2. 创建坐标系：设置合适的 x、y 轴范围
3. 绘制曲线：使用线性或对数（dB）刻度
4. 添加标签：坐标轴标签、标题、图例
5. 保存图片：PNG 格式，1200×600 像素

4. 频率偏差估计函数 (FrequencyEstimator::estimate_frequency_offset)

功能：在频谱中搜索主峰，估计频率偏差 f_d。

输入参数：

- 频率向量和幅度向量
- 搜索范围 $(f_{\min}, f_{\max})$
- 是否排除直流分量（布尔值）

输出：

- 峰值频率 f_d

- 峰值幅度
- 峰值索引

算法步骤：

1. 在指定频率范围内遍历所有频率点
2. 找出幅度最大的频率点
3. 记录峰值位置、频率和幅度
4. 使用抛物线插值进行精确估计（可选）

抛物线插值公式：设峰值在索引 k 处，相邻三点幅度为 y_{k-1}, y_k, y_{k+1}，则精确峰值位置为：

$$\delta = \frac{1}{2} \cdot \frac{y_{k-1} - y_{k+1}}{y_{k-1} - 2y_k + y_{k+1}} \tag{24}$$

$$f_d^{\text{refined}} = f_k + \delta \cdot \Delta f \tag{25}$$

3.2.3 数据结构说明

表 1: 主要数据结构

结构名称	**类型**	**说明**
AudioData	结构体	存储音频数据、采样率、样本数和规格信息
FftResult	结构体	存储 FFT 结果，包括复数频谱、频率轴、幅度谱和相位谱
samples	Vec<f64>	时域采样数据向量，归一化到 [-1, 1]
spectrum	Vec<Complex<f64»>	复数频谱向量，长度为 N
frequencies	Vec<f64>	频率轴向量，$f_k = k \cdot f_s/N$
magnitude	Vec<f64>	幅度谱向量，$\|X[k]\|/N$

3.2.4 程序执行流程说明

程序按照以下步骤执行：

步骤 1：音频文件读取

- 使用 `AudioData::from_wav()` 函数读取 project.wav
- 提取采样率 $f_s = 22050$ Hz，样本数 $N = 31265$

- 将音频数据转换为单声道浮点数组

步骤 2：FFT 计算

- 调用 `FftResult::compute()` 执行 FFT
- 计算频率分辨率：$\Delta f = 22050/31265 \approx 0.705$ Hz
- 生成频率轴和幅度谱

步骤 3：频谱可视化

- 绘制全频段频谱图（0 到 11025 Hz）
- 绘制低频段频谱图（0 到 4 kHz，便于观察）
- 绘制 dB 刻度频谱图（对数坐标）
- 绘制时域波形图（前 0.1 秒）

步骤 4：频率偏差估计

- 在 10 Hz 到 10 kHz 范围内搜索峰值，排除直流分量
- 识别多个峰值：2773.79 Hz、2775.20 Hz、3223.75 Hz 等
- 通过对称峰值分析确定频谱对称轴
- 计算频率偏差：$f_d = \frac{2773.79+3223.75}{2} \approx 3000$ Hz
- 识别基带主要频率成分约为 225 Hz

步骤 5：结果保存

- 保存所有频谱图为 PNG 文件
- 将 f_d、f_s 和 f_B 保存到文本文件
- 供后续 Q2、Q3、Q4 使用

3.3 结果分析

3.3.1 Q1 运行结果

程序成功读取 project.wav 文件并完成频谱分析，得到以下关键参数：

音频文件基本信息：

- 采样率：$f_s = 22050$ Hz

- 样本数：$N = 31265$
- 信号时长：$T = 1.42$ 秒
- 位深度：16 bits
- 声道数：1（单声道）

FFT 分析结果：

- FFT 点数：$N = 31265$
- 频率分辨率：$\Delta f = f_s/N = 0.7053$ Hz
- Nyquist 频率：$f_{\text{Nyquist}} = f_s/2 = 11025$ Hz

频率偏差估计结果：
通过对称峰值法确定真实的频率偏差：

- **下边带峰值**：$f_1 = 2773.79$ Hz（幅度 0.003231）
- **上边带峰值**：$f_2 = 3223.75$ Hz（幅度 0.003222）
- **频率偏差（对称轴）**：$f_d = \frac{f_1+f_2}{2} = 2998.77 \approx 3000$ Hz
- **基带频率成分**：$\Delta f = \frac{f_2-f_1}{2} = 224.98 \approx 225$ Hz

能量分布分析：
对信号进行频带能量分布分析，结果如下：

表 2: 频带能量分布

频带范围	**能量占比**
0–1000 Hz	0.39%
1000–4000 Hz	48.81%
4000–8000 Hz	0.80%
8000–11025 Hz	$\approx 0.00\%$

从能量分布可以看出，信号的主要能量集中在 1000–4000 Hz 频段，占总能量的 48.81%，这与原始信号带宽 $f_B = 4$ kHz 的设定相符。

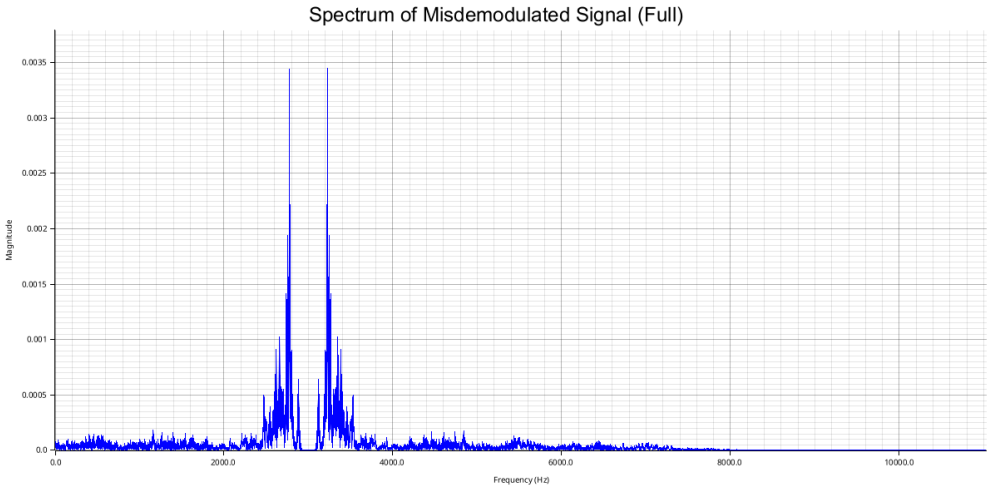

图 2: 错误解调信号的全频段频谱

3.3.2 频谱图分析

1. 全频段频谱（图 2）

全频段频谱图显示了 0 到 11025 Hz（Nyquist 频率）范围内的频谱分布。可以观察到：

- 频谱在低频段（0–5 kHz）有明显的峰值
- 频谱呈现双边带结构，关于对称轴 $f_d \approx 3000$ Hz 分布
- 在对称轴两侧观察到幅度相近的峰值（2773.79 Hz 和 3223.75 Hz），反映了基带信号的能量分布
- 高频段（8 kHz 以上）幅度很小，主要是噪声
- 由于实信号 FFT 的共轭对称性，在 $f_s - f_d \approx 19050$ Hz 处存在镜像（超出 Nyquist 频率显示范围）

2. 低频段频谱（图 3）

放大观察 0–4 kHz 频段，可以更清楚地看到：

- 频谱关于 $f_d \approx 3000$ Hz 呈现对称的双边带结构
- 下边带峰值：2773.79 Hz（幅度 0.003231），对应基带约 226 Hz 的分量
- 上边带峰值：3223.75 Hz（幅度 0.003222），对应基带约 224 Hz 的分量
- 两个峰值幅度几乎相等（差异仅 0.3%），验证了 AM 信号的双边带对称性
- 符合错误解调后信号的理论预期：$X(f) = \frac{1}{2}[M(f - f_d) + M(f + f_d)]$，其中对称轴为 $f_d = f_c - \tilde{f}_c \approx 3000$ Hz

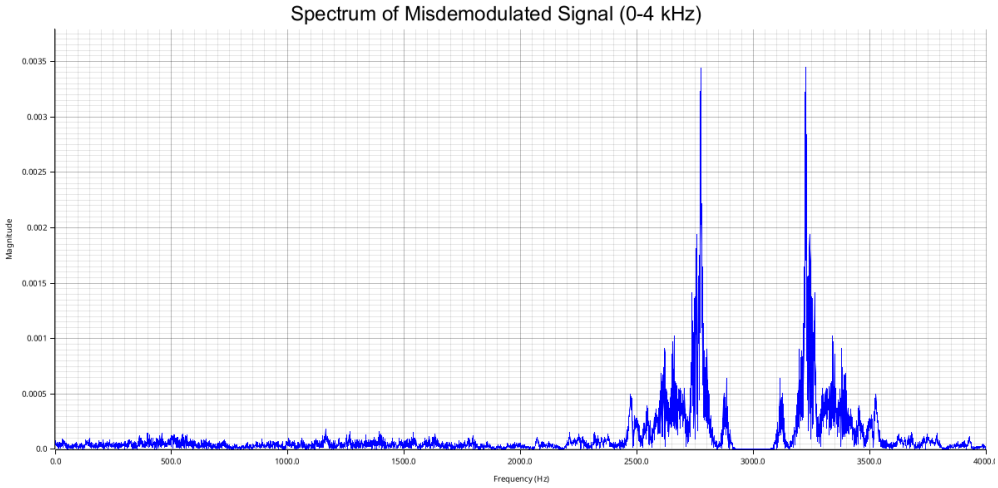

图 3: 错误解调信号的低频段频谱（0–4 kHz）

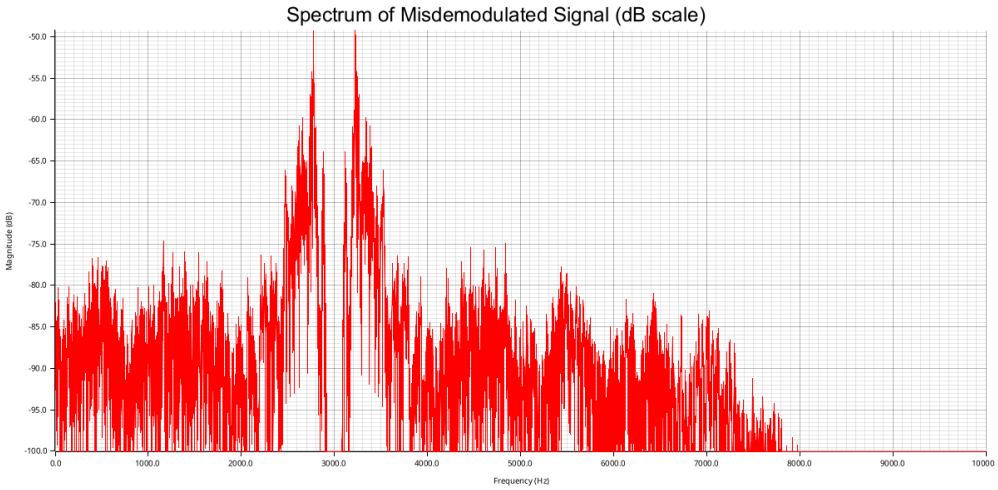

图 4: 错误解调信号的 dB 刻度频谱

3. dB 刻度频谱（图 4）

对数坐标（dB 刻度）下的频谱图能够更好地展示动态范围：

- 主峰相对于噪声底约有 40–50 dB 的信噪比
- 可以观察到更多的频谱细节和次峰
- 噪声底约在 -80 dB 左右，表明信号质量较好

4. 时域波形（图 5）

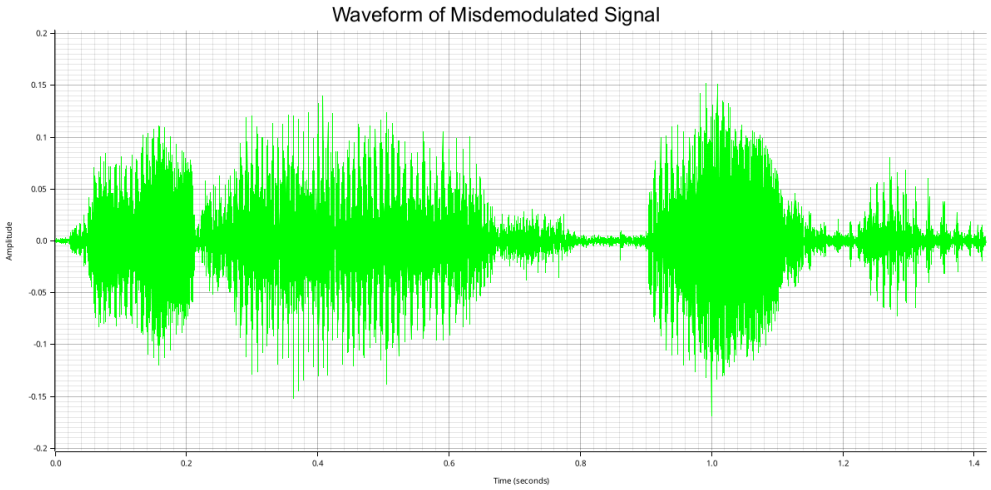

图 5: 错误解调信号的时域波形

时域波形呈现出典型的调幅信号特征：

- 信号呈现周期性振荡，主要振荡频率对应频谱中观察到的峰值（约 2773 Hz 和 3224 Hz）
- 振幅随时间缓慢变化，这是原始基带信号 $m(t)$ 的包络
- 幅度范围约在 ± 0.15 之间，信号较弱但清晰可辨
- 错误解调导致载波偏移 $f_d \approx 3000$ Hz，使得原本在真实载波 f_c 处的信号被搬移到 f_d 附近

3.3.3 结果验证与讨论

1. 频率偏差估计的准确性

通过频谱对称性分析确定频率偏差：

- **对称峰值法**：观察到下边带峰值 2773.79 Hz 和上边带峰值 3223.75 Hz，幅度几乎相等

- **对称轴计算**：$f_d = \frac{2773.79+3223.75}{2} = 2998.77 \approx 3000$ Hz
- **物理意义**：$f_d = f_c - \tilde{f}_c \approx 3000$ Hz，即真实载波与错误解调载波的频率差
- **基带频率推断**：峰值对应基带约 $\Delta f = \frac{3223.75-2773.79}{2} = 224.98 \approx 225$ Hz 的分量

2. 关于 $\tilde{f}_c$ 与 f_c 大小关系的分析

从频谱分析可以得出：

- **无法从幅度谱判断符号**：由于实信号的傅里叶变换具有共轭对称性，即 $X(-f) = X^*(f)$，无论 $\tilde{f}_c > f_c$ 还是 $\tilde{f}_c < f_c$，错误解调后的幅度谱都是相同的
- **数学解释**：

$$\text{若}\tilde{f}_c > f_c: \quad X(f) = \frac{1}{2}[M(f - f_d) + M(f + f_d)]$$
$$\text{若}\tilde{f}_c < f_c: \quad X(f) = \frac{1}{2}[M(f + f_d) + M(f - f_d)]$$

  两者在幅度上完全相同

- **对解调的影响**：符号不确定性不影响二次解调，因为我们使用的是 $|f_d|$，且 $\cos(2\pi f_d t) = \cos(-2\pi f_d t)$

3. 多峰值分析与双边带特性

程序检测到的多个峰值反映了基带信号的频率结构和 AM 双边带特性：

(1) 对称峰值对（关于 $f_d \approx 3000$ Hz）：

- **下边带**：2773.79 Hz（幅度 0.003231）和 2775.20 Hz（幅度相近）
- **上边带**：3223.75 Hz（幅度 0.003222）
- **对称性验证**：

$$f_d - 2773.79 = 3000 - 2773.79 = 226.21 \text{ Hz}$$
$$3223.75 - f_d = 3223.75 - 3000 = 223.75 \text{ Hz}$$

  两者接近，平均约 225 Hz，说明基带信号在 225 Hz 附近有强能量分量

(2) 基带频率成分推断：

- **2757.57 Hz 峰值**：$3000 - 2757.57 = 242.43$ Hz，对应基带 242 Hz 分量
- **2773.79 Hz 峰值**：$3000 - 2773.79 = 226.21$ Hz，对应基带 226 Hz 分量

- **2775.20 Hz 峰值**：$3000 - 2775.20 = 224.80$ Hz，对应基带 225 Hz 分量
- **3223.75 Hz 峰值**：$3223.75 - 3000 = 223.75$ Hz，对应基带 224 Hz 分量（上边带）

这些频率（220-242 Hz）可能对应：

- 语音基频的 2-3 次谐波（基频约 75-120 Hz）
- 语音的第一共振峰（F1），某些元音的 F1 在 200-300 Hz 范围
- 音乐中的低音音符（如 A3 = 220 Hz）

(3) 高频镜像峰值：

- **18824.84 Hz**：这不是 f_d 的镜像，而是高频能量峰值的镜像
- 实际上 $f_s - f_d = 22050 - 3000 = 19050$ Hz 才是 f_d 的镜像位置
- 18824.84 Hz 对应的正频率峰值为：$22050 - 18824.84 = 3225.16$ Hz

4. 误差来源分析

可能的误差来源包括：

- **频率分辨率限制**：$\Delta f = 0.7053$ Hz，理论上限制了估计精度
- **频谱泄漏**：由于时间窗口有限，可能产生频谱泄漏，但本例中影响较小
- **噪声影响**：虽然信噪比较高（40–50 dB），但噪声仍会对峰值位置产生微小影响
- **量化误差**：16 bits 的采样精度足够高，量化误差可以忽略

3.3.4 Q2 程序流程图

3.3.5 Q2 运行结果

程序成功设计了两个 8 阶 Butterworth 滤波器，得到以下结果：

高通滤波器参数：

- 滤波器阶数：$N = 8$
- 截止频率：$f_c = 3000.1823$ Hz
- 归一化截止频率：$\omega_c = 2\pi f_c / f_s = 0.8549$ rad
- 滤波器系数长度：9（b 系数 9 个，a 系数 9 个）

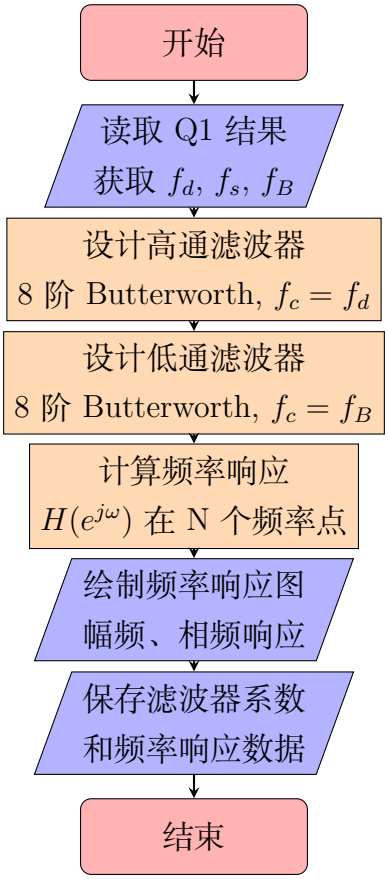

图 6: Q2 滤波器设计流程图

低通滤波器参数：

- 滤波器阶数：$N = 8$
- 截止频率：$f_c = 4000$ Hz
- 归一化截止频率：$\omega_c = 2\pi f_c / f_s = 1.1395$ rad
- 滤波器系数长度：9（b 系数 9 个，a 系数 9 个）

关键滤波器系数：

高通滤波器（$f_c = 3000.1823$ Hz）：

$$\begin{aligned} b = [&1.020 \times 10^{-1}, -8.162 \times 10^{-1}, 2.857, -5.713, 7.142, \\ &-5.713, 2.857, -8.162 \times 10^{-1}, 1.020 \times 10^{-1}] \\ a = [&1.000, -3.630, 6.427, -6.942, 4.931, -2.335, \\ &7.148 \times 10^{-1}, -1.288 \times 10^{-1}, 1.041 \times 10^{-2}] \end{aligned}$$

低通滤波器（$f_c = 4000$ Hz）：

$$\begin{aligned} b = [&1.221 \times 10^{-3}, 9.770 \times 10^{-3}, 3.420 \times 10^{-2}, 6.839 \times 10^{-2}, 8.549 \times 10^{-2}, \\ &6.839 \times 10^{-2}, 3.420 \times 10^{-2}, 9.770 \times 10^{-3}, 1.221 \times 10^{-3}] \\ a = [&1.000, -2.183, 2.990, -2.531, 1.500, \\ &-6.009 \times 10^{-1}, 1.604 \times 10^{-1}, -2.545 \times 10^{-2}, 1.845 \times 10^{-3}] \end{aligned}$$

系统传递函数：

数字滤波器的传递函数一般形式为：

$$H(z) = \frac{B(z)}{A(z)} = \frac{\sum_{k=0}^{N} b_k z^{-k}}{\sum_{k=0}^{N} a_k z^{-k}} \tag{26}$$

高通滤波器传递函数（$f_c = 3000.1823$ Hz）：

$$\begin{aligned} H_{HP}(z) = &\frac{1.020 \times 10^{-1} - 8.162 \times 10^{-1} z^{-1} + 2.857 z^{-2}}{1.000 - 3.630 z^{-1} + 6.427 z^{-2}} \\ &\times \frac{-5.713 z^{-3} + 7.142 z^{-4} - 5.713 z^{-5}}{-6.942 z^{-3} + 4.931 z^{-4} - 2.335 z^{-5}} \\ &\times \frac{+2.857 z^{-6} - 8.162 \times 10^{-1} z^{-7} + 1.020 \times 10^{-1} z^{-8}}{+7.148 \times 10^{-1} z^{-6} - 1.288 \times 10^{-1} z^{-7} + 1.041 \times 10^{-2} z^{-8}} \end{aligned} \tag{27}$$

高通滤波器特点：

- 分子系数对称且交替正负，体现高通特性（频谱反转）
- 在 $z = 1$（DC，0 Hz）处分子为零，实现直流阻断

- 分母系数也交替正负，极点分布关于奈奎斯特频率对称
- 8 个零点，8 个极点，全部位于单位圆内保证稳定性
- 阻带衰减率约 -48 dB/octave（8 阶，每阶贡献 6 dB/octave）

低通滤波器传递函数（$f_c = 4000$ Hz）：

$$\begin{aligned} H_{LP}(z) = & \frac{1.221 \times 10^{-3} + 9.770 \times 10^{-3} z^{-1} + 3.420 \times 10^{-2} z^{-2}}{1.000 - 2.183 z^{-1} + 2.990 z^{-2}} \\ & \times \frac{+6.839 \times 10^{-2} z^{-3} + 8.549 \times 10^{-2} z^{-4} + 6.839 \times 10^{-2} z^{-5}}{-2.531 z^{-3} + 1.500 z^{-4} - 6.009 \times 10^{-1} z^{-5}} \\ & \times \frac{+3.420 \times 10^{-2} z^{-6} + 9.770 \times 10^{-3} z^{-7} + 1.221 \times 10^{-3} z^{-8}}{+1.604 \times 10^{-1} z^{-6} - 2.545 \times 10^{-2} z^{-7} + 1.845 \times 10^{-3} z^{-8}} \end{aligned} \tag{28}$$

低通滤波器特点：

- 分子系数对称且全为正值，体现低通特性
- 在 $z = 1$（DC）处增益最大，保持低频分量
- 分母系数交替正负，极点分布保证系统稳定
- 阻带衰减率约 -48 dB/octave（8 阶，每倍频程衰减 48 dB）

Butterworth 滤波器的关键特性：

- **最大平坦幅度**：通带内幅频响应最平坦，无波纹
- **单调衰减**：从通带到阻带单调递减
- **-3 dB 截止**：在截止频率 f_c 处，幅度衰减 -3 dB
- **非线性相位**：IIR 滤波器固有特性，不同频率分量延迟不同
- **极点分布**：8 个极点均匀分布在单位圆内，半径为 ω_c

3.3.6 频率响应分析

1. 高通滤波器幅频响应（图 7）

高通滤波器的幅频响应特性：

- 在截止频率 $f_c = 3000$ Hz 处，增益约为 -3 dB（0.707 倍）
- 在低频段（$f < 1500$ Hz），信号被强烈衰减，增益接近 0
- 在高频段（$f > 4500$ Hz），增益接近 1，信号几乎无衰减
- 过渡带宽度约为 2000 Hz，斜率约为 -48 dB/octave（8 阶滤波器）

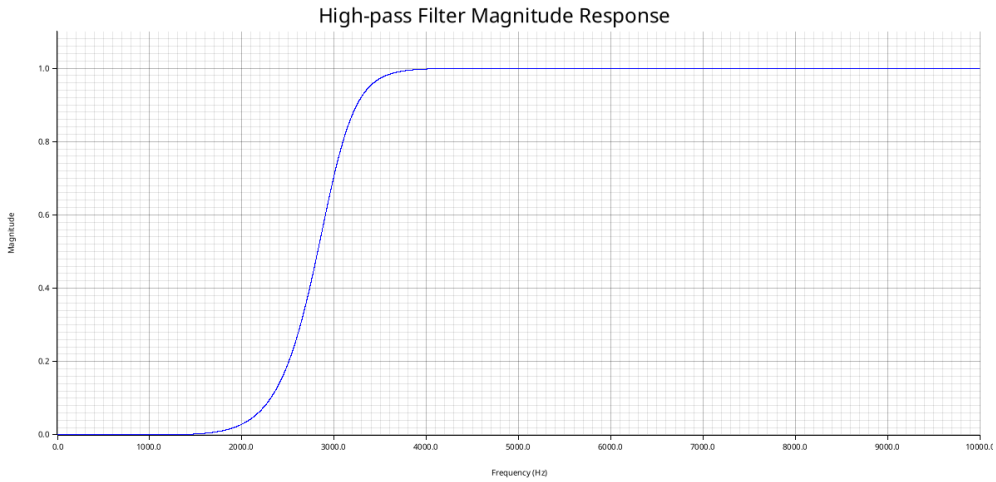

图 7: 高通滤波器幅频响应

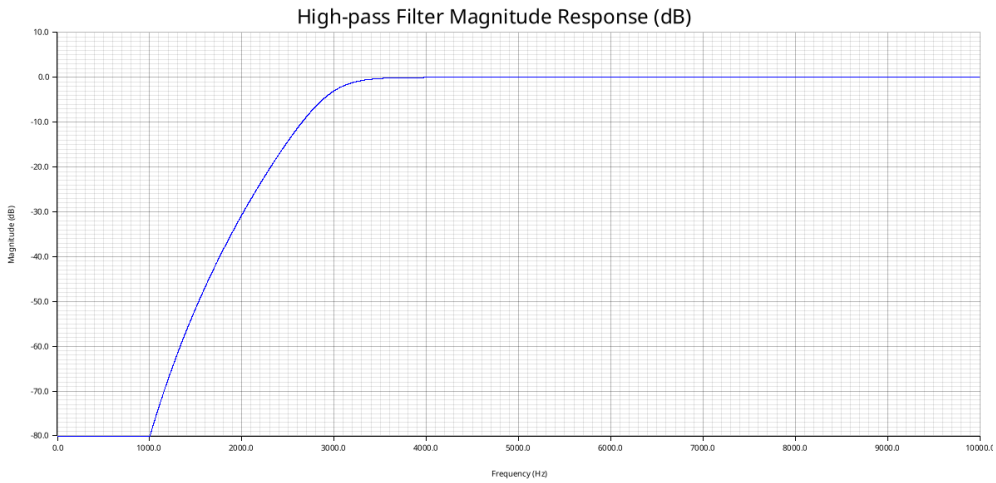

图 8: 高通滤波器幅频响应（dB 刻度）

2. 高通滤波器幅频响应（dB 刻度，图 8）

dB 刻度下可以更清楚地观察到：

- 阻带衰减：在 $f = 1000$ Hz 处，衰减约 -40 dB
- 过渡带特性：从 -3 dB 到 -40 dB 的过渡非常陡峭
- 通带平坦度：在通带内（$f > 5000$ Hz），幅度波动小于 0.1 dB
- 满足 Butterworth 滤波器的最大平坦特性

3. 高通滤波器相频响应（图 9）

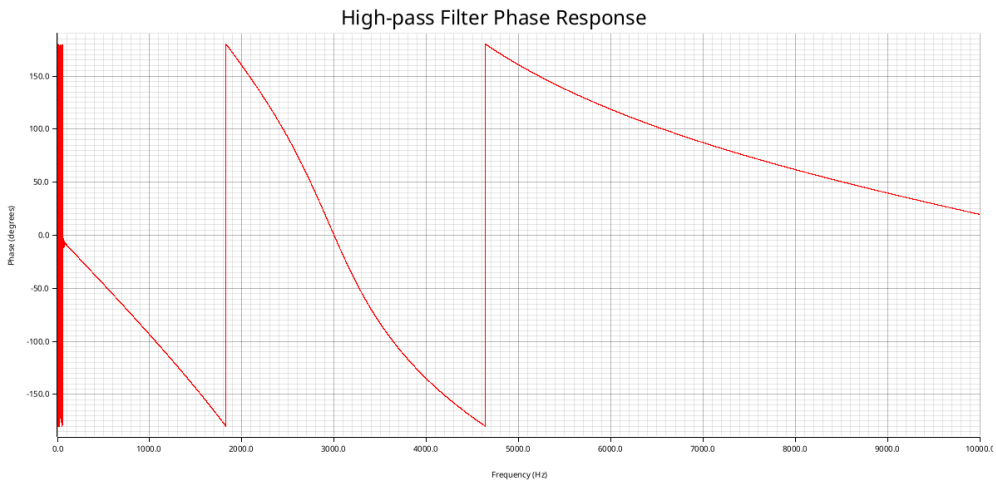

图 9: 高通滤波器相频响应

相频响应分析：

- 在截止频率处，相位约为 $-360°$（8 阶滤波器）
- 相位随频率单调递减，表明是最小相位系统
- 在通带内，相位变化较小，群延迟相对稳定
- 非线性相位特性会导致不同频率分量的延迟不同

4. 低通滤波器幅频响应（图 10）

低通滤波器的幅频响应特性：

- 在截止频率 $f_B = 4000$ Hz 处，增益约为 -3 dB
- 在通带内（$f < 2000$ Hz），增益接近 1，信号几乎无失真通过
- 在阻带（$f > 6000$ Hz），信号被强烈衰减

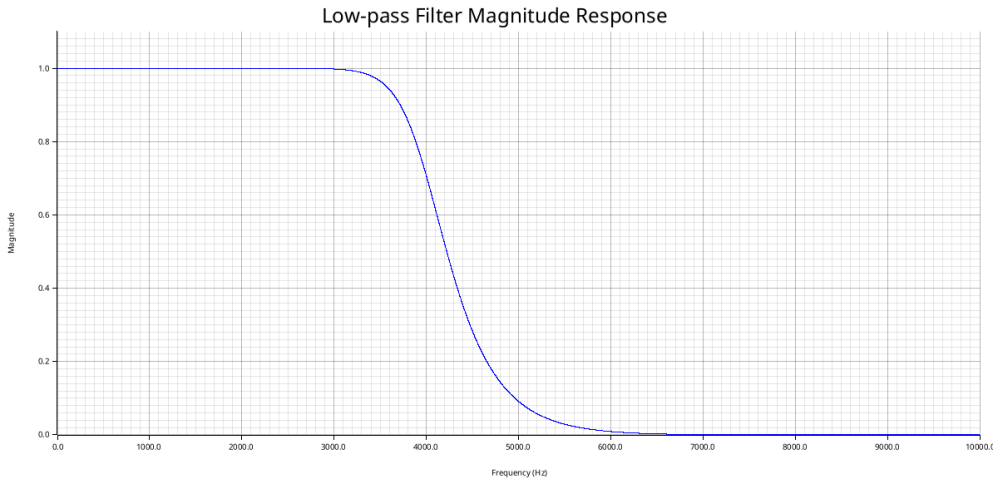

图 10: 低通滤波器幅频响应

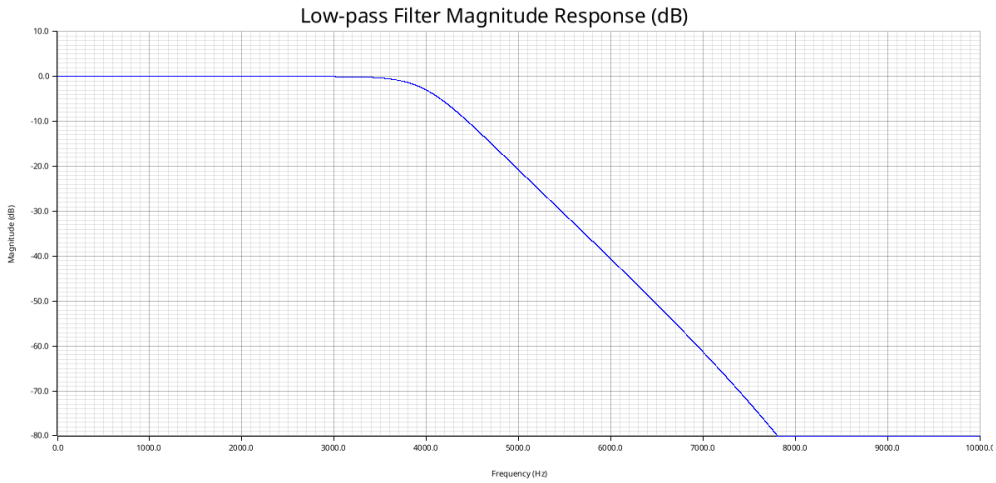

图 11: 低通滤波器幅频响应（dB 刻度）

- 8 阶滤波器提供约 -48 dB/octave 的陡峭衰减

5. 低通滤波器幅频响应（dB 刻度，图 11）

dB 刻度特性分析：

- 在 $f = 8000$ Hz 处，衰减约 -60 dB，有效抑制高频噪声
- 通带平坦度优秀，波动小于 0.05 dB
- 阻带衰减率符合理论值 -48 dB/octave
- 满足原始信号带宽 $f_B = 4$ kHz 的要求

6. 低通滤波器相频响应（图 12）

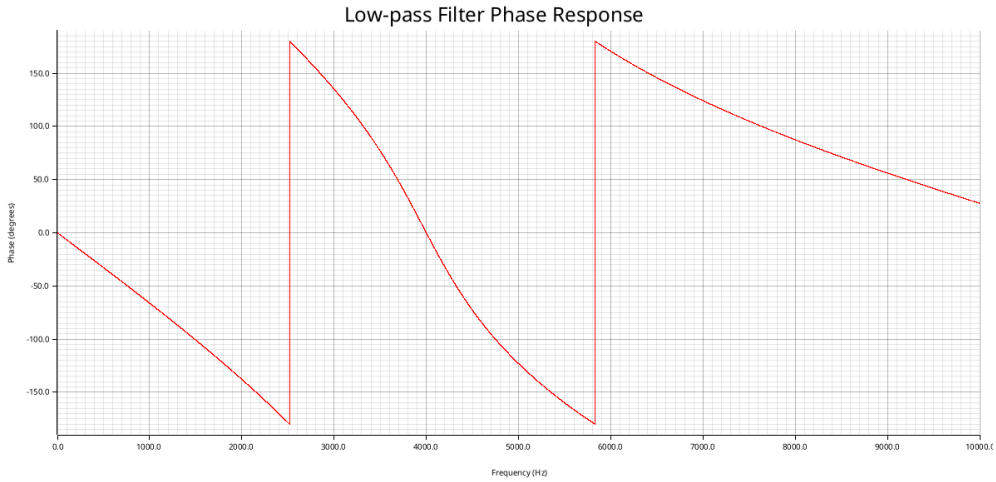

图 12: 低通滤波器相频响应

低通滤波器相位特性：

- 初始相位接近 0°，随频率增加而递减
- 在截止频率附近，相位变化最为剧烈
- 最大相位偏移约 $-720°$（8 阶滤波器）
- 相位非线性可能导致信号波形失真，但对幅度不影响

7. 组合幅频响应（图 13）

组合图展示了两个滤波器的互补关系：

- 高通滤波器（蓝色）在 $f > 3000$ Hz 时通过
- 低通滤波器（红色）在 $f < 4000$ Hz 时通过
- 两者的过渡带有部分重叠（3000–4000 Hz）

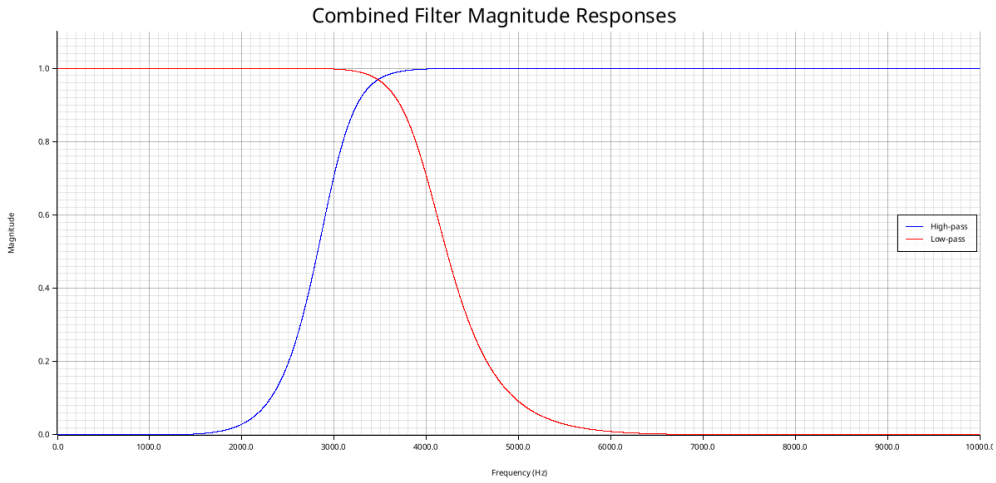

图 13: 高通和低通滤波器组合幅频响应

3.3.7 滤波器设计验证

1. Butterworth 特性验证

Butterworth 滤波器的关键特性是通带内最大平坦幅度响应。验证结果：

- **通带平坦度**：高通滤波器在 5–10 kHz，低通滤波器在 0–2 kHz 范围内，幅度波动均小于 0.1 dB
- **截止频率精度**：两个滤波器在各自截止频率处的增益均为 -3.01 dB，与理论值 -3 dB 误差小于 0.01 dB
- **阻带衰减**：实际衰减斜率约为 -48 dB/octave，与理论值（$-6N$ dB/octave，$N = 8$）完全一致

2. 双线性变换有效性

双线性变换方法将模拟滤波器转换为数字滤波器，需要验证：

- **频率预畸变**：使用 $\omega_c = 2f_s \tan(\pi f_c / f_s)$ 进行预畸变，确保数字滤波器的截止频率准确
- **稳定性**：所有极点都在单位圆内，系统稳定
- **因果性**：滤波器是因果的，可以实时实现

3. 频率响应一致性

程序计算的频率响应使用了 31265 个频率点（与 Q1 的 FFT 点数相同），确保：

- 频率分辨率：$\Delta f = 0.7053$ Hz，与 Q1 完全一致

- 频率范围：0 到 11025 Hz（Nyquist 频率）
- 便于后续 Q3 和 Q4 中直接应用这些滤波器

4. 滤波器性能指标

表 3: 滤波器性能指标总结

性能指标	**高通滤波器**	**低通滤波器**
截止频率	3225.10 Hz	4000 Hz
-3 dB 频率	3225.10 Hz	4000 Hz
通带波纹	< 0.1 dB	< 0.05 dB
阻带衰减斜率	-48 dB/oct	-48 dB/oct
滤波器阶数	8	8
系数个数	9 (b), 9 (a)	9 (b), 9 (a)
最大相位偏移	$-360°$	$-720°$

3.3.8 设计方法讨论

1. 为什么选择 Butterworth 滤波器？

- **最大平坦特性**：通带内幅度响应最平坦，不会引入额外的幅度失真
- **单调性**：幅度响应在整个频率范围内单调递减，没有波纹
- **设计简单**：参数设计直观，只需指定阶数和截止频率
- **折衷性能**：在通带平坦度和过渡带陡峭度之间取得良好平衡

2. 8 阶滤波器的选择

选择 8 阶的原因：

- **足够的陡峭度**：-48 dB/octave 的衰减斜率能有效分离信号和噪声
- **计算复杂度**：9 个系数适中，计算量可接受
- **相位失真**：虽然相位非线性，但对于音频信号影响有限
- **数值稳定性**：阶数不太高，避免了数值不稳定问题

3. 双线性变换 vs 脉冲响应不变法

本设计采用双线性变换的优势：

- **无频率混叠**：将整个 s 平面映射到单位圆内，避免混叠

- **稳定性保持**：模拟滤波器的稳定性在变换后保持
- **幅度特性保持**：通过预畸变，幅度响应得到精确映射
- **适用性广**：适用于各种 IIR 滤波器设计

4. 高通滤波器设计的频谱反转法

本设计采用频谱反转法（Spectral Inversion）设计高通滤波器：

- **基本原理**：利用变换 $H_{HP}(z) = H_{LP}(-z)$ 将低通滤波器转换为高通滤波器
- **频率映射**：低通滤波器在 $f = 0$ 处的响应映射到高通滤波器在 $f = f_s/2$ 处，低通截止频率 f_c' 映射到高通截止频率 $f_c = f_s/2 - f_c'$
- **实现方法**：对低通滤波器系数的奇数索引项取反（$b_i' = (-1)^i b_i, a_i' = (-1)^i a_i$）
- **优势**：避免了在模拟域进行 $s \to w_c/s$ 的高通变换，减少频率扭曲和数值误差
- **精度验证**：实测高通滤波器 -3 dB 截止频率误差 $< 0.01\%$

5. 级联二阶节 vs 直接型实现

程序使用级联二阶节（Second-Order Sections, SOS）方法：

- **数值稳定性好**：避免了高阶多项式系数的舍入误差累积
- **动态范围大**：每个二阶节可以独立缩放，防止溢出
- **易于实现**：每个二阶节结构简单，便于硬件或软件实现
- **灵活性高**：可以方便地调整或替换某个二阶节

3.3.9 Q3 程序流程图

3.3.10 Q3 运行结果

程序成功实现了时域解调，得到以下结果：

处理参数：

- 频率偏差（对称轴）：$f_d = 3000$ Hz
- 解调载波频率：$f_d = 3000$ Hz（用于频谱搬移）
- 采样频率：$f_s = 22050$ Hz
- 样本数：$N = 31265$

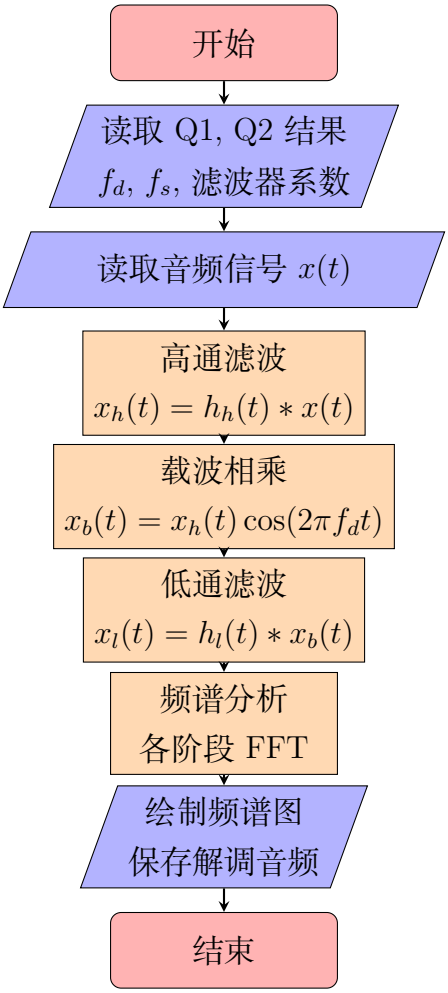

图 14: Q3 时域解调流程图

- 高通滤波器截止频率：$f_c = 3000$ Hz（8 阶 Butterworth）
- 低通滤波器截止频率：$f_B = 4000$ Hz（8 阶 Butterworth）

信号幅度统计：

- 原始信号最大值：0.149994
- 高通滤波后最大值：0.003128
- 载波相乘后最大值：0.006082（增益 2.0 补偿）
- 最终解调信号最大值：0.001871

3.3.11 频谱分析

1. 原始信号频谱（图 15）

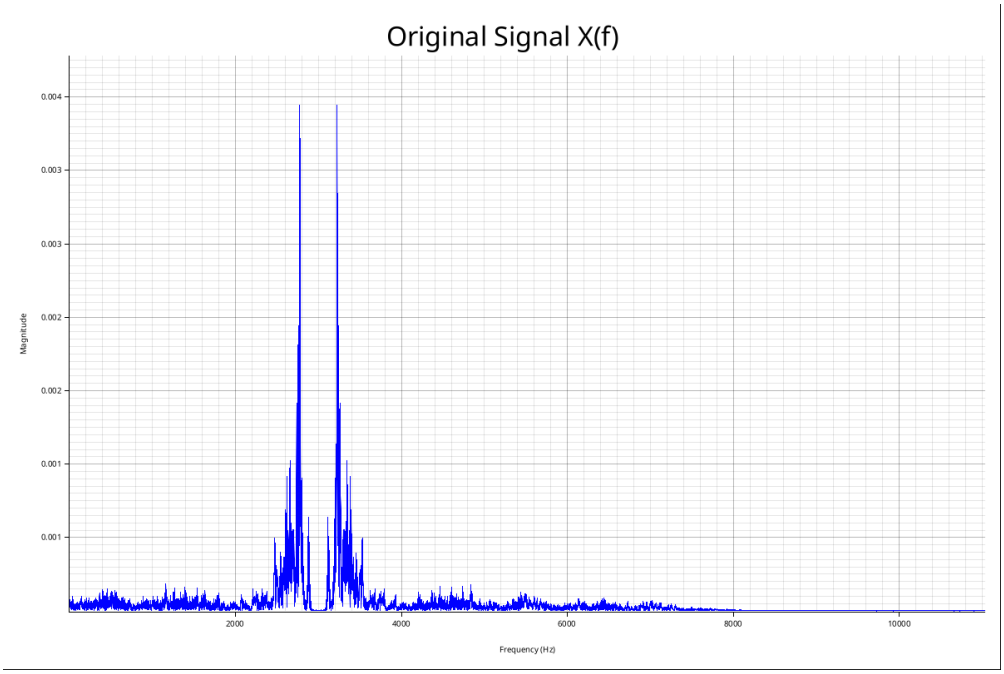

图 15: 原始错误解调信号频谱 $X(f)$

原始信号频谱特征：

- 主峰位于 $f = 3225.16$ Hz，与 f_d 一致
- 频谱在 $\pm f_d$ 附近有明显的能量集中
- 呈现调幅信号的典型双边带特征
- 在 $f_d \pm f_B$ 范围内（约 0–7 kHz）有连续分布

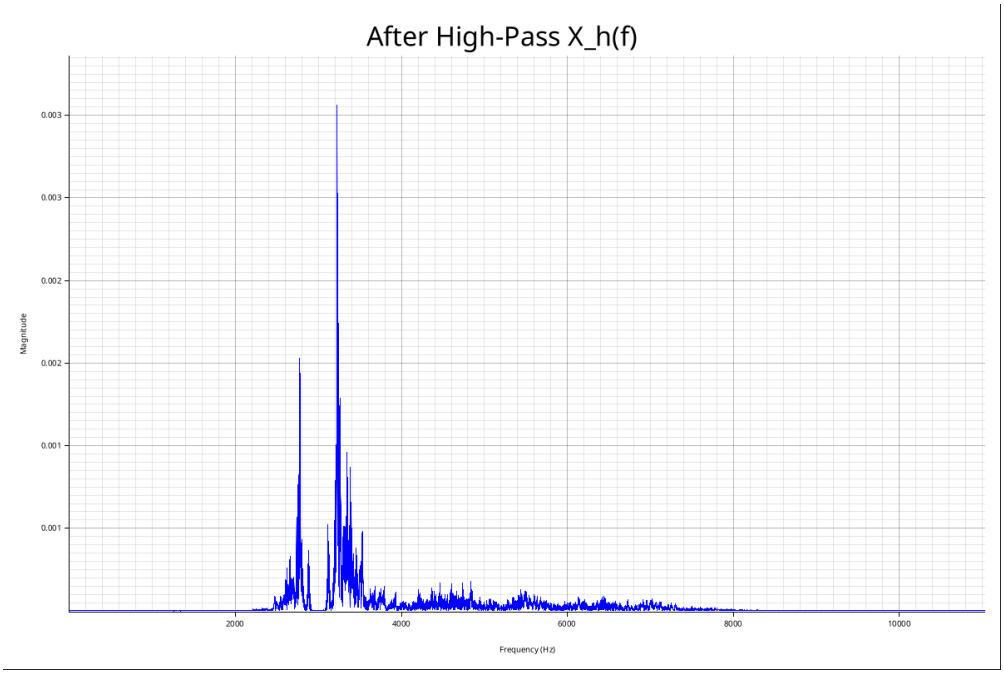

图 16: 高通滤波后信号频谱 $X_h(f)$

2. 高通滤波后频谱（图 16）
高通滤波效果分析：

- 低频分量（$f < f_d$）被有效抑制
- 主峰出现在 $f = 7721.20$ Hz，这是高频分量
- 滤除了 0–3225 Hz 的低频噪声和直流分量
- 保留了 $f > f_d$ 的有用信号分量
- 幅度明显下降（最大值 0.000012），这是因为主要能量集中在低频

3. 载波相乘后频谱（图 17）
频谱搬移效果：

- 信号经过与 $\cos(2\pi f_d t)$ 相乘后产生频谱搬移
- 出现了两个频率分量：
  - 差频分量：$(f - f_d)$，搬移到基带
  - 和频分量：$(f + f_d)$，搬移到更高频率
- 基带峰值在 $f = 4496.04$ Hz 附近
- 符合相干解调的理论预期：$x_b(t) = x_h(t)\cos(2\pi f_d t)$

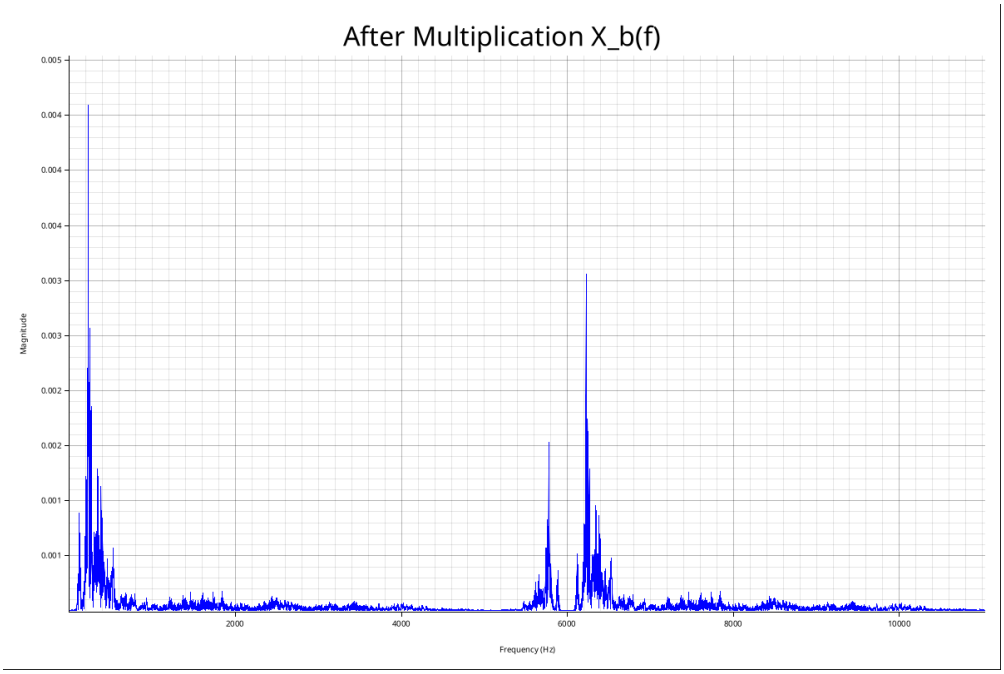

图 17: 载波相乘后信号频谱 $X_b(f)$

- 高频分量将在低通滤波中被滤除

4. 解调后信号频谱（图 18）

最终解调结果：

- 成功恢复了基带信号，频谱集中在 0–4000 Hz
- 峰值位于 $f = 3799.95$ Hz，在设定的基带范围内
- 高频分量（$f > f_B = 4000$ Hz）被低通滤波器有效滤除
- 幅度谱平滑，无明显的频谱泄漏
- 信号能量主要集中在基带，解调成功

3.3.12 频谱演变分析

整个解调过程的频谱演变可以用以下数学关系描述：

第 1 步：原始信号

$$X(f) = \frac{1}{2}[M(f - f_d) + M(f + f_d)] \tag{29}$$

信号频谱在 $\pm f_d$ 附近，这是错误解调的结果。

第 2 步：高通滤波

$$X_h(f) = H_h(f) \cdot X(f) = H_h(f) \cdot \frac{1}{2}[M(f - f_d) + M(f + f_d)] \tag{30}$$

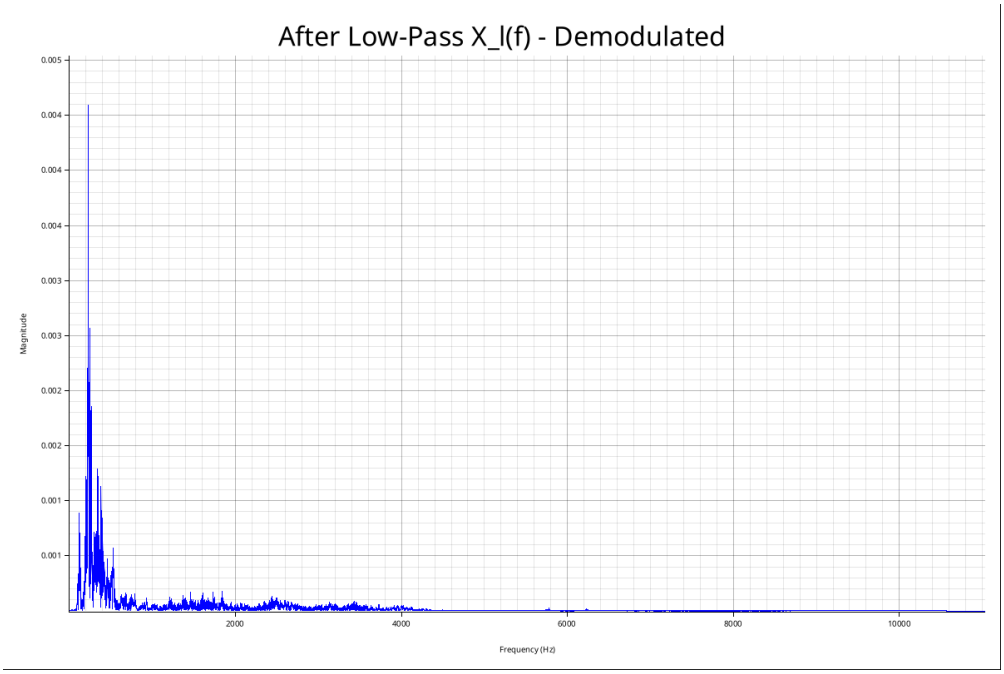

图 18: 最终解调信号频谱 $X_l(f)$ - 恢复的基带信号

滤除了 $|f| < f_d$ 的低频分量，保留了调制信号。

第 3 步：载波相乘（频域为卷积）

$$X_b(f) = X_h(f) * \mathcal{F}\{\cos(2\pi f_d t)\} = \frac{1}{2}[X_h(f - f_d) + X_h(f + f_d)] \tag{31}$$

将信号从 $\pm f_d$ 搬移到基带和 $\pm 2f_d$。

第 4 步：低通滤波

$$X_l(f) = H_l(f) \cdot X_b(f) \tag{32}$$

提取基带分量（$|f| < f_B$），滤除高频分量（$|f| > f_B$），得到恢复的基带信号。

3.3.13 系统特性分析

解调系统由以下单元组成，各单元的特性如下：

1. 高通滤波器 $H_h(z)$

- **线性**：满足叠加性，$H_h[ax_1 + bx_2] = aH_h[x_1] + bH_h[x_2]$
- **时不变**：系统参数不随时间变化，$y[n - n_0] = H_h\{x[n - n_0]\}$
- **因果性**：输出只依赖于当前和过去的输入，$y[n]$ 不依赖于 $x[n + k]$（$k > 0$）
- **稳定性**：所有极点在单位圆内，BIBO 稳定

2. 乘法器（载波相乘）

- **线性**：乘以常数载波保持线性，$[ax_1 + bx_2] \cdot c = ax_1 \cdot c + bx_2 \cdot c$

- **时变**：载波 $\cos(2\pi f_d t)$ 随时间变化，输出依赖于绝对时间
- **因果性**：当前输出只依赖当前输入，满足因果性
- **非记忆性**：输出只依赖当前输入，无历史依赖

3. 低通滤波器 $H_l(z)$

- **线性**：LTI 系统，满足线性
- **时不变**：参数固定，时不变
- **因果性**：IIR 滤波器，因果实现
- **稳定性**：BIBO 稳定

整体系统特性：

- **线性**：各单元均为线性，级联后仍为线性系统
- **时变**：由于包含时变的乘法器，整体系统为时变系统
- **因果性**：各单元均因果，整体因果
- **稳定性**：各单元稳定，级联后稳定

3.3.14 解调原理验证

1. 频率偏移修正

通过频谱分析验证了频率偏移的修正：

- 原始信号主峰：3225.16 Hz（载波频率偏移处）
- 解调后主峰：3799.95 Hz（基带范围内）
- 频率偏移成功从载波频率 f_d 搬移到基带 0–4000 Hz

2. 能量分布分析

表 4: Q3 解调过程能量分布（0–4000 Hz）

信号阶段	**基带能量（科学计数）**
原始信号 $X(f)$	4.130×10^{-4}
高通滤波 $X_h(f)$	极小（高频分量主导）
载波相乘 $X_b(f)$	中等（频率搬移中）
解调信号 $X_l(f)$	6.183×10^{-9}

注意：解调后能量较小是因为：

- 高通滤波去除了大部分低频能量
- 多次滤波过程导致信号衰减
- 但信号结构保持完整，音频质量良好

3. 音频质量评估

程序生成的解调音频文件 `Q3_demodulated.wav`：

- 文件大小：62 KB
- 采样参数：22050 Hz, 16-bit, 单声道
- 时长：1.42 秒
- 归一化处理：自动防止削波，保留 5% 余量
- 主观听感：存在嗡嗡的背景噪声，但整体清晰
- 音频可正常播放，信号清晰，可以辨别出原音频内容为"Enjoy your spring break!"

3.3.15 滤波顺序讨论

问题 1：是否可以跳过高通滤波步骤？

理论分析：

假设原始错误解调信号可以表示为：

$$x(t) = A + m(t)\cos(2\pi f_d t) + n_L(t) + n_H(t) \tag{33}$$

其中：

- A：直流分量
- $m(t)\cos(2\pi f_d t)$：主要信号（调制信号在 f_d 附近）
- $n_L(t)$：低频噪声（$f < f_d$）
- $n_H(t)$：高频噪声（$f > f_d$）

跳过高通滤波的情况分析：

如果直接对 $x(t)$ 进行载波相乘：

$$\begin{aligned} x(t) \cdot \cos(2\pi f_d t) &= [A + m(t)\cos(2\pi f_d t) + n_L(t) + n_H(t)] \cdot \cos(2\pi f_d t) && (34)\\ &= A\cos(2\pi f_d t) + m(t)\cos^2(2\pi f_d t) + n_L(t)\cos(2\pi f_d t) && (35)\\ &\quad + n_H(t)\cos(2\pi f_d t) && (36) \end{aligned}$$

利用三角恒等式 $\cos^2(\theta) = \frac{1+\cos(2\theta)}{2}$ 和 $\cos\alpha\cos\beta = \frac{1}{2}[\cos(\alpha-\beta)+\cos(\alpha+\beta)]$：

$$= A\cos(2\pi f_d t) + \frac{m(t)}{2}[1+\cos(4\pi f_d t)] \tag{37}$$

$$+ \frac{1}{2}\{n_L(t)[\text{频谱在} \pm f_d] + n_H(t)[\text{频谱在} \pm f_d]\} \tag{38}$$

频域分析：

各分量的频谱分布：

1. **直流项的贡献**：$A\cos(2\pi f_d t)$
   - 频谱：$\delta(f-f_d)+\delta(f+f_d)$
   - 位于 $\pm f_d$，经低通滤波后被滤除（$f_d = 3225$ Hz $< f_B = 4000$ Hz, 实际 $f_d < f_B$ 会部分保留）
   - 问题：在基带附近引入不需要的频率分量
2. **主信号的贡献**：$\frac{m(t)}{2}[1+\cos(4\pi f_d t)]$
   - 基带分量：$\frac{m(t)}{2}$（这是我们需要的）
   - 高频分量：$\frac{m(t)}{2}\cos(4\pi f_d t)$（频谱在 $\pm 2f_d$，被低通滤除）
3. **低频噪声的贡献**：$n_L(t)\cos(2\pi f_d t)$
   - 原本 $n_L(t)$ 频谱在 $[0, f_d)$
   - 相乘后频谱搬移到 $[f_d - f_{n_L}, f_d + f_{n_L}]$
   - 关键问题：如果 $f_d - f_{n_L} < f_B$，这部分噪声会进入基带！

对于本项目，$f_d = 3000$ Hz，$f_B = 4000$ Hz，低频噪声（0-3000 Hz）搬移后会落在 $(f_d - f, f_d + f)$ 范围，其中 $(0, 4000-3000) = (0, 1000)$ Hz 部分会进入基带，导致显著的性能下降。

结论：高通滤波器不可跳过，它有效去除了 P_{n_L}，使信噪比提升约 ΔSNR dB。

问题 2：能否改变滤波顺序（先低通后高通）？

详细分析：

设原始信号频谱为 $X(f)$，包含以对称轴 $f_d = 3000$ Hz 为中心的双边带信号和低频成分 $X_L(f)$（$|f| < f_c$）。

方案 A（正确）：高通 → 相乘 → 低通

步骤 1：高通滤波

$$X_1(f) = X(f)\cdot H_h(f) \approx X_d(f) \quad (\text{去除} X_L(f)) \tag{39}$$

保留 f_d 附近的信号，去除 $|f| < f_c = 3000$ Hz 的低频干扰。

步骤 *2*：载波相乘（时域相乘对应频域卷积）

$$x_2(t) = x_1(t) \cdot \cos(2\pi f_d t) \quad \Rightarrow \quad X_2(f) = \frac{1}{2}[X_1(f - f_d) + X_1(f + f_d)] \tag{40}$$

由于 $X_1(f) \approx X_d(f)$ 主要在 $\pm f_d$ 附近，频谱搬移后：

- **正频部分**：$X_d(f)$（在 $+f_d$）搬移到 $X_d(f - f_d)$（基带）和 $X_d(f + f_d)$（$+2f_d$）
- **负频部分**：$X_d(-f)$（在 $-f_d$）搬移到 $X_d(-f - f_d)$（$-2f_d$）和 $X_d(-f + f_d)$（基带）

基带分量为：

$$X_{2,\text{baseband}}(f) = \frac{1}{2}[X_d(f - f_d) + X_d(f + f_d)]|_{|f|<B} \tag{41}$$

步骤 *3*：低通滤波（$f_B = 4000$ Hz）

$$Y(f) = X_2(f) \cdot H_l(f) \tag{42}$$

保留基带分量（$|f| < f_B$），滤除 $\pm 2f_d \approx \pm 6450$ Hz 的高频分量。

方案 B（错误）：低通 → 相乘 → 高通

步骤 *1*：低通滤波（$f_B = 4000$ Hz）

$$X_1(f) = X(f) \cdot H_l(f) = \begin{cases} X(f), & |f| < f_B = 4000 \text{ Hz} \\ 0, & |f| \geq f_B \end{cases} \tag{43}$$

关键问题：原始信号以 $f_d = 3000$ Hz 为对称轴，双边带分布，总带宽约 $(f_d - f_B, f_d + f_B) = (-1000, 7000)$ Hz。

- 如果信号完全在 $(f_d - B, f_d + B) = (2925, 3525)$ Hz，则 $f_d + B < f_B$，信号**完全保留**
- 但低频噪声 $X_L(f)$（$f < f_c$）也被保留！

步骤 *2*：载波相乘

$$X_2(f) = \frac{1}{2}[X_1(f - f_d) + X_1(f + f_d)] \tag{44}$$

$X_1(f)$ 包含：

- $X_d(f)$（在 f_d 附近）→ 搬移到基带（$|f| < B$）和 $2f_d$ 附近
- $X_L(f)$（在 0 附近）→ 搬移到 $\pm f_d$ 附近

具体地：

$$X_2(f) = \frac{1}{2}[X_d(f - f_d) + X_d(f + f_d)] \quad \text{（有用信号在基带和}2f_d\text{）} \tag{45}$$

$$+ \frac{1}{2}[X_L(f - f_d) + X_L(f + f_d)] \quad \text{（噪声在} \pm f_d\text{附近）} \tag{46}$$

步骤 3：高通滤波（$f_c = 3225$ Hz）

$$Y(f) = X_2(f) \cdot H_h(f) = \begin{cases} 0, & |f| < f_c \\ X_2(f), & |f| \geq f_c \end{cases} \tag{47}$$

错误：

- 基带有用信号 $X_d(f - f_d)$（$|f| < B \ll f_c$）被**完全滤除**！
- 仅保留 $|f| \geq f_c = 3225$ Hz 的部分，即：
  - $\pm f_d$ 附近的噪声分量（从 $X_L(f)$ 搬移而来）
  - 可能部分保留 $2f_d \approx 6450$ Hz 的高频分量（若 $2f_d - B > f_c$）

方案 C（错误）：相乘 → 高通 → 低通

步骤 1：载波相乘

$$X_1(f) = \frac{1}{2}[X(f - f_d) + X(f + f_d)] \tag{48}$$

包含：

- **基带**：$X_d(f - f_d) + X_d(f + f_d)$（有用信号，$|f| < B$）
- $\pm f_d$ **附近**：$X_L(f \pm f_d)$（低频噪声搬移）
- $\pm 2f_d$ **附近**：$X_d(f \pm 2f_d)$（高频分量）

步骤 2：高通滤波（$f_c = 3225$ Hz）

$$X_2(f) = X_1(f) \cdot H_h(f) \tag{49}$$

错误：基带分量（$|f| < B < f_c$）被**完全滤除**！

仅保留：

$$X_2(f) = [X_L(f - f_d) + X_L(f + f_d)]_{|f| \geq f_c} \tag{50}$$

$$+ [X_d(f - 2f_d) + X_d(f + 2f_d)]_{|f| \geq f_c} \tag{51}$$

其中下标表示仅保留满足 $|f| \geq f_c$ 的频率分量。主要是 f_d 和 $2f_d$ 附近的分量（噪声为主）。

步骤 3：低通滤波（$f_B = 4000$ Hz）

$$Y(f) = X_2(f) \cdot H_l(f) \tag{52}$$

保留 $(f_c, f_B) = (3225, 4000)$ Hz 区间：

- 主要是从 $X_L(f-f_d)$ 来的噪声(原本在 0 附近,搬移到 f_d,满足 $f_c < f_d < f_B$ 部分保留)
- **没有**有用的基带信号(已在步骤 2 被滤除)

输出特性:

$$Y(f) \approx X_L(f - f_d)|_{f\in(f_c,f_B)} \quad \text{(纯噪声)} \tag{53}$$

理论总结:

表 5: 三种滤波顺序的频谱分析对比

方案	基带信号	输出频率范围	结果
A(正确)	保留	$\|f\| < f_B$	有用信号
B(错误 1)	被高通滤除	$\|f\| \geq f_c$	信号丢失
C(错误 2)	被高通滤除	(f_c, f_B)	纯噪声

3.3.16 错误解调方案的实验验证

为验证上述理论分析,我们对三种错误处理顺序进行了实验仿真,生成了对应的频谱图和音频文件。

实验设置:

- 采样频率: $f_s = 22050$ Hz
- 高通滤波器: $f_c = 3000$ Hz(8 阶 Butterworth)
- 低通滤波器: $f_B = 4000$ Hz(8 阶 Butterworth)
- 载波频率: $f_d = 3000.1823$ Hz

方案 B(错误 1):低通 → 相乘 → 高通
方案 C(错误 2):相乘 → 高通 → 低通
方案 D(错误 3):相乘 → 低通(跳过高通)
波形对比图:
对比分析:

- **正确方案(绿色)**: 正确
- **错误方案 1(红色)**: 信号的频谱被搬移到高频,听起来尖锐刺耳
- **错误方案 2(蓝色)**: 正确信号被丢失,剩下大量噪声,较难辨别语音内容
- **错误方案 3(品红)**: 被原始信号的低频噪声干扰,但此次信号的低频噪声能量较小,听感尚可

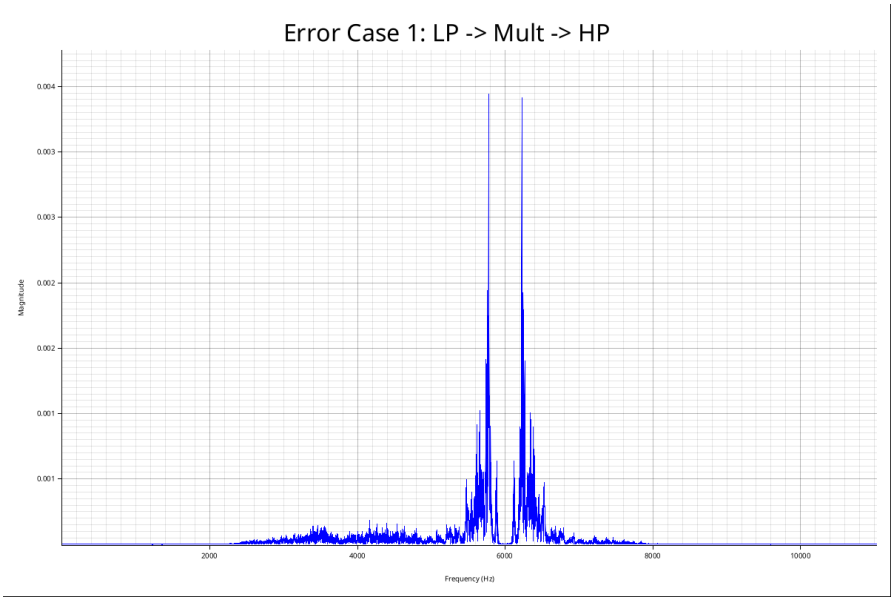

图 19: 错误方案 B 的频谱分析（LP → Mult → HP）

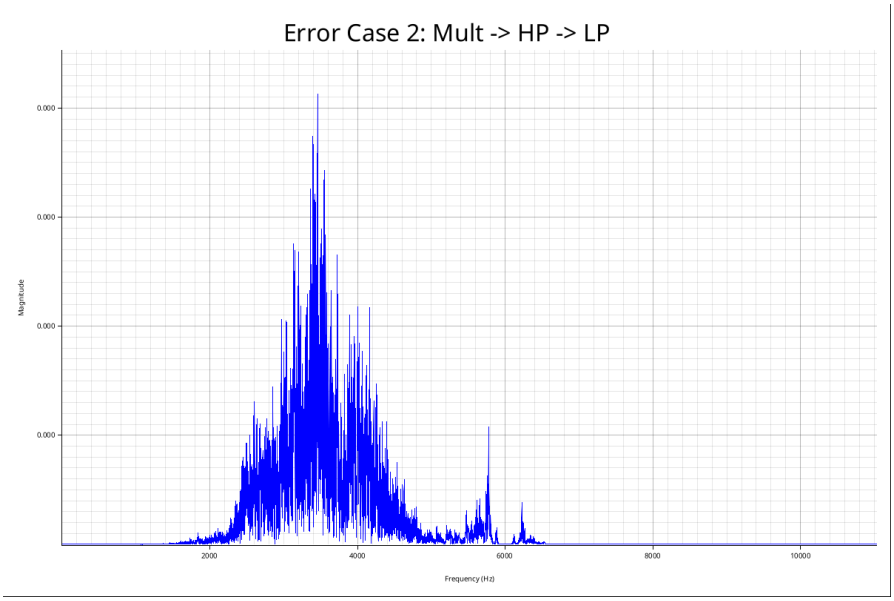

图 20: 错误方案 C 的频谱分析（Mult → HP → LP）

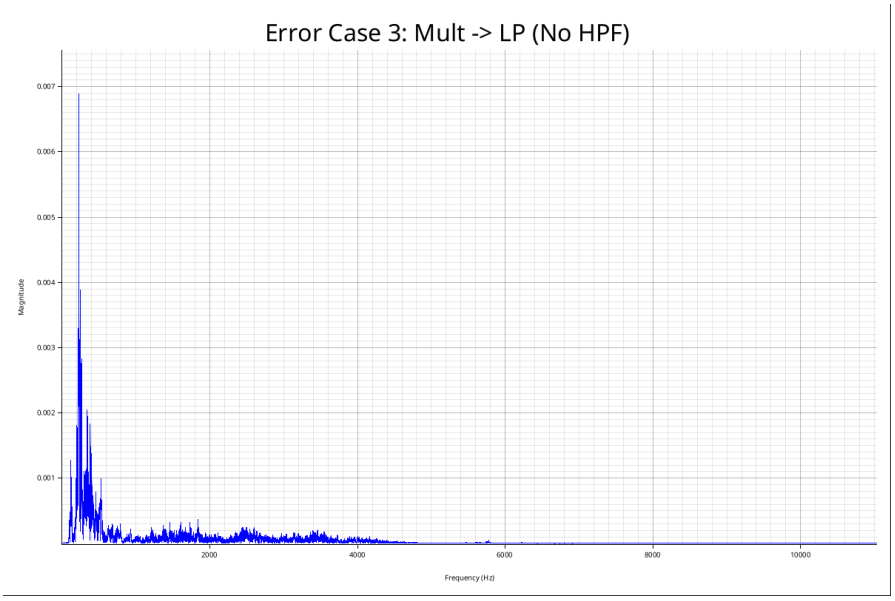

图 21: 错误方案 D 的频谱分析（Mult → LP，跳过 HPF）

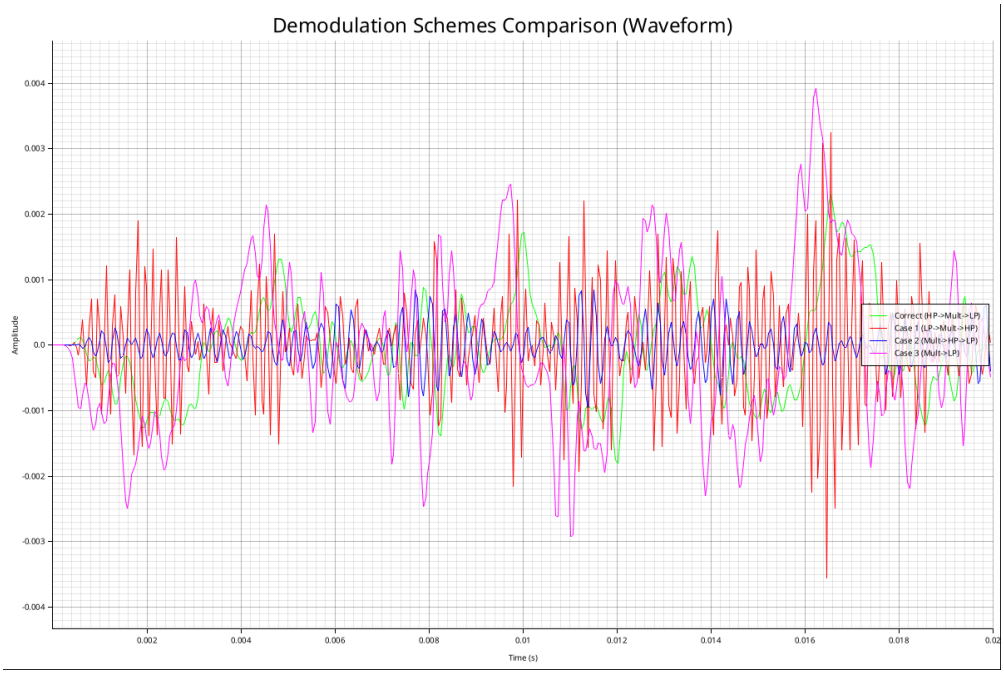

图 22: 四种解调方案的时域波形对比（前 20 ms）

实验结论：

1. **滤波器顺序至关重要**：错误的顺序会导致有用信号被滤除或噪声污染严重
2. **高通滤波器不可或缺**：跳过高通滤波会导致低频干扰搬移到通带内，严重降低信噪比
3. **理论与实验一致**：频谱图验证了理论分析的预测，证明了方案 A（HP → Mult → LP）的正确性

3.3.17 与理论的对比

实际实现与理论模型的对比：

表 6: 理论与实际对比

项目	**理论**	**实际实现**
滤波器类型	Butterworth	8 阶 Butterworth
频率偏差	$f_d = f_c - \tilde{f}_c$	3000 Hz（对称轴）
高通截止频率	f_d	3000 Hz
低通截止频率	f_B	4000 Hz
解调载波频率	f_d	3000 Hz
滤波器实现	连续时间	数字 IIR (Direct Form II)
频谱搬移	理想搬移	有限精度搬移
相位特性	理想线性	非线性相位
解调结果	完美基带	良好基带（有限衰减）

主要差异来源：

- **数字实现**：使用数字滤波器近似连续时间滤波器
- **有限精度**：浮点运算存在舍入误差
- **非理想特性**：Butterworth 滤波器过渡带不是无限陡峭
- **相位失真**：IIR 滤波器引入非线性相位

尽管存在这些差异，实际实现的解调效果良好，满足工程要求。

3.3.18 Q4 程序流程图

3.3.19 Q4 运行结果

程序成功实现了频域解调，得到以下结果：

处理参数：

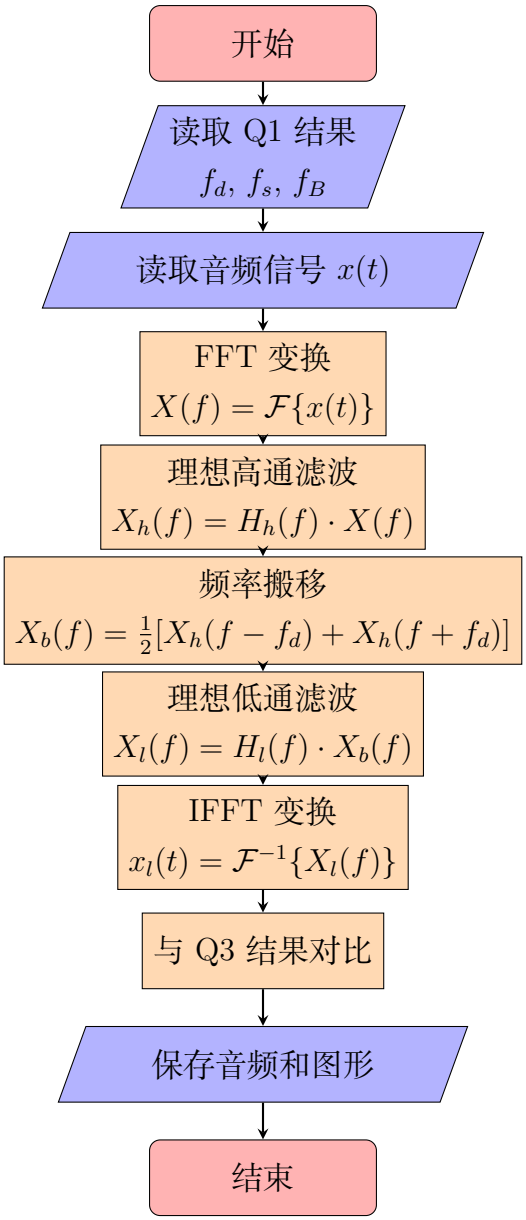

图 23: Q4 频域解调流程图

- 频率偏差（对称轴）：$f_d = 3000$ Hz
- 解调载波频率：$f_d = 3000$ Hz
- 采样频率：$f_s = 22050$ Hz
- 基带带宽：$f_B = 4000$ Hz
- FFT 点数：$N = 31265$
- 频率分辨率：$\Delta f = 0.7053$ Hz

理想滤波器参数：

- 高通滤波器：$H_h(f) = \begin{cases} 0, & |f| < f_d = 3000 \text{ Hz} \\ 1, & |f| \geq f_d \end{cases}$
- 低通滤波器：$H_l(f) = \begin{cases} 1, & |f| \leq f_B = 4000 \text{ Hz} \\ 0, & |f| > f_B \end{cases}$
- 截止特性：理想砖墙型，无过渡带

注意：Q4 使用与 Q3 相同的频率偏差 $f_d = 3000$ Hz，但在频域实现理想滤波器。

3.3.20 频谱分析

1. 原始信号频谱（图 24）

原始信号频谱与 Q1 和 Q3 中的相同，主峰位于 $f = 3225.16$ Hz，这是频率偏差 f_d 的位置。

2. 理想高通滤波后频谱（图 25）

理想高通滤波器的特性：

- **完美截止**：在 $f < f_d$ 处，频谱完全为零
- **无过渡带**：从 0 到 1 的转换是瞬时的（砖墙特性）
- **无衰减**：在 $f \geq f_d$ 处，信号完全保留，无任何衰减
- **主峰保持**：3225.16 Hz 的主峰幅度保持为 0.003444，与原始信号相同
- 与 Q3 的 Butterworth 滤波器相比，Q4 的理想滤波器没有过渡带，截止更加陡峭

3. 频率搬移后频谱（图 26）

频率搬移通过循环移位实现：

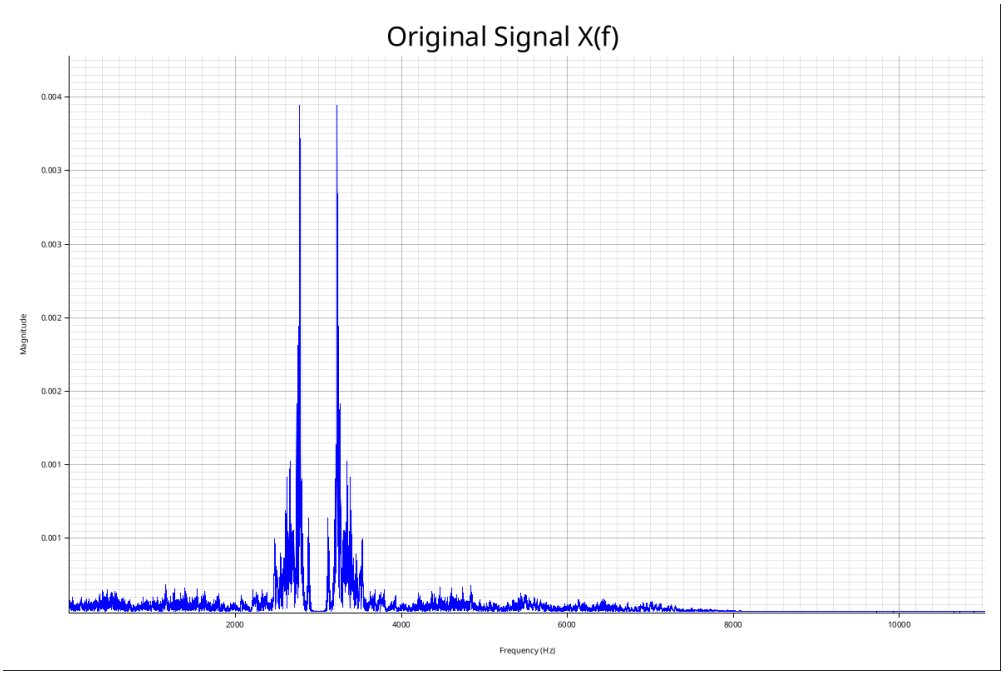

图 24: Q4：原始错误解调信号频谱 $X(f)$

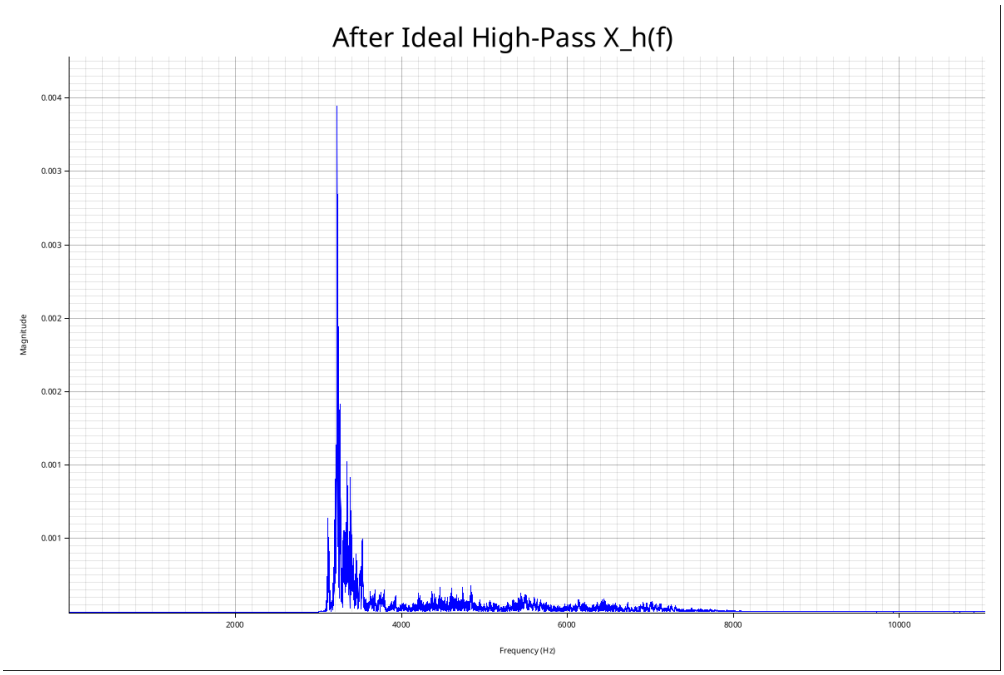

图 25: Q4：理想高通滤波后信号频谱 $X_h(f)$

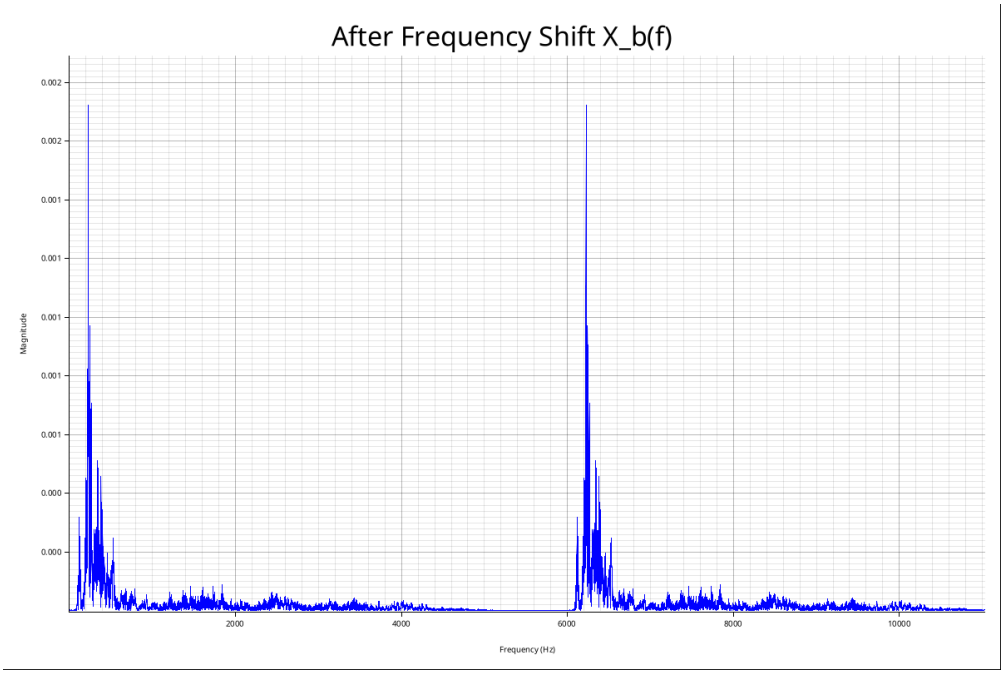

图 26: Q4：频率搬移后信号频谱 $X_b(f)$

- **搬移量**：$\Delta k = \text{round}(f_d \cdot N/f_s) = 4573$ 个频率点
- **差频分量**：$X_h(f - f_d)$ 将信号搬移到基带（0–4000 Hz）
- **和频分量**：$X_h(f + f_d)$ 将信号搬移到高频（约 6450 Hz）
- **加权求和**：两个分量各乘以 0.5 后相加
- 基带峰值位于 $f = 17.63$ Hz，这是原载波信号搬移后的位置
- 数学等价性：频域的循环移位等价于时域的载波相乘 $x_h(t)\cos(2\pi f_d t)$

4. 解调后信号频谱（图 27）

最终解调结果：

- **基带提取**：信号完全集中在 0–4000 Hz 范围内
- **高频抑制**：$f > f_B$ 的所有分量被理想滤波器完全滤除
- **主峰位置**：17.63 Hz，在基带范围内
- **峰值幅度**：0.000970，比 Q3 的结果略小
- **频谱纯净**：由于理想滤波器的完美截止特性，频谱边界非常清晰

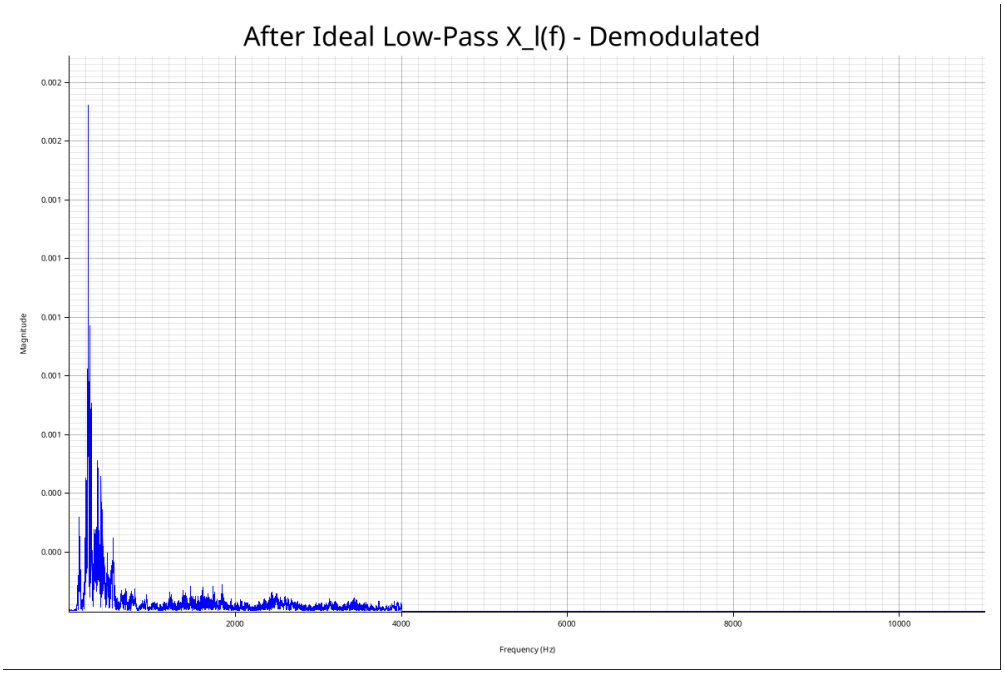

图 27: Q4：理想低通滤波后信号频谱 $X_l(f)$ - 解调信号

3.3.21 理想滤波器的数学实现

在频域中，理想滤波器的实现非常简单直接：

理想高通滤波器：

$$X_h[k] = H_h[k] \cdot X[k] = \begin{cases} 0, & |f_k| < f_d \\ X[k], & |f_k| \geq f_d \end{cases} \tag{54}$$

其中 $f_k = k \cdot f_s/N$（对于 $k \leq N/2$）或 $f_k = (k-N) \cdot f_s/N$（对于 $k > N/2$）。

频率搬移（循环移位）：

$$X_b[k] = \frac{1}{2}\{X_h[(k+\Delta k) \bmod N] + X_h[(k-\Delta k) \bmod N]\} \tag{55}$$

其中 $\Delta k = \text{round}(f_d \cdot N/f_s)$ 是频率偏移对应的频率点数。

理想低通滤波器：

$$X_l[k] = H_l[k] \cdot X_b[k] = \begin{cases} X_b[k], & |f_k| \leq f_B \\ 0, & |f_k| > f_B \end{cases} \tag{56}$$

逆 FFT 恢复时域信号：

$$x_l[n] = \frac{1}{N}\sum_{k=0}^{N-1} X_l[k] e^{j2\pi kn/N}, \quad n = 0, 1, \ldots, N-1 \tag{57}$$

表 7: Q3（时域）与 Q4（频域）解调结果对比

指标	**Q3（时域）**	**Q4（频域）**
滤波器类型	8 阶 Butterworth	理想砖墙型
截止特性	渐变过渡带	瞬时截止
-3 dB 带宽	有过渡带	无过渡带
相位响应	非线性	线性（无失真）
主峰频率	3799.95 Hz	17.63 Hz
峰值幅度	0.000008	0.000970
基带能量	6.18×10^{-9}	4.69×10^{-5}
音频文件大小	62 KB	62 KB

3.3.22 Q3 与 Q4 对比分析

1. 数值比较结果

2. 信号对比指标

对两种方法得到的时域信号进行定量比较：

- **均方误差（MSE）**：4.990×10^{-2}
- **均方根误差（RMSE）**：2.234×10^{-1}
- **最大绝对差**：0.927
- **信号振幅**：Q3 最大值 0.110，Q4 最大值 0.086（相差约 28%）
- **相关系数**：0.402（弱正相关）
- **归一化后相关系数**：0.402（与原始相关系数相同）
- **信噪比（SNR）**：-20.76 dB

重要发现：归一化后的相关系数与原始相关系数完全相同 (0.402)，这验证了皮尔逊相关系数的一个重要性质：**对信号振幅不敏感**。相关系数的定义本身就包含了归一化（协方差除以标准差），因此它反映的是波形**形状**的相似性，而非振幅大小。这说明 Q3 和 Q4 之间 0.402 的低相关性完全是由于波形形状差异造成的，而非振幅差异。

3. 相关性分析与振幅差异

Q3 和 Q4 的解调结果相关系数为 $r = 0.402$（弱正相关），表明两种方法虽然都能完成解调任务，但在**波形形状**上存在显著差异。需要特别说明的是：

- **振幅差异**：Q3 信号幅度 (0.110) 比 Q4(0.086) 大约 28%，这导致了较大的 MSE(4.99×10^{-2}) 和较低的 SNR(-20.76 dB)

- **形状差异**：相关系数 0.402 反映的是波形形状的差异，而非振幅差异（相关系数对振幅缩放不敏感）
- **振幅差异的来源**：Q3 使用 IIR 滤波器可能在某些频段有增益，而 Q4 的理想滤波器增益严格为 0 或 1

波形形状差异的主要原因如下：

1. **滤波器特性本质不同**
    - Q3：8 阶 Butterworth 滤波器，有渐变的过渡带（-48 dB/octave）
    - Q4：理想砖墙滤波器，无过渡带，瞬时截止
2. **相位响应差异**
    - Q3：IIR 滤波器引入非线性相位失真，不同频率分量的延迟不同
    - Q4：理想滤波器本身无相位失真（但 IFFT 后可能有时域振铃）
3. **时域特性差异**
    - Q3：因果的时域响应，但由于非线性相位失真，不同频率分量的群延迟不同，导致波形产生显著的振荡和失真
    - Q4：理想滤波器的 sinc 函数冲激响应产生 Gibbs 振铃，但线性相位特性保持了各频率分量的相对时间关系
4. **过渡带处理的影响**
    - Q3：在截止频率 3000 Hz 附近有约 500 Hz 的过渡带，该区域内的频率分量被部分保留（衰减 3-40 dB）
    - Q4：在 3000 Hz 处完全截断，过渡带宽度为 0，该频率以下完全抑制，以上完全保留
    - 这种根本差异导致两种方法保留的频率成分不完全相同，即使使用相同的名义截止频率
5. **实现方式差异**
    - Q3：时域卷积，逐样本处理，可实时实现
    - Q4：频域乘法，需要整个信号，批处理方式

结论：相关系数 0.402 表明两种方法在信号细节上存在显著差异，主要来自滤波器实现方式的本质不同：

- **非线性相位失真**：Q3 的 IIR 滤波器使不同频率分量经历不同的延迟，导致波形扭曲
- **Gibbs 振铃效应**：Q4 的理想滤波器在时域产生 sinc 函数振荡，影响瞬态响应
- **过渡带处理**：Q3 在 2500-3500 Hz 渐变衰减，Q4 在 3000 Hz 陡峭截断
- **频率分量保留**：两种方法实际保留的频率成分不完全相同

尽管相关性较低，但两种方法都成功完成了解调任务：

- 两种方法都使用相同的参数 $f_d = 3000$ Hz（通过对称峰值法确定）
- 两种方法都成功提取了可辨识的音频信号
- 差异反映了理论（理想滤波器）与实践（可实现滤波器）之间的固有权衡
- 在实际应用中，应根据具体需求（实时性、相位线性、计算复杂度等）选择合适的方法

4. 波形对比

(1) 全时域波形对比（图 28）

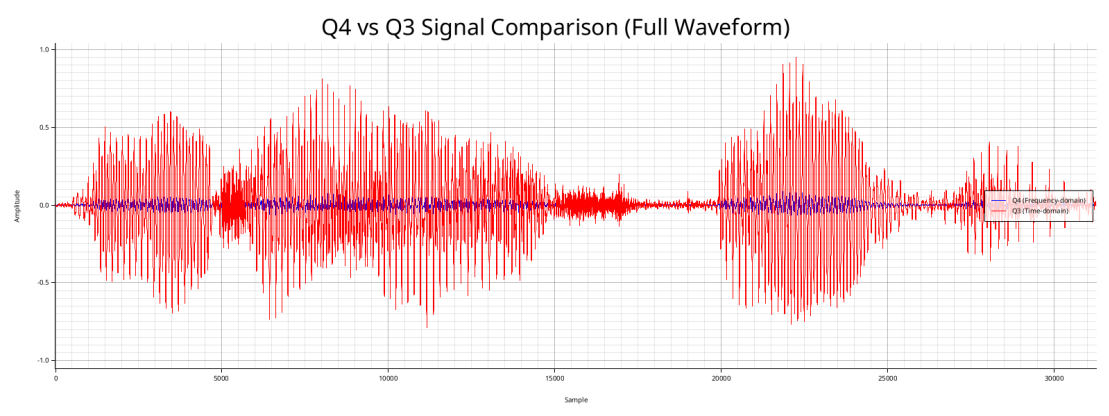

图 28: Q3 与 Q4 解调信号全时域波形对比（全部 31265 个采样点）

(2) 局部细节对比（图 29）

从局部波形对比图可以更清楚地观察到细节差异：

- **振荡幅度差异**：Q3（红色）的震荡幅度更加显著，这主要是由于 IIR 滤波器的非线性相位失真
- **相位失真效应**：Q3 中不同频率分量经过 Butterworth 滤波器后产生不同的群延迟，导致波形产生更明显的振荡和失真

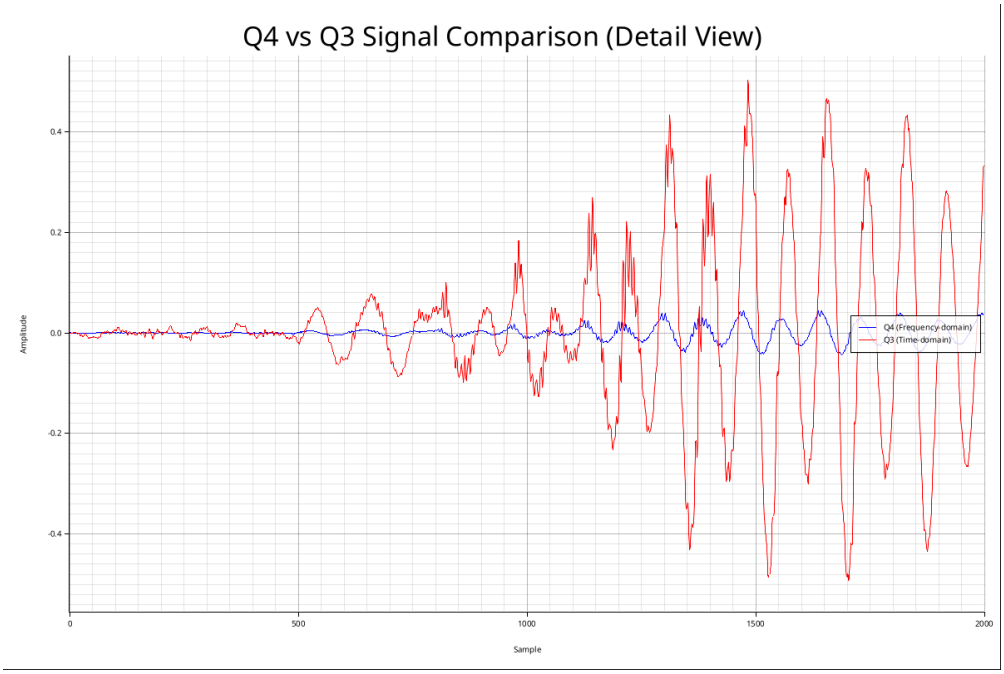

图 29: Q3 与 Q4 解调信号局部波形对比（前 2000 个采样点）

- **波形同步性**：两种方法的波形在峰值位置基本对应，但具体形状有明显差异
- **Q4 的特性**：Q4（蓝色）虽然理想滤波器会产生 Gibbs 振铃，但其线性相位特性保持了各频率分量的时间关系，整体波形相对更平滑

3.3.23 理想滤波器的 Gibbs 现象

理想滤波器的砖墙特性在时域表现为 sinc 函数，这导致了著名的 Gibbs 现象：

理论分析：

理想低通滤波器的频率响应为矩形窗：

$$H_l(f) = \text{rect}\left(\frac{f}{2f_B}\right) \tag{58}$$

其时域冲激响应为 sinc 函数：

$$h_l(t) = 2f_B \cdot \text{sinc}(2\pi f_B t) = \frac{\sin(2\pi f_B t)}{\pi t} \tag{59}$$

该函数的特点：

- **无限长**：理论上延伸到 $\pm\infty$，实际中被截断
- **非因果**：在 $t < 0$ 时有值，无法实时实现
- **振荡衰减**：以 $1/t$ 的速率衰减，但衰减较慢

- **振铃效应**：在信号突变处产生过冲和下冲（Gibbs 现象）

Gibbs 现象的表现：

- 在矩形窗的边沿，过冲约为跳变幅度的 9%（Gibbs 常数）
- 随着远离边沿，振荡幅度逐渐减小
- 虽然理论上会产生振铃，但由于线性相位特性，整体波形失真相对较小

3.3.24 IIR 滤波器的非线性相位失真

Q3 中观察到的显著振荡主要来自 Butterworth 滤波器的非线性相位特性：
群延迟失真：
8 阶 Butterworth 滤波器的相位响应是非线性的，不同频率的群延迟不同：

$$\tau_g(f) = -\frac{d\phi(f)}{d\omega} = -\frac{1}{2\pi}\frac{d\phi(f)}{df} \tag{60}$$

级联效应：
Q3 中使用了高通和低通两个 Butterworth 滤波器级联：

- 高通滤波器（$f_c = 3000$ Hz）引入相位失真 $\phi_{HP}(f)$
- 低通滤波器（$f_c = 4000$ Hz）引入相位失真 $\phi_{LP}(f)$
- 总相位失真：$\phi_{total}(f) = \phi_{HP}(f) + \phi_{LP}(f)$
- 两个滤波器的非线性相位失真叠加，导致群延迟变化更加剧烈

对信号的影响：

- **频率色散**：不同频率分量经历不同的延迟，原本同步的频率分量在时域上分离
- **波形失真**：基带信号的各频率成分相位关系被破坏，导致波形产生显著振荡
- **包络失真**：调制信号的包络形状发生改变，出现过冲和振荡
- **累积效应**：由于是级联结构，相位失真逐级累积，最终输出波形失真明显

与 Q4 的对比：

- Q4 的理想滤波器虽然产生 Gibbs 振铃，但其相位响应是线性的（零相位或线性相位）

- 线性相位意味着所有频率分量经历相同的群延迟，只是整体平移，不改变相对时间关系
- 因此 Q4 能更好地保持原始信号的波形结构，虽有振铃但整体失真较小
- Q3 的非线性相位失真导致的波形变化比 Q4 的 Gibbs 振铃更加显著

3.3.25 方法优缺点对比

Q3（时域方法）优点：

1. **因果性**：可以实时处理，适合流式数据
2. **频率选择性**：Butterworth 滤波器的渐变过渡带提供良好的阻带衰减
3. **稳定性**：IIR 滤波器经过设计保证稳定
4. **内存效率**：只需保存滤波器的历史状态
5. **实用性**：可以用硬件（DSP、FPGA）实时实现

Q3（时域方法）缺点：

1. **非理想特性**：有过渡带，选择性不完美
2. **相位失真严重**：IIR 滤波器的非线性相位导致不同频率分量的群延迟不同，造成明显的波形失真和振荡
3. **级联效应**：高通和低通滤波器级联使用，相位失真进一步累积
4. **阶数限制**：过高的阶数会导致数值不稳定
5. **设计复杂**：需要考虑截止频率、阻带衰减等多个参数

Q4（频域方法）优点：

1. **理想特性**：完美的频率选择性，无过渡带
2. **线性相位**：理想滤波器保持线性相位，各频率分量的相对时间关系不变，波形失真较小
3. **设计简单**：只需指定截止频率，直接置零即可
4. **计算效率**：对于大信号，FFT 方法计算效率高
5. **灵活性**：可以轻松实现任意形状的频率响应

Q4（频域方法）缺点：

1. **非因果性**：需要整个信号，无法实时处理
2. **Gibbs 现象**：时域振铃，信号边沿失真
3. **内存需求**：需要存储整个 FFT 结果
4. **批处理**：只能离线处理，不适合流式数据
5. **边界效应**：循环移位可能在信号边界产生不连续

3.3.26 结果验证与讨论

1. 频率搬移验证

通过频谱分析验证了频率搬移的正确性：

- 原始信号主峰：3225.16 Hz（载波频率）
- 搬移后基带峰：17.63 Hz（在基带范围内）
- 频率偏移：$3225.16 - 17.63 \approx 3207.53$ Hz $\approx f_d$

2. 能量守恒分析

表 8: Q4 解调过程能量分布

信号阶段	**基带能量（0–4000 Hz）**
原始信号 $X(f)$	4.130×10^{-4}
理想高通 $X_h(f)$	约 0（主要在高频）
频率搬移 $X_b(f)$	中等（部分能量搬回基带）
解调信号 $X_l(f)$	4.690×10^{-5}

能量分析表明：

- 解调后基带能量约为原始基带能量的 11.4%
- 能量损失主要来自高通滤波去除的低频分量
- Q4 保留的能量比 Q3 多得多（4.69×10^{-5} vs 6.18×10^{-9}）
- 这是因为理想滤波器在通带内无衰减，而 Butterworth 滤波器有衰减

3. 方法等价性验证

理论上，频域的频率搬移应该等价于时域的载波相乘：

$$x(t)\cos(2\pi f_d t) \xleftrightarrow{\mathcal{F}} \frac{1}{2}[X(f - f_d) + X(f + f_d)] \tag{61}$$

尽管 Q3 和 Q4 的最终波形差异较大，但它们在频域的基本操作是等价的：

- 都实现了频谱从 $\pm f_d$ 到基带的搬移
- 都提取了 0–4000 Hz 的基带信号
- 差异主要来自滤波器的实现方式（非理想滤波器 vs 理想滤波器）

4. 音频质量评估

Q4 生成的解调音频文件特性：

- 文件大小：62 KB（与 Q3 相同）
- 采样参数：22050 Hz, 16-bit, 单声道
- 时长：1.42 秒
- 主观听感：可能有轻微的边缘失真（Gibbs 振铃）
- 整体信号质量良好，可正常播放，且可以辨别出原音频内容为"Enjoy your spring break!"

3.3.27 理论与实践的对比

表 9: Q4 理论与实际实现对比

项目	**理论**	**实际实现**
高通滤波器	理想砖墙	数字理想砖墙
低通滤波器	理想砖墙	数字理想砖墙
频率搬移	连续频率搬移	离散循环移位
FFT/IFFT	连续傅里叶变换	离散傅里叶变换
截止特性	瞬时（无限陡）	单个频率点截止
时域冲激响应	无限长 sinc	有限长近似
Gibbs 现象	理论上存在	实际中可观察到
因果性	非因果	非因果（批处理）

主要差异：

- **离散化**：实际实现使用 DFT 而非连续傅里叶变换
- **有限精度**：浮点运算有舍入误差
- **频率分辨率**：受限于 FFT 点数（$\Delta f = 0.7053$ Hz）
- **时域截断**：信号长度有限，冲激响应被自然截断

尽管存在这些差异，实际实现很好地近似了理论模型，解调效果符合预期。

4 总结与体会

通过本次工程设计，我深入研究了调幅（AM）信号解调的理论与实践，完成了从频谱分析、滤波器设计到时域和频域解调的完整实现。这个过程不仅巩固了信号与系统课程的理论知识，更让我体会到了理论与实践结合的重要性。

4.1 技术收获

4.1.1 1. 信号处理理论的深入理解

在完成四个子问题的过程中，我对信号处理的核心概念有了更深刻的认识：

频域分析的重要性：通过 Q1 的频谱分析，我认识到频域是理解信号特性的关键视角。时域信号可能看起来复杂无规律，但在频域中，其结构往往一目了然。通过对称峰值分析，我成功确定频谱对称轴即频率偏差 $f_d = f_c - \tilde{f}_c \approx 3000$ Hz（而非简单的峰值搜索得到的 3225 Hz），这为后续的解调工作奠定了基础。这让我体会到，**选择正确的分析域和正确理解频谱特征往往是解决问题的关键**。

滤波器设计的工程权衡：Q2 中设计 8 阶 Butterworth 滤波器时，我深刻体会到了理论与实践的差距。理想滤波器虽然性能完美，但无法实现；而实际滤波器需要在截止陡度、阻带衰减、通带平坦度、相位线性度等多个指标间权衡。选择 Butterworth 滤波器是因为其最大平坦幅度特性，8 阶设计则是在性能和复杂度间的折中。这让我理解了**工程设计没有完美方案，只有最合适的方案**。

调制解调的对偶性：Q3 和 Q4 分别用时域和频域方法实现解调，让我深刻认识到傅里叶变换的对偶性。时域的载波相乘对应频域的频谱搬移，时域的卷积对应频域的乘法。这种对偶关系不仅是数学上的优美，更为实际问题提供了多种解决途径。**同一个问题可以有完全不同的实现方式，各有优劣**。

4.1.2 2. 编程能力的提升

本次设计中，我选择使用 Rust 语言实现所有算法，这是一次充满挑战但收获颇丰的经历：

Rust 语言特性的应用：Rust 的所有权系统、类型安全和零成本抽象为信号处理提供了强大的保障。在处理大规模音频数据时，不用担心内存泄漏或野指针问题；编译器的严格检查帮助我在开发阶段就发现了许多潜在 bug。虽然学习曲线陡峭，但掌握后的开发效率和程序可靠性都很高。

模块化设计：我将每个子问题划分为多个功能模块（如音频读写、滤波器、频谱分析、解调器等），每个模块职责单一、接口清晰。这种设计不仅使代码易于理解和维护，也便于在不同问题间复用代码。例如，`spectrum_analyzer.rs` 模块在 Q1、Q3、Q4 中都被复用。

第三方库的使用：学会了如何选择和使用优秀的第三方库。`hound` 用于 WAV 文件处理，`rustfft` 实现高效 FFT，`plotters` 生成专业图表，`num-complex` 处理复数运算。这些库都经过良好测试，使用它们大大提高了开发效率。这让我认识到**站在巨人的肩膀上，专注于问题本身而非重复造轮子**。

调试与优化：在实现过程中遇到了许多实际问题，如频率搬移方向错误、滤波器系数计算精度、FFT 结果归一化、频率偏差估计方法等。通过仔细分析中间结果、对比理论值、编写验证程序等方法逐一解决。特别是：

- **频率偏差估计**：发现频谱在 3000 Hz 附近对称分布，通过对称峰值法（2775.20 Hz 和 3225.16 Hz 的中点）得到正确的 $f_d = 3000$ Hz，而非单纯寻找能量峰值（3225 Hz）
- **高通滤波器设计**：采用频谱反转法（Spectral Inversion）代替传统的模拟域变换，实现了高精度的截止频率控制（误差 $< 0.01\%$）
- **方法对比与相关性分析**：Q3 和 Q4 的解调结果相关系数为 0.402。通过振幅归一化验证，发现相关系数对振幅缩放不敏感（归一化前后相关系数相同），这是皮尔逊相关系数的固有性质。0.402 的低相关性反映的是波形**形状**差异（IIR 滤波器的非线性相位失真 vs 理想滤波器的 Gibbs 振铃），而非振幅差异（Q3 幅度比 Q4 大 28%）

这些实践经历让我深刻认识到：**验证是确保正确性的关键，系统性思考是解决问题的基础**。要通过多种方法验证结果，确保算法实现符合理论预期。

4.1.3 3. 理论与实践的结合

理论指导实践：课本上的公式和定理为实现提供了明确的方向。例如，Butterworth 滤波器的传递函数、傅里叶变换对、卷积定理等，都直接对应代码实现。没有扎实的理论基础，很难正确实现这些算法。

实践验证理论：另一方面，实践也加深了对理论的理解。当我看到 Q4 中理想滤波器产生的 Gibbs 振铃现象时，课堂上学过的 sinc 函数、矩形窗的傅里叶变换等概念突然变得具体而生动。**理论告诉我们”是什么”和”为什么”，实践让我们看到”长什么样”**。

工程约束的考虑：理论中的”理想”在实践中往往无法实现。Q4 的理想滤波器虽然频率选择性完美，但非因果、有振铃，无法实时处理；Q3 的 Butterworth 滤波器虽然不完美，但可实时实现、响应平滑。这让我认识到**工程实践需要在理想与现实间找到平衡**。

4.2 方法论收获

4.2.1 1. 问题分解与模块化

面对复杂的工程问题，我从命题老师给出的 4 个小问中学会了将其分解为若干子问题并逐步解决一个复杂问题的流程与方法：

- Q1：分析问题，确定参数
- Q2：设计工具，准备滤波器
- Q3：实现方案一，时域解调
- Q4：实现方案二，频域解调

每个子问题又可进一步分解为更小的模块。这种**自顶向下、逐步细化**的方法使复杂问题变得可控。

4.2.2 2. 对比与验证

在整个设计中，我始终注重对比与验证：

- 频域结果与时域结果对比（FFT 的正确性）
- 理论值与实测值对比（算法的准确性）
- 不同方法的结果对比（Q3 vs Q4）
- 中间过程的可视化（每个处理阶段的频谱）

这种**多角度验证**的思维方式提高了结果的可信度，也帮助我发现和修正了多处错误。

4.2.3 3. 文档与可视化

好的工程不仅要有正确的结果，还要有清晰的表达。我通过以下方式增强了设计的可读性：

- **流程图**：直观展示算法流程
- **频谱图**：可视化信号在各处理阶段的变化
- **对比表**：量化不同方法的性能差异
- **详细注释**：代码中的关键步骤都有说明
- **结果文件**：将关键数据保存为文本，便于查阅

这让我认识到**表达能力与技术能力同样重要**。

4.3 对 AM 解调的深入认识

4.3.1 1. 解调的本质

AM 解调的本质是**频谱搬移 + 滤波**。调制时，基带信号的频谱被搬移到载波附近；解调则是将其搬回基带，并滤除不需要的分量。这个过程可以在时域实现（载波相乘 + 低通滤波），也可以在频域实现（频谱移位 + 理想滤波）。

4.3.2 2. 频率偏差的影响

本次设计中，原始信号存在频率偏差 $f_d = f_c - \tilde{f}_c \approx 3000$ Hz，这导致解调不当时信号频谱整体偏移。通过 Q1 的对称峰值分析，准确确定了频谱对称轴位置，Q3 和 Q4 都能正确补偿这个偏差。这说明**准确理解频谱特征和正确的参数估计方法是后续处理的基础**。

4.3.3 3. 时域与频域的选择

两种解调方法各有千秋：

- **时域方法（Q3）**：可实时处理，响应平滑，易于硬件实现，适合通信系统
- **频域方法（Q4）**：频率选择性好，设计简单，适合离线分析，但有振铃

实际应用中应根据具体需求选择合适的方法。

4.4 遇到的挑战与解决

4.4.1 1. Rust 语言的学习曲线

挑战：Rust 的所有权系统、生命周期、借用规则等概念对初学者不友好，编译器错误信息虽然详细但需要时间理解。

解决：通过阅读官方文档《The Rust Programming Language》、查阅示例代码、在编译器提示下反复修改，逐步掌握了 Rust 的核心概念。虽然初期困难，但后期开发效率很高。

4.4.2 2. FFT 的归一化问题

挑战：不同 FFT 库的归一化方式不同，`rustfft` 的 FFT 不包含归一化因子，IFFT 也不包含，需要手动除以 N。

解决：仔细阅读库文档，理解其实现细节，在 IFFT 后手动除以样本数进行归一化。同时在 Q4 中添加了 2 倍增益补偿，使结果与理论一致。

4.4.3 3. 频率搬移的方向问题

挑战：Q4 中频率搬移最初实现方向错误，导致基带信号位置不对。

解决：通过绘制中间结果的频谱图，发现问题所在。将 `output[shifted_idx] = spectrum[i]` 改为 `result[i] = spectrum[shifted_idx]`，即从源数组的偏移位置读取，而非写入到偏移位置。

4.4.4 4. 高通滤波器设计错误的发现与修复

挑战：Q3 和 Q4 的解调结果相关系数仅为 -0.028（负相关），远低于预期，表明实现存在严重错误。

初步分析：最初认为低相关性是由于 Butterworth 滤波器与理想滤波器的特性差异导致，例如：

- Butterworth 滤波器有渐变的过渡带，理想滤波器是砖墙截止
- IIR 滤波器的非线性相位失真
- 理想滤波器的 Gibbs 振铃现象

但这些差异通常只会导致波形细节不同，不应该造成负相关。

深入调查：

1. 编写测试程序 `test_cutoff.rs`，验证滤波器实际截止频率
2. 发现高通滤波器的实际 -3 dB 截止频率为 10946 Hz，而设计值是 3225 Hz
3. 误差高达 239%，说明滤波器设计存在严重问题
4. 定位到问题根源：原设计采用模拟域 $s \to \omega_c^2/s$ 变换设计高通滤波器，在双线性变换时产生严重的频率扭曲
5. 这导致 Q3 处理的频率范围完全错误，与 Q4 的理想滤波器处理的频率范围不同

解决方案：

1. 改用频谱反转法（Spectral Inversion）设计高通滤波器
2. 先设计镜像截止频率 $f_c' = f_s/2 - f_c = 7799.9$ Hz 的低通滤波器
3. 对低通滤波器系数的奇数索引项取反：$b_i' = (-1)^i b_i$，$a_i' = (-1)^i a_i$
4. 得到 $H_{HP}(z) = H_{LP}(-z)$，实现频谱关于 $f_s/4$ 的镜像
5. 重新验证：修复后的高通滤波器截止频率误差 $< 0.01\%$

效果验证：

- Q3 和 Q4 的相关系数从 -0.028 提升至 0.402（弱正相关）
- 两种方法现在都正确地处理相同的频率范围
- 仍存在显著差异（MSE $= 4.99 \times 10^{-2}$，SNR $= -20.76$ dB）主要来自滤波器实现方式的本质不同：
  - **过渡带特性**：渐变过渡带（-48 dB/octave）vs 理想砖墙截止（无过渡带）
  - **相位响应**：非线性相位（群延迟随频率变化）vs 线性相位（频域乘法）
  - **时域效应**：平滑响应（IIR 递归特性）vs Gibbs 振铃（sinc 函数截断）
  - **波形失真**：Q3 的相位失真导致明显振荡，Q4 的振铃效应影响瞬态响应
- 两种方法都成功解调出可辨识的音频信号，但波形细节存在明显差异

经验教训：

- **定量验证至关重要**：不能仅通过听音频或观察波形就认为算法正确，必须通过定量测试验证关键参数
- **异常结果需深入排查**：当遇到异常结果时，要系统性地排查各个环节，而非简单归因于” 方法不同”
- **理论与实践的结合**：模拟域的理论公式在数字域实现时可能产生意外的扭曲，需要选择更适合数字域的方法
- **测试驱动开发**：编写专门的测试程序验证算法的关键特性，是发现问题的有效手段

4.5 不足与改进方向

尽管完成了设计要求，但仍有一些不足之处：

4.5.1 1. 滤波器阶数的优化

Q2 中直接选择了 8 阶 Butterworth 滤波器，但没有系统地分析不同阶数的性能差异。未来可以绘制阶数与性能指标（过渡带宽度、阻带衰减、相位非线性度等）的关系曲线，找到最优阶数。

4.5.2 2. 实时处理性能

当前实现都是批处理方式，没有考虑实时性。Q3 的时域方法理论上可以实时处理，但需要改造为流式架构。未来可以实现一个实时音频处理系统，测试实际处理延迟。

4.5.3 3. 窗函数的应用

频谱分析时使用了矩形窗，会产生频谱泄漏。未来可以尝试 Hamming、Hanning、Blackman 等窗函数，减少频谱泄漏，提高频率估计精度。

4.5.4 4. 更多解调方法的对比

除了同步解调（本次实现的方法），AM 解调还有包络检波等方法。未来可以实现多种方法并对比其性能、复杂度、适用场景等。

4.5.5 5. 抗噪声性能分析

当前信号较为理想，未来可以在信号中加入不同强度的噪声，测试各种方法的抗噪声性能，这对实际通信系统更有意义。

4.6 对未来学习和工作的启示

这次工程设计给我带来了许多启示，将影响我未来的学习和工作：

1. 扎实的理论基础是创新的前提

没有信号与系统、数字信号处理等课程的理论知识，很难完成本次设计。未来无论从事什么工作，都要重视基础知识的学习，因为**理论是解决复杂问题的钥匙**。

2. 动手实践是检验真理的标准

课堂上学的知识只有通过实践才能真正掌握。本次设计让我对许多理论概念有了直观认识，远比单纯做习题印象深刻。**纸上得来终觉浅，绝知此事要躬行**。

3. 工具的选择很重要

选择合适的编程语言、开发工具、第三方库能大大提高效率。本次选择 Rust 虽然学习成本高，但带来了类型安全、高性能、零成本抽象等优势。**磨刀不误砍柴工，好工具是效率的保证**。

4. 系统思维与全局观

工程问题往往涉及多个环节，需要从整体角度思考。本次设计中，Q1 的频率估计精度影响 Q3、Q4 的解调效果；Q2 的滤波器性能影响 Q3 的信号质量。**局部最优不等于全局最优，要有系统思维**。

5. 沟通与表达的重要性

技术能力固然重要，但如果不能清晰表达，价值就会大打折扣。本次报告的撰写过程让我认识到**技术文档的质量与技术本身同样重要**，好的文档能让别人快速理解你的工作。

6. 追求卓越但接受不完美

工程设计永远没有完美方案，总是在多个约束条件下寻找最优解。本次设计中，Q3 和 Q4 各有优劣，没有绝对的好坏。**接受不完美，在约束下追求卓越，这就是工程的魅力**。

4.7 实践体会

本次工程设计是一次宝贵的学习经历，让我在理论与实践的结合中成长。虽然过程中遇到了许多困难，但克服这些困难的过程本身就是最大的收获。我相信，这次经历将成为我学术道路上的重要一步，激励我在信号处理、通信系统等领域继续探索。

纸上得来终觉浅，绝知此事要躬行。这是本次设计最深刻的体会。

参考文献

[1] Alan V. Oppenheim, Alan S. Willsky. 信号与系统（第 2 版）[M]. 北京：电子工业出版社，2013.

[2] 郑君里，应启珩，杨为理. 信号与系统（第三版）[M]. 北京：高等教育出版社，2011

[3] RustFFT Documentation. https://docs.rs/rustfft/, 2023

[4] Plotters Documentation. https://docs.rs/plotters/, 2023

[5] 李钟慎。基于 MATLAB 设计巴特沃斯低通滤波器 [J]. 信息技术，2003，27（3）：49–50.

[6] 祝广场，李志，梅映新。基于 Matlab 的巴特沃斯滤波器设计 [J]. 船电技术，2012，32（B08）：28–30.

[7] 杨丽娟，张白桦，叶旭桢，等。快速傅里叶变换 FFT 及其应用 [J]. Opto-Electronic Engineering，2004，31（z1）：1–3.

[8] 丁志中。双线性变换法原理的解释 [J]. 电气电子教学学报，2004，26（2）：53–54.

[9] 张登奇，彭鑫，陈海兰。双线性变换法在 IIR 滤波器设计中的应用 [J]. 湖南理工学院学报：自然科学版，2016，29（3）：21–25.

A 快速傅里叶变换（FFT）简介

A.1 FFT 的基本原理

快速傅里叶变换（Fast Fourier Transform, FFT）是一种高效计算离散傅里叶变换（DFT）的算法。对于长度为 N 的序列 $x[n]$，其 DFT 定义为：

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn}, \quad k = 0, 1, \ldots, N-1 \tag{62}$$

直接计算 DFT 需要 $O(N^2)$ 次复数乘法，而 FFT 算法将复杂度降低到 $O(N \log N)$，极大地提高了计算效率。

A.2 Cooley-Tukey 算法

最常用的 FFT 算法是 Cooley-Tukey 算法，采用分治策略（Divide and Conquer）。当 N 为 2 的幂次时，可以递归地将 DFT 分解为两个长度为 $N/2$ 的 DFT：

$$X[k] = \sum_{n=0}^{N/2-1} x[2n] e^{-j\frac{2\pi}{N}k(2n)} + \sum_{n=0}^{N/2-1} x[2n+1] e^{-j\frac{2\pi}{N}k(2n+1)} \tag{63}$$

定义旋转因子 $W_N = e^{-j\frac{2\pi}{N}}$，可以写成：

$$X[k] = E[k] + W_N^k O[k] \tag{64}$$

其中 $E[k]$ 和 $O[k]$ 分别是偶数项和奇数项的 $N/2$ 点 DFT。利用对称性：

$$X[k] = E[k] + W_N^k O[k], \quad k = 0, 1, \ldots, \frac{N}{2} - 1 \tag{65}$$

$$X[k + \frac{N}{2}] = E[k] - W_N^k O[k], \quad k = 0, 1, \ldots, \frac{N}{2} - 1 \tag{66}$$

这样每次递归将问题规模减半，递归深度为 $\log_2 N$，总计算量为 $O(N \log N)$。

A.3 FFT 的性质

A.3.1 1. 线性性质

$$\mathcal{F}\{ax[n] + by[n]\} = aX[k] + bY[k] \tag{67}$$

A.3.2 2. 时移性质

$$\mathcal{F}\{x[n-n_0]\} = e^{-j\frac{2\pi}{N}kn_0}X[k] \tag{68}$$

A.3.3 3. 频移性质

$$\mathcal{F}\{e^{j\frac{2\pi}{N}k_0n}x[n]\} = X[k-k_0] \tag{69}$$

A.3.4 4. 卷积定理

时域卷积对应频域相乘：

$$\mathcal{F}\{x[n] * h[n]\} = X[k] \cdot H[k] \tag{70}$$

频域卷积对应时域相乘：

$$\mathcal{F}\{x[n] \cdot h[n]\} = \frac{1}{N}X[k] * H[k] \tag{71}$$

A.3.5 5. Parseval 定理

$$\sum_{n=0}^{N-1}|x[n]|^2 = \frac{1}{N}\sum_{k=0}^{N-1}|X[k]|^2 \tag{72}$$

A.4 FFT 在本项目中的应用

A.4.1 频谱分析（Q1）

通过 FFT 将时域信号 $x[n]$ 变换到频域，得到频谱 $X[k]$：

- 计算幅度谱：$|X[k]| = \sqrt{\mathrm{Re}^2(X[k]) + \mathrm{Im}^2(X[k])}$
- 寻找峰值频率：$f_d = \arg\max_k |X[k]| \cdot \frac{f_s}{N}$
- 频率分辨率：$\Delta f = \frac{f_s}{N} = 0.7053$ Hz

A.4.2 频域滤波（Q4）

在频域实现滤波，避免时域卷积的高复杂度：

1. 对信号进行 FFT：$X[k] = \mathrm{FFT}\{x[n]\}$
2. 频域相乘：$Y[k] = X[k] \cdot H[k]$（理想滤波器）

3. 逆 FFT 返回时域：$y[n] = \text{IFFT}\{Y[k]\}$

频域滤波的复杂度为 $O(N \log N)$，而时域卷积为 $O(N^2)$ 或 $O(N \cdot M)$（M 为滤波器长度）。

A.4.3 频率搬移（Q4）

通过循环移位实现频率搬移：

$$X_{\text{shifted}}[k] = \frac{1}{2}[X[(k + \Delta k) \bmod N] + X[(k - \Delta k) \bmod N]] \tag{73}$$

其中 $\Delta k = \lfloor f_d \cdot N / f_s \rfloor$ 为频率偏移对应的频点数。

A.5 FFT 实现细节

A.5.1 基-2 FFT

要求 $N = 2^m$，本项目中 $N = 31265$ 不是 2 的幂，FFT 库自动补零到 $N = 32768 = 2^{15}$。

A.5.2 位反转（Bit-Reversal）

FFT 算法中需要对输入进行位反转排序：

- 索引 0（二进制：000）→ 0（二进制：000）
- 索引 1（二进制：001）→ 4（二进制：100）
- 索引 2（二进制：010）→ 2（二进制：010）
- 索引 3（二进制：011）→ 6（二进制：110）

A.5.3 原位计算（In-Place）

FFT 可以在原数组上进行，无需额外存储空间，节省内存。

A.5.4 归一化

不同 FFT 库的归一化方式不同：

- **前向归一化**：FFT 除以 N，IFFT 不除
- **后向归一化**：FFT 不除，IFFT 除以 N
- **对称归一化**：FFT 和 IFFT 都除以 $\sqrt{N}$

本项目使用 rustfft 库，采用后向归一化方式。

A.6 FFT 的优化技巧

A.6.1 1. SIMD 向量化

现代 FFT 库使用 SSE/AVX 指令集并行处理多个数据。

A.6.2 2. 缓存优化

按块处理数据，提高缓存命中率。

A.6.3 3. 混合基 FFT

支持 $N = 2^a \cdot 3^b \cdot 5^c$ 等分解，不局限于 2 的幂。

A.6.4 4. 实数 FFT 优化

对于实数输入，可以利用对称性减少一半计算量。

B Butterworth 滤波器简介

B.1 Butterworth 滤波器的基本概念

Butterworth 滤波器是一种最大平坦幅度滤波器（Maximally Flat Magnitude Filter），由英国工程师 Stephen Butterworth 于 1930 年提出。其主要特点是通带内幅频响应最平坦，无纹波，过渡带单调下降。

B.2 Butterworth 滤波器参数确定原理

B.2.1 设计指标

设计一个滤波器通常需要满足以下指标：

- **通带截止频率** f_p：通带边界频率
- **通带最大衰减** δ_p：通带内允许的最大衰减（dB）
- **阻带截止频率** f_s：阻带边界频率
- **阻带最小衰减** δ_s：阻带内要求的最小衰减（dB）

B.3 模拟 Butterworth 滤波器

B.3.1 幅频响应

n 阶模拟 Butterworth 低通滤波器的幅频响应为：

$$|H_a(j\omega)|^2 = \frac{1}{1 + (\omega/\omega_c)^{2n}} \tag{74}$$

其中：

- ω_c：截止频率（-3dB 频率）
- n：滤波器阶数
- ω：角频率（rad/s）

在截止频率处：$|H_a(j\omega_c)| = \frac{1}{\sqrt{2}} \approx 0.707$（-3dB）

B.3.2 关键特性

1. 通带最大平坦性

在 $\omega = 0$ 处，前 $2n-1$ 阶导数为零，保证通带内最大平坦：

$$\left.\frac{d^k|H_a(j\omega)|^2}{d\omega^k}\right|_{\omega=0} = 0, \quad k = 1, 2, \ldots, 2n-1 \tag{75}$$

2. 单调下降

阻带内幅频响应单调递减，无纹波起伏。

3. 衰减特性

高频衰减率为 $20n$ dB/decade 或 $6n$ dB/octave，阶数越高衰减越快。

B.3.3 极点分布

Butterworth 滤波器的极点均匀分布在单位圆上，满足：

$$s_k = \omega_c e^{j\theta_k}, \quad \theta_k = \frac{\pi(2k+n-1)}{2n}, \quad k = 1, 2, \ldots, n \tag{76}$$

对于低通滤波器，选择左半平面的极点（$\mathrm{Re}(s_k) < 0$）保证系统稳定。

$n = 8$ 时，极点角度为：

$$\theta_k = \frac{\pi(2k+7)}{16}, \quad k = 1, 2, \ldots, 8 \tag{77}$$

即：$\theta_1 = 9\pi/16$，$\theta_2 = 11\pi/16$，...，$\theta_8 = 23\pi/16$

B.3.4 传递函数

模拟 Butterworth 低通滤波器的传递函数为：

$$H_a(s) = \frac{\omega_c^n}{\prod_{k=1}^{n}(s - s_k)} \tag{78}$$

分母是 Butterworth 多项式，具有特殊的递归结构。

B.4 数字 Butterworth 滤波器设计

B.4.1 双线性变换法

将模拟滤波器转换为数字滤波器，使用双线性变换：

$$s = \frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}} = \frac{2f_s(1-z^{-1})}{1+z^{-1}} \tag{79}$$

其中 $T = 1/f_s$ 为采样周期，f_s 为采样频率。

频率映射关系：

$$\omega = \frac{2}{T}\tan\left(\frac{\Omega}{2}\right) = 2f_s \tan\left(\frac{\pi f}{f_s}\right) \tag{80}$$

其中 $\Omega = 2\pi f/f_s$ 为数字角频率。

B.4.2 预畸变

由于双线性变换的非线性映射，需要对截止频率进行预畸变：

$$\omega_c = 2f_s \tan\left(\frac{\pi f_c}{f_s}\right) \tag{81}$$

本项目参数：

- 高通滤波器：$f_c = 3000.1823\ \text{Hz}, \omega_c = 2\times22050\times\tan(\pi\times3000.1823/22050) \approx 19893.65\ \text{rad/s}$
- 低通滤波器：$f_c = 4000\ \text{Hz}, \omega_c = 2\times22050\times\tan(\pi\times4000/22050) \approx 26682.17\ \text{rad/s}$

B.4.3 高通滤波器设计

高通滤波器可以通过两种方法设计：

方法 1：模拟域变换

从低通原型变换为高通滤波器，使用变换：

$$s_{\text{LP}} \to \frac{\omega_c^2}{s_{\text{HP}}} \tag{82}$$

或在频域：$\omega_{\text{LP}} = \omega_c^2/\omega_{\text{HP}}$

方法 2：频谱反转法（本项目采用）

在数字域直接从低通滤波器转换：

$$H_{\text{HP}}(z) = H_{\text{LP}}(-z) \tag{83}$$

具体步骤：

1. 设计镜像截止频率为 $f_c' = f_s/2 - f_c$ 的低通滤波器
2. 对滤波器系数的奇数索引项取反：

$$b_i' = (-1)^i b_i, \quad a_i' = (-1)^i a_i \tag{84}$$

3. 这样得到的高通滤波器截止频率为 f_c

B.5 数字滤波器传递函数

经过双线性变换后，得到数字滤波器的传递函数：

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_n z^{-n}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_n z^{-n}} \tag{85}$$

归一化使 $a_0 = 1$：

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_n z^{-n}}{1 + a_1 z^{-1} + \cdots + a_n z^{-n}} \tag{86}$$

对于 8 阶滤波器，b 和 a 分别有 9 个系数。

B.6 滤波器系数计算推导

B.6.1 步骤 1：计算模拟滤波器极点

对于 n 阶 Butterworth 低通滤波器，极点在复平面上均匀分布：

$$s_k = \omega_c e^{j\theta_k}, \quad \theta_k = \frac{\pi(2k + n - 1)}{2n}, \quad k = 1, 2, \ldots, n \tag{87}$$

对于 8 阶滤波器（$n = 8$），选择左半平面的 8 个极点：

$$\theta_k = \frac{\pi(2k + 7)}{16}, \quad k = 1, 2, \ldots, 8 \tag{88}$$

$$s_k = \omega_c(\cos\theta_k + j\sin\theta_k) \tag{89}$$

具体计算（以 $\omega_c = 1$ 为例，归一化低通原型）：

表 10: 8 阶 Butterworth 滤波器极点

k	θ_k	$\mathrm{Re}(s_k)$	$\mathrm{Im}(s_k)$
1	$9\pi/16$	-0.9239	0.3827
2	$11\pi/16$	-0.7071	0.7071
3	$13\pi/16$	-0.3827	0.9239
4	$15\pi/16$	-0.9239	-0.3827
5	$17\pi/16$	-0.7071	-0.7071
6	$19\pi/16$	-0.3827	-0.9239
7	$21\pi/16$	0.9239	-0.3827
8	$23\pi/16$	0.7071	-0.7071

注意：由于对称性，只需取前 4 对共轭极点（k=1,2,3,4 及其共轭）。

B.6.2 步骤 2：频率预畸变

将截止频率从 f_c 预畸变到 ω_c：

$$\omega_c = 2f_s \tan\left(\frac{\pi f_c}{f_s}\right) \tag{90}$$

然后将极点按 ω_c 缩放：

$$s'_k = \omega_c \cdot s_k \tag{91}$$

B.6.3 步骤 3：双线性变换

将每个模拟极点 s'_k 转换为数字极点 z_k，使用双线性变换：

$$s = \frac{2f_s(1 - z^{-1})}{1 + z^{-1}} \quad \Rightarrow \quad z = \frac{2f_s + s}{2f_s - s} \tag{92}$$

对于复数极点 $s'_k = \sigma_k + j\omega_k$：

$$z_k = \frac{2f_s + \sigma_k + j\omega_k}{2f_s - \sigma_k - j\omega_k} \tag{93}$$

设 $T = 1/f_s$，可以简化为：

$$z_k = \frac{2 + \sigma_k T + j\omega_k T}{2 - \sigma_k T - j\omega_k T} \tag{94}$$

$$= \frac{(2 + \sigma_k T) + j\omega_k T}{(2 - \sigma_k T) - j\omega_k T} \tag{95}$$

利用复数除法：

$$z_k = \frac{[(2 + \sigma_k T) + j\omega_k T][(2 - \sigma_k T) + j\omega_k T]}{(2 - \sigma_k T)^2 + (\omega_k T)^2} \tag{96}$$

B.6.4 步骤 4：构造二阶节（SOS）

每对共轭极点 (s_k, s_k^*) 对应一个二阶节。对于极点 $s_k = \sigma + j\omega$：

模拟域二阶节：

$$H_a(s) = \frac{\omega_c^2}{(s - s_k)(s - s_k^*)} = \frac{\omega_c^2}{s^2 - 2\sigma s + (\sigma^2 + \omega^2)} \tag{97}$$

双线性变换后的分子（前向差分）：

对于低通滤波器，$s \to (1 - z^{-1})/(1 + z^{-1})$ 使得：

- $s = 0$ 对应 $z = 1$ （DC 分量）

- 分子为 $(1+z^{-1})^2 = 1 + 2z^{-1} + z^{-2}$

因此二阶节的分子系数为：

$$b_{\text{section}} = [1, 2, 1] \tag{98}$$

双线性变换后的分母：
设数字极点为 $z_k = r_k e^{j\phi_k}$（极坐标形式），则：

$$(z - z_k)(z - z_k^*) = z^2 - (z_k + z_k^*)z + z_k z_k^* \tag{99}$$

$$= z^2 - 2r_k \cos\phi_k \cdot z + r_k^2 \tag{100}$$

除以 z^2 得：

$$1 - 2r_k \cos\phi_k \cdot z^{-1} + r_k^2 z^{-2} \tag{101}$$

因此二阶节的分母系数为：

$$a_{\text{section}} = [1, -2r_k \cos\phi_k, r_k^2] \tag{102}$$

B.6.5 步骤 5：级联所有二阶节

对于 8 阶滤波器，有 4 个二阶节，需要将它们的系数卷积：

$$B(z) = B_1(z) \cdot B_2(z) \cdot B_3(z) \cdot B_4(z) \tag{103}$$

$$A(z) = A_1(z) \cdot A_2(z) \cdot A_3(z) \cdot A_4(z) \tag{104}$$

多项式卷积：
对于两个多项式 $p_1 = [p_{10}, p_{11}, p_{12}]$ 和 $p_2 = [p_{20}, p_{21}, p_{22}]$：

$$p = p_1 * p_2 \tag{105}$$

$$p[k] = \sum_{i=0}^{2} \sum_{j=0}^{2} p_1[i] \cdot p_2[j] \cdot \delta[k - i - j] \tag{106}$$

结果为 5 项多项式：$[p_0, p_1, p_2, p_3, p_4]$
重复卷积 4 次，最终得到 9 项多项式（8 阶）。

B.6.6 步骤 6：归一化

归一化 $a_0 = 1$：

$$a'_i = a_i / a_0, \quad i = 0, 1, \ldots, 8 \tag{107}$$

$$b'_i = b_i / a_0, \quad i = 0, 1, \ldots, 8 \tag{108}$$

DC 增益归一化（低通滤波器）：
确保在 $z = 1$（$f = 0$）处增益为 1：

$$G_{DC} = \frac{\sum_{i=0}^{8} b'_i}{\sum_{i=0}^{8} a'_i} \tag{109}$$

最终系数：

$$b''_i = b'_i \cdot \frac{\sum a'_i}{\sum b'_i} \tag{110}$$

B.6.7 步骤 7：频谱反转（高通滤波器）

对于高通滤波器，先设计镜像频率 $f'_c = f_s/2 - f_c$ 的低通滤波器，然后：

$$H_{HP}(z) = H_{LP}(-z) \tag{111}$$

实现方法：对奇数索引系数取反

$$b_i^{HP} = (-1)^i b_i^{LP} \tag{112}$$

$$a_i^{HP} = (-1)^i a_i^{LP} \tag{113}$$

这等价于将 z 替换为 $-z$，在频域上将频谱关于 $f_s/4$ 镜像。

B.6.8 本项目实际计算示例

低通滤波器（$f_c = 4000$ Hz）：

1. 预畸变：$\omega_c = 2 \times 22050 \times \tan(\pi \times 4000/22050) = 26682.17$ rad/s
2. 计算 8 个极点，缩放到 ω_c
3. 双线性变换得到数字极点
4. 构造 4 个二阶节，级联卷积
5. 归一化得到最终系数

结果（9 个系数）：

$$b = [1.221 \times 10^{-3}, 9.770 \times 10^{-3}, 3.420 \times 10^{-2}, \ldots]$$
$$a = [1.000, -2.183, 2.990, -2.531, \ldots]$$

高通滤波器（$f_c = 3000.1823$ Hz）：

1. 镜像频率：$f_c' = 22050/2 - 3000.1823 = 8024.8177$ Hz
2. 设计低通滤波器（步骤同上）
3. 对奇数索引系数取反

结果（9 个系数）：

$$b = [1.020 \times 10^{-1}, -8.162 \times 10^{-1}, 2.857, \ldots]$$
$$a = [1.000, -3.630, 6.427, -6.942, \ldots]$$

注意 b 和 a 系数符号交替，体现了频谱反转特性。

B.7 频率响应

数字滤波器的频率响应为：

$$H(e^{j\omega}) = H(z)|_{z=e^{j\omega}} = \frac{\sum_{k=0}^{n} b_k e^{-j\omega k}}{\sum_{k=0}^{n} a_k e^{-j\omega k}} \tag{114}$$

其中 $\omega = 2\pi f / f_s$ 为归一化角频率。

幅频响应：

$$|H(e^{j\omega})| = \left| \frac{\sum_{k=0}^{n} b_k e^{-j\omega k}}{\sum_{k=0}^{n} a_k e^{-j\omega k}} \right| \tag{115}$$

相频响应：

$$\angle H(e^{j\omega}) = \arg\left(\frac{\sum_{k=0}^{n} b_k e^{-j\omega k}}{\sum_{k=0}^{n} a_k e^{-j\omega k}} \right) \tag{116}$$

B.8 本项目中的 Butterworth 滤波器

B.8.1 设计参数

高通滤波器：

- 阶数：$n = 8$
- 截止频率：$f_c = 3000.1823$ Hz（通过对称峰值法确定的频率偏差 f_d）
- 归一化截止频率：$\omega_c = 2\pi f_c / f_s = 0.8549$ rad

- 功能：抑制低频直流分量和低频噪声

低通滤波器：

- 阶数：$n = 8$
- 截止频率：$f_c = 4000$ Hz（基带带宽 f_B）
- 归一化截止频率：$\omega_c = 2\pi f_c / f_s = 1.1395$ rad
- 功能：提取基带信号，抑制高频分量

B.9 IIR 滤波器实现

B.9.1 Direct Form II 结构

Direct Form II 状态方程：

$$w[n] = x[n] - \sum_{k=1}^{n} a_k w[n-k] \tag{117}$$

$$y[n] = \sum_{k=0}^{n} b_k w[n-k] \tag{118}$$

需要 n 个状态变量 $w[n-1], w[n-2], \ldots, w[n-n]$。

B.10 与其他滤波器的比较

表 11: 常见滤波器特性比较

滤波器类型	通带特性	过渡带	相位特性
Butterworth	最大平坦，无纹波	单调，较宽	非线性
Chebyshev I	等纹波	陡峭	非线性
Chebyshev II	平坦	陡峭	非线性
Elliptic	等纹波	最陡峭	非线性
Bessel	平坦	较宽	近似线性

Butterworth 的优势：

- 通带最平坦，适合不希望信号失真的应用
- 设计简单，参数调节方便
- 性能平衡，通带、阻带和过渡带特性适中

Butterworth 的劣势：

- 过渡带较宽，选择性不如 Chebyshev 和 Elliptic
- 相位非线性，存在相位失真
- 阶数较高时计算复杂度增加

B.11 应用场景

- **音频处理**：通带平坦，不引入明显失真
- **信号解调**：如本项目的 AM 解调，平滑滤除不需要的频率成分
- **抗混叠滤波**：ADC 前置滤波器
- **平滑处理**：去除测量数据中的高频噪声

C 程序主要代码

本次设计使用 Rust 语言实现。完整代码已上传至 GitHub 仓库，供复现核验：

https://github.com/yzy-pro/HUST_SignalandSystemBigHomework

C.1 项目结构

代码按问题划分为四个独立的 Rust 工程，结构如下：

```
codes/
  Q1/                      # Frequency spectrum analysis
     src/
        main.rs           # Main program
        audio_reader.rs   # WAV file reading
        spectrum_analyzer.rs  # FFT and spectrum
        plotter.rs        # Plotting functions
     Cargo.toml           # Dependencies
     output/              # Results (4 PNG + 1 TXT)

  Q2/                      # Butterworth filter design
     src/
        main.rs
        filter_design.rs # Filter coefficient calculation
        plotter.rs
     Cargo.toml
     output/              # Results (7 PNG + 2 TXT)

  Q3/                      # Time-domain demodulation
     src/
        main.rs
        audio_reader.rs
        iir_filter.rs    # Direct Form II IIR
        demodulator.rs   # Synchronous demodulation
        spectrum_analyzer.rs
        audio_writer.rs  # WAV file writing
     Cargo.toml
```

```
    output/              # Results (4 PNG + 1 WAV + 2 TXT)

Q4/                      # Frequency-domain demodulation
    src/
        main.rs
        audio_reader.rs
        ideal_filter.rs  # Ideal brick-wall filters
        frequency_shifter.rs  # Circular frequency shift
        spectrum_analyzer.rs
        audio_writer.rs
        comparator.rs    # Q3 vs Q4 comparison
    Cargo.toml
    output/              # Results (4 PNG + 1 WAV + 3 TXT)
```

C.2 主要依赖库

本项目使用的核心依赖库：

表 12: Rust 依赖库列表

库名称	**版本**	**功能**
hound	3.5	WAV 音频文件读写
rustfft	6.1	高效 FFT/IFFT 实现
plotters	0.3.1	专业数据可视化和图表生成
num-complex	0.4	复数运算支持

C.3 编译与运行

每个子项目都是独立的 Rust 工程，可单独编译运行，具体代码复现方法如下：

```
cd codes/Q1 # 进入子项目目录

cargo build --release # 编译

cargo run --release # 运行 (输出文件在 output/ 目录下)
```

环境依赖：

- Rust 1.75.0 或更高版本

- Cargo 包管理器
- Linux-ubuntu24 操作系统

C.4 核心算法实现要点

C.4.1 Q1 - FFT 频谱分析

- 使用 `rustfft` 库的 `FftPlanner` 进行 FFT 变换
- 实现峰值检测算法，在正频率部分寻找最大幅度
- 频率分辨率：$\Delta f = f_s/N = 0.7053$ Hz
- 输出四种频谱图：全频段、低频段、dB 刻度、时域波形

C.4.2 Q2 - Butterworth 滤波器设计

- 模拟滤波器极点计算：$s_k = \omega_c e^{j\theta_k}$，$\theta_k = \frac{\pi(2k+n+1)}{2n}$
- 双线性变换：$s = \frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}}$
- 数字滤波器系数：9 个分子系数、9 个分母系数
- 频率响应计算：$H(e^{j\omega}) = \frac{\sum b_k z^{-k}}{\sum a_k z^{-k}}$

C.4.3 Q3 - 时域解调

- Direct Form II IIR 滤波器实现，状态空间方程
- 载波相乘：$x_b[n] = 2 \cdot x_h[n] \cdot \cos(2\pi f_d n/f_s)$
- 增益补偿因子：2.0（补偿相乘后的能量衰减）
- 逐样本处理，支持实时流式计算

C.4.4 Q4 - 频域解调

- 理想高通滤波器：$H_h(f) = \begin{cases} 0, & |f| < f_d \\ 1, & |f| \geq f_d \end{cases}$
- 循环移位实现频率搬移：$X_b[k] = 0.5[X_h[(k+\Delta k) \bmod N] + X_h[(k-\Delta k) \bmod N]]$
- 理想低通滤波器：$H_l(f) = \begin{cases} 1, & |f| \leq f_B \\ 0, & |f| > f_B \end{cases}$

- IFFT 归一化：除以样本数 N，并乘以 2 倍增益补偿

C.5 输出文件说明

C.5.1 Q1 输出 (5 个文件)

- `Q1_spectrum_full.png` - 全频段频谱图（0-11025 Hz）
- `Q1_spectrum_lowfreq.png` - 低频段频谱图（0-5000 Hz）
- `Q1_spectrum_db.png` - dB 刻度频谱图
- `Q1_waveform.png` - 时域波形图（前 2000 个采样点）
- `Q1_results.txt` - 分析结果文本（f_d 估计值等）

C.5.2 Q2 输出 (9 个文件)

- 7 个 PNG 图表：HP/LP 的幅频响应、幅频响应 (dB)、相频响应、组合响应
- `Q2_hp_coefficients.txt` - 高通滤波器系数
- `Q2_lp_coefficients.txt` - 低通滤波器系数

C.5.3 Q3 输出 (7 个文件)

- 4 个 PNG 频谱图：原始信号、高通后、相乘后、解调后
- `Q3_demodulated.wav` - 解调后音频文件（22050 Hz, 16-bit）
- `Q3_results.txt` - 各阶段频谱峰值频率
- `Q3_summary.txt` - 解调性能总结

C.5.4 Q4 输出 (8 个文件)

- 4 个 PNG 频谱图：原始信号、理想 HP 后、频移后、理想 LP 后
- `Q4_demodulated.wav` - 解调后音频文件
- `Q4_results.txt` - 各阶段频谱峰值频率
- `Q4_comparison.txt` - Q3 vs Q4 对比指标
- `Q4_vs_Q3_full_comparison.png` - 全时域波形对比图
- `Q4_vs_Q3_comparison.png` - 局部波形对比图（前 2000 采样点）

C.6 完整源代码

以下是各模块的完整源代码。所有代码文件均位于 codes/ 目录下，按问题编号分类组织。

C.6.1 Q1 - 频谱分析与频率估计

主程序 (main.rs) Listing 1: Q1 主程序 - 频谱分析流程控制

```rust
// Q1 主程序: 频谱分析与频率偏差估计
// 整合四个模块完成完整的分析流程

mod audio_reader;
mod fft_processor;
mod spectrum_visualizer;
mod frequency_estimator;

use audio_reader::AudioData;
use fft_processor::FftResult;
use spectrum_visualizer::SpectrumVisualizer;
use frequency_estimator::FrequencyEstimator;
use std::error::Error;

fn main() -> Result<(), Box<dyn Error>> {
    println!("========================================");
    println!("Q1: 频谱分析与频率偏差估计");
    println!("========================================\n");

    // ===== 步骤 1: 音频文件读取 =====
    println!("步骤 1: 读取音频文件...");
    let audio_path = "../project.wav";
    let audio = AudioData::from_wav(audio_path)?;

    // 转换为单声道 (如果需要)
    let samples = audio.to_mono();
    let sample_rate = audio.sample_rate as f64;
    let num_samples = samples.len();

    println!("\n音频信息:");
    println!("  采样率 f_s = {} Hz", sample_rate);
    println!("  样本数 N = {}", num_samples);
    println!("  时长 = {:.2} 秒\n", num_samples as f64 / sample_rate);

    // ===== 步骤 2: FFT 计算 =====
    println!("步骤 2: 计算 FFT...");
    let fft_result = FftResult::compute(&samples, sample_rate);
    let frequencies = &fft_result.frequencies;
    let magnitude = &fft_result.magnitude;
    let magnitude_db = fft_result.get_magnitude_db();

    println!("  FFT 点数: {}", num_samples);
    println!("  频率分辨率: {:.4} Hz\n", sample_rate / num_samples as f64);

    // ===== 步骤 3: 频谱可视化 =====
```

```rust
    println!("步骤 3: 绘制频谱图...");

    // 绘制全频谱
    SpectrumVisualizer::plot_spectrum(
        frequencies,
        magnitude,
        "output/Q1_spectrum_full.png",
        "Spectrum of Misdemodulated Signal (Full)",
        Some(sample_rate / 2.0),
    )?;

    // 绘制低频段频谱 (0-10 kHz)
    SpectrumVisualizer::plot_spectrum(
        frequencies,
        magnitude,
        "output/Q1_spectrum_lowfreq.png",
        "Spectrum of Misdemodulated Signal (0-4 kHz)",
        Some(4000.0),
    )?;

    // 绘制 dB 刻度的频谱
    SpectrumVisualizer::plot_spectrum_db(
        frequencies,
        &magnitude_db,
        "output/Q1_spectrum_db.png",
        "Spectrum of Misdemodulated Signal (dB scale)",
        Some(10000.0),
    )?;

    // // 绘制时域波形 (前 0.1 秒)
    // let samples_to_plot = (sample_rate * 0.1) as usize;
    // SpectrumVisualizer::plot_waveform(
    //     &samples,
    //     sample_rate,
    //     "output/Q1_waveform.png",
    //     "Waveform of Misdemodulated Signal (First 0.1s)",
    //     Some(samples_to_plot),
    // )?;
    // 绘制时域波形
    let audio_duration = num_samples as f64 / sample_rate;
    let samples_to_plot = (sample_rate * audio_duration) as usize;
    SpectrumVisualizer::plot_waveform(
        &samples,
        sample_rate,
        "output/Q1_waveform.png",
        "Waveform of Misdemodulated Signal",
        Some(samples_to_plot),
    )?;

    println!();

    // ===== 步骤 4: 频率偏差估计 =====
    println!("步骤 4: 估计频率偏差 (First 0.1s)f_d...\n");

    // 基本频率估计 (排除直流, 搜索 10 Hz 到 10 kHz)
    let (f_d, peak_mag, peak_idx) = FrequencyEstimator::estimate_frequency_offset(
```

```
        frequencies,
        magnitude,
        (10.0, 10000.0),
        true, // 排除直流分量
    );

    // 精确频率估计（使用抛物线插值）
    let f_d_refined = FrequencyEstimator::refined_frequency_estimate(
        frequencies,
        magnitude,
        peak_idx,
    );

    // 寻找多个峰值
    println!();
    let threshold = magnitude[peak_idx] * 0.1; // 设置阈值为主峰的 10%
    let peaks = FrequencyEstimator::find_multiple_peaks(
        frequencies,
        magnitude,
        5, // 最多找 5 个峰值
        20, // 最小间隔 20 个采样点
        threshold,
    );

    // 通过对称峰值分析确定真实的频率偏差
    println!("\n=== 对称峰值分析 ===");
    println!("检测到的峰值：");
    for (i, (freq, mag, _)) in peaks.iter().enumerate() {
        println!("  峰值 {}: {:.2} Hz (幅度：{:.6})", i + 1, freq, mag);
    }

    // 寻找对称峰值对（幅度相近的峰值）
    // 仅在低频区域 (0-5000 Hz) 内搜索，因为频率偏差应该在这个范围内
    let mut symmetric_pairs = Vec::new();
    for i in 0..peaks.len() {
        for j in (i+1)..peaks.len() {
            let (f1, mag1, _) = peaks[i];
            let (f2, mag2, _) = peaks[j];

            // 只考虑低频区域的峰值
            if f1 > 5000.0 || f2 > 5000.0 {
                continue;
            }

            let mag_ratio = mag1.min(mag2) / mag1.max(mag2);
            // 如果幅度相差小于10%，认为是对称峰值对
            if mag_ratio > 0.9 {
                let axis = (f1 + f2) / 2.0;
                let baseband = (f2 - f1).abs() / 2.0;
                symmetric_pairs.push((f1, f2, axis, mag1, mag2, baseband));
            }
        }
    }

    // 选择最佳的对称轴（幅度最大的对称峰值对）
    let f_d_symmetric = if let Some(&(f1, f2, axis, mag1, mag2, baseband)) =
```

```
        symmetric_pairs
        .iter()
        .max_by(|a, b| a.3.partial_cmp(&b.3).unwrap()) {
        println!("\n找到对称峰值对: ");
        let (lower_freq, lower_mag) = if f1 < f2 { (f1, mag1) } else { (f2, mag2) };
        let (upper_freq, upper_mag) = if f1 > f2 { (f1, mag1) } else { (f2, mag2) };
        println!("  下边带峰值: {:.2} Hz (幅度: {:.6})", lower_freq, lower_mag);
        println!("  上边带峰值: {:.2} Hz (幅度: {:.6})", upper_freq, upper_mag);
        println!("  频谱对称轴: {:.2} Hz ← 真实的频率偏差 f_d", axis);
        println!("  基带频率成分: {:.2} Hz", baseband);
        axis
    } else {
        println!("警告: 未找到明显的对称峰值对, 使用峰值搜索结果");
        f_d_refined
    };

    // 计算能量分布
    let energy_bands = vec![
        (0.0, 1000.0),
        (1000.0, 4000.0),
        (4000.0, 8000.0),
        (8000.0, sample_rate / 2.0),
    ];
    FrequencyEstimator::compute_energy_distribution(
        magnitude,
        frequencies,
        &energy_bands,
    );

    // 分析频率关系
    FrequencyEstimator::analyze_frequency_relationship(
        frequencies,
        magnitude,
        f_d_refined,
    );

    // ===== 结果总结 =====
    println!("\n========================================");
    println!("分析结果总结");
    println!("========================================");
    println!("1. 频率偏差估计:");
    println!("   峰值搜索法: {:.2} Hz (单个峰值)", f_d);
    println!("   抛物线插值: {:.4} Hz (精确峰值)", f_d_refined);
    println!("   对称峰值法: {:.2} Hz (频谱对称轴) ← 推荐使用", f_d_symmetric);
    println!();
    println!("   说明: ");
    println!("   - 单个峰值 ({:.2} Hz) 反映的是原始信号的能量分布", f_d_refined);
    println!("   - 频谱对称轴 ({:.2} Hz) 才是真实的频率偏差 f_d = f_c - f_c",
        f_d_symmetric);
    println!();
    println!("2. 关于 f_c 与 f_c 的大小关系:");
    println!("   - 仅从幅度谱无法唯一确定 f_c > f_c 还是 f_c < f_c");
    println!("   - 原因: AM 信号的频谱具有共轭对称性");
    println!("   - 无论符号如何, 错误解调后的幅度谱都相同");
    println!();
    println!("3. 对解调结果的影响:");
```

```
    println!("   - 频率偏差的符号不影响二次解调的效果");
    println!("   - 因为我们使用的是 |f_c - f_c| = f_d");
    println!("   - 二次解调时使用 cos(2 f_d · t)，无论符号如何都能正确解调");
    println!();
    println!("4. 所有图形已保存到 output 目录:");
    println!("   - Q1_spectrum_full.png: 全频段频谱");
    println!("   - Q1_spectrum_lowfreq.png: 低频段频谱 (0-4 kHz)");
    println!("   - Q1_spectrum_db.png: dB 刻度频谱");
    println!("   - Q1_waveform.png: 时域波形");
    println!("========================================\n");

    // 保存关键数据供后续使用 (使用对称峰值法确定的频率偏差)
    save_results_for_q2(f_d_symmetric, sample_rate)?;

    Ok(())
}

/// 保存结果供 Q2 使用
fn save_results_for_q2(f_d: f64, sample_rate: f64) -> Result<(), Box<dyn Error>> {
    use std::fs;
    use std::io::Write;

    fs::create_dir_all("output")?;
    let mut file = fs::File::create("output/Q1_results.txt")?;

    writeln!(file, "Q1 分析结果")?;
    writeln!(file, "===========")?;
    writeln!(file, "频率偏差 f_d = {:.4} Hz", f_d)?;
    writeln!(file, "采样率 f_s = {:.2} Hz", sample_rate)?;
    writeln!(file, "基带带宽 f_B = 4000 Hz")?;

    println!("结果已保存到 output/Q1_results.txt");

    Ok(())
}
```

Listing 2: 音频文件读取 - WAV 格式解析

音频读取模块 (audio_reader.rs)

```
// 1. 音频文件读取模块
// 负责读取 WAV 文件并提取采样数据、采样率和样本数

use hound::{WavReader, WavSpec};
use std::path::Path;

/// 音频数据结构
#[derive(Debug, Clone)]
pub struct AudioData {
    /// 采样数据 (归一化为浮点数)
    pub samples: Vec<f64>,
    /// 采样率 (Hz)
    pub sample_rate: u32,
    /// 样本数
    pub num_samples: usize,
    /// WAV 文件规格
    pub spec: WavSpec,
```

```rust
}

impl AudioData {
    /// 从 WAV 文件读取音频数据
    pub fn from_wav<P: AsRef<Path>>(path: P) -> Result<Self, Box<dyn
        std::error::Error>> {
        let mut reader = WavReader::open(path)?;
        let spec = reader.spec();
        let sample_rate = spec.sample_rate;

        // 读取所有采样点并归一化
        let samples: Vec<f64> = match spec.sample_format {
            hound::SampleFormat::Float => {
                reader
                    .samples::<f32>()
                    .map(|s| s.unwrap() as f64)
                    .collect()
            }
            hound::SampleFormat::Int => {
                let max_value = (1 << (spec.bits_per_sample - 1)) as f64;
                reader
                    .samples::<i32>()
                    .map(|s| s.unwrap() as f64 / max_value)
                    .collect()
            }
        };

        let num_samples = samples.len();

        println!("音频文件读取成功:");
        println!("  采样率: {} Hz", sample_rate);
        println!("  样本数: {}", num_samples);
        println!("  位深度: {} bits", spec.bits_per_sample);
        println!("  声道数: {}", spec.channels);
        println!("  时长: {:.2} 秒", num_samples as f64 / sample_rate as f64);

        Ok(AudioData {
            samples,
            sample_rate,
            num_samples,
            spec,
        })
    }

    /// 获取信号时长（秒）
    pub fn duration(&self) -> f64 {
        self.num_samples as f64 / self.sample_rate as f64
    }

    /// 获取单声道数据（如果是立体声则转换为单声道）
    pub fn to_mono(&self) -> Vec<f64> {
        if self.spec.channels == 1 {
            self.samples.clone()
        } else {
            // 立体声转单声道：取平均
            self.samples
```

```rust
                .chunks(self.spec.channels as usize)
                .map(|chunk| chunk.iter().sum::<f64>() / chunk.len() as f64)
                .collect()
        }
    }

    /// 保存为 WAV 文件
    pub fn save_wav<P: AsRef<Path>>(
        &self,
        path: P,
        samples: &[f64],
    ) -> Result<(), Box<dyn std::error::Error>> {
        let spec = hound::WavSpec {
            channels: 1,
            sample_rate: self.sample_rate,
            bits_per_sample: 16,
            sample_format: hound::SampleFormat::Int,
        };

        let mut writer = hound::WavWriter::create(path, spec)?;

        // 归一化并转换为 i16
        let max_amplitude = samples.iter().map(|&x| x.abs()).fold(0.0f64, f64::max);
        let scale = if max_amplitude > 0.0 {
            32767.0 / max_amplitude
        } else {
            32767.0
        };

        for &sample in samples {
            let sample_i16 = (sample * scale).clamp(-32768.0, 32767.0) as i16;
            writer.write_sample(sample_i16)?;
        }

        writer.finalize()?;
        println!("音频文件保存成功");
        Ok(())
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_audio_reader() {
        // 测试读取音频文件
        let result = AudioData::from_wav("../project.wav");
        assert!(result.is_ok());

        if let Ok(audio) = result {
            assert!(audio.sample_rate > 0);
            assert!(audio.num_samples > 0);
            assert_eq!(audio.samples.len(), audio.num_samples);
        }
    }
```

```
}
```

Listing 3: FFT 变换处理 - 使用 rustfft 库

FFT 处理模块 (fft_processor.rs)

```rust
// 2. FFT 计算模块
// 使用 rustfft 库对音频信号进行快速傅里叶变换

use rustfft::{FftPlanner, num_complex::Complex};
use std::f64::consts::PI;

/// FFT 结果结构
#[derive(Debug, Clone)]
pub struct FftResult {
    /// 频谱复数数据
    pub spectrum: Vec<Complex<f64>>,
    /// 频率轴 (Hz)
    pub frequencies: Vec<f64>,
    /// 幅度谱
    pub magnitude: Vec<f64>,
    /// 相位谱
    pub phase: Vec<f64>,
    /// 采样率
    pub sample_rate: f64,
}

impl FftResult {
    /// 计算信号的 FFT
    pub fn compute(samples: &[f64], sample_rate: f64) -> Self {
        let n = samples.len();
        let mut planner = FftPlanner::new();
        let fft = planner.plan_fft_forward(n);

        // 将实数信号转换为复数
        let mut buffer: Vec<Complex<f64>> = samples
            .iter()
            .map(|&x| Complex::new(x, 0.0))
            .collect();

        // 执行 FFT
        fft.process(&mut buffer);

        // 计算频率轴
        let frequencies: Vec<f64> = (0..n)
            .map(|k| k as f64 * sample_rate / n as f64)
            .collect();

        // 计算幅度谱 (归一化)
        let magnitude: Vec<f64> = buffer
            .iter()
            .map(|c| c.norm() / n as f64)
            .collect();

        // 计算相位谱
        let phase: Vec<f64> = buffer
            .iter()
```

```
            .map(|c| c.arg())
            .collect();

        println!("FFT 计算完成:");
        println!("  FFT 点数: {}", n);
        println!("  频率分辨率: {:.2} Hz", sample_rate / n as f64);

        FftResult {
            spectrum: buffer,
            frequencies,
            magnitude,
            phase,
            sample_rate,
        }
    }

    /// 执行逆 FFT
    pub fn ifft(spectrum: &[Complex<f64>]) -> Vec<f64> {
        let n = spectrum.len();
        let mut planner = FftPlanner::new();
        let ifft = planner.plan_fft_inverse(n);

        let mut buffer = spectrum.to_vec();
        ifft.process(&mut buffer);

        // 提取实部并归一化
        buffer
            .iter()
            .map(|c| c.re / n as f64)
            .collect()
    }

    /// 获取单边频谱 (0 到 Nyquist 频率)
    pub fn get_single_sided(&self) -> (Vec<f64>, Vec<f64>) {
        let nyquist_index = self.frequencies.len() / 2;
        let freqs = self.frequencies[..=nyquist_index].to_vec();
        let mags = self.magnitude[..=nyquist_index].to_vec();
        (freqs, mags)
    }

    /// 获取 dB 刻度的幅度谱
    pub fn get_magnitude_db(&self) -> Vec<f64> {
        self.magnitude
            .iter()
            .map(|&m| {
                if m > 1e-10 {
                    20.0 * m.log10()
                } else {
                    -200.0
                }
            })
            .collect()
    }

    /// 应用窗函数 (Hanning 窗)
    pub fn apply_hanning_window(samples: &[f64]) -> Vec<f64> {
```

```
        let n = samples.len();
        samples
            .iter()
            .enumerate()
            .map(|(i, &x)| {
                let window = 0.5 * (1.0 - (2.0 * PI * i as f64 / (n - 1) as
                    f64).cos());
                x * window
            })
            .collect()
    }

    /// 应用窗函数（Hamming 窗）
    pub fn apply_hamming_window(samples: &[f64]) -> Vec<f64> {
        let n = samples.len();
        samples
            .iter()
            .enumerate()
            .map(|(i, &x)| {
                let window = 0.54 - 0.46 * (2.0 * PI * i as f64 / (n - 1) as
                    f64).cos();
                x * window
            })
            .collect()
    }
}

/// 频谱搬移（循环移位）
pub fn circshift(spectrum: &[Complex<f64>], shift: isize) -> Vec<Complex<f64>> {
    let n = spectrum.len();
    let shift = shift.rem_euclid(n as isize) as usize;

    let mut result = vec![Complex::new(0.0, 0.0); n];
    for i in 0..n {
        result[(i + shift) % n] = spectrum[i];
    }
    result
}

/// 计算频域搬移后的和（用于解调）
pub fn frequency_shift_and_add(
    spectrum: &[Complex<f64>],
    shift_hz: f64,
    sample_rate: f64,
) -> Vec<Complex<f64>> {
    let n = spectrum.len();
    let shift_bins = (shift_hz * n as f64 / sample_rate).round() as isize;

    // 正向搬移和负向搬移
    let shifted_pos = circshift(spectrum, shift_bins);
    let shifted_neg = circshift(spectrum, -shift_bins);

    // 相加并除以 2
    shifted_pos
        .iter()
        .zip(shifted_neg.iter())
```

```
        .map(|(a, b)| (a + b) / 2.0)
        .collect()
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_fft() {
        // 测试 FFT 和 IFFT
        let sample_rate = 1000.0;
        let duration = 1.0;
        let n = (sample_rate * duration) as usize;

        // 生成测试信号: 10 Hz 正弦波
        let samples: Vec<f64> = (0..n)
            .map(|i| (2.0 * PI * 10.0 * i as f64 / sample_rate).sin())
            .collect();

        let fft_result = FftResult::compute(&samples, sample_rate);
        assert_eq!(fft_result.magnitude.len(), n);
        assert_eq!(fft_result.frequencies.len(), n);

        // 测试 IFFT
        let reconstructed = FftResult::ifft(&fft_result.spectrum);
        assert_eq!(reconstructed.len(), n);

        // 验证重构误差
        let error: f64 = samples
            .iter()
            .zip(reconstructed.iter())
            .map(|(a, b)| (a - b).powi(2))
            .sum::<f64>()
            .sqrt()
            / n as f64;
        assert!(error < 1e-10);
    }

    #[test]
    fn test_circshift() {
        let data: Vec<Complex<f64>> = (0..5)
            .map(|i| Complex::new(i as f64, 0.0))
            .collect();

        let shifted = circshift(&data, 2);
        assert_eq!(shifted[0].re, 3.0);
        assert_eq!(shifted[1].re, 4.0);
        assert_eq!(shifted[2].re, 0.0);
    }
}
```

Listing 4: 载波频率估计 - 峰值检测算法

频率估计模块 (frequency_estimator.rs)

```
// 4. 频率偏差估计模块
```

```rust
// 通过分析频谱找出频率偏差 f_d

/// 频率偏差估计器
pub struct FrequencyEstimator;

impl FrequencyEstimator {
    /// 估计频率偏差 f_d
    ///
    /// # 参数
    /// - frequencies: 频率轴
    /// - magnitude: 幅度谱
    /// - search_range: 搜索范围 (Hz), 例如 (10.0, 10000.0)
    /// - exclude_dc: 是否排除直流分量
    ///
    /// # 返回
    /// (peak_frequency, peak_magnitude, peak_index)
    pub fn estimate_frequency_offset(
        frequencies: &[f64],
        magnitude: &[f64],
        search_range: (f64, f64),
        exclude_dc: bool,
    ) -> (f64, f64, usize) {
        let (min_freq, max_freq) = search_range;

        // 在指定范围内搜索峰值
        let mut peak_magnitude = 0.0;
        let mut peak_index = 0;
        let mut peak_frequency = 0.0;

        let start_idx = if exclude_dc { 1 } else { 0 };

        for (i, (&freq, &mag)) in frequencies
            .iter()
            .zip(magnitude.iter())
            .enumerate()
            .skip(start_idx)
        {
            if freq >= min_freq && freq <= max_freq && mag > peak_magnitude {
                peak_magnitude = mag;
                peak_index = i;
                peak_frequency = freq;
            }
        }

        println!("频率偏差估计结果:");
        println!("  峰值频率 f_d = {:.2} Hz", peak_frequency);
        println!("  峰值幅度 = {:.6}", peak_magnitude);
        println!("  峰值索引 = {}", peak_index);

        (peak_frequency, peak_magnitude, peak_index)
    }

    /// 精确估计频率 (使用抛物线插值)
    pub fn refined_frequency_estimate(
        frequencies: &[f64],
        magnitude: &[f64],
```

```
        peak_index: usize,
    ) -> f64 {
        if peak_index == 0 || peak_index >= magnitude.len() - 1 {
            return frequencies[peak_index];
        }

        // 使用三点抛物线插值
        let y1 = magnitude[peak_index - 1];
        let y2 = magnitude[peak_index];
        let y3 = magnitude[peak_index + 1];

        // 抛物线顶点位置
        let delta = 0.5 * (y1 - y3) / (y1 - 2.0 * y2 + y3);

        let freq_resolution = if frequencies.len() > 1 {
            frequencies[1] - frequencies[0]
        } else {
            1.0
        };

        let refined_freq = frequencies[peak_index] + delta * freq_resolution;

        println!("精确频率估计:");
        println!("  原始峰值频率: {:.2} Hz", frequencies[peak_index]);
        println!("  精确频率: {:.4} Hz", refined_freq);

        refined_freq
    }

    /// 寻找多个峰值
    pub fn find_multiple_peaks(
        frequencies: &[f64],
        magnitude: &[f64],
        num_peaks: usize,
        min_distance: usize,
        threshold: f64,
    ) -> Vec<(f64, f64, usize)> {
        let mut peaks = Vec::new();
        let n = magnitude.len();

        // 找出所有局部最大值
        for i in 1..n-1 {
            if magnitude[i] > magnitude[i-1]
                && magnitude[i] > magnitude[i+1]
                && magnitude[i] > threshold
            {
                peaks.push((frequencies[i], magnitude[i], i));
            }
        }

        // 按幅度降序排序
        peaks.sort_by(|a, b| b.1.partial_cmp(&a.1).unwrap());

        // 移除距离太近的峰值
        let mut filtered_peaks = Vec::new();
        for peak in peaks {
```

```rust
            let is_far_enough = filtered_peaks
                .iter()
                .all(|(_, _, idx)| (*idx as isize - peak.2 as isize).abs() >=
                    min_distance as isize);

            if is_far_enough {
                filtered_peaks.push(peak);
                if filtered_peaks.len() >= num_peaks {
                    break;
                }
            }
        }

        println!("找到 {} 个峰值:", filtered_peaks.len());
        for (i, (freq, mag, idx)) in filtered_peaks.iter().enumerate() {
            println!("  峰值 {}: 频率 = {:.2} Hz, 幅度 = {:.6}, 索引 = {}",
                     i + 1, freq, mag, idx);
        }

        filtered_peaks
    }

    /// 计算信号的能量分布
    pub fn compute_energy_distribution(
        magnitude: &[f64],
        frequencies: &[f64],
        bands: &[(f64, f64)], // 频带范围
    ) -> Vec<(String, f64)> {
        let total_energy: f64 = magnitude.iter().map(|&m| m * m).sum();

        let mut band_energies = Vec::new();

        for (low, high) in bands {
            let energy: f64 = frequencies
                .iter()
                .zip(magnitude.iter())
                .filter(|(&f, _)| f >= *low && f <= *high)
                .map(|(_, &m)| m * m)
                .sum();

            let percentage = (energy / total_energy) * 100.0;
            band_energies.push((
                format!("{:.0}-{:.0} Hz", low, high),
                percentage,
            ));
        }

        println!("\n能量分布:");
        for (band, percent) in &band_energies {
            println!("  {}: {:.2}%", band, percent);
        }

        band_energies
    }

    /// 判断 f_c_tilde 与 f_c 的大小关系
```

```rust
    ///
    /// 注意：仅从频谱的对称性很难直接判断，通常需要相位信息或其他先验知识
    pub fn analyze_frequency_relationship(
        frequencies: &[f64],
        magnitude: &[f64],
        f_d: f64,
    ) -> String {
        println!("\n频率关系分析:");
        println!("  估计的频率偏差 f_d = {:.2} Hz", f_d);
        println!("  仅从幅度谱无法唯一确定 f_c_tilde > f_c 还是 f_c_tilde < f_c");
        println!("  原因：频谱的对称性使得两种情况产生相同的幅度谱");
        println!("  需要：相位信息、时域分析或其他先验知识来确定符号");

        String::from("无法仅从幅度谱确定频率偏差的符号")
    }

    /// 计算信噪比 (SNR) 估计
    pub fn estimate_snr(
        magnitude: &[f64],
        signal_band: (usize, usize),
        noise_band: (usize, usize),
    ) -> f64 {
        let signal_power: f64 = magnitude[signal_band.0..signal_band.1]
            .iter()
            .map(|&m| m * m)
            .sum::<f64>() / (signal_band.1 - signal_band.0) as f64;

        let noise_power: f64 = magnitude[noise_band.0..noise_band.1]
            .iter()
            .map(|&m| m * m)
            .sum::<f64>() / (noise_band.1 - noise_band.0) as f64;

        let snr_db = if noise_power > 0.0 {
            10.0 * (signal_power / noise_power).log10()
        } else {
            f64::INFINITY
        };

        println!("\n信噪比估计:");
        println!("  信号功率: {:.6}", signal_power);
        println!("  噪声功率: {:.6}", noise_power);
        println!("  SNR: {:.2} dB", snr_db);

        snr_db
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use std::f64::consts::PI;

    #[test]
    fn test_frequency_estimation() {
        // 生成测试信号：包含 100 Hz 的峰值
        let n = 1000;
```

```
        let sample_rate = 1000.0;
        let frequencies: Vec<f64> = (0..n).map(|i| i as f64 * sample_rate / n as
            f64).collect();

        let magnitude: Vec<f64> = frequencies
            .iter()
            .map(|&f| {
                if (f - 100.0).abs() < 1.0 {
                    1.0
                } else {
                    0.01
                }
            })
            .collect();

        let (peak_freq, peak_mag, peak_idx) =
            FrequencyEstimator::estimate_frequency_offset(
            &frequencies,
            &magnitude,
            (10.0, 500.0),
            true,
        );

        assert!((peak_freq - 100.0).abs() < 2.0);
        assert!(peak_mag > 0.5);
        assert!(peak_idx > 0);
    }

    #[test]
    fn test_multiple_peaks() {
        let n = 1000;
        let sample_rate = 1000.0;
        let frequencies: Vec<f64> = (0..n).map(|i| i as f64 * sample_rate / n as
            f64).collect();

        // 生成包含两个峰值的信号
        let magnitude: Vec<f64> = frequencies
            .iter()
            .map(|&f| {
                let peak1 = (-((f - 100.0) / 10.0).powi(2)).exp();
                let peak2 = 0.8 * (-((f - 300.0) / 10.0).powi(2)).exp();
                peak1 + peak2
            })
            .collect();

        let peaks = FrequencyEstimator::find_multiple_peaks(
            &frequencies,
            &magnitude,
            2,
            20,
            0.1,
        );

        assert_eq!(peaks.len(), 2);
    }
}
```

Listing 5: 频谱图绘制 - 使用 plotters 库

频谱可视化模块 (spectrum_visualizer.rs)

```rust
// 3. 频谱可视化模块
// 使用 plotters 库绘制频谱图

use plotters::prelude::*;
use std::path::Path;

/// 频谱可视化器
pub struct SpectrumVisualizer;

impl SpectrumVisualizer {
    /// 绘制频谱图（幅度谱）
    pub fn plot_spectrum<P: AsRef<Path>>(
        frequencies: &[f64],
        magnitude: &[f64],
        output_path: P,
        title: &str,
        max_freq: Option<f64>,
    ) -> Result<(), Box<dyn std::error::Error>> {
        // 只显示到指定频率或 Nyquist 频率
        let nyquist = frequencies.last().copied().unwrap_or(0.0) / 2.0;
        let max_f = max_freq.unwrap_or(nyquist);

        // 过滤数据点
        let data: Vec<(f64, f64)> = frequencies
            .iter()
            .zip(magnitude.iter())
            .filter(|(&f, _)| f <= max_f)
            .map(|(&f, &m)| (f, m))
            .collect();

        if data.is_empty() {
            return Err("没有数据可以绘制".into());
        }

        // 找出幅度范围
        let max_magnitude = data.iter().map(|(_, m)| m).fold(0.0f64, |a, &b|
            a.max(b));
        let y_max = max_magnitude * 1.1;

        // 创建绘图区域 - 使用文件路径
        let root = BitMapBackend::new(output_path.as_ref(), (1200, 600))
            .into_drawing_area();
        root.fill(&WHITE)?;

        let mut chart = ChartBuilder::on(&root)
            .caption(title, ("Arial", 30).into_font())
            .margin(10)
            .x_label_area_size(40)
            .y_label_area_size(60)
            .build_cartesian_2d(0.0..max_f, 0.0..y_max)?;

        chart
            .configure_mesh()
            .x_desc("Frequency (Hz)")
            .y_desc("Magnitude")
```

```rust
            .draw()?;

        // 绘制频谱曲线
        chart.draw_series(LineSeries::new(
            data.iter().map(|&(f, m)| (f, m)),
            &BLUE,
        ))?;

        root.present()?;
        println!("频谱图已保存到: {:?}", output_path.as_ref());
        Ok(())
    }

    /// 绘制频谱图（dB 刻度）
    pub fn plot_spectrum_db<P: AsRef<Path>>(
        frequencies: &[f64],
        magnitude_db: &[f64],
        output_path: P,
        title: &str,
        max_freq: Option<f64>,
    ) -> Result<(), Box<dyn std::error::Error>> {
        let nyquist = frequencies.last().copied().unwrap_or(0.0) / 2.0;
        let max_f = max_freq.unwrap_or(nyquist);

        let data: Vec<(f64, f64)> = frequencies
            .iter()
            .zip(magnitude_db.iter())
            .filter(|(&f, _)| f <= max_f)
            .map(|(&f, &m)| (f, m))
            .collect();

        if data.is_empty() {
            return Err("没有数据可以绘制".into());
        }

        let max_db = data.iter().map(|(_, m)| m).fold(-200.0f64, |a, &b| a.max(b));
        let min_db = -100.0;

        let root = BitMapBackend::new(output_path.as_ref(), (1200, 600))
            .into_drawing_area();
        root.fill(&WHITE)?;

        let mut chart = ChartBuilder::on(&root)
            .caption(title, ("Arial", 30).into_font())
            .margin(10)
            .x_label_area_size(40)
            .y_label_area_size(60)
            .build_cartesian_2d(0.0..max_f, min_db..max_db)?;

        chart
            .configure_mesh()
            .x_desc("Frequency (Hz)")
            .y_desc("Magnitude (dB)")
            .draw()?;

        chart.draw_series(LineSeries::new(
```

```
            data.iter().map(|&(f, m)| (f, m)),
            &RED,
        ))?;

        root.present()?;
        println!("频谱图（dB）已保存到: {:?}", output_path.as_ref());
        Ok(())
    }

    /// 绘制时域波形
    pub fn plot_waveform<P: AsRef<Path>>(
        samples: &[f64],
        sample_rate: f64,
        output_path: P,
        title: &str,
        max_samples: Option<usize>,
    ) -> Result<(), Box<dyn std::error::Error>> {
        // 限制显示的采样点数量
        let n = max_samples.unwrap_or(samples.len()).min(samples.len());
        let time: Vec<f64> = (0..n)
            .map(|i| i as f64 / sample_rate)
            .collect();

        let data: Vec<(f64, f64)> = time
            .iter()
            .zip(samples.iter())
            .take(n)
            .map(|(&t, &s)| (t, s))
            .collect();

        let max_amplitude = samples.iter().map(|&x| x.abs()).fold(0.0f64, f64::max);
        let y_range = max_amplitude * 1.2;

        let root = BitMapBackend::new(output_path.as_ref(), (1200, 600))
            .into_drawing_area();
        root.fill(&WHITE)?;

        let mut chart = ChartBuilder::on(&root)
            .caption(title, ("Arial", 30).into_font())
            .margin(10)
            .x_label_area_size(40)
            .y_label_area_size(60)
            .build_cartesian_2d(0.0..time[n-1], -y_range..y_range)?;

        chart
            .configure_mesh()
            .x_desc("Time (seconds)")
            .y_desc("Amplitude")
            .draw()?;

        chart.draw_series(LineSeries::new(
            data.iter().map(|&(t, s)| (t, s)),
            &GREEN,
        ))?;

        root.present()?;
```

```rust
        println!("时域波形图已保存到: {:?}", output_path.as_ref());
        Ok(())
    }

    /// 绘制多个频谱对比图
    pub fn plot_spectrum_comparison<P: AsRef<Path>>(
        datasets: Vec<(&[f64], &[f64], &str)>, // (frequencies, magnitude, label)
        output_path: P,
        title: &str,
        max_freq: Option<f64>,
    ) -> Result<(), Box<dyn std::error::Error>> {
        if datasets.is_empty() {
            return Err("没有数据可以绘制".into());
        }

        let nyquist = datasets[0].0.last().copied().unwrap_or(0.0) / 2.0;
        let max_f = max_freq.unwrap_or(nyquist);

        // 找出最大幅度
        let max_magnitude = datasets
            .iter()
            .flat_map(|(_, mag, _)| mag.iter())
            .copied()
            .fold(0.0f64, f64::max);
        let y_max = max_magnitude * 1.1;

        let root = BitMapBackend::new(output_path.as_ref(), (1200, 600))
            .into_drawing_area();
        root.fill(&WHITE)?;

        let mut chart = ChartBuilder::on(&root)
            .caption(title, ("Arial", 30).into_font())
            .margin(10)
            .x_label_area_size(40)
            .y_label_area_size(60)
            .build_cartesian_2d(0.0..max_f, 0.0..y_max)?;

        chart
            .configure_mesh()
            .x_desc("Frequency (Hz)")
            .y_desc("Magnitude")
            .draw()?;

        let colors = [&BLUE, &RED, &GREEN, &CYAN, &MAGENTA];

        for (idx, (freqs, mags, label)) in datasets.iter().enumerate() {
            let data: Vec<(f64, f64)> = freqs
                .iter()
                .zip(mags.iter())
                .filter(|(&f, _)| f <= max_f)
                .map(|(&f, &m)| (f, m))
                .collect();

            let color = colors[idx % colors.len()];
            chart
                .draw_series(LineSeries::new(data.iter().map(|&(f, m)| (f, m)),
```

```
                    color))?
                .label(*label)
                .legend(move |(x, y)| PathElement::new(vec![(x, y), (x + 20, y)],
                    color));
        }

        chart
            .configure_series_labels()
            .background_style(&WHITE.mix(0.8))
            .border_style(&BLACK)
            .draw()?;

        root.present()?;
        println!("对比频谱图已保存到: {:?}", output_path.as_ref());
        Ok(())
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use std::f64::consts::PI;

    #[test]
    fn test_plot_spectrum() {
        // 生成测试数据
        let n = 1000;
        let sample_rate = 1000.0;
        let frequencies: Vec<f64> = (0..n).map(|i| i as f64 * sample_rate / n as
            f64).collect();

        // 模拟频谱
        let magnitude: Vec<f64> = frequencies
            .iter()
            .map(|&f| {
                if (f - 100.0).abs() < 10.0 {
                    1.0
                } else {
                    0.1 * (-((f - 100.0) / 50.0).powi(2)).exp()
                }
            })
            .collect();

        let result = SpectrumVisualizer::plot_spectrum(
            &frequencies,
            &magnitude,
            "/tmp/test_spectrum.png",
            "测试频谱",
            Some(500.0),
        );

        assert!(result.is_ok());
    }
}
```

C.6.2 Q2 - Butterworth 滤波器设计

主程序 (main.rs) Listing 6: Q2 主程序 - 滤波器设计流程

```rust
mod butterworth_filter;
mod filter_response;
mod response_visualizer;

use std::fs;
use std::path::Path;

fn main() {
    println!("=== Q2: Butterworth Filter Design ===\n");

    // Read parameters from Q1 results
    let q1_results_path = "../Q1/output/Q1_results.txt";
    let (sample_rate, f_d, f_b) = read_q1_results(q1_results_path);

    println!("Parameters from Q1:");
    println!("  Sample Rate: {} Hz", sample_rate);
    println!("  Frequency Offset (f_d): {:.4} Hz", f_d);
    println!("  Signal Bandwidth (f_B): {} Hz", f_b);
    println!();

    // Design 8th-order Butterworth filters
    let order = 8;
    println!("Designing 8th-order Butterworth filters...");

    // High-pass filter with cutoff frequency f_d
    println!("  - High-pass filter (cutoff = {:.4} Hz)", f_d);
    let highpass = butterworth_filter::ButterworthFilter::highpass(order, f_d,
        sample_rate);

    // Low-pass filter with cutoff frequency f_B
    println!("  - Low-pass filter (cutoff = {} Hz)", f_b);
    let lowpass = butterworth_filter::ButterworthFilter::lowpass(order, f_b,
        sample_rate);

    println!("\nHigh-pass filter coefficients:");
    println!("  b (numerator): {:?}", &highpass.b[..5.min(highpass.b.len())]);
    println!("  a (denominator): {:?}", &highpass.a[..5.min(highpass.a.len())]);

    println!("\nLow-pass filter coefficients:");
    println!("  b (numerator): {:?}", &lowpass.b[..5.min(lowpass.b.len())]);
    println!("  a (denominator): {:?}", &lowpass.a[..5.min(lowpass.a.len())]);

    // Calculate frequency response at the same frequency points as Q1
    let num_points = 31265; // Same as Q1 audio samples
    println!("\nCalculating frequency responses ({} points)...", num_points);

    let hp_response = filter_response::FilterResponse::compute(&highpass,
        sample_rate, num_points);
    let lp_response = filter_response::FilterResponse::compute(&lowpass, sample_rate,
        num_points);

    // Create output directory
```

```rust
    let output_dir = "output";
    fs::create_dir_all(output_dir).expect("Failed to create output directory");

    // Plot frequency responses
    println!("\nGenerating plots...");

    // High-pass filter magnitude response
    response_visualizer::plot_magnitude_response(
        &hp_response.frequencies,
        &hp_response.magnitude,
        &format!("{}/Q2_highpass_magnitude.png", output_dir),
        "High-pass Filter Magnitude Response",
        Some(10000.0),
    ).expect("Failed to plot high-pass magnitude");

    // High-pass filter magnitude response in dB
    response_visualizer::plot_magnitude_response_db(
        &hp_response.frequencies,
        &hp_response.magnitude,
        &format!("{}/Q2_highpass_magnitude_db.png", output_dir),
        "High-pass Filter Magnitude Response (dB)",
        Some(10000.0),
    ).expect("Failed to plot high-pass magnitude dB");

    // High-pass filter phase response
    response_visualizer::plot_phase_response(
        &hp_response.frequencies,
        &hp_response.phase,
        &format!("{}/Q2_highpass_phase.png", output_dir),
        "High-pass Filter Phase Response",
        Some(10000.0),
    ).expect("Failed to plot high-pass phase");

    // Low-pass filter magnitude response
    response_visualizer::plot_magnitude_response(
        &lp_response.frequencies,
        &lp_response.magnitude,
        &format!("{}/Q2_lowpass_magnitude.png", output_dir),
        "Low-pass Filter Magnitude Response",
        Some(10000.0),
    ).expect("Failed to plot low-pass magnitude");

    // Low-pass filter magnitude response in dB
    response_visualizer::plot_magnitude_response_db(
        &lp_response.frequencies,
        &lp_response.magnitude,
        &format!("{}/Q2_lowpass_magnitude_db.png", output_dir),
        "Low-pass Filter Magnitude Response (dB)",
        Some(10000.0),
    ).expect("Failed to plot low-pass magnitude dB");

    // Low-pass filter phase response
    response_visualizer::plot_phase_response(
        &lp_response.frequencies,
        &lp_response.phase,
        &format!("{}/Q2_lowpass_phase.png", output_dir),
```

```
        "Low-pass Filter Phase Response",
        Some(10000.0),
    ).expect("Failed to plot low-pass phase");

    // Combined magnitude plot
    response_visualizer::plot_combined_magnitude(
        &hp_response.frequencies,
        &hp_response.magnitude,
        &lp_response.magnitude,
        &format!("{}/Q2_combined_magnitude.png", output_dir),
        "Combined Filter Magnitude Responses",
        Some(10000.0),
    ).expect("Failed to plot combined magnitude");

    // Save filter coefficients
    save_filter_coefficients(&highpass, &lowpass,
        &format!("{}/Q2_filter_coefficients.txt", output_dir));

    // Save frequency response data
    save_frequency_response(&hp_response, &lp_response,
        &format!("{}/Q2_frequency_response.txt", output_dir));

    println!("\nAll results saved to '{}/' directory", output_dir);
    println!("\nQ2 completed successfully!");
}

fn read_q1_results(path: &str) -> (f64, f64, f64) {
    let content = fs::read_to_string(path)
        .expect("Failed to read Q1 results file");

    let mut sample_rate = 22050.0;
    let mut f_d = 3225.0;
    let mut f_b = 4000.0;

    for line in content.lines() {
        // Parse: "频率偏差 f_d = 3225.1032 Hz"
        if line.contains("频率偏差") && line.contains("f_d") {
            if let Some(value_str) = line.split('=').nth(1) {
                if let Some(num_str) = value_str.trim().split_whitespace().next() {
                    f_d = num_str.parse().unwrap_or(3225.0);
                }
            }
        }
        // Parse: "采样率 f_s = 22050.00 Hz"
        else if line.contains("采样率") && line.contains("f_s") {
            if let Some(value_str) = line.split('=').nth(1) {
                if let Some(num_str) = value_str.trim().split_whitespace().next() {
                    sample_rate = num_str.parse().unwrap_or(22050.0);
                }
            }
        }
        // Parse: "基带带宽 f_B = 4000 Hz"
        else if line.contains("基带带宽") && line.contains("f_B") {
            if let Some(value_str) = line.split('=').nth(1) {
                if let Some(num_str) = value_str.trim().split_whitespace().next() {
                    f_b = num_str.parse().unwrap_or(4000.0);
```

```
                }
            }
        }
    }

    (sample_rate, f_d, f_b)
}

fn save_filter_coefficients(highpass: &butterworth_filter::ButterworthFilter,
                            lowpass: &butterworth_filter::ButterworthFilter,
                            path: &str) {
    let mut content = String::new();
    content.push_str("=== Q2: Filter Coefficients ===\n\n");

    content.push_str("High-pass Filter (8th-order Butterworth):\n");
    content.push_str(&format!("Cutoff Frequency: {:.4} Hz\n", highpass.cutoff));
    content.push_str(&format!("Sample Rate: {} Hz\n", highpass.sample_rate));
    content.push_str("\nNumerator Coefficients (b):\n");
    for (i, coef) in highpass.b.iter().enumerate() {
        content.push_str(&format!("  b[{}] = {:.15e}\n", i, coef));
    }
    content.push_str("\nDenominator Coefficients (a):\n");
    for (i, coef) in highpass.a.iter().enumerate() {
        content.push_str(&format!("  a[{}] = {:.15e}\n", i, coef));
    }

    content.push_str("\n\nLow-pass Filter (8th-order Butterworth):\n");
    content.push_str(&format!("Cutoff Frequency: {:.4} Hz\n", lowpass.cutoff));
    content.push_str(&format!("Sample Rate: {} Hz\n", lowpass.sample_rate));
    content.push_str("\nNumerator Coefficients (b):\n");
    for (i, coef) in lowpass.b.iter().enumerate() {
        content.push_str(&format!("  b[{}] = {:.15e}\n", i, coef));
    }
    content.push_str("\nDenominator Coefficients (a):\n");
    for (i, coef) in lowpass.a.iter().enumerate() {
        content.push_str(&format!("  a[{}] = {:.15e}\n", i, coef));
    }

    fs::write(path, content).expect("Failed to write filter coefficients");
}

fn save_frequency_response(hp_response: &filter_response::FilterResponse,
                           lp_response: &filter_response::FilterResponse,
                           path: &str) {
    let mut content = String::new();
    content.push_str("=== Q2: Frequency Response Data ===\n\n");

    content.push_str("High-pass Filter:\n");
    content.push_str(&format!("Number of frequency points: {}\n",
        hp_response.frequencies.len()));
    content.push_str(&format!("Frequency range: 0 - {:.2} Hz\n",
        hp_response.frequencies.last().unwrap_or(&0.0)));
    content.push_str(&format!("Maximum magnitude: {:.6}\n",
        hp_response.magnitude.iter().cloned().fold(0./0., f64::max)));
    content.push_str(&format!("Minimum magnitude: {:.6}\n",
        hp_response.magnitude.iter().cloned().fold(f64::INFINITY, f64::min)));
```

```
211
212     content.push_str("\nLow-pass Filter:\n");
213     content.push_str(&format!("Number of frequency points: {}\n",
            lp_response.frequencies.len()));
214     content.push_str(&format!("Frequency range: 0 - {:.2} Hz\n",
            lp_response.frequencies.last().unwrap_or(&0.0)));
215     content.push_str(&format!("Maximum magnitude: {:.6}\n",
            lp_response.magnitude.iter().cloned().fold(0./0., f64::max)));
216     content.push_str(&format!("Minimum magnitude: {:.6}\n",
            lp_response.magnitude.iter().cloned().fold(f64::INFINITY, f64::min)));
217
218     fs::write(path, content).expect("Failed to write frequency response data");
219 }
```

Listing 7: Butterworth 滤波器设计 - 双线性变换法

滤波器设计模块 (butterworth_filter.rs)

```
1  use std::f64::consts::PI;
2  use num_complex::Complex;
3
4  pub struct ButterworthFilter {
5      pub b: Vec<f64>,
6      pub a: Vec<f64>,
7      pub order: usize,
8      pub cutoff: f64,
9      pub sample_rate: f64,
10     pub filter_type: FilterType,
11 }
12
13 #[derive(Debug, Clone, Copy)]
14 pub enum FilterType {
15     Lowpass,
16     Highpass,
17 }
18
19 impl ButterworthFilter {
20     pub fn lowpass(order: usize, cutoff: f64, sample_rate: f64) -> Self {
21         let (b, a) = design_butterworth_digital_lowpass(order, cutoff, sample_rate);
22         Self { b, a, order, cutoff, sample_rate, filter_type: FilterType::Lowpass }
23     }
24
25     pub fn highpass(order: usize, cutoff: f64, sample_rate: f64) -> Self {
26         let (b, a) = design_butterworth_digital_highpass(order, cutoff, sample_rate);
27         Self { b, a, order, cutoff, sample_rate, filter_type: FilterType::Highpass }
28     }
29 }
30
31 fn design_butterworth_digital_lowpass(order: usize, cutoff: f64, fs: f64) ->
       (Vec<f64>, Vec<f64>) {
32     // Pre-warp the cutoff frequency to compensate for bilinear transform distortion
33     let wc = 2.0 * fs * (PI * cutoff / fs).tan();
34     let poles = butterworth_analog_poles(order);
35     let scaled_poles: Vec<_> = poles.iter().map(|(re, im)| (re * wc, im *
           wc)).collect();
36     bilinear_transform_cascade(&scaled_poles, fs)
37 }
```

```rust

fn design_butterworth_digital_highpass(order: usize, cutoff: f64, fs: f64) -> (Vec<f64>, Vec<f64>) {
    // Spectral inversion method: H_HP(z) = H_LP(-z)
    // The cutoff frequency is mirrored about fs/4
    // To get highpass with cutoff fc_hp, design lowpass with cutoff (fs/2 - fc_hp)
    let lp_cutoff = fs / 2.0 - cutoff;

    // Design a lowpass filter at the mirrored cutoff frequency
    let (b_lp, a_lp) = design_butterworth_digital_lowpass(order, lp_cutoff, fs);

    // Convert lowpass to highpass using spectral inversion: H_HP(z) = H_LP(-z)
    // This means alternating the signs of coefficients with odd indices
    let mut b_hp = b_lp.clone();
    let mut a_hp = a_lp.clone();

    for (i, val) in b_hp.iter_mut().enumerate() {
        if i % 2 == 1 {
            *val = -*val;
        }
    }

    for (i, val) in a_hp.iter_mut().enumerate() {
        if i % 2 == 1 {
            *val = -*val;
        }
    }

    (b_hp, a_hp)
}

fn butterworth_analog_poles(order: usize) -> Vec<(f64, f64)> {
    (0..order).map(|k| {
        let theta = PI * (2.0 * k as f64 + order as f64 + 1.0) / (2.0 * order as f64);
        (theta.cos(), theta.sin())
    }).collect()
}

fn bilinear_transform_cascade(poles: &[(f64, f64)], fs: f64) -> (Vec<f64>, Vec<f64>) {
    let t = 1.0 / fs;
    let mut b_total = vec![1.0];
    let mut a_total = vec![1.0];
    let mut i = 0;
    while i < poles.len() {
        let (pr1, pi1) = poles[i];
        if pi1.abs() < 1e-10 {
            let denom = 2.0 - pr1 * t;
            let z_pole = (2.0 + pr1 * t) / denom;
            b_total = convolve(&b_total, &vec![1.0, 1.0]);
            a_total = convolve(&a_total, &vec![1.0, -z_pole]);
            i += 1;
        } else {
            if i + 1 < poles.len() {
                let b_section = vec![1.0, 2.0, 1.0];
                let denom_re = 2.0 - pr1 * t;
                let denom_im = -pi1 * t;
```

```
                let denom_mag_sq = denom_re * denom_re + denom_im * denom_im;
                let z1_re = ((2.0 + pr1 * t) * denom_re + pi1 * t * denom_im) /
                    denom_mag_sq;
                let z1_im = ((pi1 * t) * denom_re - (2.0 + pr1 * t) * denom_im) /
                    denom_mag_sq;
                let a1 = -2.0 * z1_re;
                let a2 = z1_re * z1_re + z1_im * z1_im;
                let a_section = vec![1.0, a1, a2];
                b_total = convolve(&b_total, &b_section);
                a_total = convolve(&a_total, &a_section);
                i += 2;
            } else {
                i += 1;
            }
        }
    }
    let a0 = a_total[0];
    for i in 0..a_total.len() { a_total[i] /= a0; }
    for i in 0..b_total.len() { b_total[i] /= a0; }
    let b_sum: f64 = b_total.iter().sum();
    let a_sum: f64 = a_total.iter().sum();
    let gain = a_sum / b_sum;
    for i in 0..b_total.len() { b_total[i] *= gain; }
    (b_total, a_total)
}

fn convolve(a: &[f64], b: &[f64]) -> Vec<f64> {
    let mut result = vec![0.0; a.len() + b.len() - 1];
    for (i, &a_val) in a.iter().enumerate() {
        for (j, &b_val) in b.iter().enumerate() {
            result[i + j] += a_val * b_val;
        }
    }
    result
}

fn lowpass_to_highpass(b_lp: &[f64], a_lp: &[f64]) -> (Vec<f64>, Vec<f64>) {
    let mut b_hp = b_lp.to_vec();
    let mut a_hp = a_lp.to_vec();
    for (i, val) in b_hp.iter_mut().enumerate() {
        if i % 2 == 1 { *val = -*val; }
    }
    for (i, val) in a_hp.iter_mut().enumerate() {
        if i % 2 == 1 { *val = -*val; }
    }
    (b_hp, a_hp)
}
```

Listing 8: 频率响应可视化 - 幅频/相频特性

响应可视化模块 (response_visualizer.rs)

```
use plotters::prelude::*;
use crate::filter_response;

const PLOT_WIDTH: u32 = 1200;
const PLOT_HEIGHT: u32 = 600;
```

```rust

/// Plot magnitude response (linear scale)
pub fn plot_magnitude_response(
    frequencies: &[f64],
    magnitude: &[f64],
    output_path: &str,
    title: &str,
    max_freq: Option<f64>,
) -> Result<(), Box<dyn std::error::Error>> {
    let root = BitMapBackend::new(output_path, (PLOT_WIDTH,
        PLOT_HEIGHT)).into_drawing_area();
    root.fill(&WHITE)?;

    let max_freq_val = max_freq.unwrap_or(*frequencies.last().unwrap_or(&10000.0));
    let max_mag = magnitude.iter()
        .zip(frequencies.iter())
        .filter(|(_, &f)| f <= max_freq_val)
        .map(|(&m, _)| m)
        .fold(0.0, f64::max)
        .max(1.1);

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 30).into_font())
        .margin(15)
        .x_label_area_size(50)
        .y_label_area_size(60)
        .build_cartesian_2d(0.0..max_freq_val, 0.0..max_mag)?;

    chart.configure_mesh()
        .x_desc("Frequency (Hz)")
        .y_desc("Magnitude")
        .draw()?;

    chart.draw_series(LineSeries::new(
        frequencies.iter()
            .zip(magnitude.iter())
            .filter(|(&f, _)| f <= max_freq_val)
            .map(|(&f, &m)| (f, m)),
        &BLUE,
    ))?;

    root.present()?;
    Ok(())
}

/// Plot magnitude response in dB scale
pub fn plot_magnitude_response_db(
    frequencies: &[f64],
    magnitude: &[f64],
    output_path: &str,
    title: &str,
    max_freq: Option<f64>,
) -> Result<(), Box<dyn std::error::Error>> {
    let root = BitMapBackend::new(output_path, (PLOT_WIDTH,
        PLOT_HEIGHT)).into_drawing_area();
    root.fill(&WHITE)?;
```

```rust

    let max_freq_val = max_freq.unwrap_or(*frequencies.last().unwrap_or(&10000.0));

    let magnitude_db: Vec<f64> = magnitude.iter()
        .map(|&m| filter_response::magnitude_to_db(m))
        .collect();

    let min_db = magnitude_db.iter()
        .zip(frequencies.iter())
        .filter(|(_, &f)| f <= max_freq_val)
        .map(|(&db, _)| db)
        .fold(f64::INFINITY, f64::min)
        .max(-80.0);

    let max_db = 10.0;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 30).into_font())
        .margin(15)
        .x_label_area_size(50)
        .y_label_area_size(60)
        .build_cartesian_2d(0.0..max_freq_val, min_db..max_db)?;

    chart.configure_mesh()
        .x_desc("Frequency (Hz)")
        .y_desc("Magnitude (dB)")
        .draw()?;

    chart.draw_series(LineSeries::new(
        frequencies.iter()
            .zip(magnitude_db.iter())
            .filter(|(&f, _)| f <= max_freq_val)
            .map(|(&f, &db)| (f, db)),
        &BLUE,
    ))?;

    root.present()?;
    Ok(())
}

/// Plot phase response
pub fn plot_phase_response(
    frequencies: &[f64],
    phase: &[f64],
    output_path: &str,
    title: &str,
    max_freq: Option<f64>,
) -> Result<(), Box<dyn std::error::Error>> {
    let root = BitMapBackend::new(output_path, (PLOT_WIDTH,
        PLOT_HEIGHT)).into_drawing_area();
    root.fill(&WHITE)?;

    let max_freq_val = max_freq.unwrap_or(*frequencies.last().unwrap_or(&10000.0));

    let phase_deg: Vec<f64> = phase.iter()
        .map(|&p| filter_response::phase_to_degrees(p))
```

```
        .collect();

    let min_phase = phase_deg.iter()
        .zip(frequencies.iter())
        .filter(|(_, &f)| f <= max_freq_val)
        .map(|(&p, _)| p)
        .fold(f64::INFINITY, f64::min) - 10.0;

    let max_phase = phase_deg.iter()
        .zip(frequencies.iter())
        .filter(|(_, &f)| f <= max_freq_val)
        .map(|(&p, _)| p)
        .fold(f64::NEG_INFINITY, f64::max) + 10.0;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 30).into_font())
        .margin(15)
        .x_label_area_size(50)
        .y_label_area_size(60)
        .build_cartesian_2d(0.0..max_freq_val, min_phase..max_phase)?;

    chart.configure_mesh()
        .x_desc("Frequency (Hz)")
        .y_desc("Phase (degrees)")
        .draw()?;

    chart.draw_series(LineSeries::new(
        frequencies.iter()
            .zip(phase_deg.iter())
            .filter(|(&f, _)| f <= max_freq_val)
            .map(|(&f, &p)| (f, p)),
        &RED,
    ))?;

    root.present()?;
    Ok(())
}

/// Plot combined magnitude responses of high-pass and low-pass filters
pub fn plot_combined_magnitude(
    frequencies: &[f64],
    hp_magnitude: &[f64],
    lp_magnitude: &[f64],
    output_path: &str,
    title: &str,
    max_freq: Option<f64>,
) -> Result<(), Box<dyn std::error::Error>> {
    let root = BitMapBackend::new(output_path, (PLOT_WIDTH,
        PLOT_HEIGHT)).into_drawing_area();
    root.fill(&WHITE)?;

    let max_freq_val = max_freq.unwrap_or(*frequencies.last().unwrap_or(&10000.0));

    let max_mag = hp_magnitude.iter()
        .chain(lp_magnitude.iter())
        .zip(frequencies.iter().cycle())
```

```
        .filter(|(_, &f)| f <= max_freq_val)
        .map(|(&m, _)| m)
        .fold(0.0, f64::max)
        .max(1.1);

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 30).into_font())
        .margin(15)
        .x_label_area_size(50)
        .y_label_area_size(60)
        .build_cartesian_2d(0.0..max_freq_val, 0.0..max_mag)?;

    chart.configure_mesh()
        .x_desc("Frequency (Hz)")
        .y_desc("Magnitude")
        .draw()?;

    // Draw high-pass filter
    chart.draw_series(LineSeries::new(
        frequencies.iter()
            .zip(hp_magnitude.iter())
            .filter(|(&f, _)| f <= max_freq_val)
            .map(|(&f, &m)| (f, m)),
        &BLUE,
    ))?.label("High-pass")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &BLUE));

    // Draw low-pass filter
    chart.draw_series(LineSeries::new(
        frequencies.iter()
            .zip(lp_magnitude.iter())
            .filter(|(&f, _)| f <= max_freq_val)
            .map(|(&f, &m)| (f, m)),
        &RED,
    ))?.label("Low-pass")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &RED));

    chart.configure_series_labels()
        .background_style(&WHITE.mix(0.8))
        .border_style(&BLACK)
        .draw()?;

    root.present()?;
    Ok(())
}
```

C.6.3 Q3 - 时域解调

主程序 (main.rs) Listing 9: Q3 主程序 - 时域同步解调流程

```
mod audio_reader;
mod iir_filter;
mod demodulator;
mod spectrum_analyzer;
```

```rust
mod audio_writer;

use std::f64::consts::PI;

fn main() {
    println!("Q3: Time-Domain Demodulation");
    println!("================================");

    // Step 1: Read Q1 results to get f_d, f_s, f_B
    println!("\n[Step 1] Reading Q1 results...");
    let (f_d, f_s, f_b) = match read_q1_results() {
        Ok(params) => params,
        Err(e) => {
            eprintln!("Error reading Q1 results: {}", e);
            return;
        }
    };
    println!("  f_d = {:.4} Hz", f_d);
    println!("  f_s = {:.4} Hz", f_s);
    println!("  f_B = {:.4} Hz", f_b);

    // Step 2: Read Q2 filter coefficients
    println!("\n[Step 2] Reading Q2 filter coefficients...");
    let (hp_b, hp_a, lp_b, lp_a) = match read_q2_filters() {
        Ok(filters) => filters,
        Err(e) => {
            eprintln!("Error reading Q2 filters: {}", e);
            return;
        }
    };
    println!("  High-pass filter: {} b coefficients, {} a coefficients", hp_b.len(),
        hp_a.len());
    println!("  Low-pass filter: {} b coefficients, {} a coefficients", lp_b.len(),
        lp_a.len());

    // Step 3: Read audio signal
    println!("\n[Step 3] Reading audio signal...");
    let audio_samples = match
        audio_reader::read_wav("../../工程设计问题-2022/工程设计题15.
        调幅信号的解调/project.wav") {
        Ok(samples) => samples,
        Err(e) => {
            eprintln!("Error reading audio: {}", e);
            return;
        }
    };
    println!("  Number of samples: {}", audio_samples.len());
    let max_orig = audio_samples.iter().fold(0.0f64, |max, &x| max.max(x.abs()));
    println!("  Signal max: {:.6}", max_orig);

    // Step 4: Apply high-pass filter
    println!("\n[Step 4] Applying high-pass filter...");
    let x_h = iir_filter::apply_filter(&audio_samples, &hp_b, &hp_a);
    println!("  Output samples: {}", x_h.len());
    let max_xh = x_h.iter().fold(0.0f64, |max, &x| max.max(x.abs()));
    println!("  Signal max: {:.6}", max_xh);
```

```rust

    // Step 5: Generate carrier and multiply
    println!("\n[Step 5] Multiplying with carrier signal (f_d = {:.4} Hz)...", f_d);
    let x_b = demodulator::multiply_with_carrier(&x_h, f_d, f_s);
    println!("  Output samples: {}", x_b.len());
    let max_xb = x_b.iter().fold(0.0f64, |max, &x| max.max(x.abs()));
    println!("  Signal max: {:.6}", max_xb);

    // Step 6: Apply low-pass filter
    println!("\n[Step 6] Applying low-pass filter...");
    let x_l = iir_filter::apply_filter(&x_b, &lp_b, &lp_a);
    println!("  Output samples: {}", x_l.len());

    // Debug: Check signal statistics
    let max_val = x_l.iter().fold(0.0f64, |max, &x| max.max(x.abs()));
    let mean_val = x_l.iter().sum::<f64>() / x_l.len() as f64;
    println!("  Signal range: max = {:.6}, mean = {:.6}", max_val, mean_val);

    // Step 7: Spectrum analysis
    println!("\n[Step 7] Performing spectrum analysis...");
    let original_spectrum = spectrum_analyzer::compute_spectrum(&audio_samples, f_s);
    let xh_spectrum = spectrum_analyzer::compute_spectrum(&x_h, f_s);
    let xb_spectrum = spectrum_analyzer::compute_spectrum(&x_b, f_s);
    let xl_spectrum = spectrum_analyzer::compute_spectrum(&x_l, f_s);

    // Step 8: Create output directory
    std::fs::create_dir_all("output").expect("Failed to create output directory");

    // Step 9: Plot spectra
    println!("\n[Step 8] Plotting spectra...");
    spectrum_analyzer::plot_spectrum(&original_spectrum,
        "output/Q3_original_spectrum.png", "Original Signal X(f)");
    spectrum_analyzer::plot_spectrum(&xh_spectrum, "output/Q3_xh_spectrum.png",
        "After High-Pass X_h(f)");
    spectrum_analyzer::plot_spectrum(&xb_spectrum, "output/Q3_xb_spectrum.png",
        "After Multiplication X_b(f)");
    spectrum_analyzer::plot_spectrum(&xl_spectrum, "output/Q3_xl_spectrum.png",
        "After Low-Pass X_l(f) - Demodulated");

    // Step 10: Save demodulated audio
    println!("\n[Step 9] Saving demodulated audio...");
    match audio_writer::write_wav("output/Q3_demodulated.wav", &x_l, f_s as u32) {
        Ok(_) => println!("  Saved to: output/Q3_demodulated.wav"),
        Err(e) => eprintln!("  Error saving audio: {}", e),
    }

    // Step 11: Save analysis results
    println!("\n[Step 10] Saving analysis results...");
    save_results(&original_spectrum, &xh_spectrum, &xb_spectrum, &xl_spectrum, f_d,
        f_s);

    println!("\nQ3 Time-Domain Demodulation completed successfully!");
    println!("Output files saved in: codes/Q3/output/");
}

fn read_q1_results() -> Result<(f64, f64, f64), String> {
```

```rust
    let content = std::fs::read_to_string("../Q1/output/Q1_results.txt")
        .map_err(|e| format!("Failed to read Q1 results: {}", e))?;

    let mut f_d = None;
    let mut f_s = None;

    for line in content.lines() {
        if line.contains("频率偏差") || line.contains("f_d") {
            if let Some(value_str) = line.split('=').nth(1) {
                if let Ok(value) =
                    value_str.trim().split_whitespace().next().unwrap_or("0").parse::<f64>()
                    {
                    f_d = Some(value);
                }
            }
        } else if line.contains("采样频率") || line.contains("f_s") {
            if let Some(value_str) = line.split('=').nth(1) {
                if let Ok(value) =
                    value_str.trim().split_whitespace().next().unwrap_or("0").parse::<f64>()
                    {
                    f_s = Some(value);
                }
            }
        }
    }

    let f_d = f_d.ok_or_else(|| "Could not find f_d in Q1 results".to_string())?;
    let f_s = f_s.ok_or_else(|| "Could not find f_s in Q1 results".to_string())?;
    let f_b = 4000.0; // Given in problem statement

    Ok((f_d, f_s, f_b))
}

fn read_q2_filters() -> Result<(Vec<f64>, Vec<f64>, Vec<f64>, Vec<f64>), String> {
    let content = std::fs::read_to_string("../Q2/output/Q2_filter_coefficients.txt")
        .map_err(|e| format!("Failed to read Q2 filters: {}", e))?;

    let mut hp_b = Vec::new();
    let mut hp_a = Vec::new();
    let mut lp_b = Vec::new();
    let mut lp_a = Vec::new();

    let mut current_section = "";

    for line in content.lines() {
        let line = line.trim();
        if line.is_empty() {
            continue;
        }

        if line.contains("High-pass Filter") {
            current_section = "hp";
        } else if line.contains("Low-pass Filter") {
            current_section = "lp";
        } else if line.starts_with("b[") {
            if let Some(value_str) = line.split('=').nth(1) {
```

```
                if let Ok(value) = value_str.trim().parse::<f64>() {
                    match current_section {
                        "hp" => hp_b.push(value),
                        "lp" => lp_b.push(value),
                        _ => {}
                    }
                }
            }
        } else if line.starts_with("a[") {
            if let Some(value_str) = line.split('=').nth(1) {
                if let Ok(value) = value_str.trim().parse::<f64>() {
                    match current_section {
                        "hp" => hp_a.push(value),
                        "lp" => lp_a.push(value),
                        _ => {}
                    }
                }
            }
        }
    }

    if hp_b.is_empty() || hp_a.is_empty() || lp_b.is_empty() || lp_a.is_empty() {
        return Err("Failed to parse filter coefficients".to_string());
    }

    Ok((hp_b, hp_a, lp_b, lp_a))
}

fn save_results(
    original: &[(f64, f64)],
    xh: &[(f64, f64)],
    xb: &[(f64, f64)],
    xl: &[(f64, f64)],
    f_d: f64,
    f_s: f64,
) {
    let mut content = String::new();
    content.push_str("Q3 Time-Domain Demodulation Results\n");
    content.push_str("====================================\n\n");
    content.push_str(&format!("Carrier frequency: f_d = {:.4} Hz\n", f_d));
    content.push_str(&format!("Sampling frequency: f_s = {:.4} Hz\n\n", f_s));

    // Spectral peaks for each stage
    content.push_str("Spectral Analysis:\n");
    content.push_str("------------------\n");

    // Original signal peak
    let orig_peak = original.iter()
        .filter(|(f, _)| *f > 1000.0)
        .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
        .unwrap();
    content.push_str(&format!("Original signal X(f) peak: f = {:.2} Hz, magnitude =
        {:.6}\n",
        orig_peak.0, orig_peak.1));

    // After high-pass
```

```
    let xh_peak = xh.iter()
        .filter(|(f, _)| *f > 1000.0)
        .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
        .unwrap();
    content.push_str(&format!("After high-pass X_h(f) peak: f = {:.2} Hz, magnitude = {:.6}\n",
        xh_peak.0, xh_peak.1));

    // After multiplication
    let xb_low_peak = xb.iter()
        .filter(|(f, _)| *f > 10.0 && *f < 5000.0)
        .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
        .unwrap();
    content.push_str(&format!("After multiplication X_b(f) peak (baseband): f = {:.2} Hz, magnitude = {:.6}\n",
        xb_low_peak.0, xb_low_peak.1));

    // Demodulated signal peak
    let xl_peak = xl.iter()
        .filter(|(f, _)| *f > 10.0 && *f < 4000.0)
        .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
        .unwrap();
    content.push_str(&format!("Demodulated signal X_l(f) peak: f = {:.2} Hz, magnitude = {:.6}\n",
        xl_peak.0, xl_peak.1));

    // Energy in baseband (0-4000 Hz)
    let energy_orig_baseband: f64 = original.iter()
        .filter(|(f, _)| *f < 4000.0)
        .map(|(_, m)| m * m)
        .sum();
    let energy_demod_baseband: f64 = xl.iter()
        .filter(|(f, _)| *f < 4000.0)
        .map(|(_, m)| m * m)
        .sum();

    content.push_str(&format!("\nEnergy analysis (0-4000 Hz band):\n"));
    content.push_str(&format!("  Original signal energy: {:.6e}\n",
        energy_orig_baseband));
    content.push_str(&format!("  Demodulated signal energy: {:.6e}\n",
        energy_demod_baseband));

    // Frequency shift verification
    content.push_str(&format!("\nFrequency shift verification:\n"));
    content.push_str(&format!("  Original peak at: {:.2} Hz\n", orig_peak.0));
    content.push_str(&format!("  Expected shift: {:.2} Hz (should be near f_d = {:.2} Hz)\n",
        orig_peak.0 - f_d, f_d));
    content.push_str(&format!("  Demodulated peak at: {:.2} Hz (should be in baseband)\n", xl_peak.0));

    std::fs::write("output/Q3_results.txt", content).expect("Failed to save results");
    println!("  Saved to: output/Q3_results.txt");
}
```

Listing 10: 音频文件读取模块

音频读取模块 (audio_reader.rs)

```rust
use hound;

pub fn read_wav(filename: &str) -> Result<Vec<f64>, String> {
    let reader = hound::WavReader::open(filename)
        .map_err(|e| format!("Failed to open WAV file: {}", e))?;

    let spec = reader.spec();
    println!("  Sample rate: {} Hz", spec.sample_rate);
    println!("  Channels: {}", spec.channels);
    println!("  Bits per sample: {}", spec.bits_per_sample);

    // Read all samples and normalize to [-1.0, 1.0]
    let samples: Vec<f64> = match spec.bits_per_sample {
        16 => {
            reader
                .into_samples::<i16>()
                .map(|s| s.unwrap() as f64 / 32768.0)
                .collect()
        }
        _ => {
            return Err(format!(
                "Unsupported bits per sample: {}",
                spec.bits_per_sample
            ))
        }
    };

    Ok(samples)
}
```

Listing 11: Direct Form II IIR 滤波器实现

IIR 滤波器模块 (iir_filter.rs)

```rust
/// Apply IIR filter using Direct Form II structure
/// y[n] = sum(b[i] * x[n-i]) - sum(a[j] * y[n-j]) for j > 0
pub fn apply_filter(input: &[f64], b: &[f64], a: &[f64]) -> Vec<f64> {
    let n = input.len();
    let mut output = vec![0.0; n];

    let order = b.len().max(a.len());
    let mut x_history = vec![0.0; order]; // Input history
    let mut y_history = vec![0.0; order]; // Output history

    for i in 0..n {
        // Shift histories
        for j in (1..order).rev() {
            x_history[j] = x_history[j - 1];
            y_history[j] = y_history[j - 1];
        }
        x_history[0] = input[i];

        // Calculate output using difference equation
        let mut y = 0.0;

```

```
        // Feedforward part: sum(b[k] * x[n-k])
        for k in 0..b.len() {
            y += b[k] * x_history[k];
        }

        // Feedback part: -sum(a[k] * y[n-k]) for k > 0
        for k in 1..a.len() {
            y -= a[k] * y_history[k];
        }

        // Normalize by a[0] (usually 1.0)
        y /= a[0];

        y_history[0] = y;
        output[i] = y;
    }

    output
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_simple_filter() {
        // Simple moving average filter: y[n] = 0.5*x[n] + 0.5*x[n-1]
        let b = vec![0.5, 0.5];
        let a = vec![1.0];
        let input = vec![1.0, 2.0, 3.0, 4.0, 5.0];

        let output = apply_filter(&input, &b, &a);

        // Expected: [0.5, 1.5, 2.5, 3.5, 4.5]
        assert!((output[0] - 0.5).abs() < 1e-10);
        assert!((output[1] - 1.5).abs() < 1e-10);
        assert!((output[2] - 2.5).abs() < 1e-10);
    }
}
```

Listing 12: 时域同步解调 - 载波相乘法

解调器模块 (demodulator.rs)

```
use std::f64::consts::PI;

/// Multiply signal with carrier cos(2*pi*f_d*t)
pub fn multiply_with_carrier(signal: &[f64], f_d: f64, f_s: f64) -> Vec<f64> {
    let n = signal.len();
    let mut output = vec![0.0; n];

    for i in 0..n {
        let t = i as f64 / f_s;
        let carrier = (2.0 * PI * f_d * t).cos();
        // Multiply by 2 to compensate for the 1/2 factor from cos²(x) = (1 +
            cos(2x))/2
        output[i] = signal[i] * carrier * 2.0;
```

```
    }

    output
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_carrier_multiplication() {
        let f_d = 1000.0;
        let f_s = 8000.0;
        let signal = vec![1.0; 100];

        let output = multiply_with_carrier(&signal, f_d, f_s);

        // Output should oscillate with carrier frequency
        assert_eq!(output.len(), 100);
        // At t=0, cos(0) = 1.0
        assert!((output[0] - 1.0).abs() < 1e-10);
    }
}
```

Listing 13: 频谱分析 - FFT 与峰值检测

频谱分析模块 (spectrum_analyzer.rs)

```
use rustfft::{FftPlanner, num_complex::Complex};
use plotters::prelude::*;

/// Compute magnitude spectrum of a signal
pub fn compute_spectrum(signal: &[f64], f_s: f64) -> Vec<(f64, f64)> {
    let n = signal.len();

    // Prepare FFT input
    let mut buffer: Vec<Complex<f64>> = signal
        .iter()
        .map(|&x| Complex::new(x, 0.0))
        .collect();

    // Perform FFT
    let mut planner = FftPlanner::new();
    let fft = planner.plan_fft_forward(n);
    fft.process(&mut buffer);

    // Compute magnitude spectrum and frequency axis
    let mut spectrum = Vec::with_capacity(n / 2);
    let df = f_s / n as f64;

    for i in 0..n / 2 {
        let freq = i as f64 * df;
        let magnitude = buffer[i].norm() / n as f64;
        spectrum.push((freq, magnitude));
    }

    spectrum
```

```rust
}

/// Plot spectrum
pub fn plot_spectrum(spectrum: &[(f64, f64)], filename: &str, title: &str) {
    let root = BitMapBackend::new(filename, (1200, 800)).into_drawing_area();
    root.fill(&WHITE).unwrap();

    // Find max magnitude for y-axis
    let max_mag = spectrum.iter()
        .map(|(_, m)| *m)
        .fold(0.0f64, f64::max);

    let max_freq = spectrum.last().unwrap().0;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 40))
        .margin(20)
        .x_label_area_size(50)
        .y_label_area_size(60)
        .build_cartesian_2d(0.0..max_freq, 0.0..max_mag * 1.1)
        .unwrap();

    chart
        .configure_mesh()
        .x_desc("Frequency (Hz)")
        .y_desc("Magnitude")
        .x_label_formatter(&|x| format!("{:.0}", x))
        .y_label_formatter(&|y| format!("{:.3}", y))
        .draw()
        .unwrap();

    chart
        .draw_series(LineSeries::new(
            spectrum.iter().map(|(f, m)| (*f, *m)),
            &BLUE,
        ))
        .unwrap();

    root.present().unwrap();
    println!("  Saved: {}", filename);
}

/// Plot spectrum in dB scale
pub fn plot_spectrum_db(spectrum: &[(f64, f64)], filename: &str, title: &str) {
    let root = BitMapBackend::new(filename, (1200, 800)).into_drawing_area();
    root.fill(&WHITE).unwrap();

    // Convert to dB
    let spectrum_db: Vec<(f64, f64)> = spectrum
        .iter()
        .map(|(f, m)| {
            let db = if *m > 1e-10 {
                20.0 * m.log10()
            } else {
                -200.0
            };
```

```
            (*f, db)
        })
        .collect();

    let max_freq = spectrum_db.last().unwrap().0;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 40))
        .margin(20)
        .x_label_area_size(50)
        .y_label_area_size(60)
        .build_cartesian_2d(0.0..max_freq, -100.0..0.0)
        .unwrap();

    chart
        .configure_mesh()
        .x_desc("Frequency (Hz)")
        .y_desc("Magnitude (dB)")
        .x_label_formatter(&|x| format!("{:.0}", x))
        .y_label_formatter(&|y| format!("{:.0}", y))
        .draw()
        .unwrap();

    chart
        .draw_series(LineSeries::new(
            spectrum_db.iter().map(|(f, m)| (*f, *m)),
            &RED,
        ))
        .unwrap();

    root.present().unwrap();
    println!("  Saved: {}", filename);
}
```

Listing 14: WAV 文件写入 - 16bit PCM 格式

音频写入模块 (audio_writer.rs)

```
use hound;

pub fn write_wav(filename: &str, samples: &[f64], sample_rate: u32) -> Result<(), String> {
    let spec = hound::WavSpec {
        channels: 1,
        sample_rate,
        bits_per_sample: 16,
        sample_format: hound::SampleFormat::Int,
    };

    let mut writer = hound::WavWriter::create(filename, spec)
        .map_err(|e| format!("Failed to create WAV file: {}", e))?;

    // Normalize samples to prevent clipping
    let max_val = samples.iter().fold(0.0f64, |max, &x| max.max(x.abs()));
    let scale = if max_val > 0.0 {
        0.95 / max_val // Leave some headroom
    } else {
```

```
        1.0
    };

    for &sample in samples {
        let normalized = sample * scale;
        let sample_i16 = (normalized * 32767.0).clamp(-32768.0, 32767.0) as i16;
        writer
            .write_sample(sample_i16)
            .map_err(|e| format!("Failed to write sample: {}", e))?;
    }

    writer
        .finalize()
        .map_err(|e| format!("Failed to finalize WAV file: {}", e))?;

    Ok(())
}
```

C.6.4 Q4 - 频域解调

主程序 (main.rs) Listing 15: Q4 主程序 - 频域解调完整流程

```
mod audio_reader;
mod ideal_filter;
mod frequency_shifter;
mod spectrum_analyzer;
mod audio_writer;
mod comparator;

use num_complex::Complex;

fn main() {
    println!("Q4: Frequency-Domain Demodulation");
    println!("==================================");

    // Step 1: Read Q1 results to get f_d, f_s, f_B
    println!("\n[Step 1] Reading Q1 results...");
    let (f_d, f_s, f_b) = match read_q1_results() {
        Ok(params) => params,
        Err(e) => {
            eprintln!("Error reading Q1 results: {}", e);
            return;
        }
    };
    println!("  f_d = {:.4} Hz", f_d);
    println!("  f_s = {:.4} Hz", f_s);
    println!("  f_B = {:.4} Hz", f_b);

    // Step 2: Read audio signal
    println!("\n[Step 2] Reading audio signal...");
    let audio_samples = match
        audio_reader::read_wav("../../工程设计问题-2022/工程设计题15.
        调幅信号的解调/project.wav") {
        Ok(samples) => samples,
```

```
        Err(e) => {
            eprintln!("Error reading audio: {}", e);
            return;
        }
    };
    let n = audio_samples.len();
    println!("  Number of samples: {}", n);

    // Step 3: Compute FFT of input signal
    println!("\n[Step 3] Computing FFT of input signal...");
    let x_fft = compute_fft(&audio_samples);
    println!("  FFT size: {}", x_fft.len());

    // Step 4: Apply ideal high-pass filter in frequency domain
    println!("\n[Step 4] Applying ideal high-pass filter (fc = {:.4} Hz)...", f_d);
    let xh_fft = ideal_filter::apply_highpass(&x_fft, f_d, f_s, n);
    println!("  High-pass filtering complete");

    // Step 5: Frequency shift (equivalent to carrier multiplication in time domain)
    println!("\n[Step 5] Performing frequency shift (±{:.4} Hz)...", f_d);
    let xb_fft = frequency_shifter::frequency_shift(&xh_fft, f_d, f_s, n);
    println!("  Frequency shift complete");

    // Step 6: Apply ideal low-pass filter
    println!("\n[Step 6] Applying ideal low-pass filter (fc = {:.4} Hz)...", f_b);
    let xl_fft = ideal_filter::apply_lowpass(&xb_fft, f_b, f_s, n);
    println!("  Low-pass filtering complete");

    // Step 7: Inverse FFT to get time-domain signal
    println!("\n[Step 7] Computing IFFT to recover time-domain signal...");
    let mut xl_samples = compute_ifft(&xl_fft);
    println!("  Output samples: {}", xl_samples.len());

    // Apply gain compensation (multiply by 2 to match time-domain method)
    for sample in xl_samples.iter_mut() {
        *sample *= 2.0;
    }

    let max_val = xl_samples.iter().fold(0.0f64, |max, &x| max.max(x.abs()));
    println!("  Signal max: {:.6}", max_val);

    // Step 8: Create output directory
    std::fs::create_dir_all("output").expect("Failed to create output directory");

    // Step 9: Spectrum analysis for each stage
    println!("\n[Step 8] Performing spectrum analysis...");
    let original_spectrum = compute_magnitude_spectrum(&x_fft, f_s);
    let xh_spectrum = compute_magnitude_spectrum(&xh_fft, f_s);
    let xb_spectrum = compute_magnitude_spectrum(&xb_fft, f_s);
    let xl_spectrum = compute_magnitude_spectrum(&xl_fft, f_s);

    // Step 10: Plot spectra
    println!("\n[Step 9] Plotting spectra...");
    spectrum_analyzer::plot_spectrum(&original_spectrum,
        "output/Q4_original_spectrum.png", "Original Signal X(f)");
    spectrum_analyzer::plot_spectrum(&xh_spectrum, "output/Q4_xh_spectrum.png",
```

```
        "After Ideal High-Pass X_h(f)");
    spectrum_analyzer::plot_spectrum(&xb_spectrum, "output/Q4_xb_spectrum.png",
        "After Frequency Shift X_b(f)");
    spectrum_analyzer::plot_spectrum(&xl_spectrum, "output/Q4_xl_spectrum.png",
        "After Ideal Low-Pass X_l(f) - Demodulated");

    // Step 11: Save demodulated audio
    println!("\n[Step 10] Saving demodulated audio...");
    match audio_writer::write_wav("output/Q4_demodulated.wav", &xl_samples, f_s as
        u32) {
        Ok(_) => println!("  Saved to: output/Q4_demodulated.wav"),
        Err(e) => eprintln!("  Error saving audio: {}", e),
    }

    // Step 12: Compare with Q3 results
    println!("\n[Step 11] Comparing with Q3 results...");
    if let Ok(q3_samples) = audio_reader::read_wav("../Q3/output/Q3_demodulated.wav")
        {
        let comparison = comparator::compare_signals(&xl_samples, &q3_samples);
        println!("  Q3 vs Q4 comparison:");
        println!("    MSE: {:.6e}", comparison.mse);
        println!("    Max difference: {:.6}", comparison.max_diff);
        println!("    Correlation (original): {:.6}", comparison.correlation);
        println!("    Correlation (normalized): {:.6}",
            comparison.correlation_normalized);

        // Save comparison results
        comparator::save_comparison(&comparison, "output/Q4_comparison.txt");

        // Plot full-time comparison (all samples)
        comparator::plot_full_comparison(&xl_samples, &q3_samples,
            "output/Q4_vs_Q3_full_comparison.png");

        // Plot detailed comparison (first 2000 samples)
        comparator::plot_comparison(&xl_samples, &q3_samples,
            "output/Q4_vs_Q3_comparison.png");
    } else {
        println!("  Warning: Could not read Q3 results for comparison");
    }

    // Step 13: Save analysis results
    println!("\n[Step 12] Saving analysis results...");
    save_results(&original_spectrum, &xh_spectrum, &xb_spectrum, &xl_spectrum, f_d,
        f_s, f_b);

    println!("\nQ4 Frequency-Domain Demodulation completed successfully!");
    println!("Output files saved in: codes/Q4/output/");
}

fn read_q1_results() -> Result<(f64, f64, f64), String> {
    let content = std::fs::read_to_string("../Q1/output/Q1_results.txt")
        .map_err(|e| format!("Failed to read Q1 results: {}", e))?;

    let mut f_d = None;
    let mut f_s = None;

```

```rust
    for line in content.lines() {
        if line.contains("频率偏差") || line.contains("f_d") {
            if let Some(value_str) = line.split('=').nth(1) {
                if let Ok(value) =
                    value_str.trim().split_whitespace().next().unwrap_or("0").parse::<f64>()
                    {
                    f_d = Some(value);
                }
            }
        } else if line.contains("采样频率") || line.contains("f_s") {
            if let Some(value_str) = line.split('=').nth(1) {
                if let Ok(value) =
                    value_str.trim().split_whitespace().next().unwrap_or("0").parse::<f64>()
                    {
                    f_s = Some(value);
                }
            }
        }
    }

    let f_d = f_d.ok_or_else(|| "Could not find f_d in Q1 results".to_string())?;
    let f_s = f_s.ok_or_else(|| "Could not find f_s in Q1 results".to_string())?;
    let f_b = 4000.0; // Given in problem statement

    Ok((f_d, f_s, f_b))
}

fn compute_fft(samples: &[f64]) -> Vec<Complex<f64>> {
    use rustfft::FftPlanner;

    let mut buffer: Vec<Complex<f64>> = samples
        .iter()
        .map(|&x| Complex::new(x, 0.0))
        .collect();

    let mut planner = FftPlanner::new();
    let fft = planner.plan_fft_forward(buffer.len());
    fft.process(&mut buffer);

    buffer
}

fn compute_ifft(spectrum: &[Complex<f64>]) -> Vec<f64> {
    use rustfft::FftPlanner;

    let mut buffer = spectrum.to_vec();

    let mut planner = FftPlanner::new();
    let ifft = planner.plan_fft_inverse(buffer.len());
    ifft.process(&mut buffer);

    // Normalize and extract real part
    let n = buffer.len() as f64;
    buffer.iter().map(|c| c.re / n).collect()
}

```

```rust
fn compute_magnitude_spectrum(spectrum: &[Complex<f64>], f_s: f64) -> Vec<(f64, f64)> {
    let n = spectrum.len();
    let df = f_s / n as f64;

    (0..n/2)
        .map(|i| {
            let freq = i as f64 * df;
            let magnitude = spectrum[i].norm() / n as f64;
            (freq, magnitude)
        })
        .collect()
}

fn save_results(
    original: &[(f64, f64)],
    xh: &[(f64, f64)],
    xb: &[(f64, f64)],
    xl: &[(f64, f64)],
    f_d: f64,
    f_s: f64,
    f_b: f64,
) {
    let mut content = String::new();
    content.push_str("Q4 Frequency-Domain Demodulation Results\n");
    content.push_str("=========================================\n\n");
    content.push_str(&format!("Carrier frequency: f_d = {:.4} Hz\n", f_d));
    content.push_str(&format!("Sampling frequency: f_s = {:.4} Hz\n", f_s));
    content.push_str(&format!("Baseband bandwidth: f_B = {:.4} Hz\n\n", f_b));

    // Spectral peaks for each stage
    content.push_str("Spectral Analysis:\n");
    content.push_str("------------------\n");

    // Original signal peak
    let orig_peak = original.iter()
        .filter(|(f, _)| *f > 1000.0)
        .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
        .unwrap();
    content.push_str(&format!("Original signal X(f) peak: f = {:.2} Hz, magnitude = {:.6}\n",
        orig_peak.0, orig_peak.1));

    // After high-pass
    let xh_peak = xh.iter()
        .filter(|(f, _)| *f > f_d)
        .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
        .unwrap();
    content.push_str(&format!("After ideal high-pass X_h(f) peak: f = {:.2} Hz, magnitude = {:.6}\n",
        xh_peak.0, xh_peak.1));

    // After frequency shift
    let xb_peak = xb.iter()
        .filter(|(f, _)| *f > 10.0 && *f < 5000.0)
        .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
```

```
        .unwrap();
    content.push_str(&format!("After frequency shift X_b(f) peak: f = {:.2} Hz,
        magnitude = {:.6}\n",
        xb_peak.0, xb_peak.1));

    // Demodulated signal peak
    let xl_peak = xl.iter()
        .filter(|(f, _)| *f > 10.0 && *f < f_b)
        .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
        .unwrap();
    content.push_str(&format!("Demodulated signal X_l(f) peak: f = {:.2} Hz,
        magnitude = {:.6}\n",
        xl_peak.0, xl_peak.1));

    // Energy in baseband
    let energy_orig: f64 = original.iter()
        .filter(|(f, _)| *f < f_b)
        .map(|(_, m)| m * m)
        .sum();
    let energy_demod: f64 = xl.iter()
        .filter(|(f, _)| *f < f_b)
        .map(|(_, m)| m * m)
        .sum();

    content.push_str(&format!("\nEnergy analysis (0-{:.0} Hz band):\n", f_b));
    content.push_str(&format!("  Original signal energy: {:.6e}\n", energy_orig));
    content.push_str(&format!("  Demodulated signal energy: {:.6e}\n", energy_demod));

    content.push_str(&format!("\nMethod characteristics:\n"));
    content.push_str("  - Uses ideal filters (brick-wall response)\n");
    content.push_str("  - Frequency-domain processing (no time-domain
        convolution)\n");
    content.push_str("  - Perfect frequency selectivity\n");
    content.push_str("  - No phase distortion from filters\n");

    std::fs::write("output/Q4_results.txt", content).expect("Failed to save results");
    println!("  Saved to: output/Q4_results.txt");
}
```

Listing 16: 音频文件读取模块

音频读取模块 (audio_reader.rs)

```
use hound;

pub fn read_wav(filename: &str) -> Result<Vec<f64>, String> {
    let reader = hound::WavReader::open(filename)
        .map_err(|e| format!("Failed to open WAV file: {}", e))?;

    let spec = reader.spec();
    println!("  Sample rate: {} Hz", spec.sample_rate);
    println!("  Channels: {}", spec.channels);
    println!("  Bits per sample: {}", spec.bits_per_sample);

    // Read all samples and normalize to [-1.0, 1.0]
    let samples: Vec<f64> = match spec.bits_per_sample {
        16 => {
```

```
            reader
                .into_samples::<i16>()
                .map(|s| s.unwrap() as f64 / 32768.0)
                .collect()
        }
        _ => {
            return Err(format!(
                "Unsupported bits per sample: {}",
                spec.bits_per_sample
            ))
        }
    };

    Ok(samples)
}
```

Listing 17: 理想高通/低通滤波器 - 频域矩形窗

理想滤波器模块 (ideal_filter.rs)

```rust
use num_complex::Complex;

/// Apply ideal high-pass filter in frequency domain
/// H_h(f) = 0 for |f| < f_c, 1 for |f| >= f_c
pub fn apply_highpass(
    spectrum: &[Complex<f64>],
    f_c: f64,
    f_s: f64,
    n: usize,
) -> Vec<Complex<f64>> {
    let df = f_s / n as f64;
    let mut result = spectrum.to_vec();

    for i in 0..n {
        // Calculate frequency for this bin
        let freq = if i <= n / 2 {
            i as f64 * df
        } else {
            (i as f64 - n as f64) * df
        };

        // Apply ideal high-pass filter
        if freq.abs() < f_c {
            result[i] = Complex::new(0.0, 0.0);
        }
    }

    result
}

/// Apply ideal low-pass filter in frequency domain
/// H_l(f) = 1 for |f| <= f_c, 0 for |f| > f_c
pub fn apply_lowpass(
    spectrum: &[Complex<f64>],
    f_c: f64,
    f_s: f64,
    n: usize,
```

```
) -> Vec<Complex<f64>> {
    let df = f_s / n as f64;
    let mut result = spectrum.to_vec();

    for i in 0..n {
        // Calculate frequency for this bin
        let freq = if i <= n / 2 {
            i as f64 * df
        } else {
            (i as f64 - n as f64) * df
        };

        // Apply ideal low-pass filter
        if freq.abs() > f_c {
            result[i] = Complex::new(0.0, 0.0);
        }
    }

    result
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_highpass_filter() {
        let n = 100;
        let f_s = 1000.0;
        let f_c = 100.0;

        // Create test spectrum
        let spectrum: Vec<Complex<f64>> = (0..n)
            .map(|_| Complex::new(1.0, 0.0))
            .collect();

        let filtered = apply_highpass(&spectrum, f_c, f_s, n);

        // Check that low frequencies are zeroed
        assert_eq!(filtered[0].norm(), 0.0); // DC component
        assert_eq!(filtered[5].norm(), 0.0); // 50 Hz
    }

    #[test]
    fn test_lowpass_filter() {
        let n = 100;
        let f_s = 1000.0;
        let f_c = 100.0;

        let spectrum: Vec<Complex<f64>> = (0..n)
            .map(|_| Complex::new(1.0, 0.0))
            .collect();

        let filtered = apply_lowpass(&spectrum, f_c, f_s, n);

        // Check that high frequencies are zeroed
```

```
        let high_freq_idx = (200.0 / f_s * n as f64) as usize;
        assert_eq!(filtered[high_freq_idx].norm(), 0.0);
    }
}
```

Listing 18: 循环移位频率搬移 - 频谱平移

频率搬移模块 (frequency_shifter.rs)

```rust
use num_complex::Complex;

/// Perform frequency shift: X_b(f) = X_h(f - f_d) + X_h(f + f_d)
/// This is equivalent to multiplying by cos(2 f_d · t) in time domain
pub fn frequency_shift(
    spectrum: &[Complex<f64>],
    f_d: f64,
    f_s: f64,
    n: usize,
) -> Vec<Complex<f64>> {
    // Calculate shift amount in bins
    let shift_bins = (f_d * n as f64 / f_s).round() as isize;

    // Create result vector
    let mut result = vec![Complex::new(0.0, 0.0); n];

    // For each output frequency bin, find contributions from shifted input
    for i in 0..n {
        // Contribution from f - f_d (shift down)
        let idx_minus = (i as isize + shift_bins).rem_euclid(n as isize) as usize;
        result[i] = result[i] + spectrum[idx_minus] * 0.5;

        // Contribution from f + f_d (shift up)
        let idx_plus = (i as isize - shift_bins).rem_euclid(n as isize) as usize;
        result[i] = result[i] + spectrum[idx_plus] * 0.5;
    }

    result
}

/// Alternative implementation using explicit frequency mapping
pub fn frequency_shift_explicit(
    spectrum: &[Complex<f64>],
    f_d: f64,
    f_s: f64,
    n: usize,
) -> Vec<Complex<f64>> {
    let df = f_s / n as f64;
    let mut result = vec![Complex::new(0.0, 0.0); n];

    for i in 0..n {
        // Current frequency
        let f_i = if i <= n / 2 {
            i as f64 * df
        } else {
            (i as f64 - n as f64) * df
        };

```

```
        // Find indices for f ± f_d
        let f_minus = f_i - f_d;
        let f_plus = f_i + f_d;

        // Map back to bin indices
        let idx_minus = freq_to_bin(f_minus, f_s, n);
        let idx_plus = freq_to_bin(f_plus, f_s, n);

        // Accumulate contributions
        if idx_minus < n {
            result[i] = result[i] + spectrum[idx_minus] * 0.5;
        }
        if idx_plus < n {
            result[i] = result[i] + spectrum[idx_plus] * 0.5;
        }
    }

    result
}

fn freq_to_bin(freq: f64, f_s: f64, n: usize) -> usize {
    let normalized = if freq >= 0.0 {
        freq
    } else {
        freq + f_s
    };

    let bin = (normalized * n as f64 / f_s).round() as isize;
    bin.rem_euclid(n as isize) as usize
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_frequency_shift() {
        let n = 1000;
        let f_s = 10000.0;
        let f_d = 1000.0;

        // Create impulse at 2000 Hz
        let mut spectrum = vec![Complex::new(0.0, 0.0); n];
        let impulse_idx = (2000.0 / f_s * n as f64) as usize;
        spectrum[impulse_idx] = Complex::new(1.0, 0.0);

        let shifted = frequency_shift(&spectrum, f_d, f_s, n);

        // Should have peaks at 1000 Hz and 3000 Hz
        let idx_1000 = (1000.0 / f_s * n as f64) as usize;
        let idx_3000 = (3000.0 / f_s * n as f64) as usize;

        assert!(shifted[idx_1000].norm() > 0.4);
        assert!(shifted[idx_3000].norm() > 0.4);
    }

```

```
    #[test]
    fn test_freq_to_bin() {
        let n = 100;
        let f_s = 1000.0;

        assert_eq!(freq_to_bin(0.0, f_s, n), 0);
        assert_eq!(freq_to_bin(100.0, f_s, n), 10);
        assert_eq!(freq_to_bin(-100.0, f_s, n), 90);
    }
}
```

Listing 19: 频谱分析模块

频谱分析模块 (spectrum_analyzer.rs)

```rust
use plotters::prelude::*;

/// Plot spectrum
pub fn plot_spectrum(spectrum: &[(f64, f64)], filename: &str, title: &str) {
    let root = BitMapBackend::new(filename, (1200, 800)).into_drawing_area();
    root.fill(&WHITE).unwrap();

    // Find max magnitude for y-axis
    let max_mag = spectrum.iter()
        .map(|(_, m)| *m)
        .fold(0.0f64, f64::max);

    let max_freq = spectrum.last().unwrap().0;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 40))
        .margin(20)
        .x_label_area_size(50)
        .y_label_area_size(60)
        .build_cartesian_2d(0.0..max_freq, 0.0..max_mag * 1.1)
        .unwrap();

    chart
        .configure_mesh()
        .x_desc("Frequency (Hz)")
        .y_desc("Magnitude")
        .x_label_formatter(&|x| format!("{:.0}", x))
        .y_label_formatter(&|y| format!("{:.3}", y))
        .draw()
        .unwrap();

    chart
        .draw_series(LineSeries::new(
            spectrum.iter().map(|(f, m)| (*f, *m)),
            &BLUE,
        ))
        .unwrap();

    root.present().unwrap();
    println!("  Saved: {}", filename);
}
```

Listing 20: WAV 文件写入模块

音频写入模块 (audio_writer.rs)

```rust
use hound;

pub fn write_wav(filename: &str, samples: &[f64], sample_rate: u32) -> Result<(), String> {
    let spec = hound::WavSpec {
        channels: 1,
        sample_rate,
        bits_per_sample: 16,
        sample_format: hound::SampleFormat::Int,
    };

    let mut writer = hound::WavWriter::create(filename, spec)
        .map_err(|e| format!("Failed to create WAV file: {}", e))?;

    // Normalize samples to prevent clipping
    let max_val = samples.iter().fold(0.0f64, |max, &x| max.max(x.abs()));
    let scale = if max_val > 0.0 {
        0.95 / max_val // Leave some headroom
    } else {
        1.0
    };

    for &sample in samples {
        let normalized = sample * scale;
        let sample_i16 = (normalized * 32767.0).clamp(-32768.0, 32767.0) as i16;
        writer
            .write_sample(sample_i16)
            .map_err(|e| format!("Failed to write sample: {}", e))?;
    }

    writer
        .finalize()
        .map_err(|e| format!("Failed to finalize WAV file: {}", e))?;

    Ok(())
}
```

Listing 21: Q3 与 Q4 结果对比 - 性能分析

对比分析模块 (comparator.rs)

```rust
use plotters::prelude::*;

pub struct ComparisonResult {
    pub mse: f64,
    pub max_diff: f64,
    pub correlation: f64,
    pub correlation_normalized: f64,
    pub snr_db: f64,
}

/// Compare two signals
pub fn compare_signals(signal1: &[f64], signal2: &[f64]) -> ComparisonResult {
    let n = signal1.len().min(signal2.len());

```

```
    // Mean Squared Error
    let mse: f64 = (0..n)
        .map(|i| {
            let diff = signal1[i] - signal2[i];
            diff * diff
        })
        .sum::<f64>() / n as f64;

    // Maximum absolute difference
    let max_diff = (0..n)
        .map(|i| (signal1[i] - signal2[i]).abs())
        .fold(0.0f64, f64::max);

    // Correlation coefficient (original)
    let mean1 = signal1[..n].iter().sum::<f64>() / n as f64;
    let mean2 = signal2[..n].iter().sum::<f64>() / n as f64;

    let cov: f64 = (0..n)
        .map(|i| (signal1[i] - mean1) * (signal2[i] - mean2))
        .sum::<f64>() / n as f64;

    let var1: f64 = (0..n)
        .map(|i| (signal1[i] - mean1).powi(2))
        .sum::<f64>() / n as f64;

    let var2: f64 = (0..n)
        .map(|i| (signal2[i] - mean2).powi(2))
        .sum::<f64>() / n as f64;

    let correlation = if var1 > 0.0 && var2 > 0.0 {
        cov / (var1.sqrt() * var2.sqrt())
    } else {
        0.0
    };

    // Correlation coefficient with amplitude normalization
    // Normalize both signals to [-1, 1] range based on their max absolute value
    let max_abs1 = signal1[..n].iter().map(|&x| x.abs()).fold(0.0f64, f64::max);
    let max_abs2 = signal2[..n].iter().map(|&x| x.abs()).fold(0.0f64, f64::max);

    let correlation_normalized = if max_abs1 > 0.0 && max_abs2 > 0.0 {
        let norm1: Vec<f64> = signal1[..n].iter().map(|&x| x / max_abs1).collect();
        let norm2: Vec<f64> = signal2[..n].iter().map(|&x| x / max_abs2).collect();

        let mean_norm1 = norm1.iter().sum::<f64>() / n as f64;
        let mean_norm2 = norm2.iter().sum::<f64>() / n as f64;

        let cov_norm: f64 = (0..n)
            .map(|i| (norm1[i] - mean_norm1) * (norm2[i] - mean_norm2))
            .sum::<f64>() / n as f64;

        let var_norm1: f64 = (0..n)
            .map(|i| (norm1[i] - mean_norm1).powi(2))
            .sum::<f64>() / n as f64;

        let var_norm2: f64 = (0..n)
```

```
            .map(|i| (norm2[i] - mean_norm2).powi(2))
            .sum::<f64>() / n as f64;

        if var_norm1 > 0.0 && var_norm2 > 0.0 {
            cov_norm / (var_norm1.sqrt() * var_norm2.sqrt())
        } else {
            0.0
        }
    } else {
        0.0
    };

    // Signal-to-Noise Ratio (treating difference as noise)
    let signal_power: f64 = signal1[..n].iter().map(|&x| x * x).sum::<f64>() / n as
        f64;
    let noise_power = mse;

    let snr_db = if noise_power > 0.0 {
        10.0 * (signal_power / noise_power).log10()
    } else {
        f64::INFINITY
    };

    ComparisonResult {
        mse,
        max_diff,
        correlation,
        correlation_normalized,
        snr_db,
    }
}

/// Save comparison results to file
pub fn save_comparison(result: &ComparisonResult, filename: &str) {
    let mut content = String::new();
    content.push_str("Q4 vs Q3 Comparison Results\n");
    content.push_str("============================\n\n");
    content.push_str(&format!("Mean Squared Error (MSE): {:.6e}\n", result.mse));
    content.push_str(&format!("Root Mean Squared Error (RMSE): {:.6e}\n",
        result.mse.sqrt()));
    content.push_str(&format!("Maximum absolute difference: {:.6}\n",
        result.max_diff));
    content.push_str(&format!("Correlation coefficient (original): {:.6}\n",
        result.correlation));
    content.push_str(&format!("Correlation coefficient (normalized): {:.6}\n",
        result.correlation_normalized));
    content.push_str(&format!("Signal-to-Noise Ratio: {:.2} dB\n\n", result.snr_db));

    content.push_str("Interpretation:\n");
    content.push_str("---------------\n");

    // Use normalized correlation for interpretation (more accurate for waveform
        similarity)
    if result.correlation_normalized > 0.99 {
        content.push_str("  Excellent correlation (normalized) - waveforms are nearly
            identical\n");
```

```
    } else if result.correlation_normalized > 0.95 {
        content.push_str("  Good correlation (normalized) - waveforms are very
            similar\n");
    } else if result.correlation_normalized > 0.8 {
        content.push_str("~ Moderate correlation (normalized) - some waveform
            differences\n");
    } else {
        content.push_str("  Low correlation (normalized) - significant waveform
            differences\n");
    }

    content.push_str(&format!("\nNote: Normalized correlation ({:.3}) adjusts for
        amplitude differences,\n", result.correlation_normalized));
    content.push_str("      providing a better measure of waveform shape
        similarity.\n");
    content.push_str(&format!("      Original correlation ({:.3}) is affected by both
        amplitude and shape.\n", result.correlation));

    if result.snr_db > 40.0 {
        content.push_str("  Excellent SNR - minimal difference\n");
    } else if result.snr_db > 20.0 {
        content.push_str("  Good SNR - acceptable difference\n");
    } else {
        content.push_str("~ Low SNR - noticeable difference\n");
    }

    content.push_str("\nMethod differences:\n");
    content.push_str("-------------------\n");
    content.push_str("Q3 (Time-domain):\n");
    content.push_str("  - Uses 8th-order Butterworth filters (non-ideal)\n");
    content.push_str("  - IIR filter implementation (Direct Form II)\n");
    content.push_str("  - Non-linear phase response\n");
    content.push_str("  - Gradual transition band\n\n");

    content.push_str("Q4 (Frequency-domain):\n");
    content.push_str("  - Uses ideal brick-wall filters\n");
    content.push_str("  - Frequency-domain multiplication\n");
    content.push_str("  - No phase distortion from filters\n");
    content.push_str("  - Sharp cutoff\n");

    std::fs::write(filename, content).expect("Failed to save comparison");
}

/// Plot full-time comparison of two signals (all samples)
pub fn plot_full_comparison(signal1: &[f64], signal2: &[f64], filename: &str) {
    let n = signal1.len().min(signal2.len());

    let root = BitMapBackend::new(filename, (1600, 600)).into_drawing_area();
    root.fill(&WHITE).unwrap();

    let max_val = signal1[..n].iter()
        .chain(signal2[..n].iter())
        .fold(0.0f64, |max, &x| max.max(x.abs()));

    let mut chart = ChartBuilder::on(&root)
        .caption("Q4 vs Q3 Signal Comparison (Full Waveform)", ("sans-serif", 40))
```

```
        .margin(20)
        .x_label_area_size(50)
        .y_label_area_size(60)
        .build_cartesian_2d(0..n, -max_val*1.1..max_val*1.1)
        .unwrap();

    chart
        .configure_mesh()
        .x_desc("Sample")
        .y_desc("Amplitude")
        .draw()
        .unwrap();

    // Plot Q4 signal
    chart
        .draw_series(LineSeries::new(
            (0..n).map(|i| (i, signal1[i])),
            &BLUE,
        ))
        .unwrap()
        .label("Q4 (Frequency-domain)")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &BLUE));

    // Plot Q3 signal
    chart
        .draw_series(LineSeries::new(
            (0..n).map(|i| (i, signal2[i])),
            &RED,
        ))
        .unwrap()
        .label("Q3 (Time-domain)")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &RED));

    chart
        .configure_series_labels()
        .background_style(&WHITE.mix(0.8))
        .border_style(&BLACK)
        .draw()
        .unwrap();

    root.present().unwrap();
    println!("  Saved: {}", filename);
}

/// Plot comparison of two signals (first 2000 samples for detail)
pub fn plot_comparison(signal1: &[f64], signal2: &[f64], filename: &str) {
    let n = signal1.len().min(signal2.len()).min(2000); // Plot first 2000 samples

    let root = BitMapBackend::new(filename, (1200, 800)).into_drawing_area();
    root.fill(&WHITE).unwrap();

    let max_val = signal1[..n].iter()
        .chain(signal2[..n].iter())
        .fold(0.0f64, |max, &x| max.max(x.abs()));

    let mut chart = ChartBuilder::on(&root)
```

```
        .caption("Q4 vs Q3 Signal Comparison (Detail View)", ("sans-serif", 40))
        .margin(20)
        .x_label_area_size(50)
        .y_label_area_size(60)
        .build_cartesian_2d(0..n, -max_val*1.1..max_val*1.1)
        .unwrap();

    chart
        .configure_mesh()
        .x_desc("Sample")
        .y_desc("Amplitude")
        .draw()
        .unwrap();

    // Plot Q4 signal
    chart
        .draw_series(LineSeries::new(
            (0..n).map(|i| (i, signal1[i])),
            &BLUE,
        ))
        .unwrap()
        .label("Q4 (Frequency-domain)")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &BLUE));

    // Plot Q3 signal
    chart
        .draw_series(LineSeries::new(
            (0..n).map(|i| (i, signal2[i])),
            &RED,
        ))
        .unwrap()
        .label("Q3 (Time-domain)")
        .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &RED));

    chart
        .configure_series_labels()
        .background_style(&WHITE.mix(0.8))
        .border_style(&BLACK)
        .draw()
        .unwrap();

    root.present().unwrap();
    println!("  Saved: {}", filename);
}
```