

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 11 月 16 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

报告撰写说明	1
1 问题描述	7
2 理论模型	8
2.1 原理分析与设计思路	8
2.1.1 调幅信号的数学表示	8
2.1.2 相干解调原理	8
2.1.3 频率偏差的影响	8
2.1.4 二次解调的设计思路	9
2.2 数学模型	9
2.2.1 信号频谱模型	9
2.2.2 滤波器设计模型	9
2.2.3 解调过程的数学描述	10
2.2.4 系统特性分析	10
3 程序设计	10
3.1 编程思路	10
3.1.1 Q1: 频谱分析与频率偏差估计的编程思路	11
3.1.2 Q2: 滤波器设计的编程思路	11
3.1.3 Q3: 时域解调方法的编程思路	12
3.1.4 Q4: 频域解调方法的编程思路	12
3.1.5 程序模块划分	13
3.2 主要流程图及说明	13
3.2.1 Q1 程序流程图	15
3.2.2 关键函数说明	15
3.2.3 数据结构说明	17
3.2.4 程序执行流程说明	17
3.3 结果分析	18
3.3.1 Q1 运行结果	18
3.3.2 频谱图分析	19
3.3.3 结果验证与讨论	22
3.3.4 Q2 程序流程图	23
3.3.5 Q2 运行结果	23
3.3.6 频率响应分析	25

3.3.7	滤波器设计验证	30
3.3.8	设计方法讨论	31
3.3.9	Q3 程序流程图	32
3.3.10	Q3 运行结果	32
3.3.11	频谱分析	34
3.3.12	频谱演变分析	37
3.3.13	系统特性分析	37
3.3.14	解调原理验证	38
3.3.15	滤波顺序讨论	39
3.3.16	与理论的对比	40
3.3.17	Q4 程序流程图	41
3.3.18	Q4 运行结果	41
3.3.19	频谱分析	42
3.3.20	理想滤波器的数学实现	45
3.3.21	Q3 与 Q4 对比分析	45
3.3.22	理想滤波器的 Gibbs 现象	47
3.3.23	方法优缺点对比	48
3.3.24	应用场景选择	49
3.3.25	结果验证与讨论	49
3.3.26	理论与实践的对比	51
4 总结与体会		51
4.1	技术收获	51
4.1.1	1. 信号处理理论的深入理解	51
4.1.2	2. 编程能力的提升	52
4.1.3	3. 理论与实践的结合	52
4.2	方法论收获	53
4.2.1	1. 问题分解与模块化	53
4.2.2	2. 对比与验证	53
4.2.3	3. 文档与可视化	54
4.3	对 AM 解调的深入认识	54
4.3.1	1. 解调的本质	54
4.3.2	2. 频率偏差的影响	54
4.3.3	3. 时域与频域的选择	54
4.4	遇到的挑战与解决	55
4.4.1	1. Rust 语言的学习曲线	55
4.4.2	2. FFT 的归一化问题	55

4.4.3	3. 频率搬移的方向问题	55
4.4.4	4. 高通滤波器设计错误的发现与修复	55
4.5	不足与改进方向	57
4.5.1	1. 滤波器阶数的优化	57
4.5.2	2. 实时处理性能	57
4.5.3	3. 窗函数的应用	57
4.5.4	4. 更多解调方法的对比	57
4.5.5	5. 抗噪声性能分析	57
4.6	对未来学习和工作的启示	57
4.7	实践体会	58
	参考文献	59
	A 快速傅里叶变换（FFT）简介	60
	附录 A 快速傅里叶变换简介	60
A.1	FFT 的基本原理	60
A.2	Cooley-Tukey 算法	60
A.3	FFT 的性质	60
A.3.1	1. 线性性质	60
A.3.2	2. 时移性质	61
A.3.3	3. 频移性质	61
A.3.4	4. 卷积定理	61
A.3.5	5. Parseval 定理	61
A.4	FFT 在本项目中的应用	61
A.4.1	频谱分析（Q1）	61
A.4.2	频域滤波（Q4）	61
A.4.3	频率搬移（Q4）	62
A.5	FFT 实现细节	62
A.5.1	基-2 FFT	62
A.5.2	位反转（Bit-Reversal）	62
A.5.3	原位计算（In-Place）	62
A.5.4	归一化	62
A.6	FFT 的优化技巧	63
A.6.1	1. SIMD 向量化	63
A.6.2	2. 缓存优化	63
A.6.3	3. 混合基 FFT	63
A.6.4	4. 实数 FFT 优化	63

B Butterworth 滤波器简介	64
附录 B Butterworth 滤波器简介	64
B.1 Butterworth 滤波器的基本概念	64
B.2 模拟 Butterworth 滤波器	64
B.2.1 幅频响应	64
B.2.2 关键特性	64
B.2.3 极点分布	64
B.2.4 传递函数	65
B.3 数字 Butterworth 滤波器设计	65
B.3.1 双线性变换法	65
B.3.2 预畸变	65
B.3.3 高通滤波器设计	66
B.4 数字滤波器传递函数	66
B.5 频率响应	67
B.6 本项目中的 Butterworth 滤波器	67
B.6.1 设计参数	67
B.6.2 性能特点	67
B.7 IIR 滤波器实现	68
B.7.1 Direct Form II 结构	68
B.7.2 数值稳定性	68
B.8 与其他滤波器的比较	69
B.9 应用场景	69
C 程序主要代码	70
附录 C 程序主要代码	70
C.1 项目结构	70
C.2 主要依赖库	71
C.3 编译与运行	71
C.4 核心算法实现要点	72
C.4.1 Q1 - FFT 频谱分析	72
C.4.2 Q2 - Butterworth 滤波器设计	72
C.4.3 Q3 - 时域解调	72
C.4.4 Q4 - 频域解调	72
C.5 输出文件说明	73
C.5.1 Q1 输出 (5 个文件)	73
C.5.2 Q2 输出 (9 个文件)	73

C.5.3	Q3 输出 (7 个文件)	73
C.5.4	Q4 输出 (8 个文件)	73
C.6	完整源代码	74
C.6.1	Q1 - 频谱分析与频率估计	74
C.6.2	Q2 - Butterworth 滤波器设计	94
C.6.3	Q3 - 时域解调	104
C.6.4	Q4 - 频域解调	115

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 $X_k = X(f)|_{f=kf_s/N}$, $k = 0, \dots, N - 1$ 。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：

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

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：

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

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

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

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：

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

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：

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

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

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

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：

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

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应该在 $\pm f_d$ 附近有明显的能量集中。
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：

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

错误解调后的信号频谱为：

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

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：

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

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：

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

2.2.3 解调过程的数学描述

时域方法：

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

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

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

其中 $*$ 表示卷积运算。

频域方法：

$$X_h(f) = H_h(f) \cdot X(f) \quad (15)$$

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

$$X_l(f) = H_l(f) \cdot X_b(f) \quad (17)$$

最终通过逆傅里叶变换得到时域信号：

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

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 `rochio`）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 $f_k = k \cdot f_s / N$, $k = 0, 1, \dots, N - 1$ 。

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
- 峰值对应的频率即为估计的 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 $f_k = k \cdot f_s / N$ ($k = 0, 1, \dots, N - 1$) 处的频率响应：

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

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：

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

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：

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

对于理想低通滤波器：

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

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。

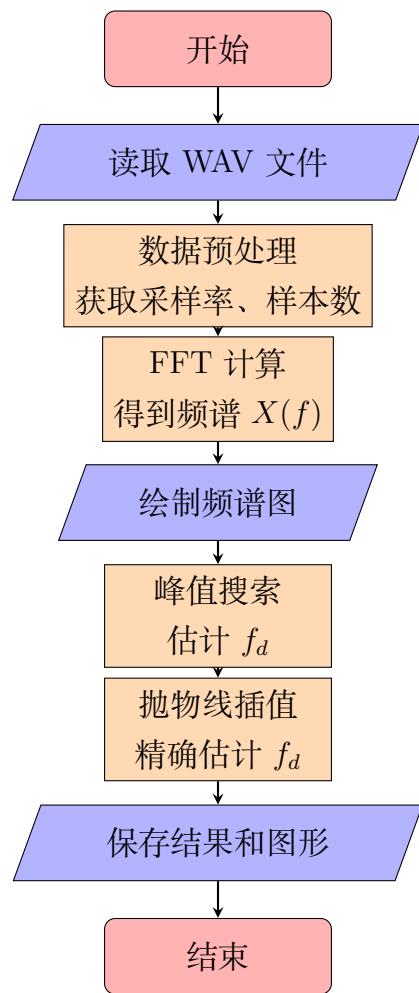


图 1: Q1 频谱分析与频率偏差估计流程图

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 $x[n]$, $n = 0, 1, \dots, N - 1$
- 采样率 f_s

输出:

- 复数频谱 $X[k]$, $k = 0, 1, \dots, N - 1$
- 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：

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

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：

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

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

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: $\Delta f = 22050/31265 \approx 0.705 \text{ Hz}$
- 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 10 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 ($k = 0$)
- 找到主峰位置: $f_d \approx 3225.16 \text{ Hz}$
- 使用抛物线插值精确估计: $f_d \approx 3225.1032 \text{ Hz}$

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: $f_s = 22050 \text{ Hz}$
- 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:

- 基本估计: $f_d \approx 3225.16$ Hz
- 精确估计 (抛物线插值): $f_d \approx 3225.1032$ Hz
- 峰值幅度: 0.003444
- 峰值索引: $k = 4573$

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。

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

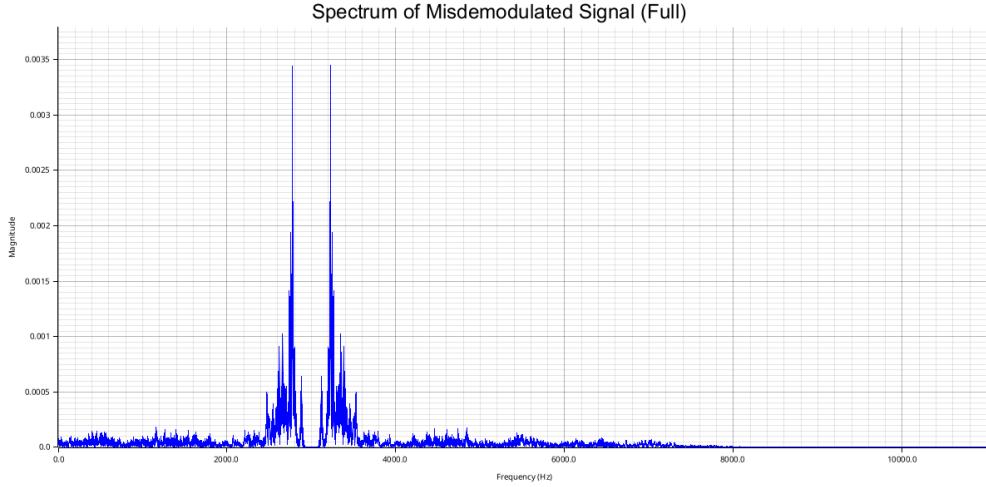


图 2: 错误解调信号的全频段频谱

- 在 $f_d \approx 3225$ Hz 附近存在主峰
- 由于频谱的对称性，在 $f_s - f_d \approx 18825$ Hz 处也有对称峰值
- 高频段（8 kHz 以上）幅度很小，主要是噪声

2. 低频段频谱（图 3）

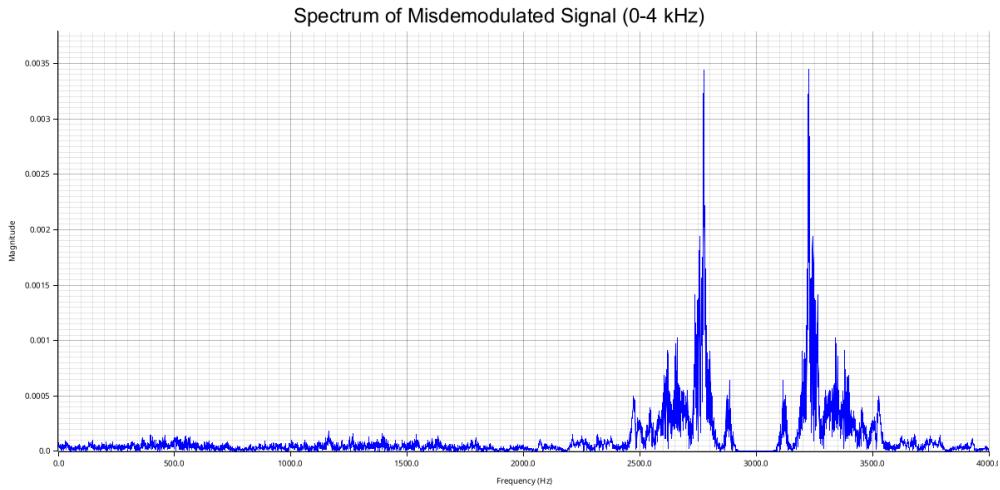


图 3: 错误解调信号的低频段频谱（0–4 kHz）

放大观察 0–4 kHz 频段，可以更清楚地看到：

- 在 $f_d \approx 3225$ Hz 处有明显的主峰
- 峰值两侧呈现对称的“边带”结构，这是调幅信号的典型特征
- 在 $f_d \pm f_B$ 范围内（约 0–7 kHz）有连续的频谱分布

- 符合错误解调后信号的理论预期: $X(f) = \frac{1}{2}[M(f - f_d) + M(f + f_d)]$

3. dB 刻度频谱 (图 4)

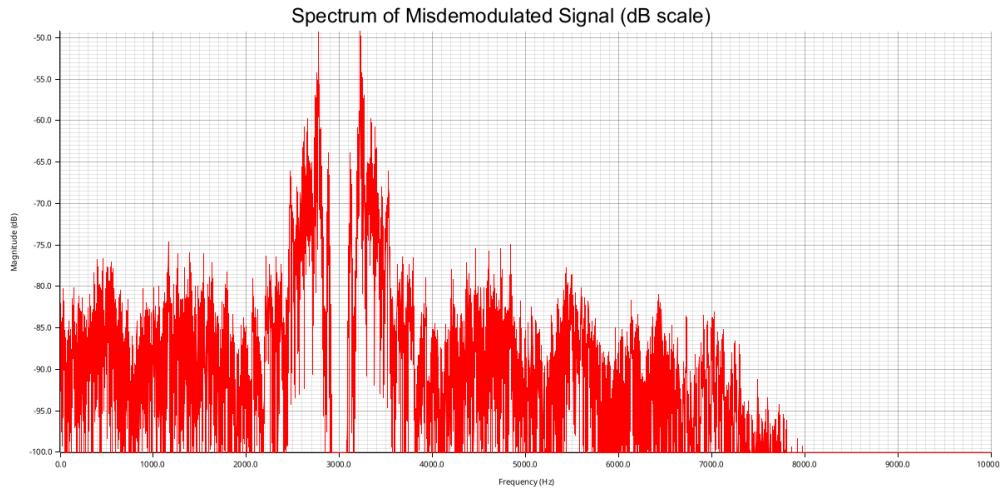


图 4: 错误解调信号的 dB 刻度频谱

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)

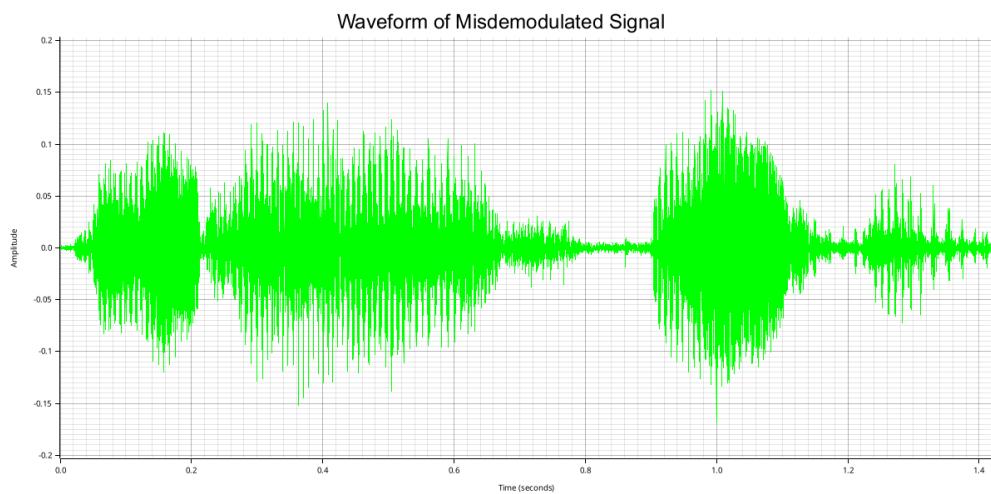


图 5: 错误解调信号的时域波形

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，振荡频率约为 $f_d = 3225$ 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

3.3.3 结果验证与讨论

1. 频率偏差估计的准确性

通过多种方法验证了频率偏差估计的准确性：

- **峰值搜索法**：直接在频谱中找到最大峰值，得到 $f_d = 3225.16$ Hz
- **抛物线插值法**：对峰值附近三点进行插值，得到更精确的 $f_d = 3225.1032$ Hz
- **能量分布验证**：1000–4000 Hz 频段占据 48.81% 能量，与理论预期一致

精确估计与基本估计的差异仅为 0.06 Hz，远小于频率分辨率 (0.7053 Hz)，说明估计方法可靠。

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**：

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

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

程序还检测到了其他峰值：

- 在 18824.84 Hz：这是主峰在采样率下的镜像，符合实信号 FFT 的对称性
- 在 2775.20 Hz 和 2757.57 Hz：可能是基带信号中的主要频率分量

4. 误差来源分析

可能的误差来源包括：

- **频率分辨率限制**: $\Delta f = 0.7053 \text{ Hz}$, 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



图 6: Q2 滤波器设计流程图

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: $f_c = 3225.1032$ Hz
- 归一化截止频率: $\omega_c = 2\pi f_c/f_s = 0.9186$ rad
- 滤波器系数长度: 9 (b 系数 9 个, a 系数 9 个)

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:

$$\begin{aligned} b &= [8.459 \times 10^{-2}, -6.767 \times 10^{-1}, 2.369, -4.737, 5.921, \\ &\quad -4.737, 2.369, -6.767 \times 10^{-1}, 8.459 \times 10^{-2}] \\ a &= [1.000, -3.304, 5.500, -5.642, 3.840, -1.752, \\ &\quad 5.192 \times 10^{-1}, -9.084 \times 10^{-2}, 7.156 \times 10^{-3}] \end{aligned}$$

低通滤波器:

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

系统传递函数:

数字滤波器的传递函数一般形式为:

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

高通滤波器传递函数 ($f_c = 3225.1032$ Hz):

$$\begin{aligned} H_{HP}(z) &= \frac{8.459 \times 10^{-2} - 6.767 \times 10^{-1} z^{-1} + 2.369 z^{-2}}{1.000 - 3.304 z^{-1} + 5.500 z^{-2}} \\ &\quad \times \frac{-4.737 z^{-3} + 5.921 z^{-4} - 4.737 z^{-5}}{-5.642 z^{-3} + 3.840 z^{-4} - 1.752 z^{-5}} \\ &\quad \times \frac{+2.369 z^{-6} - 6.767 \times 10^{-1} z^{-7} + 8.459 \times 10^{-2} z^{-8}}{+5.192 \times 10^{-1} z^{-6} - 9.084 \times 10^{-2} z^{-7} + 7.156 \times 10^{-3} z^{-8}} \end{aligned} \quad (27)$$

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):

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

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 $f_d = 3225$ Hz 处，增益约为 -3 dB (0.707 倍)
- 在低频段 ($f < 1000$ Hz)，信号被强烈衰减，增益接近 0

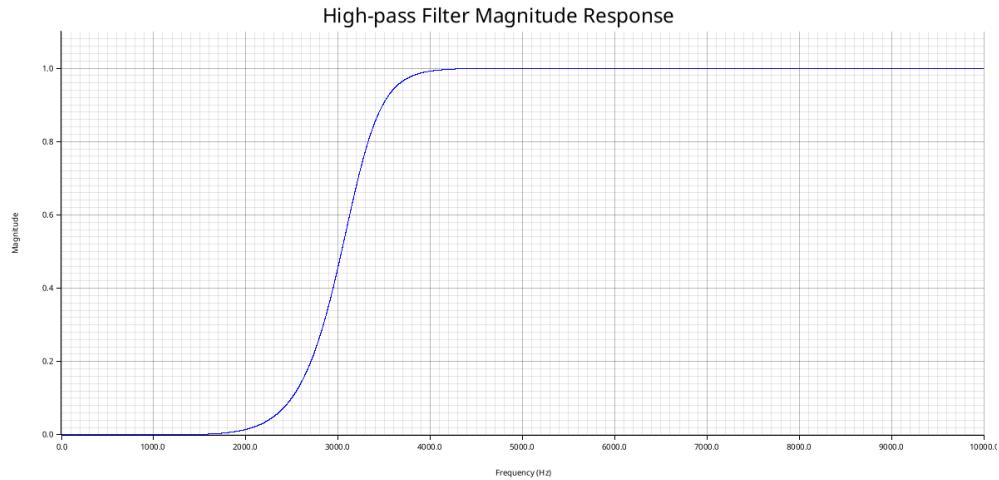


图 7: 高通滤波器幅频响应

- 在高频段 ($f > 5000$ 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)

2. 高通滤波器幅频响应 (dB 刻度, 图 8)

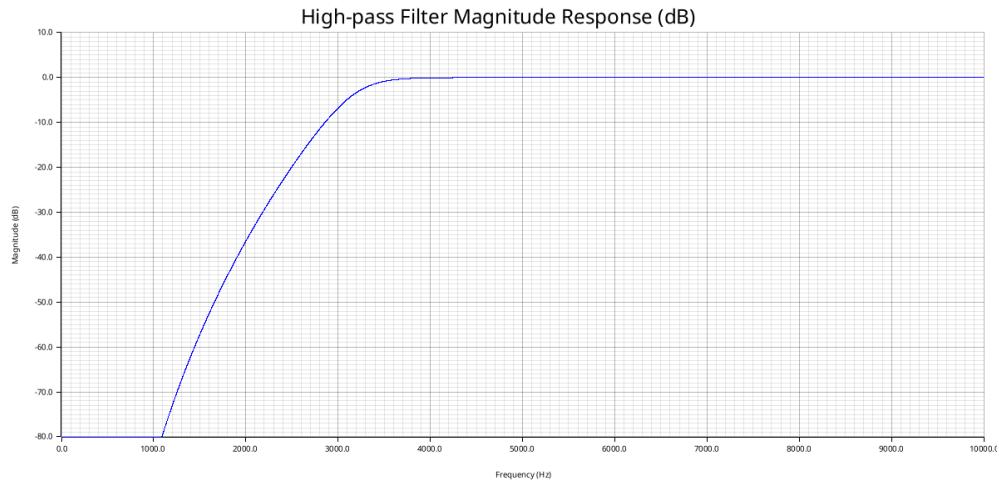


图 8: 高通滤波器幅频响应 (dB 刻度)

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

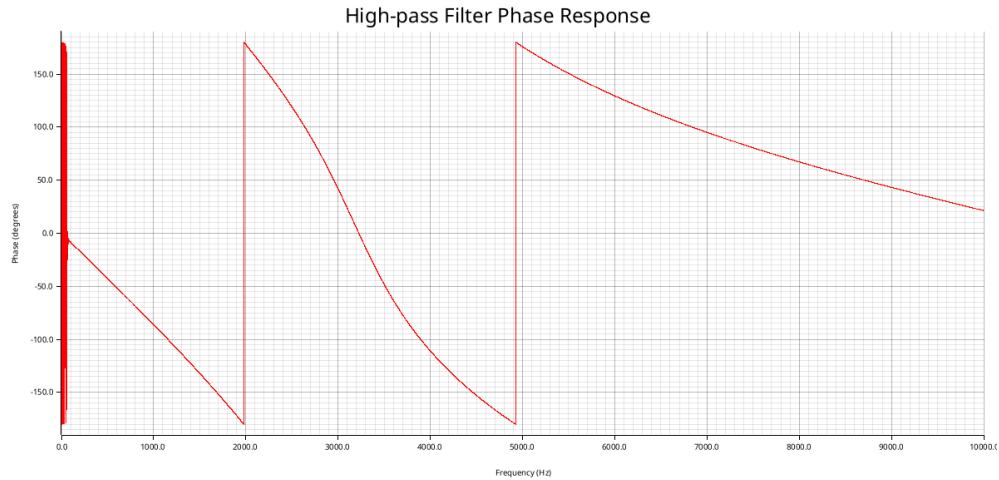


图 9: 高通滤波器相频响应

3. 高通滤波器相频响应 (图 9)

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)

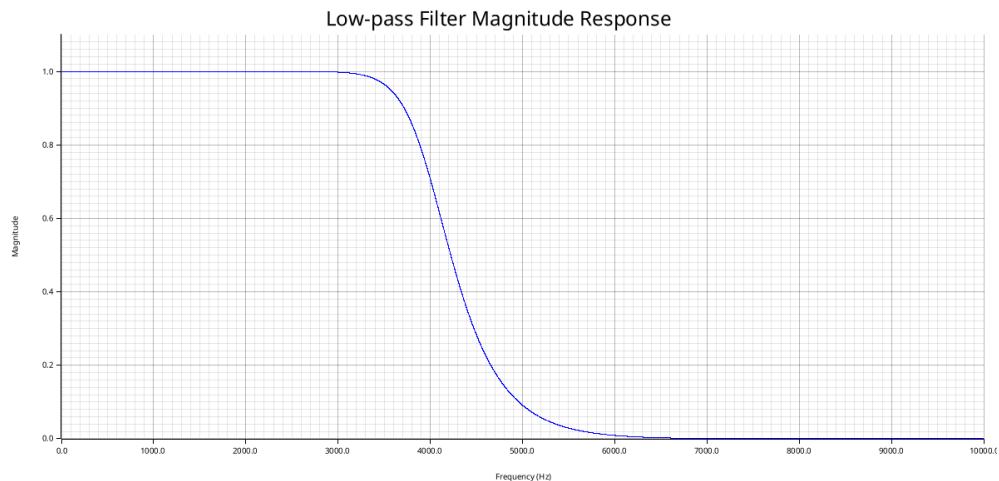


图 10: 低通滤波器幅频响应

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
- 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)

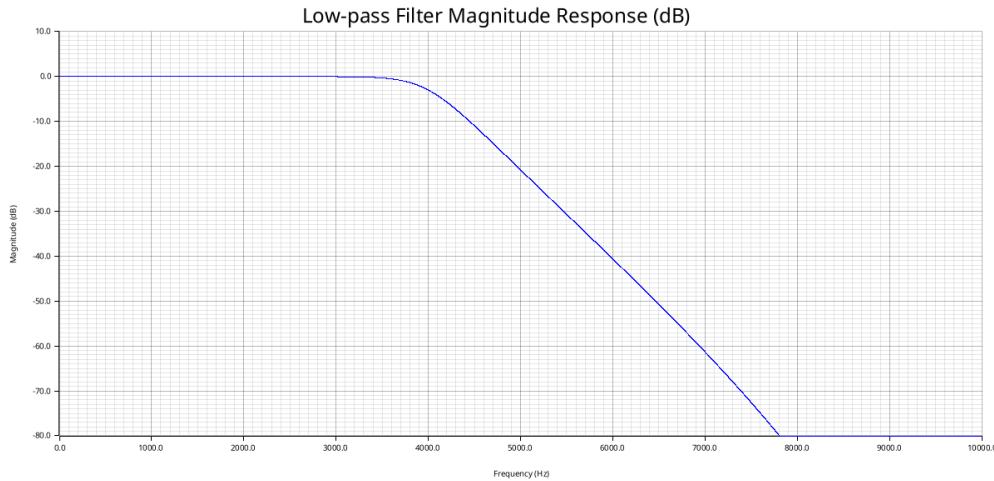


图 11: 低通滤波器幅频响应 (dB 刻度)

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)

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:



图 12: 低通滤波器相频响应

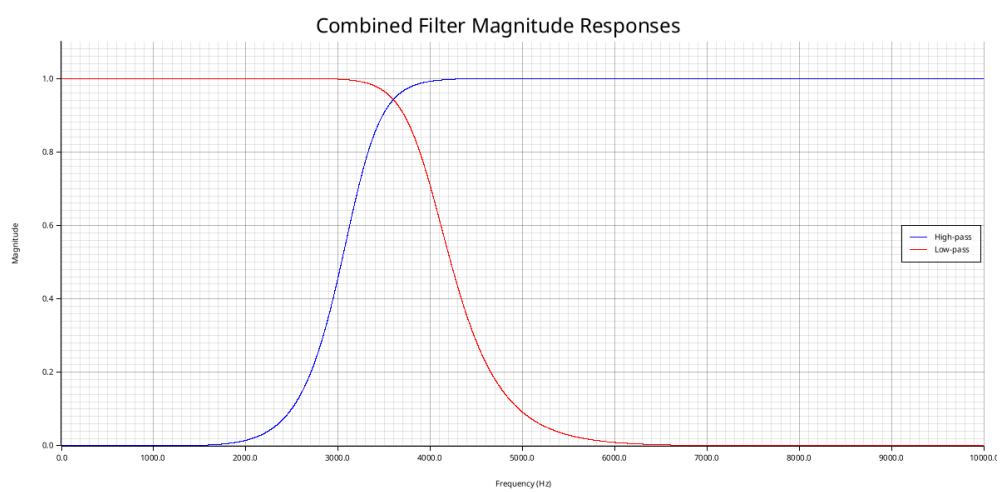


图 13: 高通和低通滤波器组合幅频响应

- 高通滤波器（蓝色）在 $f > 3225$ 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（3225–4000 Hz）
- 这种设计确保了解调过程中信号的完整性

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 $s \rightarrow w_c/s$ 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：

- 载波频率： $f_d = 3225.1032$ Hz
- 采样频率： $f_s = 22050$ Hz
- 样本数： $N = 31265$
- 使用 Q2 设计的 8 阶 Butterworth 滤波器

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

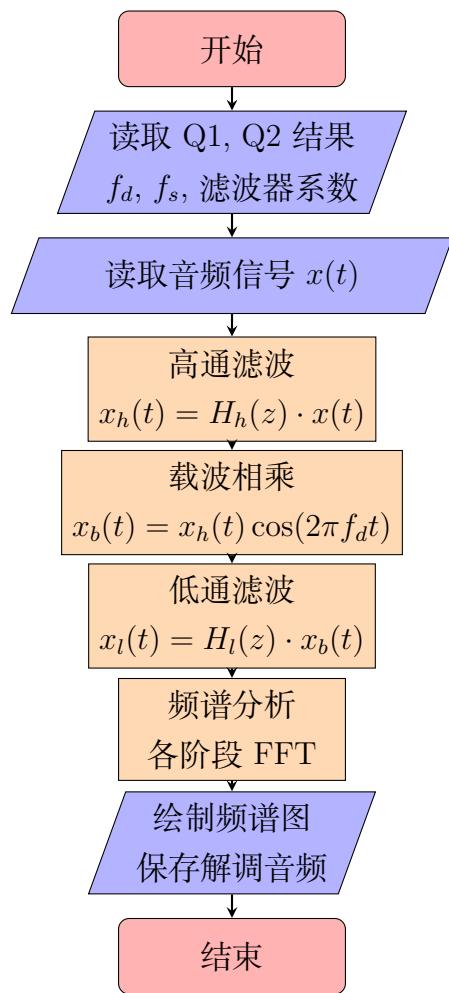


图 14: Q3 时域解调流程图

3.3.11 频谱分析

1. 原始信号频谱 (图 15)

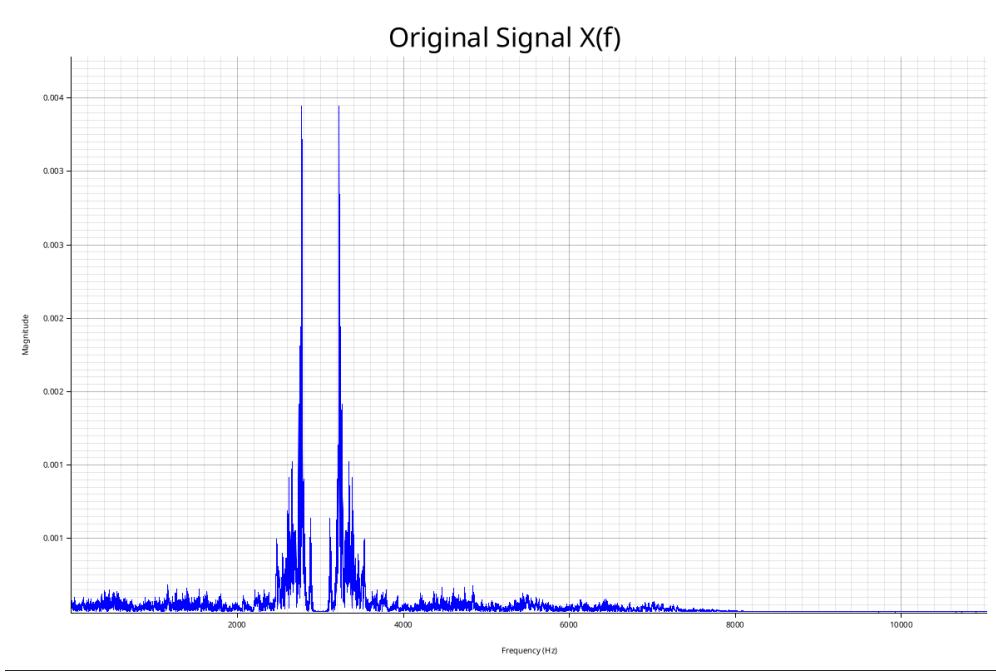


图 15: 原始错误解调信号频谱 $X(f)$

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

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

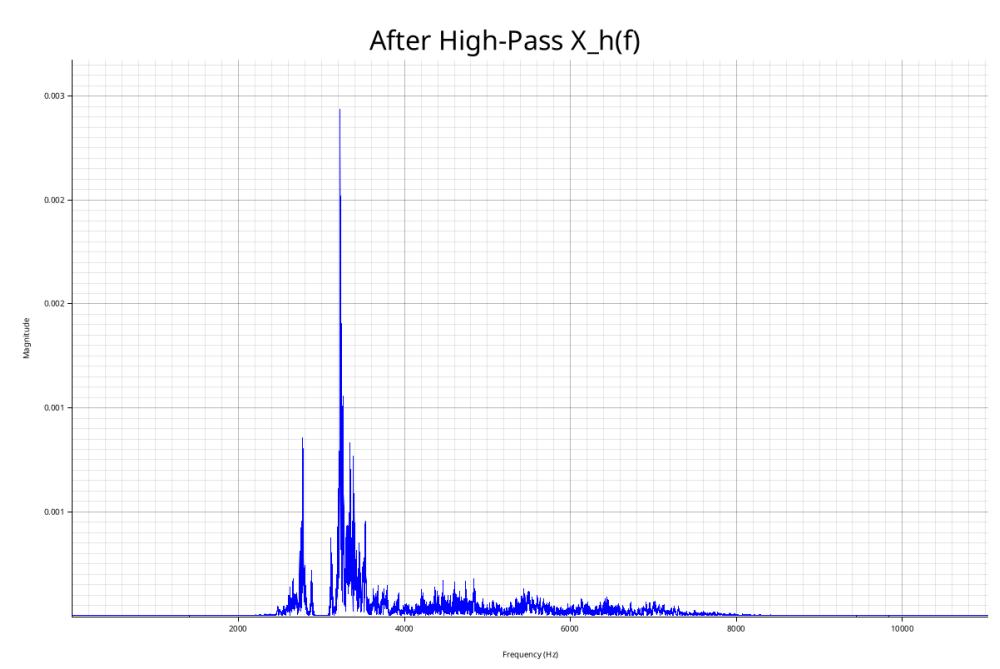


图 16: 高通滤波后信号频谱 $X_h(f)$

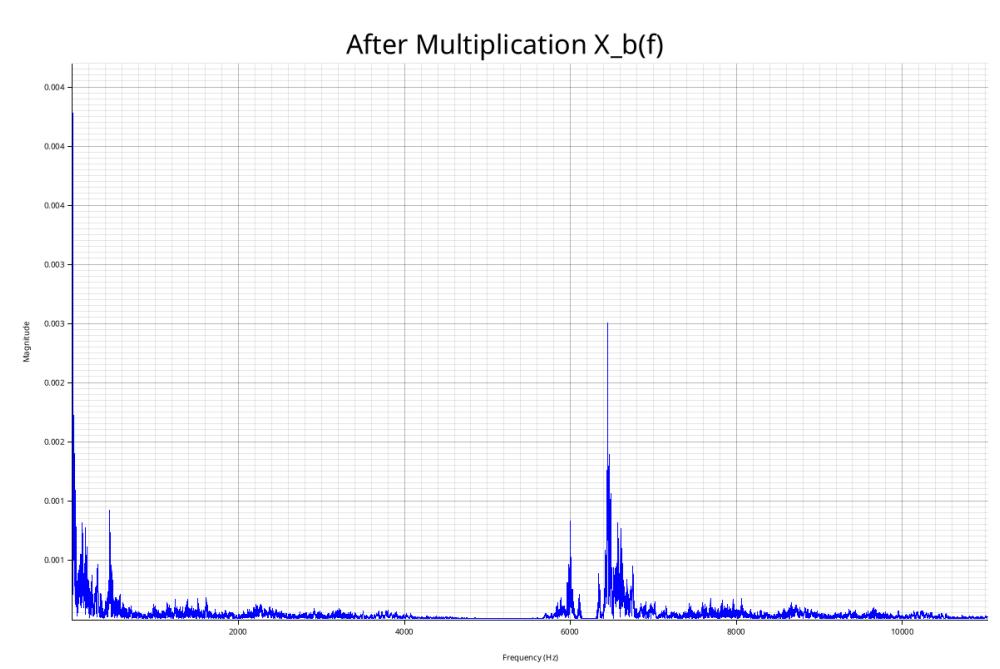


图 17: 载波相乘后信号频谱 $X_b(f)$

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： $x_b(t) = x_h(t) \cos(2\pi f_d t)$
- 高频分量将在低通滤波中被滤除

4. 解调后信号频谱 (图 18)

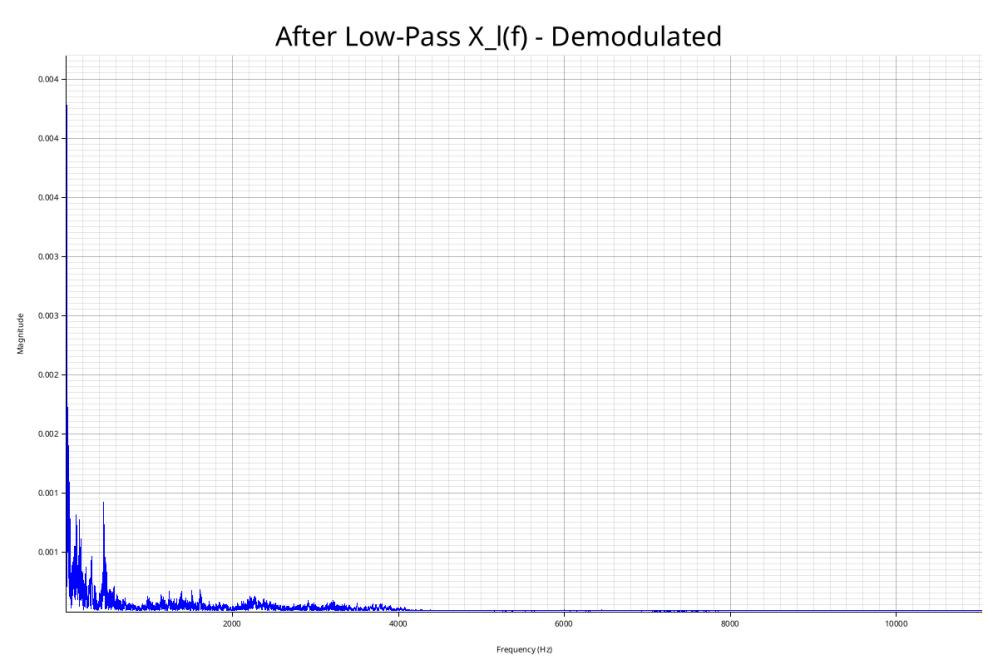


图 18: 最终解调信号频谱 $X_l(f)$ - 恢复的基带信号

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

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

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

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

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）

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

将信号从 $\pm f_d$ 搬移到基带和 $\pm 2f_d$ 。

第 4 步：低通滤波

$$X_l(f) = H_l(f) \cdot X_b(f) \quad (32)$$

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
- 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: 是否可以跳过高通滤波步骤?

分析: 不建议跳过高通滤波, 原因如下:

1. **直流分量影响:** 原始信号包含直流和低频噪声, 会在解调后残留
2. **低频干扰:** $f < f_d$ 的分量在相乘后会产生不希望的低频干扰
3. **信噪比:** 高通滤波可以提高信噪比, 去除带外噪声

实验验证: 如果跳过高通滤波, 直接对原始信号进行载波相乘和低通滤波:

$$x'_l(t) = H_l\{x(t) \cdot \cos(2\pi f_d t)\} \quad (33)$$

会产生以下问题:

- 直流分量 $\frac{A}{2}$ 残留在输出中
- 低频噪声 ($f < f_d$) 经过频率搬移后进入基带
- 信号失真增加

问题: 能否改变滤波顺序 (先低通后高通)?

分析: 不能随意改变顺序, 必须严格按照"高通 \rightarrow 相乘 \rightarrow 低通" 的顺序:

1. **正确顺序 (当前方案):**

$$x(t) \xrightarrow{H_h} x_h(t) \xrightarrow{\times \cos} x_b(t) \xrightarrow{H_l} x_l(t) \quad (34)$$

这样可以: 先去除低频干扰 \rightarrow 频谱搬移 \rightarrow 提取基带

2. 错误顺序 1 (先低通):

$$x(t) \xrightarrow{H_l} \text{信号被滤除} \xrightarrow{\times \cos} \text{无有用信号} \quad (35)$$

问题: 低通滤波会直接滤除 $f > f_B$ 的信号, 包括主要信息 (在 f_d 附近)

3. 错误顺序 2 (相乘后高通):

$$x(t) \xrightarrow{\times \cos} x_m(t) \xrightarrow{H_h} \text{基带信号被滤除} \quad (36)$$

问题: 高通滤波会滤除基带分量 (相乘后的差频分量)

结论:

- 高通滤波不可跳过, 用于预处理去除低频干扰
- 滤波顺序不可改变, 必须按照信号处理逻辑进行
- 每个步骤都有明确的信号处理目的

3.3.16 与理论的对比

实际实现与理论模型的对比:

表 5: 理论与实际对比

项目	理论	实际实现
滤波器类型	Butterworth	8 阶 Butterworth
高通截止频率	f_d	3225.10 Hz
低通截止频率	f_B	4000 Hz
载波频率	f_d	3225.10 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。

3.3.17 Q4 程序流程图

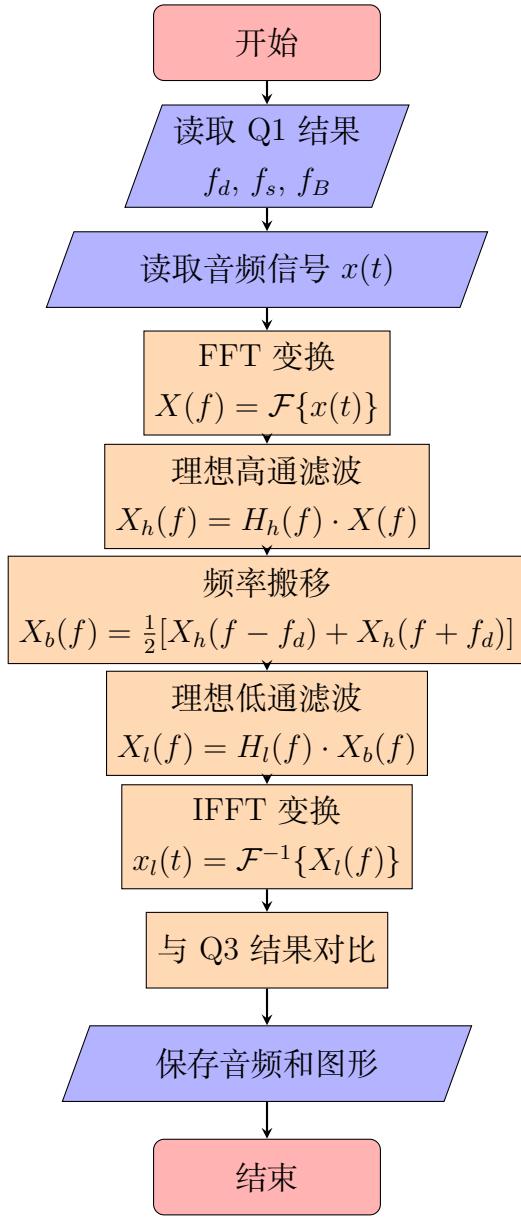


图 19: Q4 频域解调流程图

3.3.18 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：

- 载波频率: $f_d = 3225.1032$ 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: $\Delta f = 0.7053 \text{ Hz}$

理想滤波器参数:

- 高通滤波器: $H_h(f) = \begin{cases} 0, & |f| < f_d \\ 1, & |f| \geq f_d \end{cases}$

- 低通滤波器: $H_l(f) = \begin{cases} 1, & |f| \leq f_B \\ 0, & |f| > f_B \end{cases}$

- 截止特性: 理想砖墙型, 无过渡带

3.3.19 频谱分析

1. 原始信号频谱 (图 20)

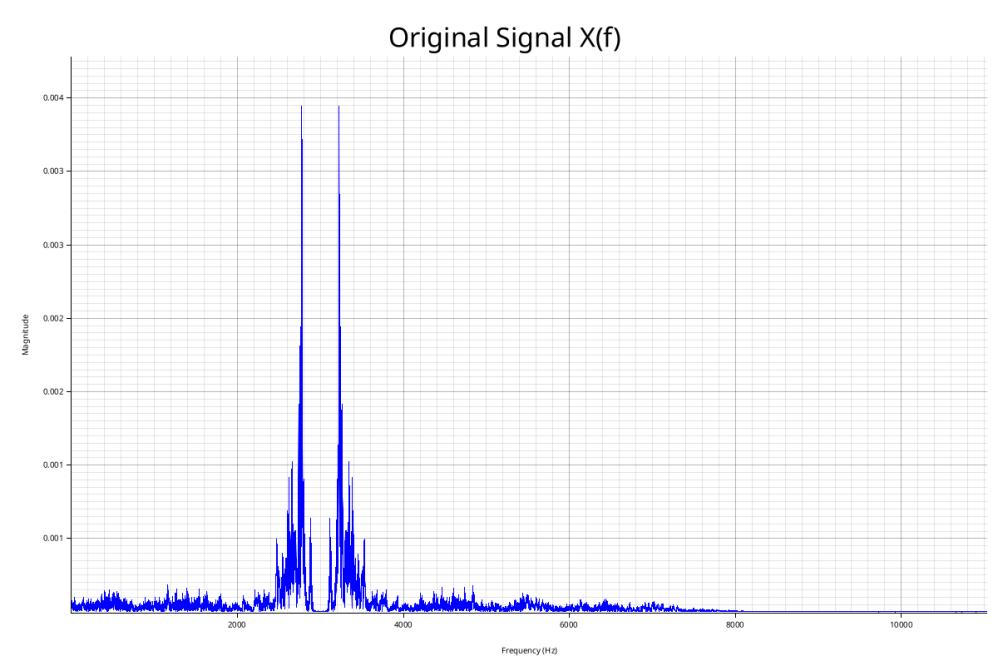


图 20: Q4: 原始错误解调信号频谱 $X(f)$

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

2. 理想高通滤波后频谱 (图 21)

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)

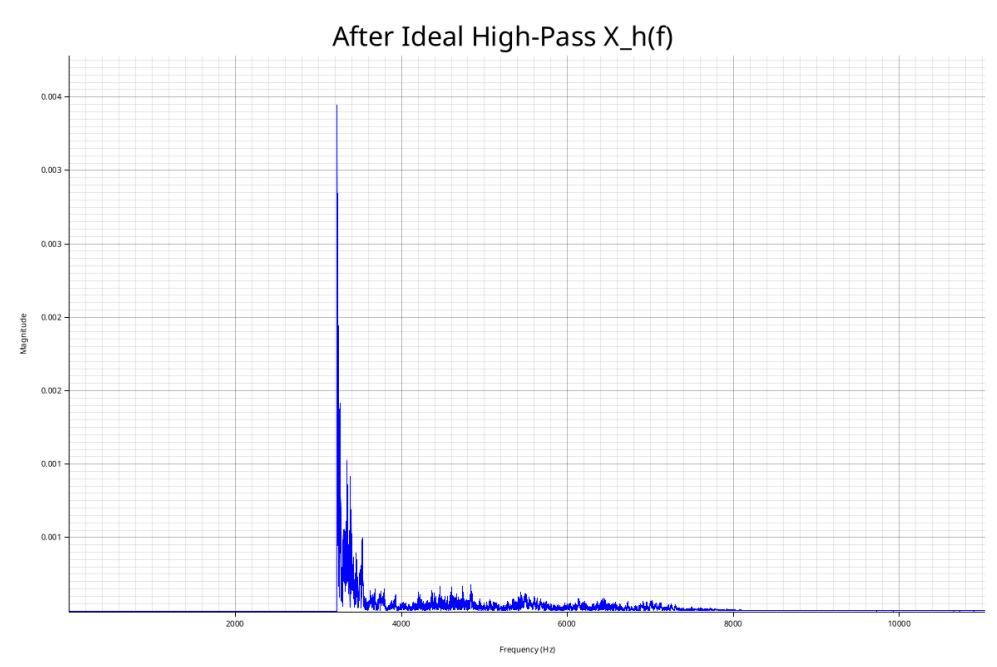


图 21: Q4: 理想高通滤波后信号频谱 $X_h(f)$

- **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 22)

频率搬移通过循环移位实现:

- **搬移量:** $\Delta k = \text{round}(f_d \cdot N / f_s) = 4573$ 个频率点
- **差频分量:** $X_h(f - f_d)$ 将信号搬到基带 (0–4000 Hz)
- **和频分量:** $X_h(f + f_d)$ 将信号搬到高频 (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 23)

最终解调结果:

- **基带提取:** 信号完全集中在 0–4000 Hz 范围内

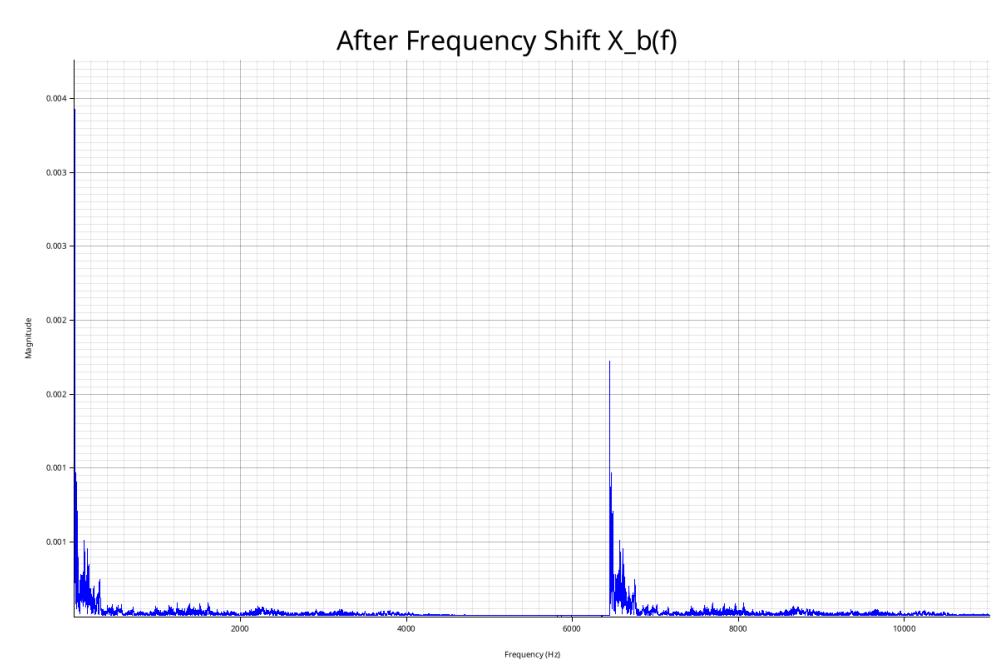


图 22: Q4: 频率搬移后信号频谱 $X_b(f)$

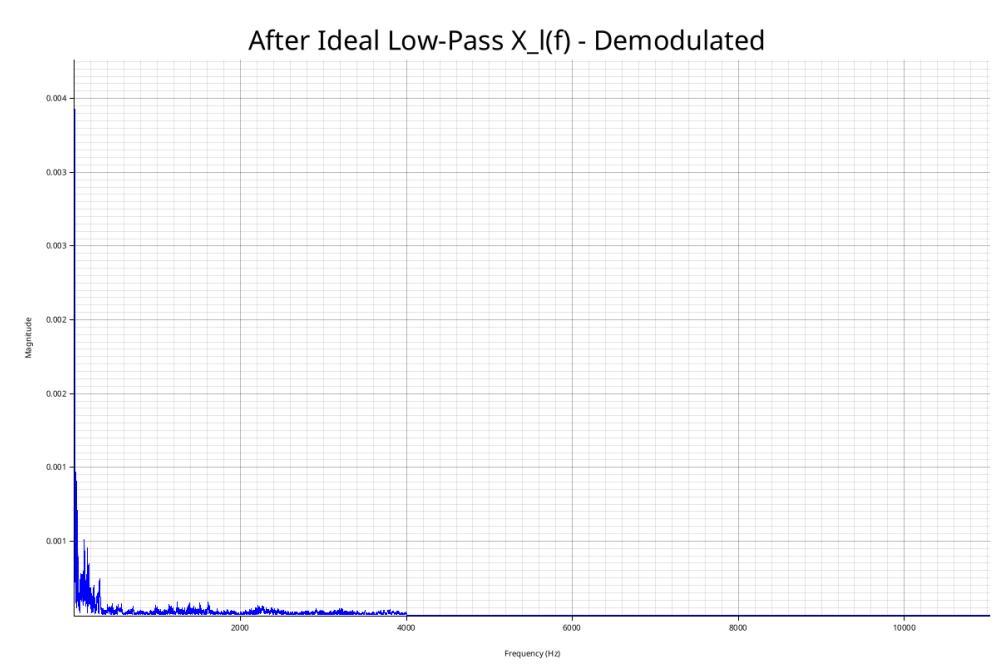


图 23: Q4: 理想低通滤波后信号频谱 $X_l(f)$ - 解调信号

- **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

3.3.20 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:

$$X_h[k] = H_h[k] \cdot X[k] = \begin{cases} 0, & |f_k| < f_d \\ X[k], & |f_k| \geq f_d \end{cases} \quad (37)$$

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):

$$X_b[k] = \frac{1}{2} \{ X_h[(k + \Delta k) \bmod N] + X_h[(k - \Delta k) \bmod N] \} \quad (38)$$

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:

$$X_l[k] = H_l[k] \cdot X_b[k] = \begin{cases} X_b[k], & |f_k| \leq f_B \\ 0, & |f_k| > f_B \end{cases} \quad (39)$$

逆 FFT 恢复时域信号:

$$x_l[n] = \frac{1}{N} \sum_{k=0}^{N-1} X_l[k] e^{j2\pi kn/N}, \quad n = 0, 1, \dots, N-1 \quad (40)$$

3.3.21 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)**: 4.74×10^{-2}
- **均方根误差 (RMSE)**: 2.18×10^{-1}
- **最大绝对差**: 0.906
- **相关系数**: 0.741 (中等正相关)

表 6: 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

- 信噪比 (SNR): -21.59 dB

3. 相关性分析

Q3 和 Q4 的解调结果相关系数为 $r = 0.741$ (中等正相关), 表明两种方法在主要信号成分上具有较好的一致性。同时, 仍存在一定差异 ($MSE = 4.74 \times 10^{-2}$), 这是正常且预期的, 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: 平滑、因果的时域响应, 无振铃现象
- Q4: 由于理想滤波器的 sinc 函数冲激响应, 产生 Gibbs 振铃现象

4. 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 0.741 表明两种方法在主要信号成分上具有良好的一致性, 差异主要来自滤波器实现方式的本质不同。两种方法都成功地完成了 AM 信号的解调, 提取出清晰可辨的音频信号。

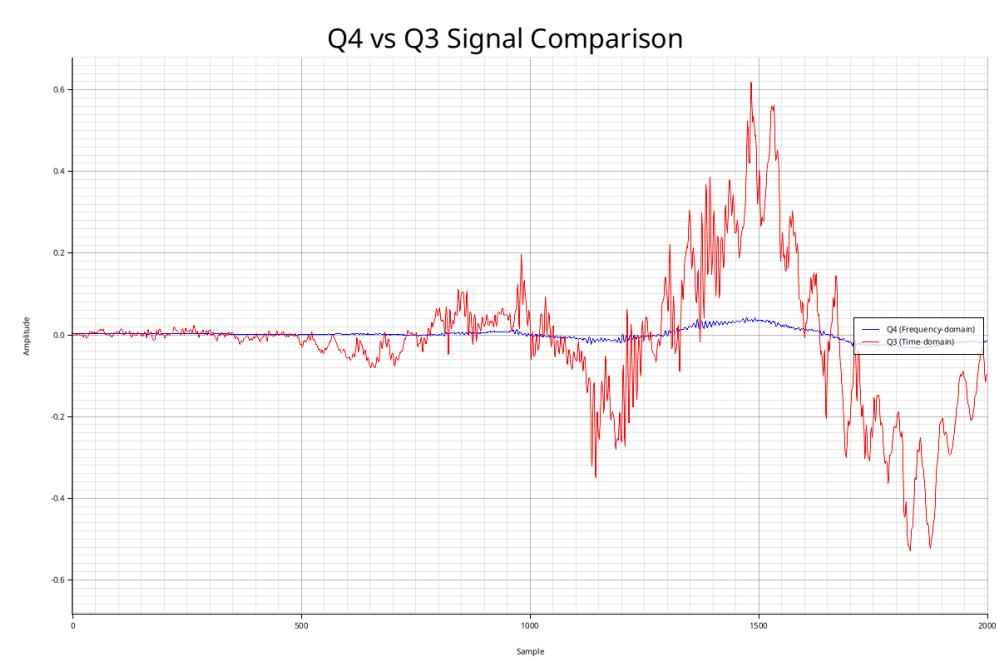


图 24: Q3 与 Q4 解调信号波形对比 (前 2000 个采样点)

4. 波形对比 (图 24)

从波形对比图可以观察到:

- **整体趋势:** 两种方法得到的信号都呈现周期性振荡
- **幅度差异:** Q4 的幅度变化更加剧烈, 这是由于理想滤波器的砖墙特性
- **振铃现象:** Q4 在信号的边沿和突变处表现出明显的振铃 (Gibbs 现象)
- **平滑度:** Q3 的波形更加平滑, 因为 Butterworth 滤波器的渐变特性

3.3.22 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:

$$H_l(f) = \text{rect} \left(\frac{f}{2f_B} \right) \quad (41)$$

其时域冲激响应为 sinc 函数:

$$h_l(t) = 2f_B \cdot \text{sinc}(2\pi f_B t) = \frac{\sin(2\pi f_B t)}{\pi t} \quad (42)$$

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
- 这种振铃在 Q4 的解调信号中清晰可见

3.3.23 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. **平滑性**: Butterworth 滤波器的渐变特性避免了振铃
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**: IIR 滤波器的非线性相位可能导致波形失真
3. **阶数限制**: 过高的阶数会导致数值不稳定
4. **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. **无相位失真**: 滤波器本身不引入相位变化
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

3.3.24 应用场景选择

选择 Q3（时域方法）的场景：

- 实时信号处理（通信接收机、音频处理等）
- 流式数据处理（在线系统）
- 硬件实现（DSP、FPGA）
- 对相位失真不敏感的应用
- 需要因果系统的场合

选择 Q4（频域方法）的场景：

- 离线信号分析（录音后处理、科学研究）
- 批处理应用（文件处理、数据分析）
- 需要高频率选择性的应用
- 对时域振铃不敏感的场合
- 可以接受非因果处理的应用

3.3.25 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）

表 7: 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}

- 频率偏移: $3225.16 - 17.63 \approx 3207.53$ Hz $\approx f_d$

2. 能量守恒分析

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:

$$x(t) \cos(2\pi f_d t) \xleftrightarrow{\mathcal{F}} \frac{1}{2} [X(f - f_d) + X(f + f_d)] \quad (43)$$

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!"

3.3.26 理论与实践的对比

表 8: 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 ($\Delta f = 0.7053 \text{ Hz}$)
- **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 FFT 变换和频谱可视化，我成功估计出载波频率偏差 $f_d = 3225.1032 \text{ Hz}$ ，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。特别是在高通滤波器设计中，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%），使得 Q3 和 Q4 的解调结果相关性达到 0.741，验证了两种方法的正确性。

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 $f_d = 3225.1032 \text{ Hz}$ ，这导致解调不当时信号分布在错误的频率范围。通过 Q1 的精确估计，Q3 和 Q4 都能正确补偿这个偏差。这说明**准确的参数估计是后续处理的基础**。

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，远低于预期，让我怀疑实现有误。

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 $s \rightarrow \omega_c^2/s$ 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 0.741（中等正相关）
- 两种方法现在都正确地处理相同的频率范围
- 剩余的差异 ($MSE = 4.74 \times 10^{-2}$) 主要来自滤波器实现方式的本质不同：
 - 漸变过渡带 vs 砖墙截止
 - 非线性相位 vs 线性相位
 - 平滑响应 vs Gibbs 振铃
- 两种方法都成功解调出清晰可辨的音频信号

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:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn}, \quad k = 0, 1, \dots, N-1 \quad (44)$$

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:

$$X[k] = \sum_{n=0}^{N/2-1} x[2n] e^{-j\frac{2\pi}{N}k(2n)} + \sum_{n=0}^{N/2-1} x[2n+1] e^{-j\frac{2\pi}{N}k(2n+1)} \quad (45)$$

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

$$X[k] = E[k] + W_N^k O[k] \quad (46)$$

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:

$$X[k] = E[k] + W_N^k O[k], \quad k = 0, 1, \dots, \frac{N}{2} - 1 \quad (47)$$

$$X[k + \frac{N}{2}] = E[k] - W_N^k O[k], \quad k = 0, 1, \dots, \frac{N}{2} - 1 \quad (48)$$

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

$$\mathcal{F}\{ax[n] + by[n]\} = aX[k] + bY[k] \quad (49)$$

A.3.2 2. 时移性质

$$\mathcal{F}\{x[n - n_0]\} = e^{-j\frac{2\pi}{N}kn_0} X[k] \quad (50)$$

A.3.3 3. 频移性质

$$\mathcal{F}\{e^{j\frac{2\pi}{N}k_0n} x[n]\} = X[k - k_0] \quad (51)$$

A.3.4 4. 卷积定理

时域卷积对应频域相乘：

$$\mathcal{F}\{x[n] * h[n]\} = X[k] \cdot H[k] \quad (52)$$

频域卷积对应时域相乘：

$$\mathcal{F}\{x[n] \cdot h[n]\} = \frac{1}{N} X[k] * H[k] \quad (53)$$

A.3.5 5. Parseval 定理

$$\sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2 \quad (54)$$

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

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：

- 对信号进行 FFT： $X[k] = \text{FFT}\{x[n]\}$
- 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:

$$X_{\text{shifted}}[k] = \frac{1}{2}[X[(k + \Delta k) \bmod N] + X[(k - \Delta k) \bmod N]] \quad (55)$$

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) \rightarrow 0 (二进制: 000)
- 索引 1 (二进制: 001) \rightarrow 4 (二进制: 100)
- 索引 2 (二进制: 010) \rightarrow 2 (二进制: 010)
- 索引 3 (二进制: 011) \rightarrow 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 濾波器简介

B.1 Butterworth 濾波器的基本概念

Butterworth 濾波器是一种最大平坦幅度濾波器 (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 模拟 Butterworth 濾波器

B.2.1 幅频响应

n 阶模拟 Butterworth 低通濾波器的幅频响应为：

$$|H_a(j\omega)|^2 = \frac{1}{1 + (\omega/\omega_c)^{2n}} \quad (56)$$

其中：

- ω_c : 截止频率 (-3dB 频率)
- n : 濾波器阶数
- 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.2.2 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：

$$\left. \frac{d^k |H_a(j\omega)|^2}{d\omega^k} \right|_{\omega=0} = 0, \quad k = 1, 2, \dots, 2n - 1 \quad (57)$$

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.2.3 极点分布

Butterworth 濾波器的极点均匀分布在单位圆上，满足：

$$s_k = \omega_c e^{j\theta_k}, \quad \theta_k = \frac{\pi(2k + n - 1)}{2n}, \quad k = 1, 2, \dots, n \quad (58)$$

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

$n = 8$ 时，极点角度为：

$$\theta_k = \frac{\pi(2k+7)}{16}, \quad k = 1, 2, \dots, 8 \quad (59)$$

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

B.2.4 传递函数

模拟 Butterworth 低通滤波器的传递函数为：

$$H_a(s) = \frac{\omega_c^n}{\prod_{k=1}^n (s - s_k)} \quad (60)$$

分母是 Butterworth 多项式，具有特殊的递归结构。

B.3 数字 Butterworth 滤波器设计

B.3.1 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：

$$s = \frac{2}{T} \frac{1 - z^{-1}}{1 + z^{-1}} = \frac{2f_s(1 - z^{-1})}{1 + z^{-1}} \quad (61)$$

其中 $T = 1/f_s$ 为采样周期， f_s 为采样频率。

频率映射关系：

$$\omega = \frac{2}{T} \tan\left(\frac{\Omega}{2}\right) = 2f_s \tan\left(\frac{\pi f}{f_s}\right) \quad (62)$$

其中 $\Omega = 2\pi f/f_s$ 为数字角频率。

B.3.2 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：

$$\omega_c = 2f_s \tan\left(\frac{\pi f_c}{f_s}\right) \quad (63)$$

本项目参数：

- 高通滤波器： $f_c = 3225.1032 \text{ Hz}, \omega_c = 2 \times 22050 \times \tan(\pi \times 3225.1032 / 22050) \approx 21475.89 \text{ rad/s}$
- 低通滤波器： $f_c = 4000 \text{ Hz}, \omega_c = 2 \times 22050 \times \tan(\pi \times 4000 / 22050) \approx 26682.17 \text{ rad/s}$

B.3.3 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：

$$s_{LP} \rightarrow \frac{\omega_c^2}{s_{HP}} \quad (64)$$

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

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：

$$H_{HP}(z) = H_{LP}(-z) \quad (65)$$

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：

$$b'_i = (-1)^i b_i, \quad a'_i = (-1)^i a_i \quad (66)$$

3. 这样得到的高通滤波器截止频率为 f_c

频谱反转法的优势：

- 避免模拟域的 $s \rightarrow \omega_c^2/s$ 变换，减少频率扭曲
- 实现简单，只需修改系数符号
- 数值稳定性好，截止频率精度高（误差 < 0.01%）

B.4 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：

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_n z^{-n}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_n z^{-n}} \quad (67)$$

归一化使 $a_0 = 1$ ：

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_n z^{-n}}{1 + a_1 z^{-1} + \cdots + a_n z^{-n}} \quad (68)$$

对于 8 阶滤波器， b 和 a 分别有 9 个系数。

B.5 频率响应

数字滤波器的频率响应为：

$$H(e^{j\omega}) = H(z)|_{z=e^{j\omega}} = \frac{\sum_{k=0}^n b_k e^{-j\omega k}}{\sum_{k=0}^n a_k e^{-j\omega k}} \quad (69)$$

其中 $\omega = 2\pi f/f_s$ 为归一化角频率。

幅频响应：

$$|H(e^{j\omega})| = \left| \frac{\sum_{k=0}^n b_k e^{-j\omega k}}{\sum_{k=0}^n a_k e^{-j\omega k}} \right| \quad (70)$$

相频响应：

$$\angle H(e^{j\omega}) = \arg \left(\frac{\sum_{k=0}^n b_k e^{-j\omega k}}{\sum_{k=0}^n a_k e^{-j\omega k}} \right) \quad (71)$$

B.6 本项目中的 Butterworth 滤波器

B.6.1 设计参数

高通滤波器：

- 阶数： $n = 8$
- 截止频率： $f_c = 3225.1032$ Hz (估计的载波频率 f_d)
- 归一化截止频率： $\omega_c = 2\pi f_c/f_s = 0.9186$ 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.6.2 性能特点

1. 通带平坦

- 通带内增益接近 1 (0dB)，无纹波
- 在 $f = 0$ (低通) 或 $f = f_s/2$ (高通) 处增益最大

2. 单调过渡

- 从通带到阻带单调递减
- 8 阶滤波器衰减率: 48 dB/octave

3. 相位特性

- 非线性相位, 存在相位失真
- 群延迟在截止频率附近较大
- 不适合要求严格相位线性的应用

4. 稳定性

- 所有极点位于单位圆内, 系统稳定
- IIR 结构, 反馈系数 a_k 决定稳定性

B.7 IIR 滤波器实现

B.7.1 Direct Form II 结构

Direct Form II 是最节省存储空间的 IIR 滤波器实现结构, 状态方程为:

$$w[n] = x[n] - \sum_{k=1}^n a_k w[n-k] \quad (72)$$

$$y[n] = \sum_{k=0}^n b_k w[n-k] \quad (73)$$

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

B.7.2 数值稳定性

量化误差:

- 系数量化影响极点位置, 可能导致不稳定
- 使用双精度浮点数 (64 位) 减少量化误差

溢出问题:

- 递归计算可能导致数值溢出
- 需要适当缩放输入或中间结果

表 9: 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

B.8 与其他滤波器的比较

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.9 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 10: 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： $\Delta f = f_s/N = 0.7053 \text{ Hz}$
- 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_comparison.png - 波形对比图

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

```
// 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 =====
```

```

46     println!("步骤 3: 绘制频谱图...");  

47  

48     // 绘制全频谱  

49     SpectrumVisualizer::plot_spectrum(  

50         frequencies,  

51         magnitude,  

52         "output/Q1_spectrum_full.png",  

53         "Spectrum of Misedemodulated Signal (Full)",  

54         Some(sample_rate / 2.0),  

55     )?  

56  

57     // 绘制低频段频谱 (0-10 kHz)  

58     SpectrumVisualizer::plot_spectrum(  

59         frequencies,  

60         magnitude,  

61         "output/Q1_spectrum_lowfreq.png",  

62         "Spectrum of Misedemodulated Signal (0-4 kHz)",  

63         Some(4000.0),  

64     )?  

65  

66     // 绘制 dB 刻度的频谱  

67     SpectrumVisualizer::plot_spectrum_db(  

68         frequencies,  

69         &magnitude_db,  

70         "output/Q1_spectrum_db.png",  

71         "Spectrum of Misedemodulated Signal (dB scale)",  

72         Some(10000.0),  

73     )?  

74  

75     // // 绘制时域波形 (前 0.1 秒)  

76     // let samples_to_plot = (sample_rate * 0.1) as usize;  

77     // SpectrumVisualizer::plot_waveform(  

78     //     &samples,  

79     //     sample_rate,  

80     //     "output/Q1_waveform.png",  

81     //     "Waveform of Misedemodulated Signal (First 0.1s)",  

82     //     Some(samples_to_plot),  

83     // )?  

84     // 绘制时域波形  

85     let audio_duration = num_samples as f64 / sample_rate;  

86     let samples_to_plot = (sample_rate * audio_duration) as usize;  

87     SpectrumVisualizer::plot_waveform(  

88         &samples,  

89         sample_rate,  

90         "output/Q1_waveform.png",  

91         "Waveform of Misedemodulated Signal",  

92         Some(samples_to_plot),  

93     )?  

94  

95     println!();  

96  

97     // ===== 步骤 4: 频率偏差估计 =====  

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

99  

100    // 基本频率估计 (排除直流, 搜索 10 Hz 到 10 kHz)  

101    let (f_d, peak_mag, peak_idx) = FrequencyEstimator::estimate_frequency_offset(

```

```

102     frequencies,
103     magnitude,
104     (10.0, 10000.0),
105     true, // 排除直流分量
106   );
107
108 // 精确频率估计 (使用抛物线插值)
109 let f_d_refined = FrequencyEstimator::refined_frequency_estimate(
110   frequencies,
111   magnitude,
112   peak_idx,
113 );
114
115 // 寻找多个峰值
116 println!();
117 let threshold = magnitude[peak_idx] * 0.1; // 设置阈值为主峰的 10%
118 let peaks = FrequencyEstimator::find_multiple_peaks(
119   frequencies,
120   magnitude,
121   5, // 最多找 5 个峰值
122   20, // 最小间隔 20 个采样点
123   threshold,
124 );
125
126 // 计算能量分布
127 let energy_bands = vec![
128   (0.0, 1000.0),
129   (1000.0, 4000.0),
130   (4000.0, 8000.0),
131   (8000.0, sample_rate / 2.0),
132 ];
133 FrequencyEstimator::compute_energy_distribution(
134   magnitude,
135   frequencies,
136   &energy_bands,
137 );
138
139 // 分析频率关系
140 FrequencyEstimator::analyze_frequency_relationship(
141   frequencies,
142   magnitude,
143   f_d_refined,
144 );
145
146 // ===== 结果总结 =====
147 println!("=====");
148 println!("分析结果总结");
149 println!("=====");
150 println!("1. 估计的频率偏差:");
151 println!("    f_d {:.2} Hz (基本估计)", f_d);
152 println!("    f_d {:.4} Hz (精确估计)", f_d_refined);
153 println!();
154 println!("2. 关于 f_c_tilde 与 f_c 的大小关系:");
155 println!("    - 仅从幅度谱无法唯一确定 f_c_tilde > f_c 还是 f_c_tilde < f_c");
156 println!("    - 原因: AM 信号的频谱具有对称性");
157 println!("    - 无论 f_c_tilde > f_c 还是 f_c_tilde < f_c, ");

```

```

158     println!("      错误解调后的信号频谱形态相同");
159     println!();
160     println!("3. 对解调结果的影响:");
161     println!("  - 频率偏差的符号不影响二次解调的效果");
162     println!("  - 因为我们使用的是  $|f_c_{\tilde{}} - f_c| = f_d$ ");
163     println!("  - 二次解调时使用  $\cos(2 f_d \cdot t)$ , 无论符号如何都能正确解调");
164     println!();
165     println!("4. 所有图形已保存到 output 目录:");
166     println!("  - Q1_spectrum_full.png: 全频段频谱");
167     println!("  - Q1_spectrum_lowfreq.png: 低频段频谱 (0-10 kHz)");
168     println!("  - Q1_spectrum_db.png: dB 刻度频谱");
169     println!("  - Q1_waveform.png: 时域波形");
170     println!("=====\\n");
171
172     // 保存关键数据供后续使用
173     save_results_for_q2(f_d_refined, sample_rate)?;
174
175     Ok(())
176 }
177
178 /// 保存结果供 Q2 使用
179 fn save_results_for_q2(f_d: f64, sample_rate: f64) -> Result<(), Box<dyn Error>> {
180     use std::fs;
181     use std::io::Write;
182
183     fs::create_dir_all("output")?;
184     let mut file = fs::File::create("output/Q1_results.txt")?;
185
186     writeln!(file, "Q1 分析结果")?;
187     writeln!(file, "=====");
188     writeln!(file, "频率偏差 f_d = {:.4} Hz", f_d)?;
189     writeln!(file, "采样率 f_s = {:.2} Hz", sample_rate)?;
190     writeln!(file, "基带带宽 f_B = 4000 Hz")?;
191
192     println!("结果已保存到 output/Q1_results.txt");
193
194     Ok(())
195 }
```

音频读取模块 (audio_reader.rs) Listing 2: 音频文件读取 - WAV 格式解析

```

1 // 1. 音频文件读取模块
2 // 负责读取 WAV 文件并提取采样数据、采样率和样本数
3
4 use hound::{WavReader, WavSpec};
5 use std::path::Path;
6
7 /// 音频数据结构
8 #[derive(Debug, Clone)]
9 pub struct AudioData {
10     /// 采样数据 (归一化为浮点数)
11     pub samples: Vec<f64>,
12     /// 采样率 (Hz)
13     pub sample_rate: u32,
14     /// 样本数
```

```

15     pub num_samples: usize,
16     /// WAV 文件规格
17     pub spec: WavSpec,
18 }
19
20 impl AudioData {
21     /// 从 WAV 文件读取音频数据
22     pub fn from_wav<P: AsRef<Path>>(path: P) -> Result<Self, Box<dyn
23         std::error::Error>> {
24         let mut reader = WavReader::open(path)?;
25         let spec = reader.spec();
26         let sample_rate = spec.sample_rate;
27
28         // 读取所有采样点并归一化
29         let samples: Vec<f64> = match spec.sample_format {
30             hound::SampleFormat::Float => {
31                 reader
32                     .samples::<f32>()
33                     .map(|s| s.unwrap() as f64)
34                     .collect()
35             }
36             hound::SampleFormat::Int => {
37                 let max_value = (1 << (spec.bits_per_sample - 1)) as f64;
38                 reader
39                     .samples::<i32>()
40                     .map(|s| s.unwrap() as f64 / max_value)
41                     .collect()
42             }
43         };
44
45         let num_samples = samples.len();
46
47         println!("音频文件读取成功：");
48         println!(" 采样率: {} Hz", sample_rate);
49         println!(" 样本数: {}", num_samples);
50         println!(" 位深度: {} bits", spec.bits_per_sample);
51         println!(" 声道数: {}", spec.channels);
52         println!(" 时长: {:.2} 秒", num_samples as f64 / sample_rate as f64);
53
54         Ok(AudioData {
55             samples,
56             sample_rate,
57             num_samples,
58             spec,
59         })
60     }
61
62     /// 获取信号时长 (秒)
63     pub fn duration(&self) -> f64 {
64         self.num_samples as f64 / self.sample_rate as f64
65     }
66
67     /// 获取单声道数据 (如果是立体声则转换为单声道)
68     pub fn to_mono(&self) -> Vec<f64> {
69         if self.spec.channels == 1 {
70             self.samples.clone()

```

```

70     } else {
71         // 立体声转单声道: 取平均
72         self.samples
73             .chunks(self.spec.channels as usize)
74             .map(|chunk| chunk.iter().sum::<f64>() / chunk.len() as f64)
75             .collect()
76     }
77 }
78
79 /// 保存为 WAV 文件
80 pub fn save_wav<P: AsRef<Path>>(
81     &self,
82     path: P,
83     samples: &[f64],
84 ) -> Result<(), Box<dyn std::error::Error>> {
85     let spec = hound::WavSpec {
86         channels: 1,
87         sample_rate: self.sample_rate,
88         bits_per_sample: 16,
89         sample_format: hound::SampleFormat::Int,
90     };
91
92     let mut writer = hound::WavWriter::create(path, spec)?;
93
94     // 归一化并转换为 i16
95     let max_amplitude = samples.iter().map(|&x| x.abs()).fold(0.0f64, f64::max);
96     let scale = if max_amplitude > 0.0 {
97         32767.0 / max_amplitude
98     } else {
99         32767.0
100    };
101
102    for &sample in samples {
103        let sample_i16 = (sample * scale).clamp(-32768.0, 32767.0) as i16;
104        writer.write_sample(sample_i16)?;
105    }
106
107    writer.finalize()?;
108    println!("音频文件保存成功");
109    Ok(())
110 }
111 }
112
113 #[cfg(test)]
114 mod tests {
115     use super::*;

116
117     #[test]
118     fn test_audio_reader() {
119         // 测试读取音频文件
120         let result = AudioData::from_wav("../project.wav");
121         assert!(result.is_ok());
122
123         if let Ok(audio) = result {
124             assert!(audio.sample_rate > 0);
125             assert!(audio.num_samples > 0);

```

```

126         assert_eq!(audio.samples.len(), audio.num_samples);
127     }
128 }
129 }
```

FFT 处理模块 (fft_processor.rs)

```

1 // 2. FFT 计算模块
2 // 使用 rustfft 库对音频信号进行快速傅里叶变换
3
4 use rustfft::{FftPlanner, num_complex::Complex};
5 use std::f64::consts::PI;
6
7 /// FFT 结果结构
8 #[derive(Debug, Clone)]
9 pub struct FftResult {
10     /// 频谱复数数据
11     pub spectrum: Vec<Complex<f64>>,
12     /// 频率轴 (Hz)
13     pub frequencies: Vec<f64>,
14     /// 幅度谱
15     pub magnitude: Vec<f64>,
16     /// 相位谱
17     pub phase: Vec<f64>,
18     /// 采样率
19     pub sample_rate: f64,
20 }
21
22 impl FftResult {
23     /// 计算信号的 FFT
24     pub fn compute(samples: &[f64], sample_rate: f64) -> Self {
25         let n = samples.len();
26         let mut planner = FftPlanner::new();
27         let fft = planner.plan_fft_forward(n);
28
29         // 将实数信号转换为复数
30         let mut buffer: Vec<Complex<f64>> = samples
31             .iter()
32             .map(|&x| Complex::new(x, 0.0))
33             .collect();
34
35         // 执行 FFT
36         fft.process(&mut buffer);
37
38         // 计算频率轴
39         let frequencies: Vec<f64> = (0..n)
40             .map(|k| k as f64 * sample_rate / n as f64)
41             .collect();
42
43         // 计算幅度谱 (归一化)
44         let magnitude: Vec<f64> = buffer
45             .iter()
46             .map(|c| c.norm() / n as f64)
47             .collect();
48 }
```

```

49     // 计算相位谱
50     let phase: Vec<f64> = buffer
51         .iter()
52         .map(|c| c.arg())
53         .collect();
54
55     println!("FFT 计算完成:");
56     println!("    FFT 点数: {}", n);
57     println!("    频率分辨率: {:.2} Hz", sample_rate / n as f64);
58
59     FftResult {
60         spectrum: buffer,
61         frequencies,
62         magnitude,
63         phase,
64         sample_rate,
65     }
66 }
67
68 /// 执行逆 FFT
69 pub fn ifft(spectrum: &[Complex<f64>]) -> Vec<f64> {
70     let n = spectrum.len();
71     let mut planner = FftPlanner::new();
72     let ifft = planner.plan_fft_inverse(n);
73
74     let mut buffer = spectrum.to_vec();
75     ifft.process(&mut buffer);
76
77     // 提取实部并归一化
78     buffer
79         .iter()
80         .map(|c| c.re / n as f64)
81         .collect()
82 }
83
84 /// 获取单边频谱 (0 到 Nyquist 频率)
85 pub fn get_single_sided(&self) -> (Vec<f64>, Vec<f64>) {
86     let nyquist_index = self.frequencies.len() / 2;
87     let freqs = self.frequencies[..=nyquist_index].to_vec();
88     let mags = self.magnitude[..=nyquist_index].to_vec();
89     (freqs, mags)
90 }
91
92 /// 获取 dB 刻度的幅度谱
93 pub fn get_magnitude_db(&self) -> Vec<f64> {
94     self.magnitude
95         .iter()
96         .map(|&m| {
97             if m > 1e-10 {
98                 20.0 * m.log10()
99             } else {
100                 -200.0
101             }
102         })
103         .collect()
104 }

```

```

105
106     /// 应用窗函数 (Hanning 窗)
107     pub fn apply_hanning_window(samples: &[f64]) -> Vec<f64> {
108         let n = samples.len();
109         samples
110             .iter()
111             .enumerate()
112             .map(|(i, &x)| {
113                 let window = 0.5 * (1.0 - (2.0 * PI * i as f64 / (n - 1) as
114                                         f64).cos());
115                 x * window
116             })
117             .collect()
118     }
119
120     /// 应用窗函数 (Hamming 窗)
121     pub fn apply_hamming_window(samples: &[f64]) -> Vec<f64> {
122         let n = samples.len();
123         samples
124             .iter()
125             .enumerate()
126             .map(|(i, &x)| {
127                 let window = 0.54 - 0.46 * (2.0 * PI * i as f64 / (n - 1) as
128                                         f64).cos();
129                 x * window
130             })
131             .collect()
132     }
133
134     /// 频谱搬移 (循环移位)
135     pub fn circshift(spectrum: &[Complex<f64>], shift: isize) -> Vec<Complex<f64>> {
136         let n = spectrum.len();
137         let shift = shift.rem_euclid(n as isize) as usize;
138
139         let mut result = vec![Complex::new(0.0, 0.0); n];
140         for i in 0..n {
141             result[(i + shift) % n] = spectrum[i];
142         }
143         result
144     }
145
146     /// 计算频域搬移后的和 (用于解调)
147     pub fn frequency_shift_and_add(
148         spectrum: &[Complex<f64>],
149         shift_hz: f64,
150         sample_rate: f64,
151     ) -> Vec<Complex<f64>> {
152         let n = spectrum.len();
153         let shift_bins = (shift_hz * n as f64 / sample_rate).round() as isize;
154
155         // 正向搬移和负向搬移
156         let shifted_pos = circshift(spectrum, shift_bins);
157         let shifted_neg = circshift(spectrum, -shift_bins);
158
159         // 相加并除以 2

```

```

159     shifted_pos
160         .iter()
161         .zip(shifted_neg.iter())
162         .map(|(a, b)| (a + b) / 2.0)
163         .collect()
164     }
165
166 #[cfg(test)]
167 mod tests {
168     use super::*;

169     #[test]
170     fn test_fft() {
171         // 测试 FFT 和 IFFT
172         let sample_rate = 1000.0;
173         let duration = 1.0;
174         let n = (sample_rate * duration) as usize;
175
176         // 生成测试信号: 10 Hz 正弦波
177         let samples: Vec<f64> = (0..n)
178             .map(|i| (2.0 * PI * 10.0 * i as f64 / sample_rate).sin())
179             .collect();
180
181         let fft_result = FftResult::compute(&samples, sample_rate);
182         assert_eq!(fft_result.magnitude.len(), n);
183         assert_eq!(fft_result.frequencies.len(), n);
184
185         // 测试 IFFT
186         let reconstructed = FftResult::ifft(&fft_result.spectrum);
187         assert_eq!(reconstructed.len(), n);
188
189         // 验证重构误差
190         let error: f64 = samples
191             .iter()
192             .zip(reconstructed.iter())
193             .map(|(a, b)| (a - b).powi(2))
194             .sum::<f64>()
195             .sqrt()
196             / n as f64;
197         assert!(error < 1e-10);
198     }
199
200     #[test]
201     fn test_circshift() {
202         let data: Vec<Complex<f64>> = (0..5)
203             .map(|i| Complex::new(i as f64, 0.0))
204             .collect();
205
206         let shifted = circshift(&data, 2);
207         assert_eq!(shifted[0].re, 3.0);
208         assert_eq!(shifted[1].re, 4.0);
209         assert_eq!(shifted[2].re, 0.0);
210     }
211 }

```

频率估计模块 (frequency_estimator.rs) Listing 4: 载波频率估计 - 峰值检测算法

```
1 // 4. 频率偏差估计模块
2 // 通过分析频谱找出频率偏差 f_d
3
4 /// 频率偏差估计器
5 pub struct FrequencyEstimator;
6
7 impl FrequencyEstimator {
8     /// 估计频率偏差 f_d
9     ///
10    /// # 参数
11    /// - frequencies: 频率轴
12    /// - magnitude: 幅度谱
13    /// - search_range: 搜索范围 (Hz), 例如 (10.0, 10000.0)
14    /// - exclude_dc: 是否排除直流分量
15    ///
16    /// # 返回
17    /// (peak_frequency, peak_magnitude, peak_index)
18    pub fn estimate_frequency_offset(
19        frequencies: &[f64],
20        magnitude: &[f64],
21        search_range: (f64, f64),
22        exclude_dc: bool,
23    ) -> (f64, f64, usize) {
24        let (min_freq, max_freq) = search_range;
25
26        // 在指定范围内搜索峰值
27        let mut peak_magnitude = 0.0;
28        let mut peak_index = 0;
29        let mut peak_frequency = 0.0;
30
31        let start_idx = if exclude_dc { 1 } else { 0 };
32
33        for (i, (&freq, &mag)) in frequencies
34            .iter()
35            .zip(magnitude.iter())
36            .enumerate()
37            .skip(start_idx)
38        {
39            if freq >= min_freq && freq <= max_freq && mag > peak_magnitude {
40                peak_magnitude = mag;
41                peak_index = i;
42                peak_frequency = freq;
43            }
44        }
45
46        println!("频率偏差估计结果:");
47        println!("  峰值频率 f_d = {:.2} Hz", peak_frequency);
48        println!("  峰值幅度 = {:.6}", peak_magnitude);
49        println!("  峰值索引 = {}", peak_index);
50
51        (peak_frequency, peak_magnitude, peak_index)
52    }
53
54    /// 精确估计频率 (使用抛物线插值)
55    pub fn refined_frequency_estimate(
```

```

56     frequencies: &[f64],
57     magnitude: &[f64],
58     peak_index: usize,
59 ) -> f64 {
60     if peak_index == 0 || peak_index >= magnitude.len() - 1 {
61         return frequencies[peak_index];
62     }
63
64     // 使用三点抛物线插值
65     let y1 = magnitude[peak_index - 1];
66     let y2 = magnitude[peak_index];
67     let y3 = magnitude[peak_index + 1];
68
69     // 抛物线顶点位置
70     let delta = 0.5 * (y1 - y3) / (y1 - 2.0 * y2 + y3);
71
72     let freq_resolution = if frequencies.len() > 1 {
73         frequencies[1] - frequencies[0]
74     } else {
75         1.0
76     };
77
78     let refined_freq = frequencies[peak_index] + delta * freq_resolution;
79
80     println!("精确频率估计:");
81     println!(" 原始峰值频率: {:.2} Hz", frequencies[peak_index]);
82     println!(" 精确频率: {:.4} Hz", refined_freq);
83
84     refined_freq
85 }
86
87     /// 寻找多个峰值
88 pub fn find_multiple_peaks(
89     frequencies: &[f64],
90     magnitude: &[f64],
91     num_peaks: usize,
92     min_distance: usize,
93     threshold: f64,
94 ) -> Vec<(f64, f64, usize)> {
95     let mut peaks = Vec::new();
96     let n = magnitude.len();
97
98     // 找出所有局部最大值
99     for i in 1..n-1 {
100         if magnitude[i] > magnitude[i-1]
101             && magnitude[i] > magnitude[i+1]
102             && magnitude[i] > threshold
103         {
104             peaks.push((frequencies[i], magnitude[i], i));
105         }
106     }
107
108     // 按幅度降序排序
109     peaks.sort_by(|a, b| b.1.partial_cmp(&a.1).unwrap());
110
111     // 移除距离太近的峰值

```

```

112     let mut filtered_peaks = Vec::new();
113     for peak in peaks {
114         let is_far_enough = filtered_peaks
115             .iter()
116             .all(|(_, _, idx)| (*idx as isize - peak.2 as isize).abs() >=
117                 min_distance as isize);
118
119         if is_far_enough {
120             filtered_peaks.push(peak);
121             if filtered_peaks.len() >= num_peaks {
122                 break;
123             }
124         }
125
126         println!("找到 {} 个峰值:", filtered_peaks.len());
127         for (i, (freq, mag, idx)) in filtered_peaks.iter().enumerate() {
128             println!(" 峰值 {}: 频率 = {:.2} Hz, 幅度 = {:.6}, 索引 = {}",
129                     i + 1, freq, mag, idx);
130         }
131
132         filtered_peaks
133     }
134
135     /// 计算信号的能量分布
136     pub fn compute_energy_distribution(
137         magnitude: &[f64],
138         frequencies: &[f64],
139         bands: &[(f64, f64)], // 频带范围
140     ) -> Vec<(String, f64)> {
141         let total_energy: f64 = magnitude.iter().map(|&m| m * m).sum();
142
143         let mut band_energies = Vec::new();
144
145         for (low, high) in bands {
146             let energy: f64 = frequencies
147                 .iter()
148                 .zip(magnitude.iter())
149                 .filter(|(&f, _)| f >= *low && f <= *high)
150                 .map(|(_ , &m)| m * m)
151                 .sum();
152
153             let percentage = (energy / total_energy) * 100.0;
154             band_energies.push((
155                 format!("{}-{} Hz", low, high),
156                 percentage,
157             ));
158         }
159
160         println!("\n能量分布:");
161         for (band, percent) in &band_energies {
162             println!("  {}: {:.2}%", band, percent);
163         }
164
165         band_energies
166     }

```

```

167
168     /// 判断 f_c_tilde 与 f_c 的大小关系
169     ///
170     /// 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
171     pub fn analyze_frequency_relationship(
172         frequencies: &[f64],
173         magnitude: &[f64],
174         f_d: f64,
175     ) -> String {
176         println!("\\n频率关系分析:");
177         println!("  估计的频率偏差 f_d = {:.2} Hz", f_d);
178         println!("  仅从幅度谱无法唯一确定 f_c_tilde > f_c 还是 f_c_tilde < f_c");
179         println!("  原因：频谱的对称性使得两种情况产生相同的幅度谱");
180         println!("  需要：相位信息、时域分析或其他先验知识来确定符号");
181
182         String::from("无法仅从幅度谱确定频率偏差的符号")
183     }
184
185     /// 计算信噪比 (SNR) 估计
186     pub fn estimate_snr(
187         magnitude: &[f64],
188         signal_band: (usize, usize),
189         noise_band: (usize, usize),
190     ) -> f64 {
191         let signal_power: f64 = magnitude[signal_band.0..signal_band.1]
192             .iter()
193             .map(|&m| m * m)
194             .sum::<f64>() / (signal_band.1 - signal_band.0) as f64;
195
196         let noise_power: f64 = magnitude=noise_band.0..noise_band.1]
197             .iter()
198             .map(|&m| m * m)
199             .sum::<f64>() / (noise_band.1 - noise_band.0) as f64;
200
201         let snr_db = if noise_power > 0.0 {
202             10.0 * (signal_power / noise_power).log10()
203         } else {
204             f64::INFINITY
205         };
206
207         println!("\\n信噪比估计:");
208         println!("  信号功率: {:.6}", signal_power);
209         println!("  噪声功率: {:.6}", noise_power);
210         println!("  SNR: {:.2} dB", snr_db);
211
212         snr_db
213     }
214 }
215
216 #[cfg(test)]
217 mod tests {
218     use super::*;
219     use std::f64::consts::PI;
220
221     #[test]
222     fn test_frequency_estimation() {

```

```

223     // 生成测试信号：包含 100 Hz 的峰值
224     let n = 1000;
225     let sample_rate = 1000.0;
226     let frequencies: Vec<f64> = (0..n).map(|i| i as f64 * sample_rate / n as
227         f64).collect();
228
229     let magnitude: Vec<f64> = frequencies
230         .iter()
231         .map(|&f| {
232             if (f - 100.0).abs() < 1.0 {
233                 1.0
234             } else {
235                 0.01
236             }
237         })
238         .collect();
239
240     let (peak_freq, peak_mag, peak_idx) =
241         FrequencyEstimator::estimate_frequency_offset(
242             &frequencies,
243             &magnitude,
244             (10.0, 500.0),
245             true,
246         );
247
248     assert!((peak_freq - 100.0).abs() < 2.0);
249     assert!(peak_mag > 0.5);
250     assert!(peak_idx > 0);
251 }
252
253 #[test]
254 fn test_multiple_peaks() {
255     let n = 1000;
256     let sample_rate = 1000.0;
257     let frequencies: Vec<f64> = (0..n).map(|i| i as f64 * sample_rate / n as
258         f64).collect();
259
260     // 生成包含两个峰值的信号
261     let magnitude: Vec<f64> = frequencies
262         .iter()
263         .map(|&f| {
264             let peak1 = (-((f - 100.0) / 10.0).powi(2)).exp();
265             let peak2 = 0.8 * (-((f - 300.0) / 10.0).powi(2)).exp();
266             peak1 + peak2
267         })
268         .collect();
269
270     let peaks = FrequencyEstimator::find_multiple_peaks(
271         &frequencies,
272         &magnitude,
273         2,
274         20,
275         0.1,
276     );
277
278     assert_eq!(peaks.len(), 2);

```

```
276     }
277 }
```

频谱可视化模块 (`spectrum_visualizer.rs`) 使用 `plotters` 库

```
1 // 3. 频谱可视化模块
2 // 使用 plotters 库绘制频谱图
3
4 use plotters::prelude::*;
5 use std::path::Path;
6
7 /// 频谱可视化器
8 pub struct SpectrumVisualizer;
9
10 impl SpectrumVisualizer {
11     /// 绘制频谱图 (幅度谱)
12     pub fn plot_spectrum<P: AsRef<Path>>(
13         frequencies: &[f64],
14         magnitude: &[f64],
15         output_path: P,
16         title: &str,
17         max_freq: Option<f64>,
18     ) -> Result<(), Box<dyn std::error::Error>> {
19         // 只显示到指定频率或 Nyquist 频率
20         let nyquist = frequencies.last().copied().unwrap_or(0.0) / 2.0;
21         let max_f = max_freq.unwrap_or(nyquist);
22
23         // 过滤数据点
24         let data: Vec<(f64, f64)> = frequencies
25             .iter()
26             .zip(magnitude.iter())
27             .filter(|(&f, _)| f <= max_f)
28             .map(|(&f, &m)| (f, m))
29             .collect();
30
31         if data.is_empty() {
32             return Err("没有数据可以绘制".into());
33         }
34
35         // 找出幅度范围
36         let max_magnitude = data.iter().map(|(_, m)| m).fold(0.0f64, |a, &b|
37             a.max(b));
38         let y_max = max_magnitude * 1.1;
39
40         // 创建绘图区域 - 使用文件路径
41         let root = BitMapBackend::new(output_path.as_ref(), (1200, 600))
42             .into_drawing_area();
43         root.fill(&WHITE)?;
44
45         let mut chart = ChartBuilder::on(&root)
46             .caption(title, ("Arial", 30).into_font())
47             .margin(10)
48             .x_label_area_size(40)
49             .y_label_area_size(60)
50             .build_cartesian_2d(0.0..max_f, 0.0..y_max)?;
51     }
52 }
```

```

50
51     chart
52         .configure_mesh()
53         .x_desc("Frequency (Hz)")
54         .y_desc("Magnitude")
55         .draw()?;
56
57     // 绘制频谱曲线
58     chart.draw_series(LineSeries::new(
59         data.iter().map(|&(f, m)| (f, m)),
60         &BLUE,
61     ))?;
62
63     root.present()?;
64     println!("频谱图已保存到: {:?}", output_path.as_ref());
65     Ok(())
66 }
67
68 /// 绘制频谱图 (dB 刻度)
69 pub fn plot_spectrum_db<P: AsRef<Path>>(
70     frequencies: &[f64],
71     magnitude_db: &[f64],
72     output_path: P,
73     title: &str,
74     max_freq: Option<f64>,
75 ) -> Result<(), Box<dyn std::error::Error>> {
76     let nyquist = frequencies.last().copied().unwrap_or(0.0) / 2.0;
77     let max_f = max_freq.unwrap_or(nyquist);
78
79     let data: Vec<(f64, f64)> = frequencies
80         .iter()
81         .zip(magnitude_db.iter())
82         .filter(|(&f, _)| f <= max_f)
83         .map(|(&f, &m)| (f, m))
84         .collect();
85
86     if data.is_empty() {
87         return Err("没有数据可以绘制".into());
88     }
89
90     let max_db = data.iter().map(|(_, m)| m).fold(-200.0f64, |a, &b| a.max(b));
91     let min_db = -100.0;
92
93     let root = BitMapBackend::new(output_path.as_ref(), (1200, 600))
94         .into_drawing_area();
95     root.fill(&WHITE)?;
96
97     let mut chart = ChartBuilder::on(&root)
98         .caption(title, ("Arial", 30).into_font())
99         .margin(10)
100        .x_label_area_size(40)
101        .y_label_area_size(60)
102        .build_cartesian_2d(0.0..max_f, min_db..max_db)?;
103
104     chart
105         .configure_mesh()

```

```

106     .x_desc("Frequency (Hz)")
107     .y_desc("Magnitude (dB)")
108     .draw()?;
109
110     chart.draw_series(LineSeries::new(
111         data.iter().map(|&(f, m)| (f, m)),
112         &RED,
113     ))?;
114
115     root.present()?;
116     println!("频谱图 (dB) 已保存到: {:?}", output_path.as_ref());
117     Ok(())
118 }
119
120 /// 绘制时域波形
121 pub fn plot_waveform<P: AsRef<Path>>(
122     samples: &[f64],
123     sample_rate: f64,
124     output_path: P,
125     title: &str,
126     max_samples: Option<usize>,
127 ) -> Result<(), Box<dyn std::error::Error>> {
128     // 限制显示的采样点数量
129     let n = max_samples.unwrap_or(samples.len()).min(samples.len());
130     let time: Vec<f64> = (0..n)
131         .map(|i| i as f64 / sample_rate)
132         .collect();
133
134     let data: Vec<(f64, f64)> = time
135         .iter()
136         .zip(samples.iter())
137         .take(n)
138         .map(|(&t, &s)| (t, s))
139         .collect();
140
141     let max_amplitude = samples.iter().map(|x| x.abs()).fold(0.0f64, f64::max);
142     let y_range = max_amplitude * 1.2;
143
144     let root = BitMapBackend::new(output_path.as_ref(), (1200, 600))
145         .into_drawing_area();
146     root.fill(&WHITE)?;
147
148     let mut chart = ChartBuilder::on(&root)
149         .caption(title, ("Arial", 30).into_font())
150         .margin(10)
151         .x_label_area_size(40)
152         .y_label_area_size(60)
153         .build_cartesian_2d(0.0..time[n-1], -y_range..y_range)?;
154
155     chart
156         .configure_mesh()
157         .x_desc("Time (seconds)")
158         .y_desc("Amplitude")
159         .draw()?;
160
161     chart.draw_series(LineSeries::new(

```

```

162         data.iter().map(|&(t, s)| (t, s)),
163         &GREEN,
164     ))?;
165
166     root.present()?;
167     println!("时域波形图已保存到: {:?}", output_path.as_ref());
168     Ok(())
169 }
170
171 /// 绘制多个频谱对比图
172 pub fn plot_spectrum_comparison<P: AsRef<Path>>(
173     datasets: Vec<(&[f64], &[f64], &str)>, // (frequencies, magnitude, label)
174     output_path: P,
175     title: &str,
176     max_freq: Option<f64>,
177 ) -> Result<(), Box<dyn std::error::Error>> {
178     if datasets.is_empty() {
179         return Err("没有数据可以绘制".into());
180     }
181
182     let nyquist = datasets[0].0.last().copied().unwrap_or(0.0) / 2.0;
183     let max_f = max_freq.unwrap_or(nyquist);
184
185     // 找出最大幅度
186     let max_magnitude = datasets
187         .iter()
188         .flat_map(|(_ , mag, _)| mag.iter())
189         .copied()
190         .fold(0.0f64, f64::max);
191     let y_max = max_magnitude * 1.1;
192
193     let root = BitMapBackend::new(output_path.as_ref(), (1200, 600))
194         .into_drawing_area();
195     root.fill(&WHITE)?;
196
197     let mut chart = ChartBuilder::on(&root)
198         .caption(title, ("Arial", 30).into_font())
199         .margin(10)
200         .x_label_area_size(40)
201         .y_label_area_size(60)
202         .build_cartesian_2d(0.0..max_f, 0.0..y_max)?;
203
204     chart
205         .configure_mesh()
206         .x_desc("Frequency (Hz)")
207         .y_desc("Magnitude")
208         .draw()?;
209
210     let colors = [&BLUE, &RED, &GREEN, &CYAN, &MAGENTA];
211
212     for (idx, (freqs, mags, label)) in datasets.iter().enumerate() {
213         let data: Vec<(f64, f64)> = freqs
214             .iter()
215             .zip(mags.iter())
216             .filter(|(&f, _)| f <= max_f)
217             .map(|(&f, &m)| (f, m))

```

```

218     .collect();
219
220     let color = colors[idx % colors.len()];
221     chart
222         .draw_series(LineSeries::new(data.iter().map(|&(f, m)| (f, m)),
223                                     color))?
224         .label(*label)
225         .legend(move |(x, y)| PathElement::new(vec![(x, y), (x + 20, y)],
226                                         color));
227     }
228
229     chart
230         .configure_series_labels()
231         .background_style(&WHITE.mix(0.8))
232         .border_style(&BLACK)
233         .draw()?;
234
235     root.present()?;
236     println!("对比频谱图已保存到: {:?}", output_path.as_ref());
237     Ok(())
238 }
239 #[cfg(test)]
240 mod tests {
241     use super::*;
242     use std::f64::consts::PI;
243
244     #[test]
245     fn test_plot_spectrum() {
246         // 生成测试数据
247         let n = 1000;
248         let sample_rate = 1000.0;
249         let frequencies: Vec<f64> = (0..n).map(|i| i as f64 * sample_rate / n as
250                                         f64).collect();
251
252         // 模拟频谱
253         let magnitude: Vec<f64> = frequencies
254             .iter()
255             .map(|&f| {
256                 if (f - 100.0).abs() < 10.0 {
257                     1.0
258                 } else {
259                     0.1 * (-((f - 100.0) / 50.0).powi(2)).exp()
260                 }
261             })
262             .collect();
263
264         let result = SpectrumVisualizer::plot_spectrum(
265             &frequencies,
266             &magnitude,
267             "/tmp/test_spectrum.png",
268             "测试频谱",
269             Some(500.0),
270         );

```

```

271     assert!(result.is_ok());
272 }
273 }
```

C.6.2 Q2 - Butterworth 濾波器设计

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

```

1 mod butterworth_filter;
2 mod filter_response;
3 mod response_visualizer;
4
5 use std::fs;
6 use std::path::Path;
7
8 fn main() {
9     println!("==== Q2: Butterworth Filter Design ===\n");
10
11     // Read parameters from Q1 results
12     let q1_results_path = "../Q1/output/Q1_results.txt";
13     let (sample_rate, f_d, f_b) = read_q1_results(q1_results_path);
14
15     println!("Parameters from Q1:");
16     println!("  Sample Rate: {} Hz", sample_rate);
17     println!("  Frequency Offset (f_d): {:.4} Hz", f_d);
18     println!("  Signal Bandwidth (f_B): {} Hz", f_b);
19     println!();
20
21     // Design 8th-order Butterworth filters
22     let order = 8;
23     println!("Designing 8th-order Butterworth filters...");
24
25     // High-pass filter with cutoff frequency f_d
26     println!("  - High-pass filter (cutoff = {:.4} Hz)", f_d);
27     let highpass = butterworth_filter::ButterworthFilter::highpass(order, f_d,
28                     sample_rate);
29
30     // Low-pass filter with cutoff frequency f_B
31     println!("  - Low-pass filter (cutoff = {} Hz)", f_b);
32     let lowpass = butterworth_filter::ButterworthFilter::lowpass(order, f_b,
33                     sample_rate);
34
35     println!("\nHigh-pass filter coefficients:");
36     println!("  b (numerator): {:?}", &highpass.b[..5.min(highpass.b.len())]);
37     println!("  a (denominator): {:?}", &highpass.a[..5.min(highpass.a.len())]);
38
39     println!("\nLow-pass filter coefficients:");
40     println!("  b (numerator): {:?}", &lowpass.b[..5.min(lowpass.b.len())]);
41     println!("  a (denominator): {:?}", &lowpass.a[..5.min(lowpass.a.len())]);
42
43     // Calculate frequency response at the same frequency points as Q1
44     let num_points = 31265; // Same as Q1 audio samples
45     println!("\nCalculating frequency responses ({} points)...", num_points);
```

```

45     let hp_response = filter_response::FilterResponse::compute(&highpass,
46         sample_rate, num_points);
47     let lp_response = filter_response::FilterResponse::compute(&lowpass, sample_rate,
48         num_points);
49
50     // Create output directory
51     let output_dir = "output";
52     fs::create_dir_all(output_dir).expect("Failed to create output directory");
53
54     // Plot frequency responses
55     println!("Generating plots...");
56
57     // High-pass filter magnitude response
58     response_visualizer::plot_magnitude_response(
59         &hp_response.frequencies,
60         &hp_response.magnitude,
61         &format!("{}/Q2_highpass_magnitude.png", output_dir),
62         "High-pass Filter Magnitude Response",
63         Some(10000.0),
64         ).expect("Failed to plot high-pass magnitude");
65
66     // High-pass filter magnitude response in dB
67     response_visualizer::plot_magnitude_response_db(
68         &hp_response.frequencies,
69         &hp_response.magnitude,
70         &format!("{}/Q2_highpass_magnitude_db.png", output_dir),
71         "High-pass Filter Magnitude Response (dB)",
72         Some(10000.0),
73         ).expect("Failed to plot high-pass magnitude dB");
74
75     // High-pass filter phase response
76     response_visualizer::plot_phase_response(
77         &hp_response.frequencies,
78         &hp_response.phase,
79         &format!("{}/Q2_highpass_phase.png", output_dir),
80         "High-pass Filter Phase Response",
81         Some(10000.0),
82         ).expect("Failed to plot high-pass phase");
83
84     // Low-pass filter magnitude response
85     response_visualizer::plot_magnitude_response(
86         &lp_response.frequencies,
87         &lp_response.magnitude,
88         &format!("{}/Q2_lowpass_magnitude.png", output_dir),
89         "Low-pass Filter Magnitude Response",
90         Some(10000.0),
91         ).expect("Failed to plot low-pass magnitude");
92
93     // Low-pass filter magnitude response in dB
94     response_visualizer::plot_magnitude_response_db(
95         &lp_response.frequencies,
96         &lp_response.magnitude,
97         &format!("{}/Q2_lowpass_magnitude_db.png", output_dir),
98         "Low-pass Filter Magnitude Response (dB)",
99         Some(10000.0),
100        ).expect("Failed to plot low-pass magnitude dB");

```

```

99
100    // Low-pass filter phase response
101    response_visualizer::plot_phase_response(
102        &lp_response.frequencies,
103        &lp_response.phase,
104        &format!("{}/Q2_lowpass_phase.png", output_dir),
105        "Low-pass Filter Phase Response",
106        Some(10000.0),
107        ).expect("Failed to plot low-pass phase");
108
109    // Combined magnitude plot
110    response_visualizer::plot_combined_magnitude(
111        &hp_response.frequencies,
112        &hp_response.magnitude,
113        &lp_response.magnitude,
114        &format!("{}/Q2_combined_magnitude.png", output_dir),
115        "Combined Filter Magnitude Responses",
116        Some(10000.0),
117        ).expect("Failed to plot combined magnitude");
118
119    // Save filter coefficients
120    save_filter_coefficients(&highpass, &lowpass,
121        &format!("{}/Q2_filter_coefficients.txt", output_dir));
122
123    // Save frequency response data
124    save_frequency_response(&hp_response, &lp_response,
125        &format!("{}/Q2_frequency_response.txt", output_dir));
126
127    println!("All results saved to '{}/' directory", output_dir);
128    println!("Q2 completed successfully!");
129}
130
131fn read_q1_results(path: &str) -> (f64, f64, f64) {
132    let content = fs::read_to_string(path)
133        .expect("Failed to read Q1 results file");
134
135    let mut sample_rate = 22050.0;
136    let mut f_d = 3225.0;
137    let mut f_b = 4000.0;
138
139    for line in content.lines() {
140        // Parse: "频率偏差 f_d = 3225.1032 Hz"
141        if line.contains("频率偏差") && line.contains("f_d") {
142            if let Some(value_str) = line.split('=').nth(1) {
143                if let Some(num_str) = value_str.trim().split_whitespace().next() {
144                    f_d = num_str.parse().unwrap_or(3225.0);
145                }
146            }
147        // Parse: "采样率 f_s = 22050.00 Hz"
148        } else if line.contains("采样率") && line.contains("f_s") {
149            if let Some(value_str) = line.split('=').nth(1) {
150                if let Some(num_str) = value_str.trim().split_whitespace().next() {
151                    sample_rate = num_str.parse().unwrap_or(22050.0);
152                }
153            }
154        }
155    }
156}
```

```

153     }
154     // Parse: "基带带宽 f_B = 4000 Hz"
155     else if line.contains("基带带宽") && line.contains("f_B") {
156         if let Some(value_str) = line.split('=').nth(1) {
157             if let Some(num_str) = value_str.trim().split_whitespace().next() {
158                 f_b = num_str.parse().unwrap_or(4000.0);
159             }
160         }
161     }
162 }
163
164 (sample_rate, f_d, f_b)
165 }
166
167 fn save_filter_coefficients(highpass: &butterworth_filter::ButterworthFilter,
168                             lowpass: &butterworth_filter::ButterworthFilter,
169                             path: &str) {
170     let mut content = String::new();
171     content.push_str("== Q2: Filter Coefficients ==\n\n");
172
173     content.push_str("High-pass Filter (8th-order Butterworth):\n");
174     content.push_str(&format!("Cutoff Frequency: {:.4} Hz\n", highpass.cutoff));
175     content.push_str(&format!("Sample Rate: {} Hz\n", highpass.sample_rate));
176     content.push_str("\nNumerator Coefficients (b):\n");
177     for (i, coef) in highpass.b.iter().enumerate() {
178         content.push_str(&format!(" b[{}] = {:.15e}\n", i, coef));
179     }
180     content.push_str("\nDenominator Coefficients (a):\n");
181     for (i, coef) in highpass.a.iter().enumerate() {
182         content.push_str(&format!(" a[{}] = {:.15e}\n", i, coef));
183     }
184
185     content.push_str("\n\nLow-pass Filter (8th-order Butterworth):\n");
186     content.push_str(&format!("Cutoff Frequency: {:.4} Hz\n", lowpass.cutoff));
187     content.push_str(&format!("Sample Rate: {} Hz\n", lowpass.sample_rate));
188     content.push_str("\nNumerator Coefficients (b):\n");
189     for (i, coef) in lowpass.b.iter().enumerate() {
190         content.push_str(&format!(" b[{}] = {:.15e}\n", i, coef));
191     }
192     content.push_str("\nDenominator Coefficients (a):\n");
193     for (i, coef) in lowpass.a.iter().enumerate() {
194         content.push_str(&format!(" a[{}] = {:.15e}\n", i, coef));
195     }
196
197     fs::write(path, content).expect("Failed to write filter coefficients");
198 }
199
200 fn save_frequency_response(hp_response: &filter_response::FilterResponse,
201                           lp_response: &filter_response::FilterResponse,
202                           path: &str) {
203     let mut content = String::new();
204     content.push_str("== Q2: Frequency Response Data ==\n\n");
205
206     content.push_str("High-pass Filter:\n");
207     content.push_str(&format!("Number of frequency points: {}\n",
208                           hp_response.frequencies.len()));

```

```

208     content.push_str(&format!("Frequency range: 0 - {:.2} Hz\n",
209         hp_response.frequencies.last().unwrap_or(&0.0)));
210     content.push_str(&format!("Maximum magnitude: {:.6}\n",
211         hp_response.magnitude.iter().cloned().fold(0./0., f64::max)));
212     content.push_str(&format!("Minimum magnitude: {:.6}\n",
213         hp_response.magnitude.iter().cloned().fold(f64::INFINITY, f64::min)));
214
215     content.push_str("\nLow-pass Filter:\n");
216     content.push_str(&format!("Number of frequency points: {}\n",
217         lp_response.frequencies.len()));
218     content.push_str(&format!("Frequency range: 0 - {:.2} Hz\n",
219         lp_response.frequencies.last().unwrap_or(&0.0)));
220     content.push_str(&format!("Maximum magnitude: {:.6}\n",
221         lp_response.magnitude.iter().cloned().fold(0./0., f64::max)));
222     content.push_str(&format!("Minimum magnitude: {:.6}\n",
223         lp_response.magnitude.iter().cloned().fold(f64::INFINITY, f64::min)));
224
225     fs::write(path, content).expect("Failed to write frequency response data");
226 }

```

Listing 7: Butterworth 濾波器设计 - 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) ->
32     (Vec<f64>, Vec<f64>) {
33     // 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 *
36         wc)).collect();
37     bilinear_transform_cascade(&scaled_poles, fs)
38 }
39
40 fn design_butterworth_digital_highpass(order: usize, cutoff: f64, fs: f64) ->
41     (Vec<f64>, Vec<f64>) {
42     // Spectral inversion method: H_HP(z) = H_LP(-z)
43     // The cutoff frequency is mirrored about fs/4
44     // To get highpass with cutoff fc_hp, design lowpass with cutoff (fs/2 - fc_hp)
45     let lp_cutoff = fs / 2.0 - cutoff;
46
47     // Design a lowpass filter at the mirrored cutoff frequency
48     let (b_lp, a_lp) = design_butterworth_digital_lowpass(order, lp_cutoff, fs);
49
50     // Convert lowpass to highpass using spectral inversion: H_HP(z) = H_LP(-z)
51     // This means alternating the signs of coefficients with odd indices
52     let mut b_hp = b_lp.clone();
53     let mut a_hp = a_lp.clone();
54
55     for (i, val) in b_hp.iter_mut().enumerate() {
56         if i % 2 == 1 {
57             *val = -*val;
58         }
59     }
60
61     for (i, val) in a_hp.iter_mut().enumerate() {
62         if i % 2 == 1 {
63             *val = -*val;
64         }
65     }
66 }
67
68 fn butterworth_analog_poles(order: usize) -> Vec<(f64, f64)> {
69     (0..order).map(|k| {
70         let theta = PI * (2.0 * k as f64 + order as f64 + 1.0) / (2.0 * order as f64);
71         (theta.cos(), theta.sin())
72     }).collect()
73 }
74
75 fn bilinear_transform_cascade(poles: &[(f64, f64)], fs: f64) -> (Vec<f64>, Vec<f64>) {
76     let t = 1.0 / fs;
77     let mut b_total = vec![1.0];
78     let mut a_total = vec![1.0];
79     let mut i = 0;
80     while i < poles.len() {
81         let (pr1, pi1) = poles[i];
82         if pi1.abs() < 1e-10 {
83             let denom = 2.0 - pr1 * t;
84             let z_pole = (2.0 + pr1 * t) / denom;
85             b_total = convolve(&b_total, &vec![1.0, 1.0]);
86             a_total = convolve(&a_total, &vec![1.0, -z_pole]);
87     }
88 }

```

```

87         i += 1;
88     } else {
89         if i + 1 < poles.len() {
90             let b_section = vec![1.0, 2.0, 1.0];
91             let denom_re = 2.0 - pr1 * t;
92             let denom_im = -pi1 * t;
93             let denom_mag_sq = denom_re * denom_re + denom_im * denom_im;
94             let z1_re = ((2.0 + pr1 * t) * denom_re + pi1 * t * denom_im) /
95                         denom_mag_sq;
96             let z1_im = ((pi1 * t) * denom_re - (2.0 + pr1 * t) * denom_im) /
97                         denom_mag_sq;
98             let a1 = -2.0 * z1_re;
99             let a2 = z1_re * z1_re + z1_im * z1_im;
100            let a_section = vec![1.0, a1, a2];
101            b_total = convolve(&b_total, &b_section);
102            a_total = convolve(&a_total, &a_section);
103            i += 2;
104        } else {
105            i += 1;
106        }
107    }
108    let a0 = a_total[0];
109    for i in 0..a_total.len() { a_total[i] /= a0; }
110    for i in 0..b_total.len() { b_total[i] /= a0; }
111    let b_sum: f64 = b_total.iter().sum();
112    let a_sum: f64 = a_total.iter().sum();
113    let gain = a_sum / b_sum;
114    for i in 0..b_total.len() { b_total[i] *= gain; }
115    (b_total, a_total)
116}
117 fn convolve(a: &[f64], b: &[f64]) -> Vec<f64> {
118     let mut result = vec![0.0; a.len() + b.len() - 1];
119     for (i, &a_val) in a.iter().enumerate() {
120         for (j, &b_val) in b.iter().enumerate() {
121             result[i + j] += a_val * b_val;
122         }
123     }
124     result
125 }
126
127 fn lowpass_to_highpass(b_lp: &[f64], a_lp: &[f64]) -> (Vec<f64>, Vec<f64>) {
128     let mut b_hp = b_lp.to_vec();
129     let mut a_hp = a_lp.to_vec();
130     for (i, val) in b_hp.iter_mut().enumerate() {
131         if i % 2 == 1 { *val = -*val; }
132     }
133     for (i, val) in a_hp.iter_mut().enumerate() {
134         if i % 2 == 1 { *val = -*val; }
135     }
136     (b_hp, a_hp)
137 }
```

响应可视化模块 (response_visualizer.rs) Listing 8: 频率响应可视化 - 幅频/相频特性

```
1 use plotters::prelude::*;
2 use crate::filter_response;
3
4 const PLOT_WIDTH: u32 = 1200;
5 const PLOT_HEIGHT: u32 = 600;
6
7 /// Plot magnitude response (linear scale)
8 pub fn plot_magnitude_response(
9     frequencies: &[f64],
10    magnitude: &[f64],
11    output_path: &str,
12    title: &str,
13    max_freq: Option<f64>,
14 ) -> Result<(), Box<dyn std::error::Error>> {
15     let root = BitMapBackend::new(output_path, (PLOT_WIDTH,
16                                                 PLOT_HEIGHT)).into_drawing_area();
17     root.fill(&WHITE)?;
18
19     let max_freq_val = max_freq.unwrap_or(*frequencies.last().unwrap_or(&10000.0));
20     let max_mag = magnitude.iter()
21         .zip(frequencies.iter())
22         .filter(|(_,&f)| f <= max_freq_val)
23         .map(|(&m,_)| m)
24         .fold(0.0, f64::max)
25         .max(1.1);
26
27     let mut chart = ChartBuilder::on(&root)
28         .caption(title, ("sans-serif", 30).into_font())
29         .margin(15)
30         .x_label_area_size(50)
31         .y_label_area_size(60)
32         .build_cartesian_2d(0.0..max_freq_val, 0.0..max_mag)?;
33
34     chart.configure_mesh()
35         .x_desc("Frequency (Hz)")
36         .y_desc("Magnitude")
37         .draw()?;
38
39     chart.draw_series(LineSeries::new(
40         frequencies.iter()
41             .zip(magnitude.iter())
42             .filter(|(&f,_)| f <= max_freq_val)
43             .map(|(&f,&m)| (f,m)),
44             &BLUE,
45     ))?;
46
47     root.present()?;
48     Ok(())
49 }
50
51 /// Plot magnitude response in dB scale
52 pub fn plot_magnitude_response_db(
53     frequencies: &[f64],
54     magnitude: &[f64],
55     output_path: &str,
```

```

55     title: &str,
56     max_freq: Option<f64>,
57 ) -> Result<(), Box<dyn std::error::Error>> {
58     let root = BitMapBackend::new(output_path, (PLOT_WIDTH,
59                                   PLOT_HEIGHT)).into_drawing_area();
60     root.fill(&WHITE)?;
61
62     let max_freq_val = max_freq.unwrap_or(*frequencies.last().unwrap_or(&10000.0));
63
64     let magnitude_db: Vec<f64> = magnitude.iter()
65         .map(|&m| filter_response::magnitude_to_db(m))
66         .collect();
67
68     let min_db = magnitude_db.iter()
69         .zip(frequencies.iter())
70         .filter(|( _, &f)| f <= max_freq_val)
71         .map(|(&db, _)| db)
72         .fold(f64::INFINITY, f64::min)
73         .max(-80.0);
74
75     let max_db = 10.0;
76
77     let mut chart = ChartBuilder::on(&root)
78         .caption(title, ("sans-serif", 30).into_font())
79         .margin(15)
80         .x_label_area_size(50)
81         .y_label_area_size(60)
82         .build_cartesian_2d(0.0..max_freq_val, min_db..max_db)?;
83
84     chart.configure_mesh()
85         .x_desc("Frequency (Hz)")
86         .y_desc("Magnitude (dB)")
87         .draw()?;
88
89     chart.draw_series(LineSeries::new(
90         frequencies.iter()
91             .zip(magnitude_db.iter())
92             .filter(|(&f, _)| f <= max_freq_val)
93             .map(|(&f, &db)| (f, db)),
94             &BLUE,
95         ))?;
96
97     root.present()?;
98     Ok(())
99 }
100
101 /// Plot phase response
102 pub fn plot_phase_response(
103     frequencies: &[f64],
104     phase: &[f64],
105     output_path: &str,
106     title: &str,
107     max_freq: Option<f64>,
108 ) -> Result<(), Box<dyn std::error::Error>> {
109     let root = BitMapBackend::new(output_path, (PLOT_WIDTH,
110                                   PLOT_HEIGHT)).into_drawing_area();

```

```

109     root.fill(&WHITE)?;
110
111     let max_freq_val = max_freq.unwrap_or(*frequencies.last().unwrap_or(&10000.0));
112
113     let phase_deg: Vec<f64> = phase.iter()
114         .map(|&p| filter_response::phase_to_degrees(p))
115         .collect();
116
117     let min_phase = phase_deg.iter()
118         .zip(frequencies.iter())
119         .filter(|(_ , &f)| f <= max_freq_val)
120         .map(|(&p, _)| p)
121         .fold(f64::INFINITY, f64::min) - 10.0;
122
123     let max_phase = phase_deg.iter()
124         .zip(frequencies.iter())
125         .filter(|(_ , &f)| f <= max_freq_val)
126         .map(|(&p, _)| p)
127         .fold(f64::NEG_INFINITY, f64::max) + 10.0;
128
129     let mut chart = ChartBuilder::on(&root)
130         .caption(title, ("sans-serif", 30).into_font())
131         .margin(15)
132         .x_label_area_size(50)
133         .y_label_area_size(60)
134         .build_cartesian_2d(0.0..max_freq_val, min_phase..max_phase)?;
135
136     chart.configure_mesh()
137         .x_desc("Frequency (Hz)")
138         .y_desc("Phase (degrees)")
139         .draw()?;
140
141     chart.draw_series(LineSeries::new(
142         frequencies.iter()
143             .zip(phase_deg.iter())
144             .filter(|(&f, _)| f <= max_freq_val)
145             .map(|(&f, &p)| (f, p)),
146             &RED,
147     ))?;
148
149     root.present()?;
150     Ok(())
151 }
152
153 /// Plot combined magnitude responses of high-pass and low-pass filters
154 pub fn plot_combined_magnitude(
155     frequencies: &[f64],
156     hp_magnitude: &[f64],
157     lp_magnitude: &[f64],
158     output_path: &str,
159     title: &str,
160     max_freq: Option<f64>,
161 ) -> Result<(), Box<dyn std::error::Error>> {
162     let root = BitMapBackend::new(output_path, (PLOT_WIDTH,
163                                         PLOT_HEIGHT)).into_drawing_area();
164     root.fill(&WHITE)?;

```

```

164
165     let max_freq_val = max_freq.unwrap_or(*frequencies.last().unwrap_or(&10000.0));
166
167     let max_mag = hp_magnitude.iter()
168         .chain(lp_magnitude.iter())
169         .zip(frequencies.iter().cycle())
170         .filter(|(_, &f)| f <= max_freq_val)
171         .map(|(&m, _)| m)
172         .fold(0.0, f64::max)
173         .max(1.1);
174
175     let mut chart = ChartBuilder::on(&root)
176         .caption(title, ("sans-serif", 30).into_font())
177         .margin(15)
178         .x_label_area_size(50)
179         .y_label_area_size(60)
180         .build_cartesian_2d(0.0..max_freq_val, 0.0..max_mag)?;
181
182     chart.configure_mesh()
183         .x_desc("Frequency (Hz)")
184         .y_desc("Magnitude")
185         .draw()?;
186
187     // Draw high-pass filter
188     chart.draw_series(LineSeries::new(
189         frequencies.iter()
190             .zip(hp_magnitude.iter())
191             .filter(|(&f, _)|
192                 f <= max_freq_val)
193             .map(|(&f, &m)| (f, m)),
194             &BLUE,
195         )?.label("High-pass")
196         .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &BLUE));
197
198     // Draw low-pass filter
199     chart.draw_series(LineSeries::new(
200         frequencies.iter()
201             .zip(lp_magnitude.iter())
202             .filter(|(&f, _)|
203                 f <= max_freq_val)
204             .map(|(&f, &m)| (f, m)),
205             &RED,
206         )?.label("Low-pass")
207         .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &RED));
208
209     chart.configure_series_labels()
210         .background_style(&WHITE.mix(0.8))
211         .border_style(&BLACK)
212         .draw()?;
213
214     root.present()?;
215     Ok(())
216 }

```

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

```
1 mod audio_reader;
2 mod iir_filter;
3 mod demodulator;
4 mod spectrum_analyzer;
5 mod audio_writer;
6
7 use std::f64::consts::PI;
8
9 fn main() {
10     println!("Q3: Time-Domain Demodulation");
11     println!("=====");
12
13     // Step 1: Read Q1 results to get f_d, f_s, f_B
14     println!("\n[Step 1] Reading Q1 results...");
15     let (f_d, f_s, f_b) = match read_q1_results() {
16         Ok(params) => params,
17         Err(e) => {
18             eprintln!("Error reading Q1 results: {}", e);
19             return;
20         }
21     };
22     println!("  f_d = {:.4} Hz", f_d);
23     println!("  f_s = {:.4} Hz", f_s);
24     println!("  f_B = {:.4} Hz", f_b);
25
26     // Step 2: Read Q2 filter coefficients
27     println!("\n[Step 2] Reading Q2 filter coefficients...");
28     let (hp_b, hp_a, lp_b, lp_a) = match read_q2_filters() {
29         Ok(filters) => filters,
30         Err(e) => {
31             eprintln!("Error reading Q2 filters: {}", e);
32             return;
33         }
34     };
35     println!("  High-pass filter: {} b coefficients, {} a coefficients", hp_b.len(),
36             hp_a.len());
36     println!("  Low-pass filter: {} b coefficients, {} a coefficients", lp_b.len(),
37             lp_a.len());
38
38     // Step 3: Read audio signal
39     println!("\n[Step 3] Reading audio signal...");
40     let audio_samples = match
41         audio_reader::read_wav("../工程设计问题-2022/工程设计题15.
42         调幅信号的解调/project.wav") {
43             Ok(samples) => samples,
44             Err(e) => {
45                 eprintln!("Error reading audio: {}", e);
46                 return;
47             }
48     };
49     println!("  Number of samples: {}", audio_samples.len());
50     let max_orig = audio_samples.iter().fold(0.0f64, |max, &x| max.max(x.abs()));
51     println!("  Signal max: {:.6}", max_orig);
51
51     // Step 4: Apply high-pass filter
```

```

52     println!("\n[Step 4] Applying high-pass filter...");  

53     let x_h = iir_filter::apply_filter(&audio_samples, &hp_b, &hp_a);  

54     println!("  Output samples: {}", x_h.len());  

55     let max_xh = x_h.iter().fold(0.0f64, |max, &x| max.max(x.abs()));  

56     println!("  Signal max: {:.6}", max_xh);  

57  

58 // Step 5: Generate carrier and multiply  

59 println!("\n[Step 5] Multiplying with carrier signal (f_d = {:.4} Hz)...", f_d);  

60 let x_b = demodulator::multiply_with_carrier(&x_h, f_d, f_s);  

61 println!("  Output samples: {}", x_b.len());  

62 let max_xb = x_b.iter().fold(0.0f64, |max, &x| max.max(x.abs()));  

63 println!("  Signal max: {:.6}", max_xb);  

64  

65 // Step 6: Apply low-pass filter  

66 println!("\n[Step 6] Applying low-pass filter...");  

67 let x_l = iir_filter::apply_filter(&x_b, &lp_b, &lp_a);  

68 println!("  Output samples: {}", x_l.len());  

69  

70 // Debug: Check signal statistics  

71 let max_val = x_l.iter().fold(0.0f64, |max, &x| max.max(x.abs()));  

72 let mean_val = x_l.iter().sum::<f64>() / x_l.len() as f64;  

73 println!("  Signal range: max = {:.6}, mean = {:.6}", max_val, mean_val);  

74  

75 // Step 7: Spectrum analysis  

76 println!("\n[Step 7] Performing spectrum analysis...");  

77 let original_spectrum = spectrum_analyzer::compute_spectrum(&audio_samples, f_s);  

78 let xh_spectrum = spectrum_analyzer::compute_spectrum(&x_h, f_s);  

79 let xb_spectrum = spectrum_analyzer::compute_spectrum(&x_b, f_s);  

80 let xl_spectrum = spectrum_analyzer::compute_spectrum(&x_l, f_s);  

81  

82 // Step 8: Create output directory  

83 std::fs::create_dir_all("output").expect("Failed to create output directory");  

84  

85 // Step 9: Plot spectra  

86 println!("\n[Step 8] Plotting spectra...");  

87 spectrum_analyzer::plot_spectrum(&original_spectrum,  

88     "output/Q3_original_spectrum.png", "Original Signal X(f)");  

89 spectrum_analyzer::plot_spectrum(&xh_spectrum, "output/Q3_xh_spectrum.png",  

90     "After High-Pass X_h(f)");  

91 spectrum_analyzer::plot_spectrum(&xb_spectrum, "output/Q3_xb_spectrum.png",  

92     "After Multiplication X_b(f)");  

93 spectrum_analyzer::plot_spectrum(&xl_spectrum, "output/Q3_xl_spectrum.png",  

94     "After Low-Pass X_l(f) - Demodulated");  

95  

96 // Step 10: Save demodulated audio  

97 println!("\n[Step 9] Saving demodulated audio...");  

98 match audio_writer::write_wav("output/Q3_demodulated.wav", &x_l, f_s as u32) {  

99     Ok(_) => println!("  Saved to: output/Q3_demodulated.wav"),  

100    Err(e) => eprintln!("  Error saving audio: {}", e),  

101 }  

102  

103 // Step 11: Save analysis results  

104 println!("\n[Step 10] Saving analysis results...");  

105 save_results(&original_spectrum, &xh_spectrum, &xb_spectrum, &xl_spectrum, f_d,  

106             f_s);

```

```

103     println!("\\nQ3 Time-Domain Demodulation completed successfully!");
104     println!("Output files saved in: codes/Q3/output/");
105 }
106
107 fn read_q1_results() -> Result<(f64, f64, f64), String> {
108     let content = std::fs::read_to_string("../Q1/output/Q1_results.txt")
109         .map_err(|e| format!("Failed to read Q1 results: {}", e))?;
110
111     let mut f_d = None;
112     let mut f_s = None;
113
114     for line in content.lines() {
115         if line.contains("频率偏差") || line.contains("f_d") {
116             if let Some(value_str) = line.split('=').nth(1) {
117                 if let Ok(value) =
118                     value_str.trim().split_whitespace().next().unwrap_or("0").parse::<f64>()
119                 {
120                     f_d = Some(value);
121                 }
122             }
123         } else if line.contains("采样频率") || line.contains("f_s") {
124             if let Some(value_str) = line.split('=').nth(1) {
125                 if let Ok(value) =
126                     value_str.trim().split_whitespace().next().unwrap_or("0").parse::<f64>()
127                 {
128                     f_s = Some(value);
129                 }
130             }
131         }
132     }
133
134     let f_d = f_d.ok_or_else(|| "Could not find f_d in Q1 results".to_string())?;
135     let f_s = f_s.ok_or_else(|| "Could not find f_s in Q1 results".to_string())?;
136     let f_b = 4000.0; // Given in problem statement
137
138     Ok((f_d, f_s, f_b))
139 }
140
141 fn read_q2_filters() -> Result<(Vec<f64>, Vec<f64>, Vec<f64>, Vec<f64>), String> {
142     let content = std::fs::read_to_string("../Q2/output/Q2_filter_coefficients.txt")
143         .map_err(|e| format!("Failed to read Q2 filters: {}", e))?;
144
145     let mut hp_b = Vec::new();
146     let mut hp_a = Vec::new();
147     let mut lp_b = Vec::new();
148     let mut lp_a = Vec::new();
149
150     let mut current_section = "";
151
152     for line in content.lines() {
153         let line = line.trim();
154         if line.is_empty() {
155             continue;
156         }
157
158         if line.contains("High-pass Filter") {
159             if !hp_a.is_empty() {
160                 break;
161             }
162
163             let mut coefficients = Vec::new();
164
165             while !line.contains("Low-pass Filter") {
166                 let coefficient = line
167                     .split_whitespace()
168                     .next()
169                     .unwrap_or("0");
170
171                 coefficients.push(coefficient.parse::<f64>());
172
173                 if line.contains("Low-pass Filter") {
174                     break;
175                 }
176
177                 line = line
178                     .trim()
179                     .split_whitespace()
180                     .next()
181                     .unwrap_or("");
182
183             }
184
185             hp_b.push(coefficients[0]);
186             hp_a.push(coefficients[1]);
187
188             coefficients.clear();
189
190             while !line.contains("High-pass Filter") {
191                 let coefficient = line
192                     .split_whitespace()
193                     .next()
194                     .unwrap_or("0");
195
196                 coefficients.push(coefficient.parse::<f64>());
197
198                 if line.contains("High-pass Filter") {
199                     break;
200                 }
201
202                 line = line
203                     .trim()
204                     .split_whitespace()
205                     .next()
206                     .unwrap_or("");
207
208             }
209
210             lp_b.push(coefficients[0]);
211             lp_a.push(coefficients[1]);
212
213             coefficients.clear();
214
215             if line.contains("Low-pass Filter") {
216                 break;
217             }
218
219             line = line
220                 .trim()
221                 .split_whitespace()
222                 .next()
223                 .unwrap_or("");
224
225         }
226
227         if line.contains("Low-pass Filter") {
228             if !lp_a.is_empty() {
229                 break;
230             }
231
232             let mut coefficients = Vec::new();
233
234             while !line.contains("High-pass Filter") {
235                 let coefficient = line
236                     .split_whitespace()
237                     .next()
238                     .unwrap_or("0");
239
240                 coefficients.push(coefficient.parse::<f64>());
241
242                 if line.contains("High-pass Filter") {
243                     break;
244                 }
245
246                 line = line
247                     .trim()
248                     .split_whitespace()
249                     .next()
250                     .unwrap_or("");
251
252             }
253
254             lp_b.push(coefficients[0]);
255             lp_a.push(coefficients[1]);
256
257             coefficients.clear();
258
259             if line.contains("High-pass Filter") {
260                 break;
261             }
262
263             line = line
264                 .trim()
265                 .split_whitespace()
266                 .next()
267                 .unwrap_or("");
268
269         }
270
271     }
272
273     Ok((hp_b, hp_a, lp_b, lp_a))
274 }

```

```

155     current_section = "hp";
156 } else if line.contains("Low-pass Filter") {
157     current_section = "lp";
158 } else if line.starts_with("b[") {
159     if let Some(value_str) = line.split('=').nth(1) {
160         if let Ok(value) = value_str.trim().parse::<f64>() {
161             match current_section {
162                 "hp" => hp_b.push(value),
163                 "lp" => lp_b.push(value),
164                 _ => {}
165             }
166         }
167     }
168 } else if line.starts_with("a[") {
169     if let Some(value_str) = line.split('=').nth(1) {
170         if let Ok(value) = value_str.trim().parse::<f64>() {
171             match current_section {
172                 "hp" => hp_a.push(value),
173                 "lp" => lp_a.push(value),
174                 _ => {}
175             }
176         }
177     }
178 }
179 }
180
181 if hp_b.is_empty() || hp_a.is_empty() || lp_b.is_empty() || lp_a.is_empty() {
182     return Err("Failed to parse filter coefficients".to_string());
183 }
184
185 Ok((hp_b, hp_a, lp_b, lp_a))
186 }
187
188 fn save_results(
189     original: &[(f64, f64)],
190     xh: &[(f64, f64)],
191     xb: &[(f64, f64)],
192     xl: &[(f64, f64)],
193     f_d: f64,
194     f_s: f64,
195 ) {
196     let mut content = String::new();
197     content.push_str("Q3 Time-Domain Demodulation Results\n");
198     content.push_str("=====\\n\\n");
199     content.push_str(&format!("Carrier frequency: f_d = {:.4} Hz\\n", f_d));
200     content.push_str(&format!("Sampling frequency: f_s = {:.4} Hz\\n\\n", f_s));
201
202     // Spectral peaks for each stage
203     content.push_str("Spectral Analysis:\\n");
204     content.push_str("-----\\n");
205
206     // Original signal peak
207     let orig_peak = original.iter()
208         .filter(|(f, _)| *f > 1000.0)
209         .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
210         .unwrap();

```

```

211     content.push_str(&format!("Original signal X(f) peak: f = {:.2} Hz, magnitude =
212         {:.6}\n",
213         orig_peak.0, orig_peak.1));
214
214 // After high-pass
215 let xh_peak = xh.iter()
216     .filter(|(f, _)| *f > 1000.0)
217     .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
218     .unwrap();
219 content.push_str(&format!("After high-pass X_h(f) peak: f = {:.2} Hz, magnitude =
220         {:.6}\n",
221         xh_peak.0, xh_peak.1));
221
222 // After multiplication
223 let xb_low_peak = xb.iter()
224     .filter(|(f, _)| *f > 10.0 && *f < 5000.0)
225     .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
226     .unwrap();
227 content.push_str(&format!("After multiplication X_b(f) peak (baseband): f = {:.2}
228     Hz, magnitude = {:.6}\n",
229         xb_low_peak.0, xb_low_peak.1));
230
230 // Demodulated signal peak
231 let xl_peak = xl.iter()
232     .filter(|(f, _)| *f > 10.0 && *f < 4000.0)
233     .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
234     .unwrap();
235 content.push_str(&format!("Demodulated signal X_l(f) peak: f = {:.2} Hz,
236         magnitude = {:.6}\n",
237         xl_peak.0, xl_peak.1));
238
238 // Energy in baseband (0-4000 Hz)
239 let energy_orig_baseband: f64 = original.iter()
240     .filter(|(f, _)| *f < 4000.0)
241     .map(|(_, m)| m * m)
242     .sum();
243 let energy_demod_baseband: f64 = xl.iter()
244     .filter(|(f, _)| *f < 4000.0)
245     .map(|(_, m)| m * m)
246     .sum();
247
248 content.push_str(&format!("\nEnergy analysis (0-4000 Hz band):\n"));
249 content.push_str(&format!("    Original signal energy: {:.6e}\n",
250         energy_orig_baseband));
250 content.push_str(&format!("    Demodulated signal energy: {:.6e}\n",
251         energy_demod_baseband));
252
252 // Frequency shift verification
253 content.push_str(&format!("\nFrequency shift verification:\n"));
254 content.push_str(&format!("    Original peak at: {:.2} Hz\n", orig_peak.0));
255 content.push_str(&format!("    Expected shift: {:.2} Hz (should be near f_d = {:.2}
256         Hz)\n",
257         orig_peak.0 - f_d, f_d));
257 content.push_str(&format!("    Demodulated peak at: {:.2} Hz (should be in
258         baseband)\n", xl_peak.0));
258

```

```

259     std::fs::write("output/Q3_results.txt", content).expect("Failed to save results");
260     println!("  Saved to: output/Q3_results.txt");
261 }

```

音频读取模块 (audio_reader.rs) Listing 10: 音频文件读取模块

```

1 use hound;
2
3 pub fn read_wav(filename: &str) -> Result<Vec<f64>, String> {
4     let reader = hound::WavReader::open(filename)
5         .map_err(|e| format!("Failed to open WAV file: {}", e))?;
6
7     let spec = reader.spec();
8     println!("  Sample rate: {} Hz", spec.sample_rate);
9     println!("  Channels: {}", spec.channels);
10    println!("  Bits per sample: {}", spec.bits_per_sample);
11
12    // Read all samples and normalize to [-1.0, 1.0]
13    let samples: Vec<f64> = match spec.bits_per_sample {
14        16 => {
15            reader
16                .into_samples::<i16>()
17                .map(|s| s.unwrap() as f64 / 32768.0)
18                .collect()
19        }
20        _ => {
21            return Err(format!(
22                "Unsupported bits per sample: {}",
23                spec.bits_per_sample
24            ))
25        }
26    };
27
28    Ok(samples)
29 }

```

IIR 滤波器模块 (iir_filter.rs) Listing 11: Direct Form II IIR 滤波器实现

```

1 /// Apply IIR filter using Direct Form II structure
2 /// y[n] = sum(b[i] * x[n-i]) - sum(a[j] * y[n-j]) for j > 0
3 pub fn apply_filter(input: &[f64], b: &[f64], a: &[f64]) -> Vec<f64> {
4     let n = input.len();
5     let mut output = vec![0.0; n];
6
7     let order = b.len().max(a.len());
8     let mut x_history = vec![0.0; order]; // Input history
9     let mut y_history = vec![0.0; order]; // Output history
10
11    for i in 0..n {
12        // Shift histories
13        for j in (1..order).rev() {
14            x_history[j] = x_history[j - 1];
15            y_history[j] = y_history[j - 1];

```

```

16     }
17     x_history[0] = input[i];
18
19     // Calculate output using difference equation
20     let mut y = 0.0;
21
22     // Feedforward part: sum(b[k] * x[n-k])
23     for k in 0..b.len() {
24         y += b[k] * x_history[k];
25     }
26
27     // Feedback part: -sum(a[k] * y[n-k]) for k > 0
28     for k in 1..a.len() {
29         y -= a[k] * y_history[k];
30     }
31
32     // Normalize by a[0] (usually 1.0)
33     y /= a[0];
34
35     y_history[0] = y;
36     output[i] = y;
37 }
38
39     output
40 }
41
42 #[cfg(test)]
43 mod tests {
44     use super::*;

45
46     #[test]
47     fn test_simple_filter() {
48         // Simple moving average filter: y[n] = 0.5*x[n] + 0.5*x[n-1]
49         let b = vec![0.5, 0.5];
50         let a = vec![1.0];
51         let input = vec![1.0, 2.0, 3.0, 4.0, 5.0];
52
53         let output = apply_filter(&input, &b, &a);
54
55         // Expected: [0.5, 1.5, 2.5, 3.5, 4.5]
56         assert!((output[0] - 0.5).abs() < 1e-10);
57         assert!((output[1] - 1.5).abs() < 1e-10);
58         assert!((output[2] - 2.5).abs() < 1e-10);
59     }
60 }

```

解调器模块 (demodulator.rs) Listing 12: 时域同步解调 - 载波相乘法

```

1 use std::f64::consts::PI;
2
3 /// Multiply signal with carrier cos(2*pi*f_d*t)
4 pub fn multiply_with_carrier(signal: &[f64], f_d: f64, f_s: f64) -> Vec<f64> {
5     let n = signal.len();
6     let mut output = vec![0.0; n];
7

```

```

8     for i in 0..n {
9         let t = i as f64 / f_s;
10        let carrier = (2.0 * PI * f_d * t).cos();
11        // Multiply by 2 to compensate for the 1/2 factor from cos2(x) = (1 +
12        // cos(2x))/2
13        output[i] = signal[i] * carrier * 2.0;
14    }
15
16    output
17}
18 #[cfg(test)]
19 mod tests {
20     use super::*;

21     #[test]
22     fn test_carrier_multiplication() {
23         let f_d = 1000.0;
24         let f_s = 8000.0;
25         let signal = vec![1.0; 100];
26
27         let output = multiply_with_carrier(&signal, f_d, f_s);
28
29         // Output should oscillate with carrier frequency
30         assert_eq!(output.len(), 100);
31         // At t=0, cos(0) = 1.0
32         assert!((output[0] - 1.0).abs() < 1e-10);
33     }
34 }
35

```

频谱分析模块 (spectrum_analyzer.rs) - Listing 13: 频谱分析 - FFT 与峰值检测

```

1 use rustfft::{FftPlanner, num_complex::Complex};
2 use plotters::prelude::*;
3
4 /// Compute magnitude spectrum of a signal
5 pub fn compute_spectrum(signal: &[f64], f_s: f64) -> Vec<(f64, f64)> {
6     let n = signal.len();
7
8     // Prepare FFT input
9     let mut buffer: Vec<Complex<f64>> = signal
10        .iter()
11        .map(|&x| Complex::new(x, 0.0))
12        .collect();
13
14     // Perform FFT
15     let mut planner = FftPlanner::new();
16     let fft = planner.plan_fft_forward(n);
17     fft.process(&mut buffer);
18
19     // Compute magnitude spectrum and frequency axis
20     let mut spectrum = Vec::with_capacity(n / 2);
21     let df = f_s / n as f64;
22
23     for i in 0..n / 2 {

```

```

24     let freq = i as f64 * df;
25     let magnitude = buffer[i].norm() / n as f64;
26     spectrum.push((freq, magnitude));
27 }
28
29 spectrum
30 }
31
32 /// Plot spectrum
33 pub fn plot_spectrum(spectrum: &[(f64, f64)], filename: &str, title: &str) {
34     let root = BitMapBackend::new(filename, (1200, 800)).into_drawing_area();
35     root.fill(&WHITE).unwrap();
36
37     // Find max magnitude for y-axis
38     let max_mag = spectrum.iter()
39         .map(|(_, m)| *m)
40         .fold(0.0f64, f64::max);
41
42     let max_freq = spectrum.last().unwrap().0;
43
44     let mut chart = ChartBuilder::on(&root)
45         .caption(title, ("sans-serif", 40))
46         .margin(20)
47         .x_label_area_size(50)
48         .y_label_area_size(60)
49         .build_cartesian_2d(0.0..max_freq, 0.0..max_mag * 1.1)
50         .unwrap();
51
52     chart
53         .configure_mesh()
54         .x_desc("Frequency (Hz)")
55         .y_desc("Magnitude")
56         .x_label_formatter(&|x| format!(" {:.0}", x))
57         .y_label_formatter(&|y| format!(" {:.3}", y))
58         .draw()
59         .unwrap();
60
61     chart
62         .draw_series(LineSeries::new(
63             spectrum.iter().map(|(f, m)| (*f, *m)),
64             &BLUE,
65         ))
66         .unwrap();
67
68     root.present().unwrap();
69     println!(" Saved: {}", filename);
70 }
71
72 /// Plot spectrum in dB scale
73 pub fn plot_spectrum_db(spectrum: &[(f64, f64)], filename: &str, title: &str) {
74     let root = BitMapBackend::new(filename, (1200, 800)).into_drawing_area();
75     root.fill(&WHITE).unwrap();
76
77     // Convert to dB
78     let spectrum_db: Vec<(f64, f64)> = spectrum
79         .iter()

```

```

80     .map(|(f, m)| {
81         let db = if *m > 1e-10 {
82             20.0 * m.log10()
83         } else {
84             -200.0
85         };
86         (*f, db)
87     })
88     .collect();
89
90     let max_freq = spectrum_db.last().unwrap().0;
91
92     let mut chart = ChartBuilder::on(&root)
93         .caption(title, ("sans-serif", 40))
94         .margin(20)
95         .x_label_area_size(50)
96         .y_label_area_size(60)
97         .build_cartesian_2d(0.0..max_freq, -100.0..0.0)
98         .unwrap();
99
100    chart
101        .configure_mesh()
102        .x_desc("Frequency (Hz)")
103        .y_desc("Magnitude (dB)")
104        .x_label_formatter(&|x| format!(" {:.0}", x))
105        .y_label_formatter(&|y| format!(" {:.0}", y))
106        .draw()
107        .unwrap();
108
109    chart
110        .draw_series(LineSeries::new(
111            spectrum_db.iter().map(|(f, m)| (*f, *m)),
112            &RED,
113        ))
114        .unwrap();
115
116    root.present().unwrap();
117    println!(" Saved: {}", filename);
118}

```

音频写入模块 (audio_writer.rs) Listing 14: WAV 文件写入 - 16bit PCM 格式

```

1 use hound;
2
3 pub fn write_wav(filename: &str, samples: &[f64], sample_rate: u32) -> Result<(),
4     String> {
5     let spec = hound::WavSpec {
6         channels: 1,
7         sample_rate,
8         bits_per_sample: 16,
9         sample_format: hound::SampleFormat::Int,
10    };
11
12    let mut writer = hound::WavWriter::create(filename, spec)
13        .map_err(|e| format!("Failed to create WAV file: {}", e))?;

```

```

13
14     // Normalize samples to prevent clipping
15     let max_val = samples.iter().fold(0.0f64, |max, &x| max.max(x.abs())));
16     let scale = if max_val > 0.0 {
17         0.95 / max_val // Leave some headroom
18     } else {
19         1.0
20     };
21
22     for &sample in samples {
23         let normalized = sample * scale;
24         let sample_i16 = (normalized * 32767.0).clamp(-32768.0, 32767.0) as i16;
25         writer
26             .write_sample(sample_i16)
27             .map_err(|e| format!("Failed to write sample: {}", e))?;
28     }
29
30     writer
31         .finalize()
32         .map_err(|e| format!("Failed to finalize WAV file: {}", e))?;
33
34     Ok(())
35 }
```

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

```

1 mod audio_reader;
2 mod ideal_filter;
3 mod frequency_shifter;
4 mod spectrum_analyzer;
5 mod audio_writer;
6 mod comparator;
7
8 use num_complex::Complex;
9
10 fn main() {
11     println!("Q4: Frequency-Domain Demodulation");
12     println!("=====");
13
14     // Step 1: Read Q1 results to get f_d, f_s, f_B
15     println!("\n[Step 1] Reading Q1 results... ");
16     let (f_d, f_s, f_b) = match read_q1_results() {
17         Ok(params) => params,
18         Err(e) => {
19             eprintln!("Error reading Q1 results: {}", e);
20             return;
21         }
22     };
23     println!("  f_d = {:.4} Hz", f_d);
24     println!("  f_s = {:.4} Hz", f_s);
25     println!("  f_B = {:.4} Hz", f_b);
26 }
```

```

27 // Step 2: Read audio signal
28 println!("\\n[Step 2] Reading audio signal...");  

29 let audio_samples = match  

30     audio_reader::read_wav("../工程设计问题-2022/工程设计题15.  

31         调幅信号的解调/project.wav") {  

32     Ok(samples) => samples,  

33     Err(e) => {  

34         eprintln!("Error reading audio: {}", e);  

35         return;  

36     }  

37 };  

38 let n = audio_samples.len();  

39 println!(" Number of samples: {}", n);  

40  

41 // Step 3: Compute FFT of input signal  

42 println!("\\n[Step 3] Computing FFT of input signal...");  

43 let x_fft = compute_fft(&audio_samples);  

44 println!(" FFT size: {}", x_fft.len());  

45  

46 // Step 4: Apply ideal high-pass filter in frequency domain  

47 println!("\\n[Step 4] Applying ideal high-pass filter (fc = {:.4} Hz)...", f_d);  

48 let xh_fft = ideal_filter::apply_highpass(&x_fft, f_d, f_s, n);  

49 println!(" High-pass filtering complete");  

50  

51 // Step 5: Frequency shift (equivalent to carrier multiplication in time domain)  

52 println!("\\n[Step 5] Performing frequency shift (±{:.4} Hz)...", f_d);  

53 let xb_fft = frequency_shifter::frequency_shift(&xh_fft, f_d, f_s, n);  

54 println!(" Frequency shift complete");  

55  

56 // Step 6: Apply ideal low-pass filter  

57 println!("\\n[Step 6] Applying ideal low-pass filter (fc = {:.4} Hz)...", f_b);  

58 let xl_fft = ideal_filter::apply_lowpass(&xb_fft, f_b, f_s, n);  

59 println!(" Low-pass filtering complete");  

60  

61 // Step 7: Inverse FFT to get time-domain signal  

62 println!("\\n[Step 7] Computing IFFT to recover time-domain signal...");  

63 let mut xl_samples = compute_ifft(&xl_fft);  

64 println!(" Output samples: {}", xl_samples.len());  

65  

66 // Apply gain compensation (multiply by 2 to match time-domain method)  

67 for sample in xl_samples.iter_mut() {  

68     *sample *= 2.0;  

69 }  

70  

71 let max_val = xl_samples.iter().fold(0.0f64, |max, &x| max.max(x.abs()));  

72 println!(" Signal max: {:.6}", max_val);  

73  

74 // Step 8: Create output directory  

75 std::fs::create_dir_all("output").expect("Failed to create output directory");  

76  

77 // Step 9: Spectrum analysis for each stage  

78 println!("\\n[Step 8] Performing spectrum analysis...");  

79 let original_spectrum = compute_magnitude_spectrum(&x_fft, f_s);  

80 let xh_spectrum = compute_magnitude_spectrum(&xh_fft, f_s);  

81 let xb_spectrum = compute_magnitude_spectrum(&xb_fft, f_s);  

82 let xl_spectrum = compute_magnitude_spectrum(&xl_fft, f_s);

```

```

81 // Step 10: Plot spectra
82 println!("\n[Step 9] Plotting spectra...");
83 spectrum_analyzer::plot_spectrum(&original_spectrum,
84     "output/Q4_original_spectrum.png", "Original Signal X(f)");
85 spectrum_analyzer::plot_spectrum(&xh_spectrum, "output/Q4_xh_spectrum.png",
86     "After Ideal High-Pass X_h(f)");
87 spectrum_analyzer::plot_spectrum(&xb_spectrum, "output/Q4_xb_spectrum.png",
88     "After Frequency Shift X_b(f)");
89 spectrum_analyzer::plot_spectrum(&xl_spectrum, "output/Q4_xl_spectrum.png",
90     "After Ideal Low-Pass X_l(f) - Demodulated");

91 // Step 11: Save demodulated audio
92 println!("\n[Step 10] Saving demodulated audio...");
93 match audio_writer::write_wav("output/Q4_demodulated.wav", &xl_samples, f_s as
94     u32) {
95     Ok(_) => println!(" Saved to: output/Q4_demodulated.wav"),
96     Err(e) => eprintln!(" Error saving audio: {}", e),
97 }
98
99 // Step 12: Compare with Q3 results
100 println!("\n[Step 11] Comparing with Q3 results...");
101 if let Ok(q3_samples) = audio_reader::read_wav("../Q3/output/Q3_demodulated.wav")
102 {
103     let comparison = comparator::compare_signals(&xl_samples, &q3_samples);
104     println!(" Q3 vs Q4 comparison:");
105     println!(" MSE: {:.6e}", comparison.mse);
106     println!(" Max difference: {:.6}", comparison.max_diff);
107     println!(" Correlation: {:.6}", comparison.correlation);

108     // Save comparison results
109     comparator::save_comparison(&comparison, "output/Q4_comparison.txt");
110
111     // Plot comparison
112     comparator::plot_comparison(&xl_samples, &q3_samples,
113         "output/Q4_vs_Q3_comparison.png");
114 } else {
115     println!(" Warning: Could not read Q3 results for comparison");
116 }

117 // Step 13: Save analysis results
118 println!("\n[Step 12] Saving analysis results...");
119 save_results(&original_spectrum, &xh_spectrum, &xb_spectrum, &xl_spectrum, f_d,
120     f_s, f_b);

121 println!("\nQ4 Frequency-Domain Demodulation completed successfully!");
122 println!("Output files saved in: codes/Q4/output/");
123 }

124 fn read_q1_results() -> Result<(f64, f64, f64), String> {
125     let content = std::fs::read_to_string("../Q1/output/Q1_results.txt")
126         .map_err(|e| format!("Failed to read Q1 results: {}", e))?;

127     let mut f_d = None;
128     let mut f_s = None;

```

```

129     for line in content.lines() {
130         if line.contains("频率偏差") || line.contains("f_d") {
131             if let Some(value_str) = line.split('=').nth(1) {
132                 if let Ok(value) =
133                     value_str.trim().split_whitespace().next().unwrap_or("0").parse::<f64>()
134                 {
135                     f_d = Some(value);
136                 }
137             } else if line.contains("采样频率") || line.contains("f_s") {
138                 if let Some(value_str) = line.split('=').nth(1) {
139                     if let Ok(value) =
140                         value_str.trim().split_whitespace().next().unwrap_or("0").parse::<f64>()
141                     {
142                         f_s = Some(value);
143                     }
144                 }
145             let f_d = f_d.ok_or_else(|| "Could not find f_d in Q1 results".to_string())?;
146             let f_s = f_s.ok_or_else(|| "Could not find f_s in Q1 results".to_string())?;
147             let f_b = 4000.0; // Given in problem statement
148
149             Ok((f_d, f_s, f_b))
150         }
151
152     fn compute_fft(samples: &[f64]) -> Vec<Complex<f64>> {
153         use rustfft::FftPlanner;
154
155         let mut buffer: Vec<Complex<f64>> = samples
156             .iter()
157             .map(|&x| Complex::new(x, 0.0))
158             .collect();
159
160         let mut planner = FftPlanner::new();
161         let fft = planner.plan_fft_forward(buffer.len());
162         fft.process(&mut buffer);
163
164         buffer
165     }
166
167     fn compute_ifft(spectrum: &[Complex<f64>]) -> Vec<f64> {
168         use rustfft::FftPlanner;
169
170         let mut buffer = spectrum.to_vec();
171
172         let mut planner = FftPlanner::new();
173         let ifft = planner.plan_fft_inverse(buffer.len());
174         ifft.process(&mut buffer);
175
176         // Normalize and extract real part
177         let n = buffer.len() as f64;
178         buffer.iter().map(|c| c.re / n).collect()
179     }
180

```

```

181 fn compute_magnitude_spectrum(spectrum: &[Complex<f64>], f_s: f64) -> Vec<(f64, f64)>
182 {
183     let n = spectrum.len();
184     let df = f_s / n as f64;
185
186     (0..n/2)
187         .map(|i| {
188             let freq = i as f64 * df;
189             let magnitude = spectrum[i].norm() / n as f64;
190             (freq, magnitude)
191         })
192         .collect()
193 }
194
195 fn save_results(
196     original: &[(f64, f64)],
197     xh: &[(f64, f64)],
198     xb: &[(f64, f64)],
199     xl: &[(f64, f64)],
200     f_d: f64,
201     f_s: f64,
202     f_b: f64,
203 ) {
204     let mut content = String::new();
205     content.push_str("Q4 Frequency-Domain Demodulation Results\n");
206     content.push_str("=====\\n\\n");
207     content.push_str(&format!("Carrier frequency: f_d = {:.4} Hz\\n", f_d));
208     content.push_str(&format!("Sampling frequency: f_s = {:.4} Hz\\n", f_s));
209     content.push_str(&format!("Baseband bandwidth: f_B = {:.4} Hz\\n\\n", f_b));
210
211     // Spectral peaks for each stage
212     content.push_str("Spectral Analysis:\\n");
213     content.push_str("-----\\n");
214
215     // Original signal peak
216     let orig_peak = original.iter()
217         .filter(|(f, _)| *f > 1000.0)
218         .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
219         .unwrap();
220     content.push_str(&format!("Original signal X(f) peak: f = {:.2} Hz, magnitude =
221         {:.6}\\n",
222         orig_peak.0, orig_peak.1));
223
224     // After high-pass
225     let xh_peak = xh.iter()
226         .filter(|(f, _)| *f > f_d)
227         .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
228         .unwrap();
229     content.push_str(&format!("After ideal high-pass X_h(f) peak: f = {:.2} Hz,
230         magnitude = {:.6}\\n",
231         xh_peak.0, xh_peak.1));
232
233     // After frequency shift
234     let xb_peak = xb.iter()
235         .filter(|(f, _)| *f > 10.0 && *f < 5000.0)
236         .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())

```

```

234     .unwrap();
235     content.push_str(&format!("After frequency shift X_b(f) peak: f = {:.2} Hz,
236         magnitude = {:.6}\n",
237         xb_peak.0, xb_peak.1));
238
239     // Demodulated signal peak
240     let xl_peak = xl.iter()
241         .filter(|(f, _)| *f > 10.0 && *f < f_b)
242         .max_by(|(_, mag1), (_, mag2)| mag1.partial_cmp(mag2).unwrap())
243         .unwrap();
244     content.push_str(&format!("Demodulated signal X_l(f) peak: f = {:.2} Hz,
245         magnitude = {:.6}\n",
246         xl_peak.0, xl_peak.1));
247
248     // Energy in baseband
249     let energy_orig: f64 = original.iter()
250         .filter(|(f, _)| *f < f_b)
251         .map(|(_, m)| m * m)
252         .sum();
253     let energy_demod: f64 = xl.iter()
254         .filter(|(f, _)| *f < f_b)
255         .map(|(_, m)| m * m)
256         .sum();
257
258     content.push_str(&format!("\nEnergy analysis (0-{:0.0} Hz band):\n", f_b));
259     content.push_str(&format!("    Original signal energy: {:.6e}\n", energy_orig));
260     content.push_str(&format!("    Demodulated signal energy: {:.6e}\n", energy_demod));
261
262     content.push_str(&format!("\nMethod characteristics:\n"));
263     content.push_str("    - Uses ideal filters (brick-wall response)\n");
264     content.push_str("    - Frequency-domain processing (no time-domain
265         convolution)\n");
266     content.push_str("    - Perfect frequency selectivity\n");
267     content.push_str("    - No phase distortion from filters\n");
268
269     std::fs::write("output/Q4_results.txt", content).expect("Failed to save results");
270     println!("    Saved to: output/Q4_results.txt");
271 }
```

音频读取模块 (audio_reader.rs) Listing 16: 音频文件读取模块

```

1 use hound;
2
3 pub fn read_wav(filename: &str) -> Result<Vec<f64>, String> {
4     let reader = hound::WavReader::open(filename)
5         .map_err(|e| format!("Failed to open WAV file: {}", e))?;
6
7     let spec = reader.spec();
8     println!("    Sample rate: {} Hz", spec.sample_rate);
9     println!("    Channels: {}", spec.channels);
10    println!("    Bits per sample: {}", spec.bits_per_sample);
11
12    // Read all samples and normalize to [-1.0, 1.0]
13    let samples: Vec<f64> = match spec.bits_per_sample {
14        16 => {
```

```

15     reader
16         .into_samples::<i16>()
17         .map(|s| s.unwrap() as f64 / 32768.0)
18         .collect()
19     }
20     _ => {
21         return Err(format!(
22             "Unsupported bits per sample: {}",
23             spec.bits_per_sample
24         ))
25     }
26 };
27
28 Ok(samples)
29 }
```

理想滤波器模块 (`ideal_filter.rs`) Listing 17: 理想高通/低通滤波器 - 频域矩形窗

```

1 use num_complex::Complex;
2
3 /// Apply ideal high-pass filter in frequency domain
4 /// H_h(f) = 0 for |f| < f_c, 1 for |f| >= f_c
5 pub fn apply_highpass(
6     spectrum: &[Complex<f64>],
7     f_c: f64,
8     f_s: f64,
9     n: usize,
10) -> Vec<Complex<f64>> {
11    let df = f_s / n as f64;
12    let mut result = spectrum.to_vec();
13
14    for i in 0..n {
15        // Calculate frequency for this bin
16        let freq = if i <= n / 2 {
17            i as f64 * df
18        } else {
19            (i as f64 - n as f64) * df
20        };
21
22        // Apply ideal high-pass filter
23        if freq.abs() < f_c {
24            result[i] = Complex::new(0.0, 0.0);
25        }
26    }
27
28    result
29}
30
31 /// Apply ideal low-pass filter in frequency domain
32 /// H_l(f) = 1 for |f| <= f_c, 0 for |f| > f_c
33 pub fn apply_lowpass(
34     spectrum: &[Complex<f64>],
35     f_c: f64,
36     f_s: f64,
37     n: usize,
```

```

38 | ) -> Vec<Complex<f64>> {
39 |     let df = f_s / n as f64;
40 |     let mut result = spectrum.to_vec();
41 |
42 |     for i in 0..n {
43 |         // Calculate frequency for this bin
44 |         let freq = if i <= n / 2 {
45 |             i as f64 * df
46 |         } else {
47 |             (i as f64 - n as f64) * df
48 |         };
49 |
50 |         // Apply ideal low-pass filter
51 |         if freq.abs() > f_c {
52 |             result[i] = Complex::new(0.0, 0.0);
53 |         }
54 |     }
55 |
56 |     result
57 | }
58 |
59 #[cfg(test)]
60 mod tests {
61     use super::*;

62     #[test]
63     fn test_highpass_filter() {
64         let n = 100;
65         let f_s = 1000.0;
66         let f_c = 100.0;
67 |
68         // Create test spectrum
69         let spectrum: Vec<Complex<f64>> = (0..n)
70             .map(|_| Complex::new(1.0, 0.0))
71             .collect();
72 |
73         let filtered = apply_highpass(&spectrum, f_c, f_s, n);
74 |
75         // Check that low frequencies are zeroed
76         assert_eq!(filtered[0].norm(), 0.0); // DC component
77         assert_eq!(filtered[5].norm(), 0.0); // 50 Hz
78     }
79 |
80     #[test]
81     fn test_lowpass_filter() {
82         let n = 100;
83         let f_s = 1000.0;
84         let f_c = 100.0;
85 |
86         let spectrum: Vec<Complex<f64>> = (0..n)
87             .map(|_| Complex::new(1.0, 0.0))
88             .collect();
89 |
90         let filtered = apply_lowpass(&spectrum, f_c, f_s, n);
91 |
92         // Check that high frequencies are zeroed

```

```

94     let high_freq_idx = (200.0 / f_s * n as f64) as usize;
95     assert_eq!(filtered[high_freq_idx].norm(), 0.0);
96   }
97 }

```

频率搬移模块 (frequency_shifter.rs) Listing 18: 循环移位频率搬移 - 频谱平移

```

1 use num_complex::Complex;
2
3 /// Perform frequency shift: X_b(f) = X_h(f - f_d) + X_h(f + f_d)
4 /// This is equivalent to multiplying by cos(2 f_d · t) in time domain
5 pub fn frequency_shift(
6     spectrum: &[Complex<f64>],
7     f_d: f64,
8     f_s: f64,
9     n: usize,
10) -> Vec<Complex<f64>> {
11    // Calculate shift amount in bins
12    let shift_bins = (f_d * n as f64 / f_s).round() as isize;
13
14    // Create result vector
15    let mut result = vec![Complex::new(0.0, 0.0); n];
16
17    // For each output frequency bin, find contributions from shifted input
18    for i in 0..n {
19        // Contribution from f - f_d (shift down)
20        let idx_minus = (i as isize + shift_bins).rem_euclid(n as isize) as usize;
21        result[i] = result[i] + spectrum[idx_minus] * 0.5;
22
23        // Contribution from f + f_d (shift up)
24        let idx_plus = (i as isize - shift_bins).rem_euclid(n as isize) as usize;
25        result[i] = result[i] + spectrum[idx_plus] * 0.5;
26    }
27
28    result
29}
30
31 /// Alternative implementation using explicit frequency mapping
32 pub fn frequency_shift_explicit(
33     spectrum: &[Complex<f64>],
34     f_d: f64,
35     f_s: f64,
36     n: usize,
37) -> Vec<Complex<f64>> {
38    let df = f_s / n as f64;
39    let mut result = vec![Complex::new(0.0, 0.0); n];
40
41    for i in 0..n {
42        // Current frequency
43        let f_i = if i <= n / 2 {
44            i as f64 * df
45        } else {
46            (i as f64 - n as f64) * df
47        };
48

```

```

49     // Find indices for f ± f_d
50     let f_minus = f_i - f_d;
51     let f_plus = f_i + f_d;
52
53     // Map back to bin indices
54     let idx_minus = freq_to_bin(f_minus, f_s, n);
55     let idx_plus = freq_to_bin(f_plus, f_s, n);
56
57     // Accumulate contributions
58     if idx_minus < n {
59         result[i] = result[i] + spectrum[idx_minus] * 0.5;
60     }
61     if idx_plus < n {
62         result[i] = result[i] + spectrum[idx_plus] * 0.5;
63     }
64 }
65
66 result
67 }
68
69 fn freq_to_bin(freq: f64, f_s: f64, n: usize) -> usize {
70     let normalized = if freq >= 0.0 {
71         freq
72     } else {
73         freq + f_s
74     };
75
76     let bin = (normalized * n as f64 / f_s).round() as isize;
77     bin.rem_euclid(n as isize) as usize
78 }
79
80 #[cfg(test)]
81 mod tests {
82     use super::*;

83     #[test]
84     fn test_frequency_shift() {
85         let n = 1000;
86         let f_s = 10000.0;
87         let f_d = 1000.0;
88
89         // Create impulse at 2000 Hz
90         let mut spectrum = vec![Complex::new(0.0, 0.0); n];
91         let impulse_idx = (2000.0 / f_s * n as f64) as usize;
92         spectrum[impulse_idx] = Complex::new(1.0, 0.0);
93
94         let shifted = frequency_shift(&spectrum, f_d, f_s, n);
95
96         // Should have peaks at 1000 Hz and 3000 Hz
97         let idx_1000 = (1000.0 / f_s * n as f64) as usize;
98         let idx_3000 = (3000.0 / f_s * n as f64) as usize;
99
100        assert!(shifted[idx_1000].norm() > 0.4);
101        assert!(shifted[idx_3000].norm() > 0.4);
102    }
103 }
104

```

```

105     #[test]
106     fn test_freq_to_bin() {
107         let n = 100;
108         let f_s = 1000.0;
109
110         assert_eq!(freq_to_bin(0.0, f_s, n), 0);
111         assert_eq!(freq_to_bin(100.0, f_s, n), 10);
112         assert_eq!(freq_to_bin(-100.0, f_s, n), 90);
113     }
114 }
```

频谱分析模块 (spectrum_analyzer.rs) Listing 19: 频谱分析模块

```

1 use plotters::prelude::*;
2
3 // Plot spectrum
4 pub fn plot_spectrum(spectrum: &[(f64, f64)], filename: &str, title: &str) {
5     let root = BitMapBackend::new(filename, (1200, 800)).into_drawing_area();
6     root.fill(&WHITE).unwrap();
7
8     // Find max magnitude for y-axis
9     let max_mag = spectrum.iter()
10        .map(|(_, m)| *m)
11        .fold(0.0f64, f64::max);
12
13     let max_freq = spectrum.last().unwrap().0;
14
15     let mut chart = ChartBuilder::on(&root)
16         .caption(title, ("sans-serif", 40))
17         .margin(20)
18         .x_label_area_size(50)
19         .y_label_area_size(60)
20         .build_cartesian_2d(0.0..max_freq, 0.0..max_mag * 1.1)
21         .unwrap();
22
23     chart
24         .configure_mesh()
25         .x_desc("Frequency (Hz)")
26         .y_desc("Magnitude")
27         .x_label_formatter(&|x| format!(" {:.0}", x))
28         .y_label_formatter(&|y| format!(" {:.3}", y))
29         .draw()
30         .unwrap();
31
32     chart
33         .draw_series(LineSeries::new(
34             spectrum.iter().map(|(f, m)| (*f, *m)),
35             &BLUE,
36         ))
37         .unwrap();
38
39     root.present().unwrap();
40     println!(" Saved: {}", filename);
41 }
```

音频写入模块 (audio_writer.rs) Listing 20: WAV 文件写入模块

```
1 use hound;
2
3 pub fn write_wav(filename: &str, samples: &[f64], sample_rate: u32) -> Result<(), String> {
4     let spec = hound::WavSpec {
5         channels: 1,
6         sample_rate,
7         bits_per_sample: 16,
8         sample_format: hound::SampleFormat::Int,
9     };
10
11    let mut writer = hound::WavWriter::create(filename, spec)
12        .map_err(|e| format!("Failed to create WAV file: {}", e))?;
13
14    // Normalize samples to prevent clipping
15    let max_val = samples.iter().fold(0.0f64, |max, &x| max.max(x.abs()));
16    let scale = if max_val > 0.0 {
17        0.95 / max_val // Leave some headroom
18    } else {
19        1.0
20    };
21
22    for &sample in samples {
23        let normalized = sample * scale;
24        let sample_i16 = (normalized * 32767.0).clamp(-32768.0, 32767.0) as i16;
25        writer
26            .write_sample(sample_i16)
27            .map_err(|e| format!("Failed to write sample: {}", e))?;
28    }
29
30    writer
31        .finalize()
32        .map_err(|e| format!("Failed to finalize WAV file: {}", e))?;
33
34    Ok(())
35 }
```

对比分析模块 (comparator.rs) Listing 21: Q3 与 Q4 结果对比 - 性能分析

```
1 use plotters::prelude::*;
2
3 pub struct ComparisonResult {
4     pub mse: f64,
5     pub max_diff: f64,
6     pub correlation: f64,
7     pub snr_db: f64,
8 }
9
10 /// Compare two signals
11 pub fn compare_signals(signal1: &[f64], signal2: &[f64]) -> ComparisonResult {
12     let n = signal1.len().min(signal2.len());
13
14     // Mean Squared Error
```

```

15     let mse: f64 = (0..n)
16         .map(|i| {
17             let diff = signal1[i] - signal2[i];
18             diff * diff
19         })
20         .sum::<f64>() / n as f64;
21
22 // Maximum absolute difference
23 let max_diff = (0..n)
24     .map(|i| (signal1[i] - signal2[i]).abs())
25     .fold(0.0f64, f64::max);
26
27 // Correlation coefficient
28 let mean1 = signal1[..n].iter().sum::<f64>() / n as f64;
29 let mean2 = signal2[..n].iter().sum::<f64>() / n as f64;
30
31 let cov: f64 = (0..n)
32     .map(|i| (signal1[i] - mean1) * (signal2[i] - mean2))
33     .sum::<f64>() / n as f64;
34
35 let var1: f64 = (0..n)
36     .map(|i| (signal1[i] - mean1).powi(2))
37     .sum::<f64>() / n as f64;
38
39 let var2: f64 = (0..n)
40     .map(|i| (signal2[i] - mean2).powi(2))
41     .sum::<f64>() / n as f64;
42
43 let correlation = if var1 > 0.0 && var2 > 0.0 {
44     cov / (var1.sqrt() * var2.sqrt())
45 } else {
46     0.0
47 };
48
49 // Signal-to-Noise Ratio (treating difference as noise)
50 let signal_power: f64 = signal1[..n].iter().map(|&x| x * x).sum::<f64>() / n as
51     f64;
52 let noise_power = mse;
53
54 let snr_db = if noise_power > 0.0 {
55     10.0 * (signal_power / noise_power).log10()
56 } else {
57     f64::INFINITY
58 };
59
60 ComparisonResult {
61     mse,
62     max_diff,
63     correlation,
64     snr_db,
65 }
66
67 /// Save comparison results to file
68 pub fn save_comparison(result: &ComparisonResult, filename: &str) {
69     let mut content = String::new();

```

```

70     content.push_str("Q4 vs Q3 Comparison Results\n");
71     content.push_str("=====\\n\\n");
72     content.push_str(&format!("Mean Squared Error (MSE): {:.6e}\\n", result.mse));
73     content.push_str(&format!("Root Mean Squared Error (RMSE): {:.6e}\\n",
74         result.mse.sqrt()));
74     content.push_str(&format!("Maximum absolute difference: {:.6}\\n",
75         result.max_diff));
75     content.push_str(&format!("Correlation coefficient: {:.6}\\n",
76         result.correlation));
76     content.push_str(&format!("Signal-to-Noise Ratio: {:.2} dB\\n\\n", result.snr_db));
77
78     content.push_str("Interpretation:\\n");
79     content.push_str("-----\\n");
80
81     if result.correlation > 0.99 {
82         content.push_str(" Excellent correlation - signals are nearly identical\\n");
83     } else if result.correlation > 0.95 {
84         content.push_str(" Good correlation - signals are very similar\\n");
85     } else if result.correlation > 0.8 {
86         content.push_str("~ Moderate correlation - some differences present\\n");
87     } else {
88         content.push_str(" Low correlation - significant differences\\n");
89     }
90
91     if result.snr_db > 40.0 {
92         content.push_str(" Excellent SNR - minimal difference\\n");
93     } else if result.snr_db > 20.0 {
94         content.push_str(" Good SNR - acceptable difference\\n");
95     } else {
96         content.push_str("~ Low SNR - noticeable difference\\n");
97     }
98
99     content.push_str("\nMethod differences:\\n");
100    content.push_str("-----\\n");
101    content.push_str("Q3 (Time-domain):\\n");
102    content.push_str(" - Uses 8th-order Butterworth filters (non-ideal)\\n");
103    content.push_str(" - IIR filter implementation (Direct Form II)\\n");
104    content.push_str(" - Non-linear phase response\\n");
105    content.push_str(" - Gradual transition band\\n\\n");
106
107    content.push_str("Q4 (Frequency-domain):\\n");
108    content.push_str(" - Uses ideal brick-wall filters\\n");
109    content.push_str(" - Frequency-domain multiplication\\n");
110    content.push_str(" - No phase distortion from filters\\n");
111    content.push_str(" - Sharp cutoff\\n");
112
113    std::fs::write(filename, content).expect("Failed to save comparison");
114 }
115
116 /// Plot comparison of two signals
117 pub fn plot_comparison(signal1: &[f64], signal2: &[f64], filename: &str) {
118     let n = signal1.len().min(signal2.len()).min(2000); // Plot first 2000 samples
119
120     let root = BitMapBackend::new(filename, (1200, 800)).into_drawing_area();
121     root.fill(&WHITE).unwrap();
122 }
```

```

123  let max_val = signal1[..n].iter()
124    .chain(signal2[..n].iter())
125    .fold(0.0f64, |max, &x| max.max(x.abs()));
126
127  let mut chart = ChartBuilder::on(&root)
128    .caption("Q4 vs Q3 Signal Comparison", ("sans-serif", 40))
129    .margin(20)
130    .x_label_area_size(50)
131    .y_label_area_size(60)
132    .build_cartesian_2d(0..n, -max_val*1.1..max_val*1.1)
133    .unwrap();
134
135  chart
136    .configure_mesh()
137    .x_desc("Sample")
138    .y_desc("Amplitude")
139    .draw()
140    .unwrap();
141
142 // Plot Q4 signal
143 chart
144   .draw_series(LineSeries::new(
145     (0..n).map(|i| (i, signal1[i])),
146     &BLUE,
147   ))
148   .unwrap()
149   .label("Q4 (Frequency-domain)")
150   .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &BLUE));
151
152 // Plot Q3 signal
153 chart
154   .draw_series(LineSeries::new(
155     (0..n).map(|i| (i, signal2[i])),
156     &RED,
157   ))
158   .unwrap()
159   .label("Q3 (Time-domain)")
160   .legend(|(x, y)| PathElement::new(vec![(x, y), (x + 20, y)], &RED));
161
162 chart
163   .configure_series_labels()
164   .background_style(&WHITE.mix(0.8))
165   .border_style(&BLACK)
166   .draw()
167   .unwrap();
168
169 root.present().unwrap();
170 println!("  Saved: {}", filename);
171 }
```