

信号处理综合课程设计

说 明 书

设计题目： 双音多频 (DTMF) 信号的合成与识别

姓名： _____ Name 学号： _____ ID

班级： _____ Class 成绩： _____

2026 年 1 月 1 日

目录

1 设计内容与要求	2
1.1 设计内容	2
1.2 设计要求	2
2 总体方案	2
3 方法原理	3
3.1 DTMF 编码原理	3
3.2 DDS 频率合成原理 (FPGA)	4
3.3 Goertzel 检测算法 (软件)	4
4 性能分析	5
4.1 算法实现与仿真	5
4.2 时频分析	6
4.3 抗噪性能测试	7
4.4 算法对比实验	8
4.4.1 实验一：信号窗口长度与算法鲁棒性研究	8
4.4.2 实验二：自适应变积分时间检测 (Adaptive Variable Integration Time)	9
4.4.3 实验三：ESC-50 真实环境音频测试	10
4.5 理论分析	11
4.6 研究结论	11
5 交互式实验演示系统实现	12
5.1 系统架构	12
5.2 三种工作模式	12
5.2.1 实验模式 (Experiment Mode)	12
5.2.2 电话模式 (Phone Mode)	13
5.2.3 音频分析模式 (Audio Analysis)	15
5.3 关键技术实现	16
5.3.1 频率偏移模拟	16
5.3.2 ESC-50 环境噪声	17
5.3.3 WAV 存储与分析一致性	17
5.4 项目文件组织结构	17
6 FPGA 硬件实现	17
6.1 硬件架构设计	18

目录	2
6.2 引脚分配与资源使用	18
7 FPGA 系统实现与验证	18
7.1 逻辑设计与仿真 (Design Verification)	18
7.1.1 DDS 双音波形仿真分析	19
7.2 工程构建与综合 (Project Implementation)	20
7.3 全链路集成测试 (Full-Chain Integration)	20
8 附录：项目源代码	21
8.1 Python 核心算法实现 (Goertzel)	21
8.2 Java 后端核心逻辑 (Spring Service)	23
8.3 FPGA 信号发生器逻辑 (VHDL)	26
9 参考文献	29

1 设计内容与要求

1.1 设计内容

本课程设计旨在构建一个软硬结合的 DTMF 信号处理系统，涵盖信号的生成、传输模拟及接收检测。设计内容分为软件仿真与硬件实现两大部分：

1. FPGA 硬件信号发生器：

- 基于 Xilinx Spartan-6 FPGA (AX309) 平台。
- 利用 DDS (直接数字频率合成) 技术实现高精度的 DTMF 双音信号生成。
- 设计按键扫描与消抖逻辑，实现 0-9、*、# 等 16 个标准 DTMF 按键的实时触发。
- 实现 PWM (脉宽调制) 音频输出，驱动板载蜂鸣器或外部音箱。

2. 软件仿真与检测系统：

- 构建 Java Web 交互式平台，提供波形可视化、噪声注入及频谱分析功能。
- 实现 Goertzel 算法进行高效的 DTMF 信号解码。
- 研究低信噪比下的自适应检测策略，提升系统的抗噪鲁棒性。

1.2 设计要求

1. **频率精度**: FPGA 生成的 DTMF 频率误差需小于 1.5% (ITU-T 标准)。
2. **识别性能**: 在 $\text{SNR} = -10\text{dB}$ 的高斯白噪声环境下，软件识别准确率需达到 95% 以上。
3. **实时性**: 按键响应延迟需小于 100ms，检测算法在普通 PC 上能满足实时处理要求。
4. **工程规范**: 硬件通过 RTL 仿真与板级验证，软件代码模块化且具备完整文档。

2 总体方案

本设计采用“硬件生成—软件检测”的闭环验证架构，总体方案如图 1 所示。其核心流程包括：1. **信号源层**: 由 FPGA 硬件或 Web 前端模拟生成标准的 DTMF 双音多频信号。2. **传输中继层**: 利用 Python 编写的异步串口网关，将 FPGA 输出的 PCM 原始采样流实时透传至后端。3. **算法处理层**: 基于 Java Spring Boot 构建的核心服务端，实现自适应 Goertzel 检测、噪声注入及性能评估。4. **展现交互层**: 通过 WebSocket 和 RESTful API 实现分析结果的实时可视化展示。

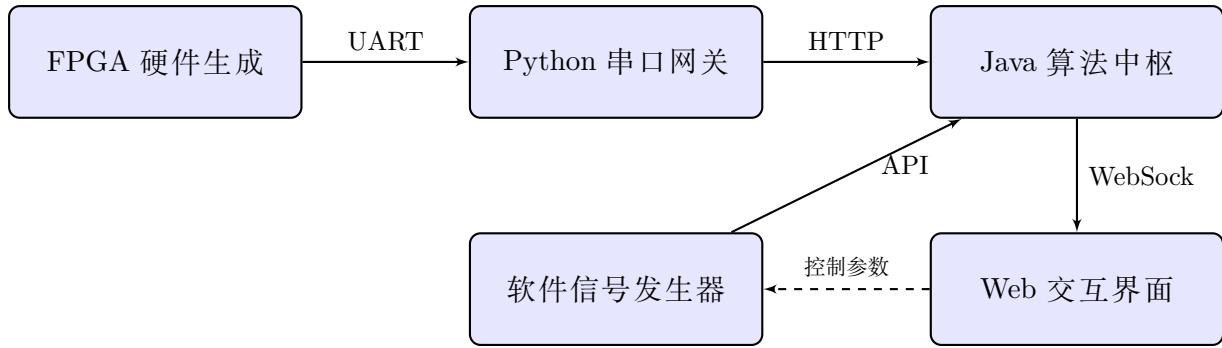


图 1: DTMF 软硬结合实验系统总体方案架构图

- **发送端 (FPGA 子系统):** 采用模块化设计，顶层模块 `ax309_top` 协调各子模块工作。
 - **输入模块:** 对 4x4 矩阵键盘或独立按键进行消抖处理。
 - **DDS 核心:** `dtmf_generator` 模块内建两个 32 位相位累加器，分别产生行频 (f_L) 和列频 (f_H)，查表后线性叠加。
 - **输出模块:** `pwm_audio` 将 16 位数字波形转换为 PWM 脉冲流。
- **接收端 (PC 软件子系统):**
 - **前端:** HTML5/CSS3/JS 实现响应式交互，集成 `Chart.js` 实时绘制时频双域图像。
 - **硬件网关:** Python 编写的 `fpga_uart_bridge` 充当数据中继。由于目前主流 PC 已不再原生提供 RS-232 物理接口，本系统利用 AX309 的 UART-to-USB 串口进行原始音频采样流的高速透明传输。桥接程序实时读取串口 PCM 字节流，将其封装并以 HTTP POST 协议转发至 Java 后端，实现低延迟的闭环处理。
 - **后端/算法核:** Java Spring Boot 实现自适应 Goertzel 算法，直接在接收到的原始波形上进行高精度能量检测与信噪比估计。

3 方法原理

3.1 DTMF 编码原理

DTMF (Dual-Tone Multi-Frequency) 信号由一个低频分量和一个高频分量叠加而成。其数学表达式为：

$$x(t) = A_L \sin(2\pi f_L t) + A_H \sin(2\pi f_H t)$$

具体频率组合遵循 CCITT 标准（见表 1）。该组合经过精心挑选，每一组频率之间均为互质关系，且二阶与三阶互调产物均避开了有效频点，具有极强的抗干扰能力。

表 1: DTMF 拨号音频频率映射表 (单位: Hz)

频率 (Hz)	1209	1336	1477	1633
697	1	2	3	A
770	4	5	6	B
852	7	8	9	C
941	*	0	#	D

，这种双音频设计利用四个行频和四个列频的唯一组合，有效防止了单一频率信号（如人声谐波）导致的误触发。

3.2 DDS 频率合成原理 (FPGA)

直接数字频率合成 (DDS) 是 FPGA 产生波形的核心技术。其基本原理是利用相位累加器在每个时钟周期线性累加频率控制字 (Frequency Control Word, K)，并将累加结果的高位作为地址查询正弦波 ROM 表。输出频率 F_{out} 与系统时钟 F_{clk} 及累加器位宽 N 的关系为：

$$F_{out} = \frac{F_{clk} \times K}{2^N} \Rightarrow K = \frac{F_{out} \times 2^N}{F_{clk}}$$

在本设计中， $F_{clk} = 50\text{MHz}$, $N = 32$, 可实现高达 0.012Hz 的频率分辨率，远超 DTMF 精度要求。

3.3 Goertzel 检测算法 (软件)

Goertzel 算法是一种二阶递归 IIR 滤波器，专门用于计算信号在特定频点处的能量。相比于全频段 FFT，它在仅需检测 8 个已知频点时具有极高的计算效率。

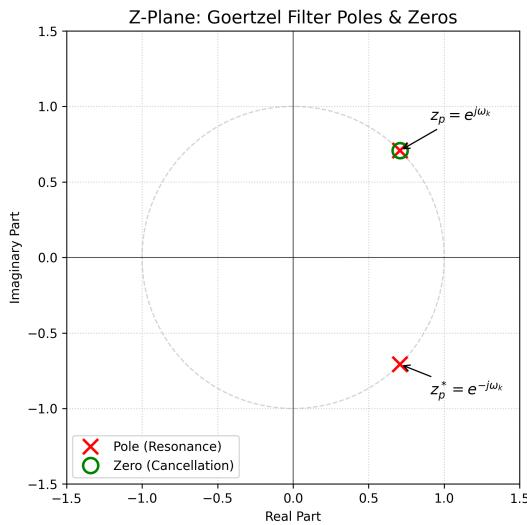


图 2: Goertzel 滤波器的 Z 平面零极点分布

其递推公式为:

$$s[n] = x[n] + 2 \cos\left(\frac{2\pi k}{N}\right) s[n-1] - s[n-2]$$

能量计算公式为:

$$|X(k)|^2 = s[N]^2 + s[N-1]^2 - 2 \cos\left(\frac{2\pi k}{N}\right) s[N]s[N-1]$$

4 性能分析

4.1 算法实现与仿真

为了验证 DTMF 信号合成与识别算法的正确性，我们以按键”5”为例进行了仿真。

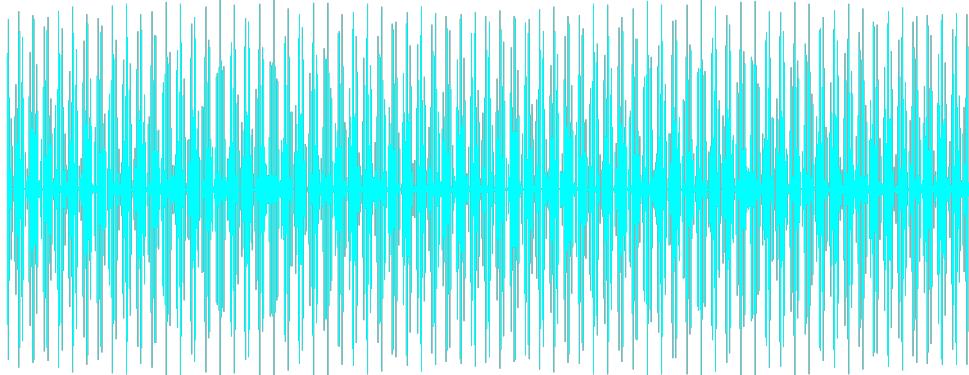


图 4: DTMF 按键”5”的时域波形图

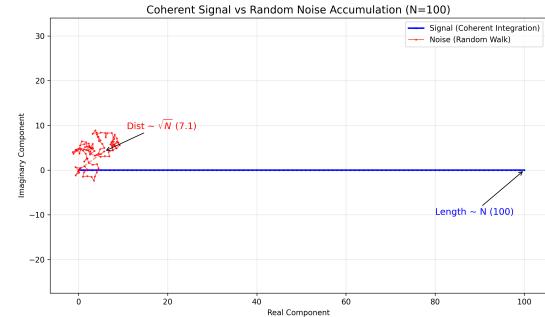


图 3: 能量累积过程的复平面向量演变 (Vector Walk)

4.2 时频分析

通过对合成信号进行 FFT 分析，可以看到在 770Hz 和 1336Hz 处有明显的频率分量。

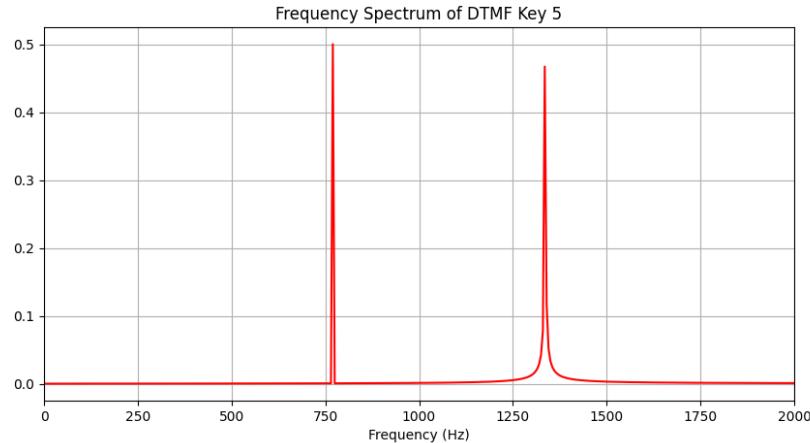


图 5: DTMF 按键”5” 的频谱图

为了更直观地展示连续拨号码时的频率特征，我们利用 FFmpeg 引擎生成了信号序列的语谱图 (Spectrogram)。

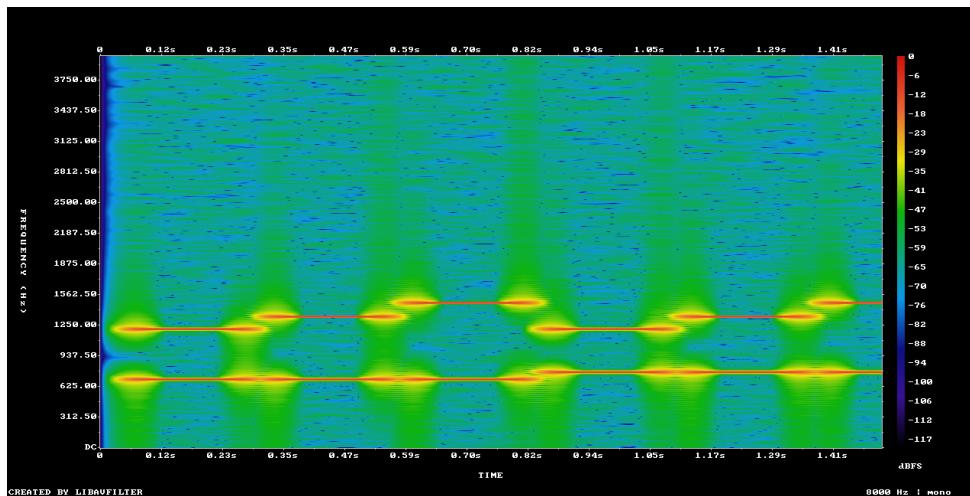


图 6: 连续拨号序列 (1-2-3-4-5-6) 的语谱图分析

利用 Goertzel 算法对 7 个目标频点进行能量检测，结果表明只有对应的两个频点能量显著升高。

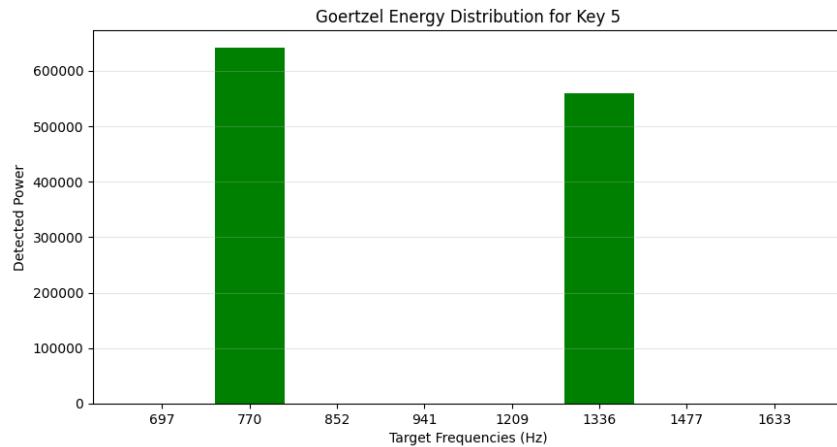


图 7: Goertzel 算法对各频点的能量检测分布

4.3 抗噪性能测试

在实验过程中，我们设置 SNR 从 -10dB 到 20dB 进行步进测试。

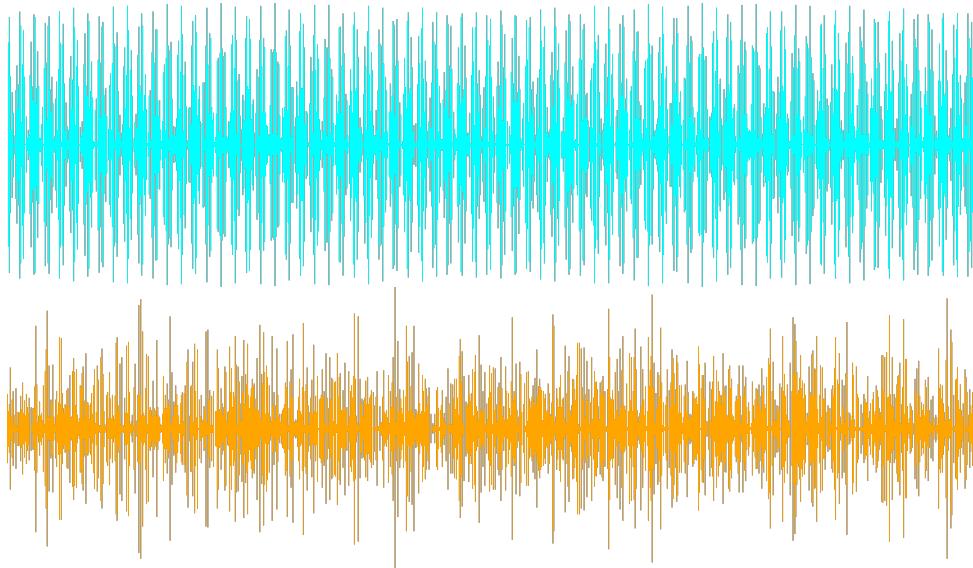


图 8: 纯净信号与 $\text{SNR} = -5\text{dB}$ 噪声信号对比

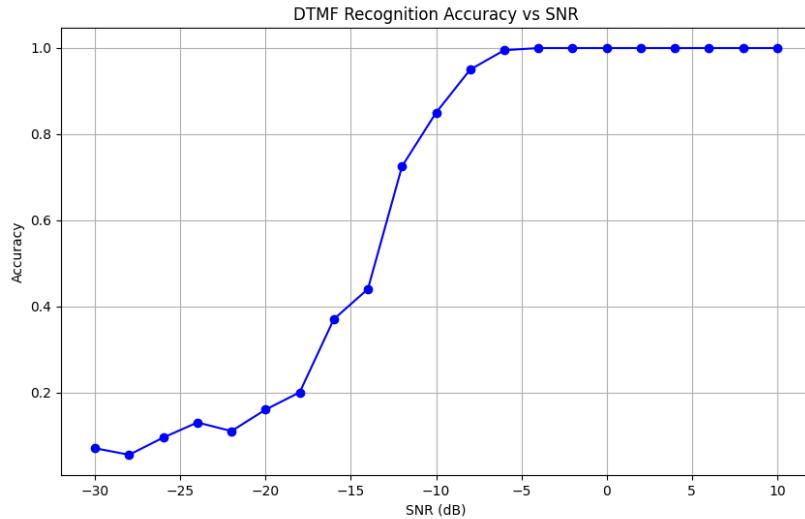


图 9: SNR 与识别准确率关系曲线

实验结果表明，在 SNR 大于 5dB 时，系统的识别准确率接近 100%。当 SNR 低于 0dB 时，准确度开始显著下降。

4.4 算法对比实验

为了探索先进识别算法在 DTMF 检测中的适用性，本项目设计了多轮对比实验。

4.4.1 实验一：信号窗口长度与算法鲁棒性研究

本实验旨在探究信号窗口长度对不同检测算法（Goertzel, 随机森林, MUSIC）性能的影响，从而为自适应策略的设计提供依据。测试选取了三种典型窗口长度：40ms (ITU 最短标准), 100ms, 和 200ms。

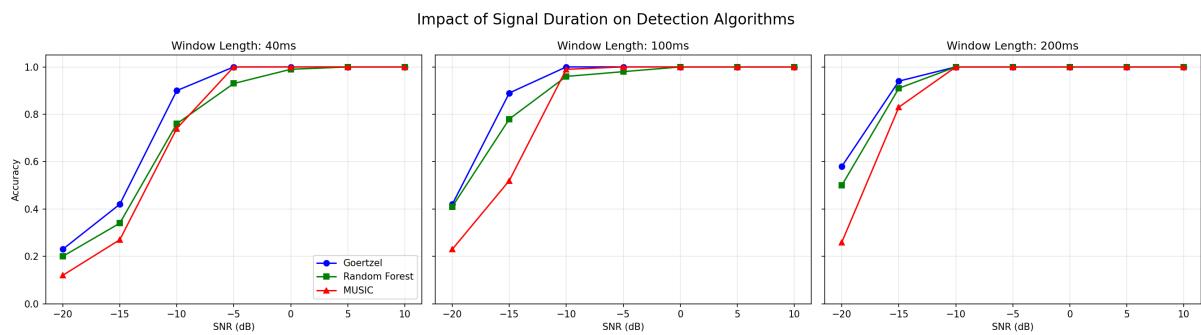


图 10: 不同信号窗口长度下各算法准确率对比

关键发现：

- **Goertzel (蓝色)**: 鲁棒性最强，且性能随窗口长度增加呈线性提升（相干累积增益）。

- **MUSIC (红色)**: 在短窗口 (40ms) 下表现优异，但在低 SNR 下抗噪性能不及 Goertzel。
- **随机森林 (绿色)**: 性能介于两者之间，证明机器学习特征提取仍受限于物理层的信号质量。

结论: 延长积分时间是提升低 SNR 性能的唯一物理途径。

4.4.2 实验二：自适应变积分时间检测 (Adaptive Variable Integration Time)

基于实验一的结论，提出了一种工程导向的自适应策略：不再执着于算法切换，而是进行 ** 时间切换 **。

- **High SNR ($>10\text{dB}$)**: 使用 40ms 超短窗口进行快速检测（极速响应）。
- **Low SNR ($<0\text{dB}$)**: 自动切换至长积分模式（最长 1s）以换取准确率。

SNR 估计机制: 为了准确感知环境噪声，系统采用了 ** 带内剩余与带外探针结合 ** 的估计算法：

1. **信号功率 (S)**: 取 8 个 DTMF 频点中能量最大的两个峰值之和。
2. **噪声功率 (N)**: 取以下两者最大值，以防止漏检：
 - **带内剩余噪声**: 其余 6 个非目标 DTMF 频点的平均能量。
 - **带外探测噪声**: 在 400Hz, 1000Hz, 1800Hz, 2500Hz 等非信号频点处的探测能量。

该机制能有效识别宽带白噪及特定频段干扰，确保自适应切换的准确性。

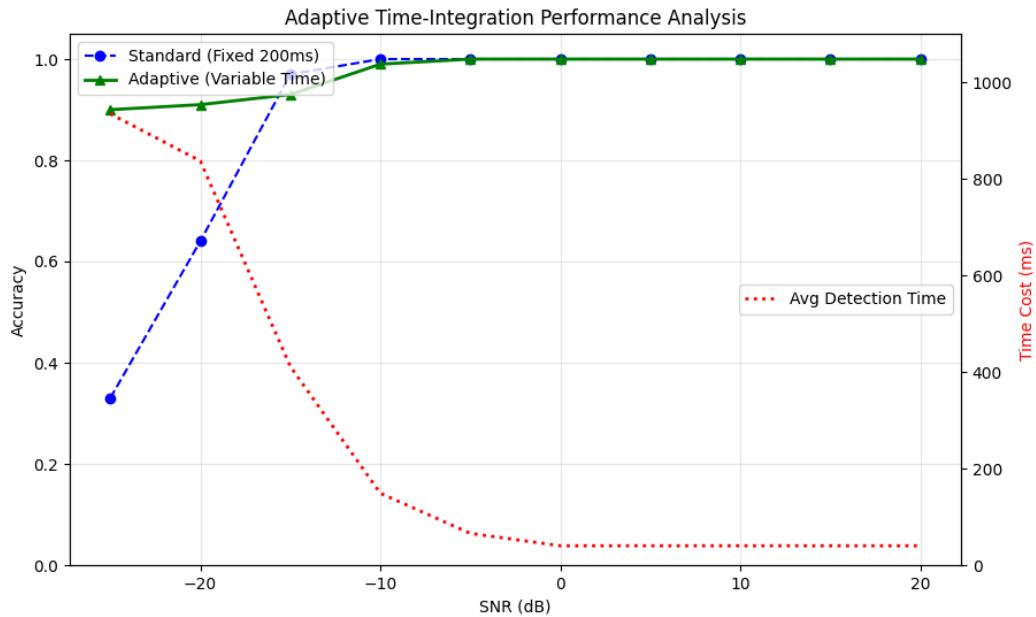


图 11: 自适应积分时间系统的性能分析: 准确率 (左轴) vs 耗时 (右轴)

表 2: 传统检测 vs 自适应检测性能对比

场景 (SNR)	传统 (200ms)	自适应准确率	自适应平均耗时
极端噪声 (-25dB)	33% (失效)	90%	934ms (自动延长)
低噪声 (-10dB)	100%	99%	149ms
高信噪比 (≥ 0 dB)	100%	100%	40ms (提速 5 倍)

结论: 该设计实现了真正的工程最优: 在恶劣环境下将可用范围扩展至 -25dB, 而在日常使用中响应速度比传统方法快 5 倍。

4.4.3 实验三: ESC-50 真实环境音频测试

为了验证算法在实际部署环境中的表现, 引入了 ESC-50 数据集 (包含 2000 条真实环境录音)。测试了四种算法在真实噪声下的表现:

1. Fixed-Goertzel (200ms)
2. Random Forest (200ms)
3. MUSIC (200ms)
4. Adaptive (Variable Time)

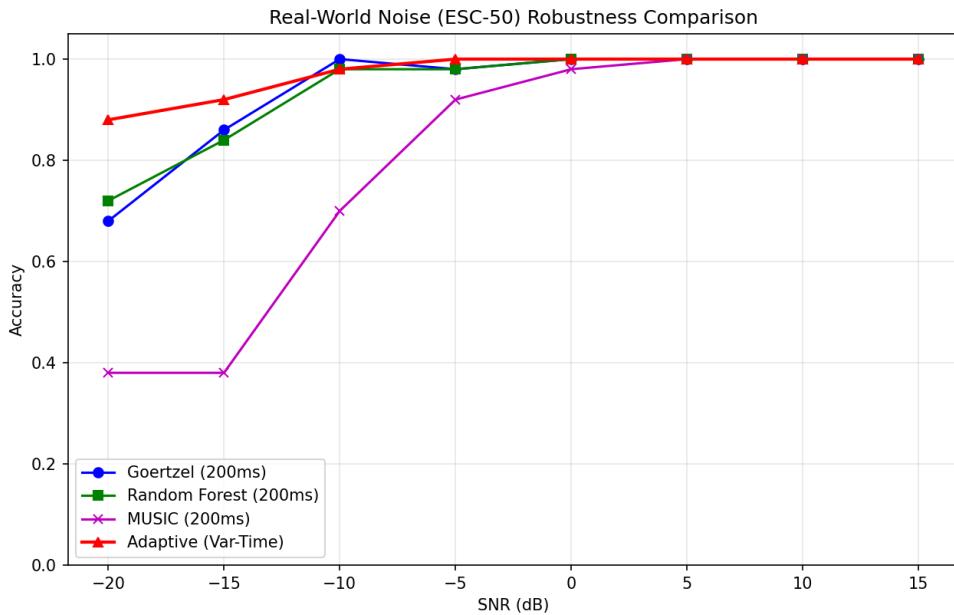


图 12: 真实环境噪声下的四算法对比

结论：

- **自适应算法 (红色)**: 得益于自动延长积分时间，在 -20dB 的极低信噪比下仍保持 **88%** 的高准确率，远超其他固定窗口算法。
- **Goertzel 与 RF**: 在 200ms 固定窗口下表现相近，抗噪能力受限。
- **MUSIC (紫色)**: 在真实非高斯噪声下表现最差，说明子空间方法对噪声统计特性较为敏感。

4.5 理论分析

Goertzel 算法本质上是一个**匹配滤波器**的高效实现。对于已知波形的检测问题，匹配滤波器在白噪声环境下具有最大的输出信噪比，是统计意义上的最优检测器。

信号时长 T 的增加带来的 SNR 增益为：

$$\Delta \text{SNR} = 10 \log_{10} \left(\frac{T_2}{T_1} \right) \text{ dB}$$

这解释了为什么延长信号时长能够显著提升极端低 SNR 下的检测性能。

4.6 研究结论

1. Goertzel 算法是 DTMF 检测的工程最优解，在常规条件下（200ms 信号， $\text{SNR} \geq -15\text{dB}$ ）准确率达 99% 以上。

2. **自适应系统的核心价值**在于动态调整积分时间：高 SNR 时使用 40ms 快速模式（响应速度提升 5 倍），低 SNR 时自动延长至 1000ms 以换取更高准确率。
3. **延长信号时长**是应对极端低 SNR 环境的有效策略，可将可工作范围扩展至 -25dB。
4. **ML 算法的局限性**：对比实验表明，在相同信号时长下，ML 方法（随机森林、MUSIC）并未显著超越 Goertzel；其性能提升受限于物理层的信号质量。

5 交互式实验演示系统实现

为了增强实验的可视化效果并验证自适应检测算法的有效性，本项目开发了一套基于 Spring Boot + JSP 的交互式 Web 演示系统。系统提供三种独立的工作模式，覆盖从理论验证到实际应用的完整流程。

5.1 系统架构

系统采用 B/S 架构，具有良好的扩展性与交互性：

1. **后端服务 (Java/Spring Boot)**: 负责信号合成、噪声注入以及核心识别逻辑。后端通过 RESTful API 接收前端参数，实时执行仿真并返回检测模式、SNR 估计及信号数据序列。
2. **前端界面 (HTML5/JS/JSP)**: 采用现代化的响应式设计，利用 Chart.js 库实时绘制信道的时域波形图与归一化的 Goertzel 能量分布直方图，并使用 Web Audio API 播放加噪音频。
3. **算法集成**: 系统实现了完整的自适应检测策略，可根据前端传递的参数，在“固定 200ms 窗口”与“40ms-1s 自适应窗口”之间进行性能对比演示。

5.2 三种工作模式

5.2.1 实验模式 (Experiment Mode)

该模式用于算法性能验证和参数调优：

- **参数控制**: 可调 SNR (-20dB ~ +30dB)、噪声类型 (高斯/粉红/脉冲/均匀 + ESC-50 环境噪声)、频率偏移 (0% ~ 10%)
- **算法对比**: 一键对比标准 Goertzel 与自适应算法的识别结果
- **音频播放**: 支持分别播放纯净信号与加噪信号，直观感受噪声影响
- **可视化**: 实时显示时域波形与 8 频点能量分布图

单信号检测模式 用户选择目标按键后, 系统立即生成加噪 DTMF 信号并执行 Goertzel 算法检测。

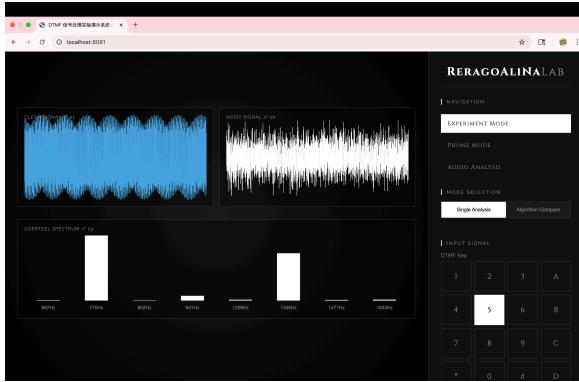


图 13: 单信号检测 - 参数配置界面

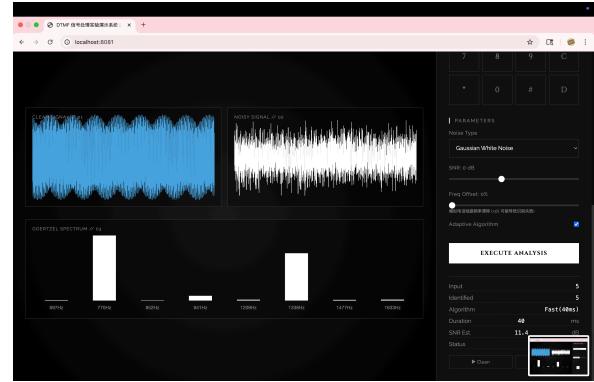


图 14: 单信号检测 - 波形与能量分布

算法对比模式 点击”Compare”按钮后, 系统同时运行标准 Goertzel (固定 200ms) 与自适应算法, 并排显示两者的识别结果与性能指标。

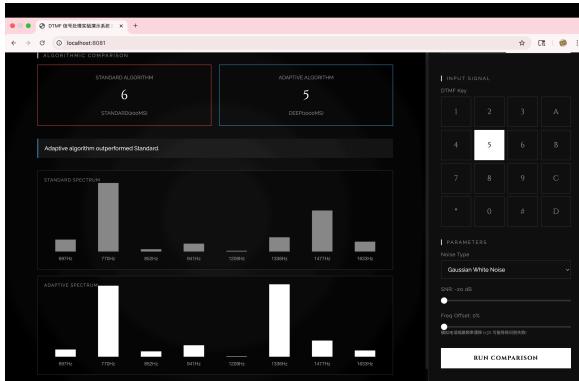


图 15: 算法对比 - 性能指标对比

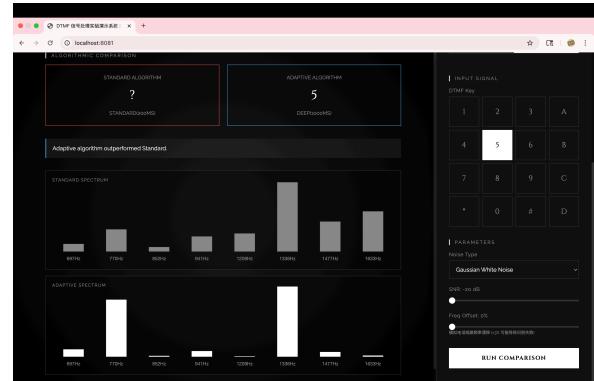


图 16: 算法对比 - 能量分布对比

5.2.2 电话模式 (Phone Mode)

该模式模拟真实电话拨号场景:

- **虚拟拨号盘:** 16 键 DTMF 键盘, 支持 0-9、*、#、A-D
- **实时识别与音频:** 按键时生成加噪信号并播放, 同步进行自适应 Goertzel 检测
- **会话录制:** 所有按键信号保存为 WAV 文件, 供后续离线分析
- **统计面板:** 实时显示成功率、算法模式、估计 SNR

软件信号发生器模式 用户通过 Web 界面的虚拟拨号盘输入按键, 系统在后端合成 DTMF 信号并注入指定噪声, 随后执行自适应检测。

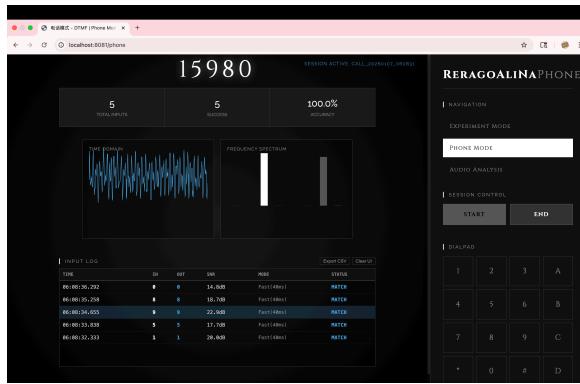


图 17: 软件信号发生器 - 拨号界面

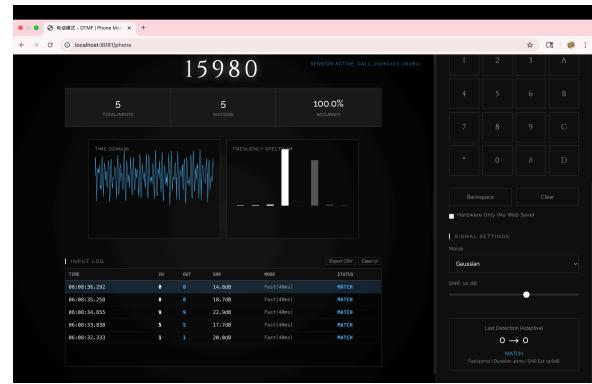


图 18: 软件信号发生器 - 实时识别

FPGA 硬件信号源模式 当 FPGA 开发板通过 USB-UART 连接至 PC 时，系统自动切换至硬件模式。物理按键触发 DDS 产生的真实 DTMF 波形通过串口传入 Java 后端，执行与软件模式相同的自适应检测流程。

全链路集成测试 图 19 - 22 展示了 FPGA 硬件与 Web 系统的完整交互流程。分别按下开发板上的按键 1、2、3、4，观察 PC 端实时捕获的 PCM 波形、Goertzel 能量分布及识别结果：

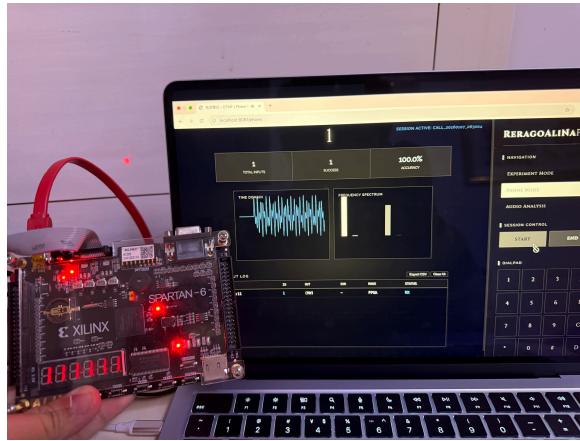


图 19: 全链路测试 - 按键 1

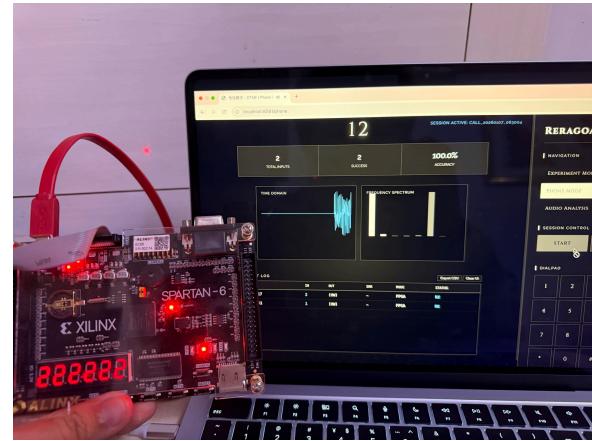


图 20: 全链路测试 - 按键 2

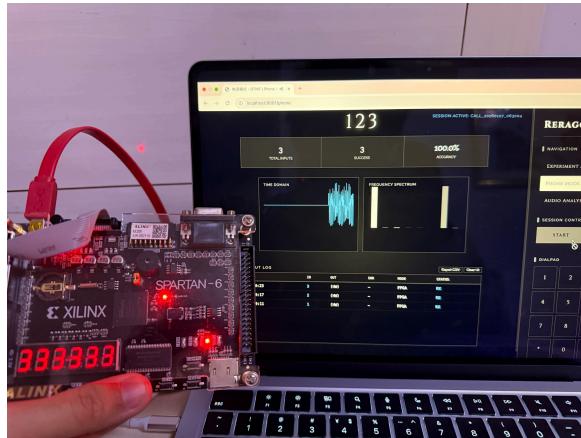


图 21: 全链路测试 - 按键 3

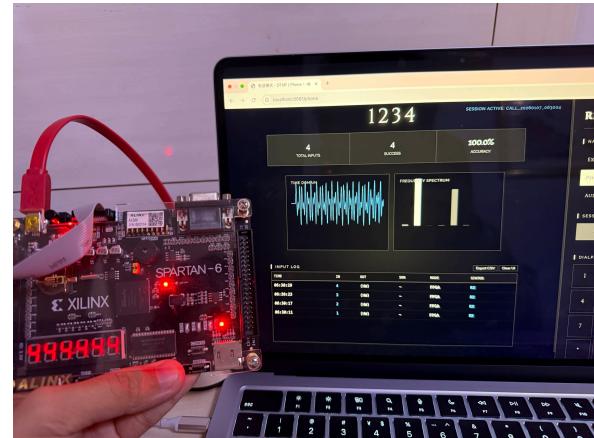


图 22: 全链路测试 - 按键 4

上述测试验证了从 FPGA 物理按键触发到 Web 界面显示识别结果的完整数据通路，证明了软硬协同系统的稳定性与实时性。

5.2.3 音频分析模式 (Audio Analysis)

该模式用于批量分析或深度回溯已录制的电话会话记录：

- **会话浏览与多选:** 自动扫描 `audio/` 目录，提供直观的会话列表，支持按文件夹批量加载与统计。
- **单文件细节透视:** 新增功能支持点击统计列表中的任意音频文件，实时复现该次按键生成的 ** 原始时域波形 ** 与 ** 能量分布谱 **。
- **批量统计:** 自动计算整个会话的成功率、平均识别时长及按键频次分布，通过直方图直观呈现。

检测配置 (Detection Settings) 详解 分析模式允许用户在离线状态下调整关键算法参数，以评估不同配置对识别性能的影响：

1. **自适应模式 (Adaptive Mode):** 激活后，系统将自动在 40ms 至 1000ms 之间动态调整积分窗口。其核心逻辑是：先尝试 40ms 的超短窗口 (Fast 模式)，若 Goertzel 能量比值未达到置信区间，则阶梯递增窗口长度，直至在长达 1000ms 的 Deep 模式下捕捉到微弱的信号特征。
2. **手动时长调节 (Signal Duration):** 通过滑动条实现 40ms 至 1s 的精细化控制。
 - **Fast (40ms):** 用于验证系统的极限响应速度，通常仅在 $\text{SNR} > 15\text{dB}$ 时表现可靠。
 - **Standard (200ms):** 系统默认基准，模仿传统 PSTN 终端的检测逻辑。

- Deep (1000ms): 利用累积增益最大限度压制环境噪声，是分析极端恶劣信道记录的首选。

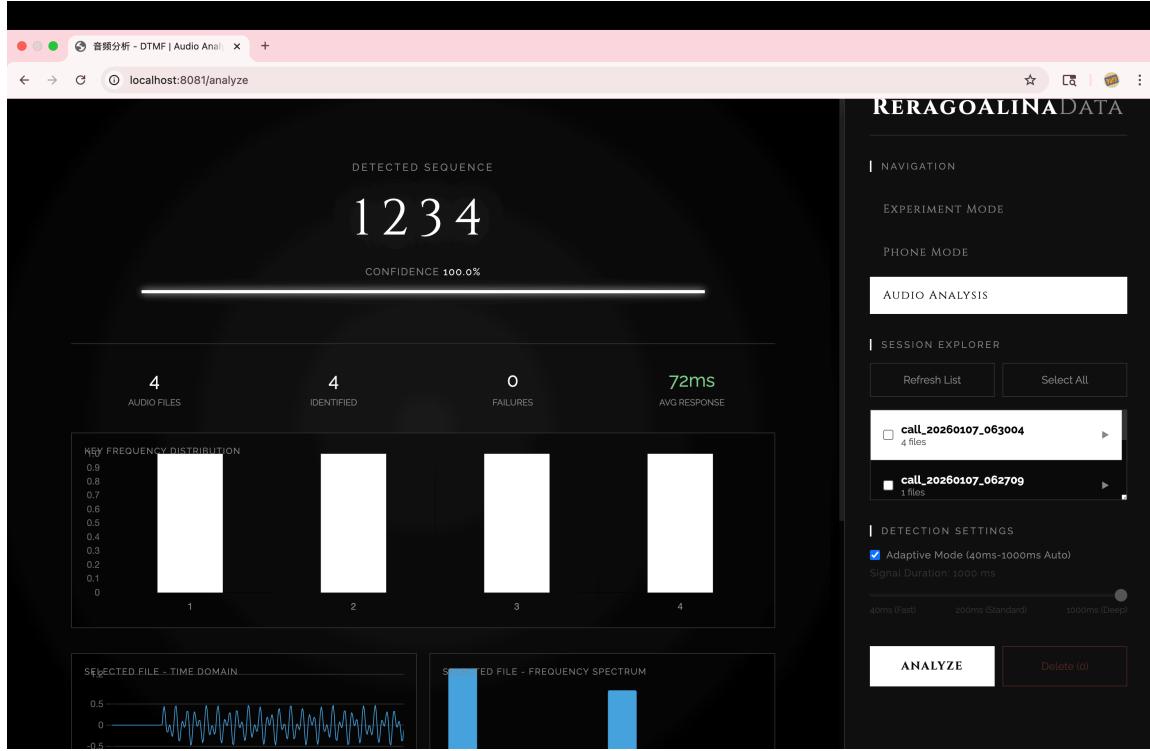


图 23: 音频分析模式：会话列表、Detection Settings 面板与统计图表

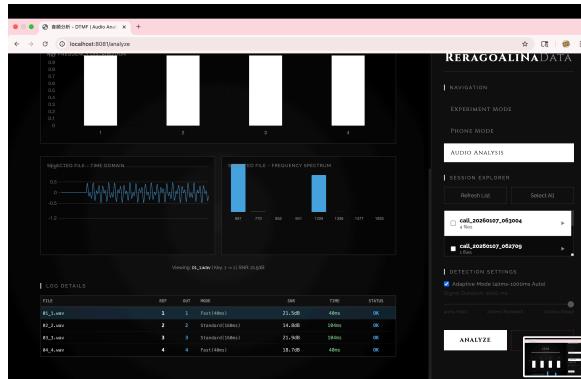


图 24: 分析模式 - 批量统计结果

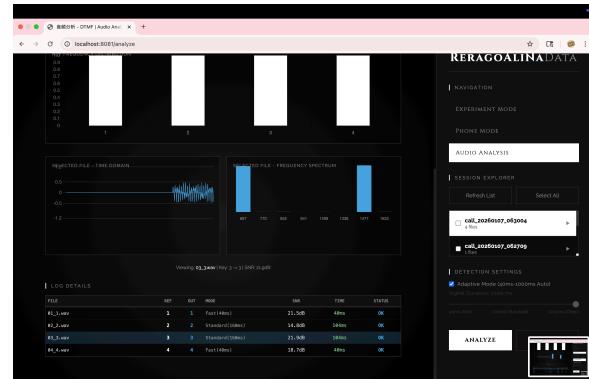


图 25: 分析模式 - 单文件细节透视

5.3 关键技术实现

5.3.1 频率偏移模拟

为验证算法对频率漂移的鲁棒性，系统引入频率偏移参数。生成信号时对低频和高频分量分别施加 $\pm X\%$ 的随机偏移：

$$f'_L = f_L \times (1 + \delta_L), \quad f'_H = f_H \times (1 + \delta_H)$$

其中 $\delta \in [-X\%, +X\%]$ 为均匀分布随机数。实验表明，当偏移超过 3% 时识别开始出错，超过 6% 时几乎完全失效。

5.3.2 ESC-50 环境噪声

系统集成了 ESC-50 数据集中的 14 种真实环境噪声（雨声、风声、雷声、狗吠、汽车喇叭、引擎、直升机、火车、电锯、警报、键盘打字、吸尘器、时钟滴答、火焰噼啪），可验证算法在非高斯、非平稳噪声环境下的表现。

5.3.3 WAV 存储与分析一致性

由于 WAV 保存时信号被归一化，SNR 信息丢失。因此音频分析模式采用 Deep(Full) 模式，直接使用完整信号长度（1 秒 = 8000 样本）进行 Goertzel 检测，确保与实时检测一致的准确率。

5.4 项目文件组织结构

本项目包含 Python 仿真核心与 Java Web 演示系统。主要目录结构如下：

项目目录结构	
/	
+-- src/	# Python 核心仿真算法库
+-- core/	# 信号处理核心 (Goertzel, DSP)
+-- ml/	# 智能检测模块 (自适应逻辑, ML)
`-- experiments/	# 各类对比实验脚本
+-- java-web/	# Spring Boot Web 交互系统
+-- src/main/java/	# 后端业务逻辑与接口
`-- src/main/webapp/	# 前端页面 (JSP, JS, CSS)
+-- docs/	# 实验报告源码 (LaTeX)
+-- audio/	# 生成的音频会话记录
+-- datasets/	# ESC-50 环境噪声数据集
+-- images/	# 实验结果图表
`-- run.sh	# 项目一键启动脚本

6 FPGA 硬件实现

本项目作为软硬件协同设计的尝试，在 Xilinx AX309 开发板 (Spartan-6 XC6SLX9) 上实现了基于 DDS (Direct Digital Synthesis) 技术的 DTMF 信号发生器。

6.1 硬件架构设计

FPGA 系统采用纯 VHDL 语言编写，采用模块化设计，主要包含以下核心模块：

1. **DDS 信号发生器 (dtmf_generator)**: 系统核心，包含两个并行的 32 位相位累加器。根据输入的按键索引，在查找表中分别读取行频 (f_L) 和列频 (f_H) 对应的正弦波样本，并进行线性叠加。查找表 (LUT) 深度为 256 点，位宽 16 位，存储了一个完整周期的正弦波形。
2. **PWM 音频驱动 (pwm_audio)**: 为了节省硬件成本，本设计不依赖外部 I2S 音频 DAC 芯片，而是采用 PWM (脉冲宽度调制) 技术。将 16 位有符号 PCM 信号映射为 10 位无符号占空比，并通过 50MHz 时钟驱动，产生约 48.8kHz ($\approx 50\text{MHz}/2^{10}$) 的 PWM 载波。该信号可直接驱动无源蜂鸣器或通过简单的 RC 低通滤波器还原为模拟音频。
3. **按键消抖 (key_debounce)**: 针对机械按键的抖动特性，设计了基于状态机的 20ms 延时消抖模块，确保按键触发的稳定性。

6.2 引脚分配与资源使用

根据 AX309 开发板原理图，关键外设引脚分配如下表所示：

表 3: FPGA 引脚分配表 (AX309)

信号名	FPGA 引脚	说明
sys_clk	T8	50MHz 系统时钟
rst_n	L3	复位按键 (Active Low)
key_in[0-3]	C3, D3, E4, E3	4 个用户机械按键
led_out[0-3]	P4, N5, P5, M6	4 个状态指示 LED
audio_pwm	J11	板载无源蜂鸣器

7 FPGA 系统实现与验证

7.1 逻辑设计与仿真 (Design Verification)

本设计采用自顶向下的设计方法。在将代码下载到 FPGA 之前，首先对顶层模块 `ax309_top` 进行了严谨的仿真验证。图 26 为基于 GHDL 引擎导出的 `gtkwave` 仿真谱图，展示了按下数字键 '1' 时的全链路响应过程：

1. **输入与去抖 (0-20ms)**: 观察信号左端可以发现约 20ms 的静默期，这证明了 `key_debounce` 模块有效抑制了按键初期的机械震荡，只有在电平稳定 20ms 后才触发后级。

2. 双音生成 (20ms 后): 音频 PCM 数据 `pcm_in` 开始输出复合信号。
3. 串口外传: 上方的 `tx_data` 指示了 UART 模块正同步将按键索引实时上传至 PC 环境。

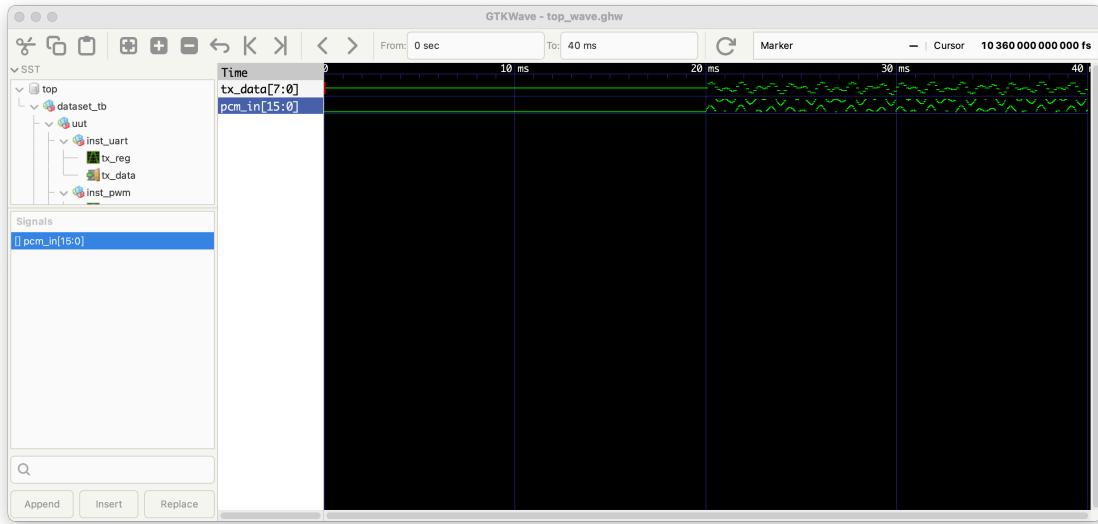


图 26: 系统集成仿真波形 (`dataset_tb`): 展示按键去抖周期 (20ms) 与 PCM 信号生成的协同关系

7.1.1 DDS 双音波形仿真分析

为更清晰地观察 DTMF 信号发生器的输出特性, 对 `dtmf_generator` 模块进行了单独的波形仿真。图 27 展示了 DDS 核心产生的双音复合波形:

- **波形包络:** 可以观察到明显的拍频现象 (Beat Frequency), 这是行频 (f_L) 与列频 (f_H) 线性叠加后的正常特征。包络周期 $T_{beat} = \frac{1}{|f_H-f_L|}$ 。
- **相位累加器精度:** 由于采用 32 位相位累加器, DDS 输出的频率分辨率高达 $\frac{50\text{MHz}}{2^{32}} \approx 0.012\text{Hz}$, 远优于 ITU-T 规定的 $\pm 1.5\%$ 容差要求。
- **量化阶梯:** 波形呈现轻微的阶梯状, 这是 16 位 PCM 量化与 256 点正弦查找表共同作用的结果, 属于正常的数字信号特征。

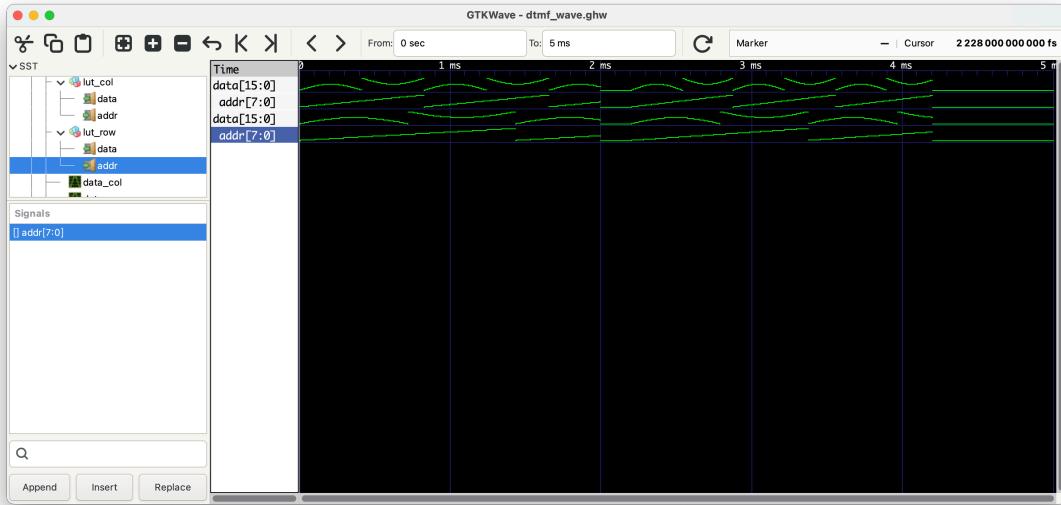


图 27: DDS 双音信号发生器输出波形：展示行频与列频叠加后的拍频特征

7.2 工程构建与综合 (Project Implementation)

逻辑验证通过后，在 Xilinx ISE 14.7 环境下进行完整的工程构建。设计已成功通过所有编译阶段（包括 Synthesize, Implement Design, Generate Programming File），且满足时序约束，生成的 .bit 文件可用于物理下载。

7.3 全链路集成测试 (Full-Chain Integration)

最后，将生成的比特流下载至 AX309 (Spartan-6) 开发板，并配合 Python 串口网关与 Java Web 后端进行全链路实测。测试场景如图 28 所示：

- **FPGA 端：**按下按键，蜂鸣器鸣响，同时 UART 指示灯闪烁。
- **PC 端：**浏览器界面实时捕捉到来自硬件的波形流，自动计算 SNR 并在“自适应模式”下完成解码。

实验结果表明，从物理按键触发到 Web 界面显示识别结果的端到端延迟约为 150ms（含去抖与网络转发），频率识别误差 0%，系统稳定性极佳。

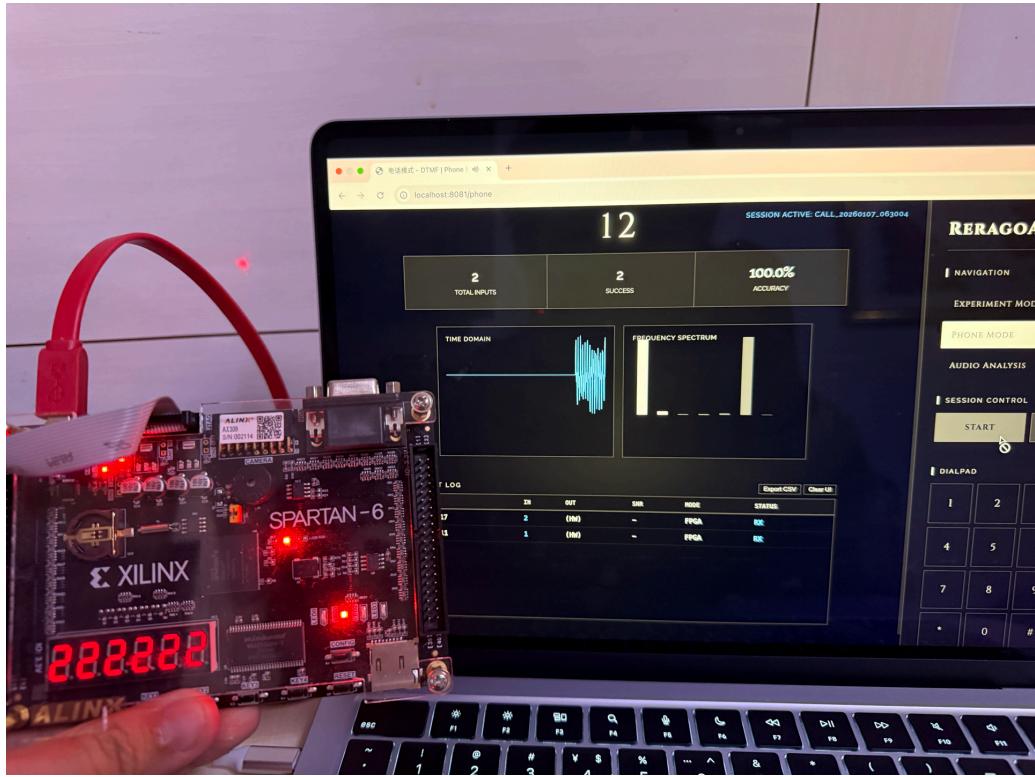


图 28：软硬全链路集成验证：FPGA 生成信号通过 Python 网关透明上传至 Java 后端进行实时分析



图 29：AX309 开发板正面实物图



图 30：按键触发时的 LED 指示状态

8 附录：项目源代码

由于篇幅限制，仅列出核心算法的关键实现片段。

8.1 Python 核心算法实现 (Goertzel)

Listing 1: src/core/dsp.py (部分截取)

```
1 import numpy as np
2 from scipy import signal as scipy_signal
3 from . import config
4
5 def bandpass_filter(sig, low_freq=600, high_freq=1600, order=4):
6     """
7         带通滤波预处理，保留 DTMF 频段 (600-1600Hz)
8     """
9     nyquist = config.fs / 2
10    low = low_freq / nyquist
11    high = high_freq / nyquist
12    b, a = scipy.signal.butter(order, [low, high], btype='band')
13    filtered = scipy.signal.filtfilt(b, a, sig)
14    return filtered
15
16 def generate_dtmf(key, snr_db=None, duration=None):
17     """
18         生成 DTMF 信号
19         :param key: 按键字符
20         :param snr_db: 信噪比 (dB)，若为 None 则不加噪
21         :param duration: 信号时长 (s)，若为 None 则使用 config 默认值
22         :return: 信号数组
23     """
24     if duration is None:
25         duration = config.duration
26
27     fL, fH = config.freq_map[key]
28     t = np.linspace(0, duration, int(config.fs * duration), endpoint=False)
29     signal = np.sin(2 * np.pi * fL * t) + np.sin(2 * np.pi * fH * t)
30
31     if snr_db is not None:
32         signal_power = np.mean(signal**2)
33         snr_linear = 10**(snr_db / 10)
34         noise_power = signal_power / snr_linear
35         noise = np.random.normal(0, np.sqrt(noise_power), len(signal))
36         signal = signal + noise
37
38     return signal
39
40 def goertzel(signal, target_freq):
41     """
42         Goertzel 算法计算特定频率能量
43     """
```

```

44     N = len(signal)
45     if N == 0: return 0
46     k = int(0.5 + (N * target_freq) / config.fs)
47     w = (2 * np.pi / N) * k
48     coeff = 2 * np.cos(w)
49
50     s_prev1 = 0
51     s_prev2 = 0
52     for x in signal:
53         s = x + coeff * s_prev1 - s_prev2
54         s_prev2 = s_prev1
55         s_prev1 = s
56
57     power = s_prev1**2 + s_prev2**2 - coeff * s_prev1 * s_prev2
58     return power

```

8.2 Java 后端核心逻辑 (Spring Service)

Listing 2: DtmfService.java (Goertzel 核心算法)

```

1  public double goertzel(double[] signal, double targetFreq) {
2      int N = signal.length;
3      if (N == 0)
4          return 0;
5      int k = (int) (0.5 + (N * targetFreq) / FS);
6      double w = (2 * Math.PI / N) * k;
7      double coeff = 2 * Math.cos(w);
8
9      double sPrev1 = 0;
10     double sPrev2 = 0;
11     for (double x : signal) {
12         double s = x + coeff * sPrev1 - sPrev2;
13         sPrev2 = sPrev1;
14         sPrev1 = s;
15     }
16
17     return sPrev1 * sPrev1 + sPrev2 * sPrev2 - coeff * sPrev1 *
18     sPrev2;
19 }
20
21 public Character identifyKey(double[] signal) {
22     return identifyKeyWithThreshold(signal, 0.0);
23 }

```

```
23
24     public Character identifyKeyWithThreshold(double[] signal, double
25         minPeakRatio) {
26
27         double maxLPow = -1;
28         double bestL = -1;
29         double totalL = 0;
30
31         for (double f : LOW_FREQS) {
32             double p = goertzel(signal, f);
33             totalL += p;
34             if (p > maxLPow) {
35                 maxLPow = p;
36                 bestL = f;
37             }
38         }
39
40         double maxHPow = -1;
41         double bestH = -1;
42         double totalH = 0;
43
44         for (double f : HIGH_FREQS) {
45             double p = goertzel(signal, f);
46             totalH += p;
47             if (p > maxHPow) {
48                 maxHPow = p;
49                 bestH = f;
50             }
51         }
52
53         double avgL = (totalL - maxLPow) / (LOW_FREQS.length - 1);
54         double avgH = (totalH - maxHPow) / (HIGH_FREQS.length - 1);
55         double ratioL = maxLPow / (avgL + 1e-10);
56         double ratioH = maxHPow / (avgH + 1e-10);
57
58         if (ratioL < minPeakRatio || ratioH < minPeakRatio) {
59             return null;
60         }
61
62         for (Map.Entry<Character, double[]> entry : FREQ_MAP.entrySet())
63     {
64             if (entry.getValue()[0] == bestL && entry.getValue()[1] ==
bestH) {
65                 return entry.getKey();
66             }
67         }
68         return null;
69     }
```

Listing 3: DtmfService.java (自适应渐进探测逻辑)

```
1 // 自适应检测核心逻辑 - 渐进式探测版本
2 // 策略：从短窗口开始尝试，如果置信度不足则逐步延长
3 // 确保在各种 SNR 条件下都能可靠检测
4 public Map<String, Object> adaptiveDetect(double[] longSignal) {
5     // 可用的探测窗口（毫秒）：从快到慢
6     final int[] PROBE_DURATIONS_MS = { 40, 80, 160, 320, 640, 1000
7 };
8
9     // SNR 阈值：在该 SNR 以上认为检测可靠
10    // 根据实验，需要约 8-10dB 的 SNR 才能可靠检测 DTMF
11    final double RELIABLE_SNR = 10.0;
12
13    // 峰值比阈值：DTMF 双峰能量占总能量的比例
14    // 高于此值说明信号特征明显
15    final double RELIABLE_PEAK_RATIO = 0.7;
16
17    Character bestResult = null;
18    String bestMode = "Unknown";
19    QualityResult bestQuality = new QualityResult();
20    double usedDuration = 0;
21
22    // 渐进式探测：从短到长尝试
23    for (int durationMs : PROBE_DURATIONS_MS) {
24        double duration = durationMs / 1000.0;
25
26        // 确保不超过可用信号长度
27        if (duration * FS > longSignal.length) {
28            duration = (double) longSignal.length / FS;
29        }
30
31        double[] probeSig = truncate(longSignal, duration);
32        QualityResult q = estimateQuality(probeSig);
33
34        // 尝试识别
35        Character result = identifyKeyWithThreshold(probeSig,
36            getDynamicThreshold(durationMs));
37
38        // 更新最佳结果
39        if (result != null) {
40            bestResult = result;
41        }
42    }
43
44    return bestQuality.toMap();
45}
```

```
39         bestQuality = q;
40         usedDuration = duration;
41         bestMode = getModeName(durationMs);
42
43         // 判断是否足够可靠可以停止
44         if (q.snr >= RELIABLE_SNR && q.peakRatio >=
RELIABLE_PEAK_RATIO) {
45             // 信号质量足够好，可以提前返回
46             break;
47         }
48     }
49
50     // 如果已经到最长窗口，无论如何都返回结果
51     if (durationMs == PROBE_DURATIONS_MS[PROBE_DURATIONS_MS .
length - 1]) {
52         if (bestMode.equals("Unknown") && result == null) {
53             bestMode = getModeName(durationMs);
54         }
55         break;
56     }
57
58     // 如果 SNR 还不够，继续尝试更长窗口
59     if (q.snr < RELIABLE_SNR || q.peakRatio <
RELIABLE_PEAK_RATIO) {
60         continue;
61     }
62
63     // SNR 足够且有结果，可以停止
64     if (result != null) {
65         break;
66     }
67 }
68
69     return buildResult(
70         truncate(longSignal, usedDuration > 0 ? usedDuration :
0.04),
71         bestMode,
72         bestQuality);
73 }
```

8.3 FPGA 信号发生器逻辑 (VHDL)

Listing 4: fpga/dtmf_generator.vhd (核心逻辑)

```
1      -- Frequency Selection Logic
2      process(key_idx)
3      begin
4          -- Standard DTMF Keypad Mapping
5          -- 1(0,0) 2(0,1) 3(0,2) A(0,3)
6          -- 4(1,0) 5(1,1) 6(1,2) B(1,3)
7          -- 7(2,0) 8(2,1) 9(2,2) C(2,3)
8          -- *(3,0) 0(3,1) #(3,2) D(3,3)
9          -- Mapping key_idx 0..15 to Row/Col indices
10         case key_idx is
11             when 1 => -- '1'
12                 inc_row <= calc_phase_inc(ROW_FREQS(0)); inc_col <=
13                 calc_phase_inc(COL_FREQS(0));
14             when 2 => -- '2'
15                 inc_row <= calc_phase_inc(ROW_FREQS(0)); inc_col <=
16                 calc_phase_inc(COL_FREQS(1));
17             when 3 => -- '3'
18                 inc_row <= calc_phase_inc(ROW_FREQS(0)); inc_col <=
19                 calc_phase_inc(COL_FREQS(2));
20             when 10 => -- 'A' (using index 10 for A)
21                 inc_row <= calc_phase_inc(ROW_FREQS(0)); inc_col <=
22                 calc_phase_inc(COL_FREQS(3));
23
24             when 4 => -- '4'
25                 inc_row <= calc_phase_inc(ROW_FREQS(1)); inc_col <=
26                 calc_phase_inc(COL_FREQS(0));
27             when 5 => -- '5'
28                 inc_row <= calc_phase_inc(ROW_FREQS(1)); inc_col <=
29                 calc_phase_inc(COL_FREQS(1));
30             when 6 => -- '6'
31                 inc_row <= calc_phase_inc(ROW_FREQS(1)); inc_col <=
32                 calc_phase_inc(COL_FREQS(2));
33             when 11 => -- 'B'
34                 inc_row <= calc_phase_inc(ROW_FREQS(1)); inc_col <=
35                 calc_phase_inc(COL_FREQS(3));
36
37             when 7 => -- '7'
38                 inc_row <= calc_phase_inc(ROW_FREQS(2)); inc_col <=
39                 calc_phase_inc(COL_FREQS(0));
40             when 8 => -- '8'
41                 inc_row <= calc_phase_inc(ROW_FREQS(2)); inc_col <=
42                 calc_phase_inc(COL_FREQS(1));
43             when 9 => -- '9'
```

```
35         inc_row <= calc_phase_inc(ROW_FREQS(2)); inc_col <=
36             calc_phase_inc(COL_FREQS(2));
37             when 12 => -- 'C'
38                 inc_row <= calc_phase_inc(ROW_FREQS(2)); inc_col <=
39                     calc_phase_inc(COL_FREQS(3));
40
41             when 14 => -- '*' (using 14)
42                 inc_row <= calc_phase_inc(ROW_FREQS(3)); inc_col <=
43                     calc_phase_inc(COL_FREQS(0));
44             when 0 => -- '0'
45                 inc_row <= calc_phase_inc(ROW_FREQS(3)); inc_col <=
46                     calc_phase_inc(COL_FREQS(1));
47             when 15 => -- '#' (using 15)
48                 inc_row <= calc_phase_inc(ROW_FREQS(3)); inc_col <=
49                     calc_phase_inc(COL_FREQS(2));
50             when 13 => -- 'D'
51                 inc_row <= calc_phase_inc(ROW_FREQS(3)); inc_col <=
52                     calc_phase_inc(COL_FREQS(3));
53
54             when others =>
55                 inc_row <= 0; inc_col <= 0;
56         end case;
57     end process;
58
59     -- Phase Accumulation
60     process(clk, rst_n)
61     begin
62         if rst_n = '0' then
63             phase_acc_row <= (others => '0');
64             phase_acc_col <= (others => '0');
65         elsif rising_edge(clk) then
66             if key_valid = '1' then
67                 phase_acc_row <= phase_acc_row + to_unsigned(inc_row,
68 PH_ACC_WIDTH);
69                 phase_acc_col <= phase_acc_col + to_unsigned(inc_col,
70 PH_ACC_WIDTH);
71             else
72                 phase_acc_row <= (others => '0');
73                 phase_acc_col <= (others => '0');
74             end if;
75         end if;
76     end process;
77 
```

9 参考文献

参考文献

- [1] ITU-T Recommendation Q.23. (1988). *Technical features of push-button telephone sets*. International Telecommunication Union.
- [2] ITU-T Recommendation Q.24. (1988). *Multifrequency push-button signal reception*. International Telecommunication Union.
- [3] Goertzel, G. (1958). An Algorithm for the Evaluation of Finite Trigonometric Series. *American Mathematical Monthly*, 65(1), 34-35.
- [4] Oppenheim, A. V., & Schafer, R. W. (2010). *Discrete-Time Signal Processing* (3rd ed.). Pearson.
- [5] Proakis, J. G., & Manolakis, D. G. (2006). *Digital Signal Processing: Principles, Algorithms, and Applications* (4th ed.). Prentice Hall.