第 3 章

声纹识别与声源定位

项目描述

在多人环境中，如会议室或公共场所，准确捕捉并跟踪特定说话人的声音是一项挑战，尤其是当存在多个声源和背景噪声时。本章将指导读者学习基于话筒阵列的先进语音分离和说话人跟踪技术。该技术通过形成针对感兴趣说话人的波束来增强其声音信号，同时利用方向置零来抑制其他说话人和环境噪声。此外，通过自适应算法实时跟踪说话人的位置变化，确保声音质量的持续优化。这些技术的应用不仅可以提高会议记录的质量，还可以在安全监控和智能家居系统中提供有效的声音识别与跟踪，从而增强系统的响应能力和准确性。

学习目标

- ❑ 理解端点检测的概念和基本原理。
- ❑ 理解音频特征提取的原理和方法。
- ❑ 理解基于话筒阵列的声源定位原理。
- ❑ 掌握端点检测技术。
- ❑ 掌握声纹识别技术与应用。
- ❑ 掌握基于话筒阵列的声源定位技术。

3.1 任务 1：端点检测


23min

语音端点检测（Voice Activity Detection，VAD）是识别音频流中语音存在的起始和终点的技术。该技术对于改进语音识别系统、减少处理无声段的计算资源消耗及优化音频数据存储都非常关键。通过精确的端点检测，可以有效地区分语音和非语音部分，支持更复杂的声音处理任务，如语音识别和声音追踪。本任务将指导读者使用基本的音频信号处理技

术检测音频中的有效语音活动的方法,实现一个简单的端点检测模型,这是构建任何高效语音处理应用的基础。

3.1.1 任务分析

语音端点检测任务需要分析音频信号以确定哪些部分包含语音,这是语音处理应用中的一个基本且关键步骤,音频信号处理在此任务中扮演着至关重要的角色。

3.1.2 必备知识:音频信号处理

声音是自然界的物理信号,经过转换可得到数字语音信号。语音信号有 3 个重要的参数:声道数、采样频率和量化位数。

声道数:指的是音频信号中独立声道的数量,它决定了声音的空间分布和定位能力。常见的声道数包括单声道、双声道、2.1 声道、四声道、5.1 声道、7.1 声道、杜比全景声。

采样频率:指在数字音频系统中,每秒对模拟声音信号进行采样的次数。它决定了数字音频信号能够准确再现的最高频率。根据奈奎斯特-香农采样定理,为了无失真地再现一个信号,采样频率至少应该是信号最高频率的两倍。

常见的采样频率如下。

8kHz:电话系统中常用的采样频率,可以提供基本的语音清晰度。

16kHz:在一些语音应用和低质量的音乐流媒体服务中使用,可以提供比电话更好的音质。

22.05kHz:早期的 CD 音频标准之一,但现在已经不常用。

44.1kHz:CD 音频的标准采样频率,可以提供高质量的音乐。

48kHz:电影和视频制作中常用的采样频率,它可以与视频的帧率很好地同步。

96kHz:高分辨率音频中常用的采样频率,可以提供更宽的频率响应和更好的音质。

192kHz:在一些高端音频设备和专业录音中使用,可以提供极高的音质和频率响应。

采样频率越高,数字音频信号能够准确再现的频率范围就越宽,音质通常也越好,但是,更高的采样频率也会导致更大的数据量和更高的处理要求,因此在实际应用中,需要根据具体需求和资源限制来选择合适的采样频率。

量化位数:指的是在数字化过程中,用来表示每个样本的二进制位数(Binary Digit,bit)。位数越高,可以表示的动态范围越大,声音的细节也就越丰富。常见的有 8 位、16 位、24 位、32 位等。

声音是由物体振动产生的机械波,通过介质(如空气、水或固体)传播。当物体振动时,它会使周围的介质分子或原子产生压缩和稀疏,形成连续的压缩波和稀疏波,这些波就是声波。声波图是声波的一种图形化表示方法,用来展示声波的物理特性,包括振幅、频率和时间等。声波(Waveform)如图 3-1 所示。在波形图中,横轴表示时间,纵轴表示音频振动幅度,在每个时刻都有一个对应的音频值。波形图可以直观地展示声音的波形,包括波峰和波谷,以及声音的起始和结束。

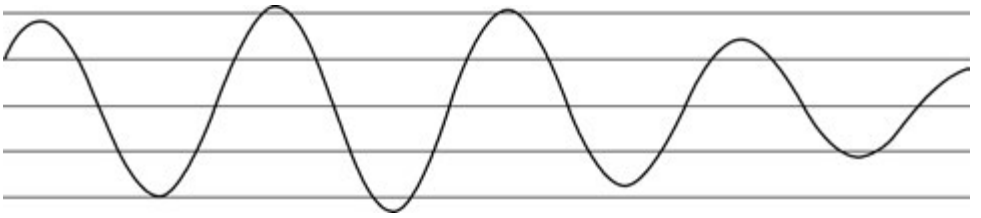

图 3-1　波形图

音频信号是模拟信号，需要将其保存为数字信号，才能对语音进行算法操作。

模拟信号是指用连续变化的物理量表示的信息，其信号的幅度，或频率，或相位随时间连续变化，或在一段连续的时间间隔内，其代表信息的特征量可以在任意瞬间呈现为任意数值的信号。

与模拟信号相对的是数字信号，数字信号是模拟信号的离散化表示，数字信号是在时间和幅度上都离散的信号。

1. 音频信号采集

在某些特定的时刻对模拟信号进行测量叫作采样，所得信号叫作离散时间信号。

采样率越高，越接近模拟信号。每秒读取数千次样本，并记录表示该时间点声波高度的数字，可以用 WAV 格式的音频文件(非压缩)进行保存。

声波转换为数字，需要等间距点记录声波的高度，如图 3-2 所示。

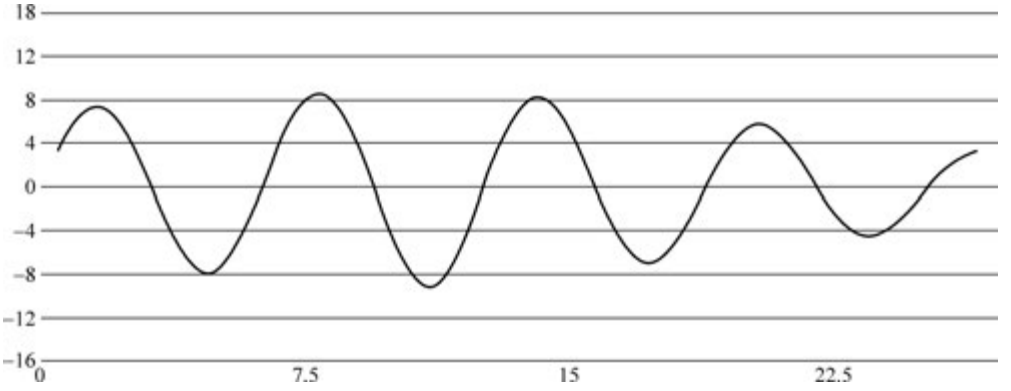

图 3-2　音频信号记录

离散时间信号的特点：时间上离散，幅度上连续。

例如输入电压的范围是 0～0.7V，在 t_1 时刻得到的采样值可以是 0.2718V。

将 0.2718V 交给计算机去处理，如果计算机只能处理 2 位浮点型数值，就会出现信息丢失的问题。

把信号幅度取值的数目加以限定，由有限个数值组成的信号称为离散幅度信号，如图 3-3 所示。

图 3-3 中的 0.2718V，量化后就可以取值为 0.3V 或者 0.2V。

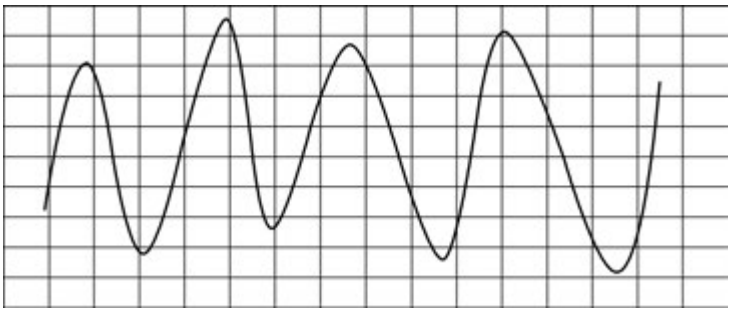

图 3-3　离散幅度信号

编码的含义是用预先规定的方法将文字、数字或其他信息编成二进制数码；用少量的基本符号，通过简单的组合表示大量复杂的信息。

在计算机内部，信息只有经过数字化编码后才能进行表示、存放和传递。

量化位数和采样率都是时域中的参数。一段音频(声波)的变化曲线，从时域上看，其横轴表示时间 t，纵轴表示幅度 v(一般是电压)。那么，采样率 44.1kHz 表示每秒采样 44 100 个点，也就是横轴上每隔 1/44 100 秒采集一个点，而采到的每个点都用一个数值来表示其幅度(电压)。假设整个音频信号的变化幅度范围是 $-5\sim5$V，我们将 $-5\sim5$V 分成 65 536 份，采到的这些点的数值 n(16 位)转换成电压，也就是($n\times10/65\ 536-5$)V，因此采样位数分解的是音频电压的幅度。

采样率越高，量化位数越高，信号失真度就越小，所得数据占用存储空间就越大，如图 3-4 所示。

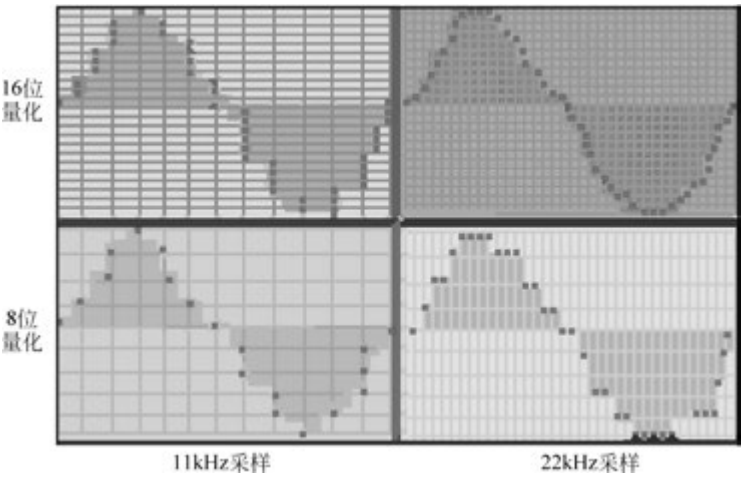

图 3-4　采样率、量化位数与存储空间

如果采样频率不低于信号最高频率的两倍，就能把以数字表达的声音还原为原来的声音。

例如声音信号的最高频率是3400Hz，若采样频率为8000Hz，则数字信号就能还原为原来的声音。

声音文件的数据量计算公式如下：

$$\text{声音文件数据量(字节 / 秒)} = \frac{\text{采样率(Hz)} \times \text{量化位数(bit)} \times \text{声道数}}{8} \tag{3-1}$$

例如计算对于5min双声道、16位量化位数、44.1kHz采样频率声音的不压缩数据量是多少？

$$\text{解：声音文件数据量(字节/秒)} = \frac{44.1 \times 1000 \times 16 \times 2 \times (5 \times 60)}{8 \times 1024 \times 1024} \approx 50.47\text{MB}$$

2. 音频信号读写

在数字语音信号处理中，既可以实时采集语音信号进行处理，也可以从文件中读取语音数据进行处理；采集到的数字语音信号或处理后的语音数据通常需要写入文件进行保存。语音信号读写通常涉及音频格式转换和数据压缩技术，以适应不同的存储和处理需求。常见的音频文件格式包括WAV、MP3、AAC等，其中WAV格式通常用于保持原始数据的完整性，而MP3和AAC则涉及数据压缩，可能会丢失部分音质信息。使用适当的库，如Python中的wave或pydub，可以方便地实现音频的读写操作。

3. 音频信号分析

在内容分析领域，音频信号可以用于声音场景识别、情感分析和语音识别等，例如，通过分析音频特征，如旋律、节奏和动态范围，可以推断音乐作品所要表达的情感。进行音频信号分析需要很多专业知识，音频信号分析的一些概念及术语如下。

(1) 波形：波形表示信号的形状、形式，波形可以千变万化，常见的基础波形包括正弦波、方波、三角波、锯齿波等。

(2) 声波：声音的传播形式。声波是一种平行波，由物体(声源)振动产生。

(3) 声场：声波传播的空间。

(4) 时域：描述数学函数或物理信号与时间的关系，例如一个信号的时域波形可以表达信号随着时间的变化。

(5) 频域：指在对函数或信号进行分析时，分析其和频率有关的部分，而不是和时间有关的部分，和时域一词相对。声波在频域上表现为振荡的频率。

音频信号分析是一个涉及多个方面的复杂过程，在完成音频信号采集后，进行音频信号分析的主要方法有以下几种。

1) 频谱分析

频谱分析是分析音频信号中不同频率成分的方法。快速傅里叶变换(FFT)是实现频域分析的核心算法，它可以将时间域信号转换为频域信号，帮助我们理解音频信号的频率分布。

2) 时域处理

时域处理包括基本的时域操作，如截取、拼接音频信号，以及声音的播放与录制。音频

信号的分析与可视化也是时域处理的一部分,可以通过绘制波形图来观察音频信号的振幅随时间的变化。

3) 时频分析

分析方法如短时傅里叶变换(STFT)和小波变换,用于研究信号在不同时间的频率分布情况。STFT 通过滑动窗口将信号分解为一系列短时段,对每个短时段进行傅里叶变换,从而得到时频图。

4) 音频特征提取

音频特征提取是描述音频信号的量,可以用于识别、分类和处理音频信号。常见的音频特征包括能量、零颈摇头指数、频带能量分布和模式识别等。

5) 音频信号增强与滤波技术

音频信号增强技术包括增益控制与均衡化,以及回声消除和噪声抑制。音频滤波主要是从音频信号或者数据中过滤掉某些成分(频率)的音频信号或数据。音频滤波在实现上可分为频域滤波、时域滤波。滤波器设计是音频信号处理中的重要部分,包括 FIR 和 IIR 滤波器理论,以及滤波器设计与实现。

6) 傅里叶定理

傅里叶变换是一种线性积分变换,用于信号在时域(或空域)和频域之间变换。任何连续测量的时序或信号都可以表示为不同频率的正弦波信号的无限叠加。

傅里叶定理揭示了一个信号可以分解成若干正弦波之和,它在信号处理领域有着极重要的意义。依据傅里叶定理可知,复杂的信号可以分解成很多正弦波,因此可以将处理复杂的信号转换为处理简单的正弦波信号。

频谱图:一个波可以分解成若干正弦波,这些正弦波是有固定的频率的(当然,它还有其他重要的性质,例如周期性、相位、振幅等),将这些正弦波的频率放到横坐标上,将它们的振幅放到纵坐标上,可以得到一个频谱图。

7) 语音端点检测

端点检测是语音处理中的基础步骤,其目标是在一段包含语音和非语音的混合信号中,准确地确定语音的起始点和终止点,从而将语音段与静音或背景噪声段区分开来。常用的端点检测算法主要包括短时能量和短时过零率两种方法。

(1) 短时能量:表示语音信号在一帧时间内的能量水平,通常用于检测语音段的大致范围。计算方法为对每帧中所有采样点的幅度平方求和。

(2) 短时过零率(Zero Crossing Rate,ZCR):表示语音信号在一帧中经过零值(正负号变化)的次数,通常用于识别高频成分较多的语音段(如清辅音)。

在实际应用中,由于语音两端常包含清音(如清辅音),这类声音的能量与静音部分相近,导致短时能量难以准确检测其边界。然而,清音的过零率通常显著高于静音段,因此可以在初步基于能量检测出的语音段基础上,继续向前后搜索,结合短时过零率来补充检测清音区域,从而更精确地确定语音的完整范围。

3.1.3 任务实施

1. 任务目标

- ❑ 能够计算音频信号的短时能量。
- ❑ 能够计算音频信号的过零率。
- ❑ 能够使用双门限法进行端点检测。

2. 实施环境

任务实施环境见表 3-1。

表 3-1 任务 1 实施环境

硬 件	软 件	资 源
高性能个人计算机 人工智能边缘计算实训设备 扬声器	Debian/Ubuntu/Windows 10(或 11)/macOS Python 3.10 wave numpy==1.23.5	wave/1～10.wav

3. 实施步骤

按照以下步骤完成本次实验：

(1) 定义帧能量计算函数。

(2) 定义过零率计算函数。

(3) 定义端点检测函数。

(4) 执行端点检测。

定义帧能量计算函数,代码如下：

```
#第 3 章/任务 1/lab.py
# - * - coding: utf - 8 - * -
import wave
import os
import numpy as np

def sgn(data):
    if data >= 0:
        return 1
    else:
        return 0
#计算每帧的能量,256 个采样点为一帧
def calEnergy(wave_data):
    energy = []
    sum = 0
    for i in range(len(wave_data)):
        sum = sum + (int(wave_data[i]) * int(wave_data[i]))
        if (i + 1) % 256 == 0:
            energy.append(sum)
            sum = 0
        elif i == len(wave_data) - 1:
```

```
            energy.append(sum)
    return energy
```

定义过零率计算函数,代码如下:

```
#第 3 章/任务 1/lab.py
#计算过零率
def calZeroCrossingRate(wave_data):
    zeroCrossingRate = []
    sum = 0
    for i in range(len(wave_data)):
        if i % 256 == 0:
            continue
        sum = sum + np.abs(sgn(wave_data[i]) - sgn(wave_data[i - 1]))
        if (i + 1) % 256 == 0:
            zeroCrossingRate.append(float(sum)/255)
            sum = 0
        elif i == len(wave_data) - 1:
            zeroCrossingRate.append(float(sum)/255)
    return zeroCrossingRate
```

定义端点检测函数,代码如下:

```
#第 3 章/任务 1/lab.py
#利用短时能量和短时过零率,使用双门限法进行端点检测
def endPointDetect(wave_data, energy, zeroCrossingRate):
    sum = 0
    energyAverage = 0
    for en in energy:
        sum = sum + en
    energyAverage = sum / len(energy)

    sum = 0
    for en in energy[:5]:
        sum = sum + en
    ML = sum / 5
    MH = energyAverage / 4                  #较高的能量阈值
    ML = (ML + MH) / 4                      #较低的能量阈值
    sum = 0
    for zcr in zeroCrossingRate[:5]:
        sum = float(sum) + zcr
    Zs = sum / 5                            #过零率阈值

    A = []
    B = []
    C = []
```

```
    #首先利用较大能量阈值 MH 进行初步检测
    flag = 0
    for i in range(len(energy)):
        if len(A) == 0 and flag == 0 and energy[i] > MH:
            A.append(i)
            flag = 1
        elif flag == 0 and energy[i] > MH and i - 21 > A[len(A) - 1]:
            A.append(i)
            flag = 1
        elif flag == 0 and energy[i] > MH and i - 21 <= A[len(A) - 1]:
            A = A[:len(A) - 1]
            flag = 1

        if flag == 1 and energy[i] < MH:
            A.append(i)
            flag = 0
    print("较高能量阈值,计算后的浊音 A:" + str(A))

    #利用较小能量阈值 ML 进行第 2 次能量检测
    for j in range(len(A)):
        i = A[j]
        if j % 2 == 1:
            while i < len(energy) and energy[i] > ML:
                i = i + 1
            B.append(i)
        else:
            while i > 0 and energy[i]> ML:
                i = i - 1
            B.append(i)
    print("较低能量阈值,增加一段语言 B:" + str(B))

    #利用过零率进行最后一次检测
    for j in range(len(B)):
        i = B[j]
        if j % 2 == 1:
            while i < len(zeroCrossingRate) and zeroCrossingRate[i]>= 3 * Zs:
                i = i + 1
            C.append(i)
        else :
            while i > 0 and zeroCrossingRate[i] >= 3 * Zs:
                i = i - 1
            C.append(i)
    print("过零率阈值,最终语音分段 C:" + str(C))
return C
```

执行端点检测,代码如下:

```
#第 3 章/任务 1/lab.py
for i in range(10):
    f = wave.open("wave/" + str(i+1) + ".wav", "rb")
    #调用 getparams() 一次性返回所有的 WAV 文件的格式信息
params = f.getparams()

    #nframes 为采样点数目
nchannels, sampwidth, framerate, nframes = params[:4]

    #readframes() 按照采样点读取数据
str_data = f.readframes(nframes)                    #str_data 是二进制字符串

    #以上两条可以直接写成 str_data = f.readframes(f.getnframes())

    #转换成二字节数组形式(每个采样点占两字节)
    wave_data = np.fromstring(str_data, dtype = np.short)
    print( "采样点数目:" + str(len(wave_data)))    #输出应为采样点数目
f.close()

    energy = calEnergy(wave_data)
    with open("energy/" + str(i+1) + "_en.txt", "w") as f:
        for en in energy:
            f.write(str(en) + "\n")
    zeroCrossingRate = calZeroCrossingRate(wave_data)
    with open("zeroCrossingRate/" + str(i+1) + "_zero.txt","w") as f:
        for zcr in zeroCrossingRate:
            f.write(str(zcr) + "\n")
    N = endPointDetect(wave_data, energy, zeroCrossingRate)
    J = 0
    frames = []
    while j < len(N):
        for num in wave_data[N[j] * 256:N[j+1] * 256]:
            frames.append(num)
        j = j + 2
    ff = wave.open('output/' + str(i+1) + '.wav', 'wb')
    ff.setnchannels(1)
    ff.setsampwidth(sampwidth)
    ff.setframerate(framerate)
    ff.writeframes(b''.join(frames))
    ff.close()
    print('output/' + str(i+1) + '.wav 保存完成')
```

在终端执行的命令如下：

```
python lab.py
```

执行过程部分的截图如图 3-5 所示。

执行结果保存在 output 目录下。

```
采样点数目：55245
较高能量阈值，计算后的浊音A:[60, 184]
较低能量阈值，增加一段语言B:[58, 184]
过零率阈值，最终语音分段C:[58, 184]
output/6.wav保存完成
采样点数目：87714
较高能量阈值，计算后的浊音A:[50, 290]
较低能量阈值，增加一段语言B:[49, 291]
过零率阈值，最终语音分段C:[49, 291]
output/7.wav保存完成
采样点数目：83922
较高能量阈值，计算后的浊音A:[51, 127, 159, 265]
较低能量阈值，增加一段语言B:[50, 127, 158, 266]
过零率阈值，最终语音分段C:[50, 127, 158, 266]
output/8.wav保存完成
采样点数目：47986
较高能量阈值，计算后的浊音A:[29, 159]
较低能量阈值，增加一段语言B:[27, 161]
过零率阈值，最终语音分段C:[27, 161]
output/9.wav保存完成
采样点数目：150762
较高能量阈值，计算后的浊音A:[316, 344, 404, 471, 521, 567]
较低能量阈值，增加一段语言B:[313, 351, 401, 472, 519, 569]
过零率阈值，最终语音分段C:[313, 351, 401, 472, 519, 569]
output/10.wav保存完成
```

图 3-5　任务 1 执行过程

3.2　任务 2：声纹识别

声纹识别任务旨在通过分析个体的声音特征来确认其身份，类似于指纹或面部识别技术在生物识别领域的应用。该任务利用声音的独特性，如音调、节奏、发音习惯等音频特征，对个体进行识别和验证。本任务将实现一个基本的声纹识别系统，使用常见的音频特征提取技术，如 Mel 频率倒谱系数（Mel-Frequency Cepstral Coefficients，MFCC），以及简单的分类算法来区分不同个体的声纹。

此任务的执行结果如图 3-6 所示。

```
(asr) E:\AI\YYAL\CH03\T2\speaker_id\src>python lab.py
请输入对应数字
1、录入声音信息
2、声音识别
请输入1
请输入姓名VIVIAN
第1次录音开始...
recording ok
第2次录音开始...
recording ok
第3次录音开始...
recording ok
第4次录音开始...
recording ok
第5次录音开始...
recording ok
+ 说话者建模完成: VIVIAN.gmm  相关数据点=  (4985, 40)
```

(a) 录入声音信息

```
(asr) E:\AI\YYAL\CH03\T2\speaker_id\src>python lab.py
请输入对应数字
1、录入声音信息
2、声音识别
请输入2
录音开始...
录音完成
        识别为 -  VIVIAN
```

(b) 声音识别

图 3-6　声纹识别任务

3.2.1 任务分析

声纹识别的核心在于利用个体声音中的独特特征来确认身份,这些特征主要来自音频信号的特征提取。与一般的语音识别不同,声纹识别更关注声音本身的物理和声学特性,而不是语义内容。为了实现这一任务,准确地进行音频特征提取至关重要。

每个人的声音都有独特的频率模式、发音习惯和韵律节奏。通过从原始的语音信号中提取出这些特征,可以生成一个用来表示该个体声纹的特征向量。这些特征能够捕捉声音的个体差异,并通过数学模型将其转换为可用于识别和分类的特征。

3.2.2 必备知识:音频特征提取

1. 频带能量(FBand)特征

语音信号可以被分为很多帧,每帧语音都对应于一个频谱,通过短时傅里叶变换计算,频谱表示频率与能量的关系。在实际使用中,频谱图有 3 种,即线性振幅谱、对数振幅谱、自功率谱。由于对数振幅谱中各谱线的振幅都进行了对数计算,所以其纵坐标的单位是分贝(dB)。对数变换的目的是使那些振幅较低的成分相对高振幅成分得以拉高,以便观察掩盖在低幅噪声中的周期信号。

频谱计算过程如图 3-7 所示。

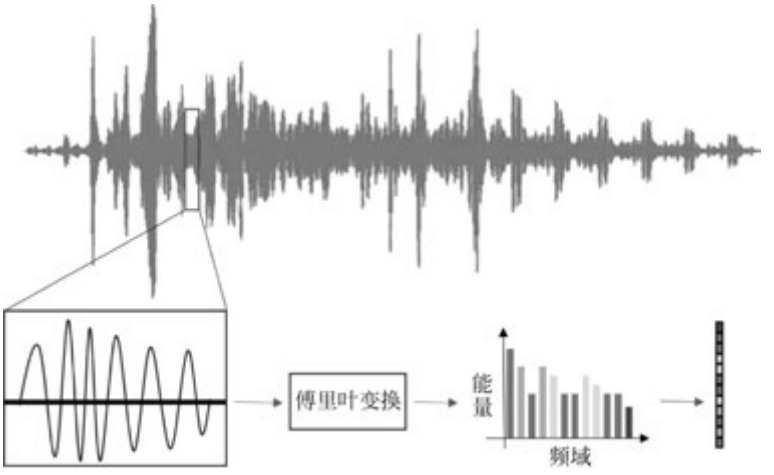

图 3-7 频谱计算

先将其中一帧语音的频谱通过坐标表示出来,如图 3-8(a)所示。将左边的频谱旋转 90°,得到图 3-8(b),然后把这些幅度映射到一个灰度级表示,也可以理解为将连续的幅度量化为 256 个离散值,0 表示黑,255 表示白色。幅度值越大,相应的区域越黑。这样就得到了如图 3-8(c)所示的结果。

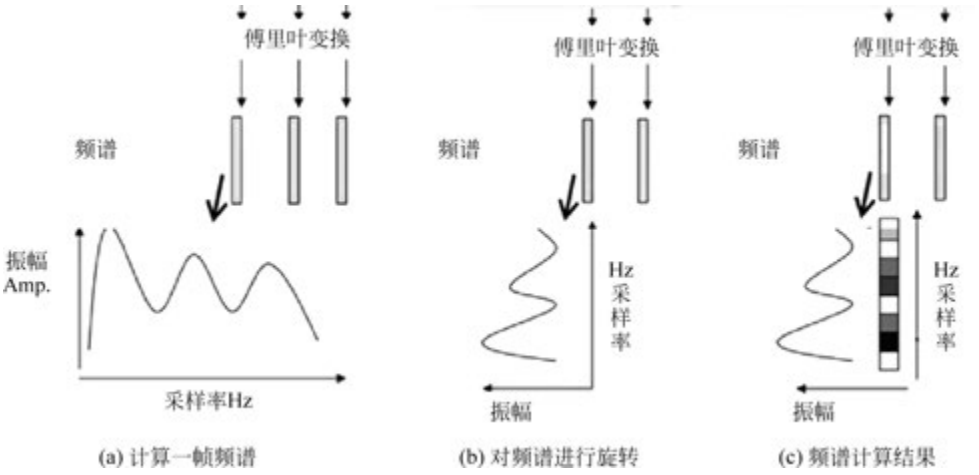

图 3-8　频谱计算过程

最终得到的结果是一个随着时间变化的频谱图，又称语谱图或者声谱图。语谱图分为宽带语谱图和窄带语谱图，常见的是窄带语谱图，如图 3-9 所示，可以识别出语谱图上的共振峰。

深颜色的较粗带称为“横杠”，它是共振峰，也是浊音部分。横杠之间的距离是基音频率，在带有“横杠”的竖条的最底部的“横杠”是基频，因为浊音需要基频起振，所以浊音的竖条带有基频。因为元音的持续时间较长和协同效应较强，所以各个谐波表现为横向的波纹。颜色较浅而且竖条上不具有深色“横杠”的是清音，因为没有引起声道谐振，所以没有共振频率，如图 3-10 所示，较为清晰地显示了一个基频和 3 个共振峰“横杠”，前两个共振峰较为突出，作用最为明显。

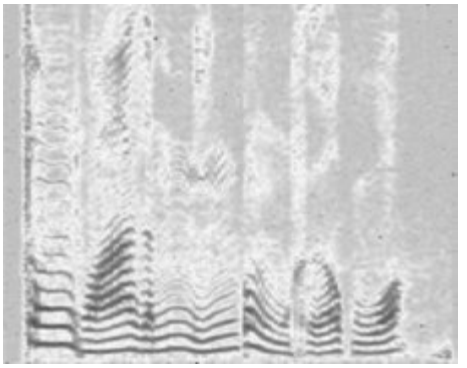

图 3-9　语谱图上的共振峰

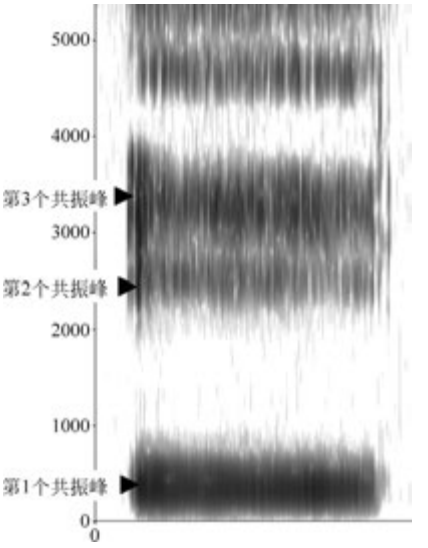

图 3-10　共振峰

在能量谱上应用梅尔(Mel)滤波器组,就能提取 FBank 特征。

在介绍 Mel 滤波器组之前,先介绍 Mel 刻度,它是一个能模拟人耳接收声音规律的刻度,人耳在接收声音时呈现非线性状态,对高频更不敏感,因此 Mel 刻度在低频区分辨度较高,在高频区分辨度较低,与频率之间的换算关系公式如下:

$$f = 700 \times (10^{\frac{m}{2595}} - 1) \tag{3-2}$$

其中,f 表示频率,单位为 Hz;m 表示 Mel 刻度的频率值。

Mel 滤波器组就是一系列的三角形滤波器,通常有 40 个或 80 个,在中心频率点响应值为 1,在两边的滤波器中心点衰减到 0,如图 3-11 所示。

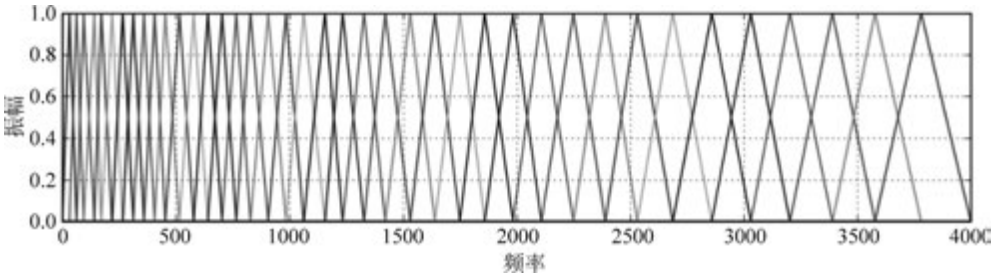

图 3-11 Mel 滤波器组

2. MFCC

FBank 提取的信号特征往往是高度相关的,在传统声学模型中,通常只需使用共振峰,就可以识别不同的声音。

由于要提取共振峰的位置,同时还要提取它们转变的过程,所以提取的是频谱的包络,这个包络就是一条连接这些共振峰点的平滑曲线。

提取了 FBank 特征后,可以继续使用离散余弦变换(Discrete Cosine Transform,DCT)对这些相关的滤波器组系数进行压缩。对于声纹识别来讲,通常取 2～13 维,过滤掉的信息里面包含滤波器组系数快速变化部分。

DCT 其实是逆傅里叶变换的等价替代,常用于对信号和图像进行数据压缩。

3. 声纹识别

人声会携带有关说话人不同特征(健康、年龄、性别、心理、语言和身份等)及语音记录环境的信息,因此语音信号在许多领域(取证、电话银行、电话购物、安全控制、计算机的语音控制等)用作可靠且独特的功能。

利用人声作为身份认证的相关应用有说话人验证和说话人识别,说话人验证通常与具体说话内容有关,说话人识别又称声纹识别,与说话内容无关。

声纹识别的目的是提取说话人声音特征以区分不同的说话人。

1) 高斯分布

高斯分布(Gaussian Distribution)也被称为正态分布(Normal Distribution),是一个常见的连续概率分布,若随机变量服从一个均值为 μ、标准差为 σ 的正态分布,则记为

$$X \sim N(\mu, \sigma^2) \tag{3-3}$$

其概率密度函数公式如下：

$$f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3-4}$$

2）参数估计

高斯分布最常见的应用之一就是参数估计，是利用部分符合高斯分布的部分数据对分布的参数进行估计。

例如对大量人口进行身高数据的随机采样，并且将采得的身高数据画成柱状图，如图 3-12 所示，可以判断均值在 180cm 附近。

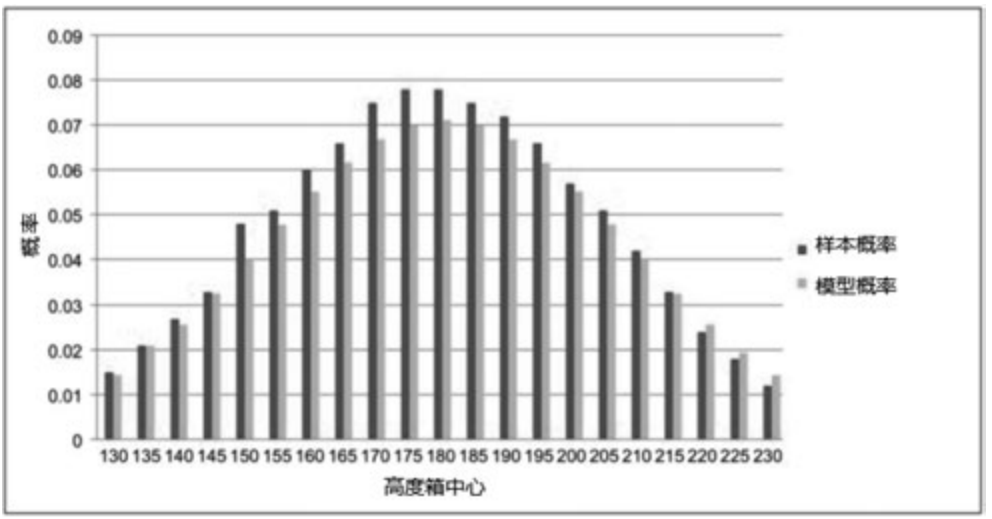

图 3-12　高斯分布中的参数估计

假如样本数量为 334，每个柱状线表示相应身高值在 334 个人中的分布概率，用每个身高值对应的人数除以总数就可以得到对应概率值，即图中所示样本概率。

通过对人口身高数据的高斯分布标准差进行估计，可以得到一个模型概率。

3）高斯混合模型

高斯混合模型（Gaussian Mixture Model，GMM）是对高斯模型进行简单扩展，是多个高斯分布函数的线性组合。

基于 GMM 的语音识别技术主要通过对语音信号进行建模，将语音信号的特征向量看作由多个高斯分布混合而成的，然后利用 GMM 对语音信号进行分类和识别，具体步骤如下。

（1）语音信号预处理：去除语音信号中的噪声和冗余信息，将其转换为适合特征提取的信号。

（2）特征提取：提取语音信号的特征向量，常见的特征包括梅尔频率倒谱系数

(MFCC)等。

(3) 模型训练：利用已知类别的语音特征向量训练 GMM 模型，得到各个高斯分布的参数。

(4) 语音识别：将待识别的语音特征向量输入已经训练好的 GMM 模型中，根据最大概率原则进行分类和识别。

3.2.3 任务实施

1. 任务目标

- ❑ 能够对语音进行 MFCC 特征提取。
- ❑ 能够对训练数据进行 GMM-UBM 模型训练。
- ❑ 能够对说话人声音进行注册和识别。

2. 实施环境

任务实施环境见表 3-2。

表 3-2 任务实施环境

硬　　件	软　　件	资　　源
高性能个人计算机 人工智能边缘计算实训设备 单声道话筒	Debian/Ubuntu/Windows 10(或 11)/macOS Python 3.8 scikit-learn==1.3.2 sklearn==0.0 python_speech_features==0.6 numpy==1.19.5 scipy==1.4.1 PyAudio==0.2.13 wave	223 个说话人语音

3. 实施步骤

按照如下步骤完成本次任务：

(1) 语音特征提取。

(2) GMM-UBM 模型训练。

(3) 说话人声音注册和识别

1) 语音特征提取

在本任务中，语音特征有 40 维，包含 20 维 MFCC 特征、20 维的一阶差分和二阶差分的增量特征。

编写 featureextraction.py 脚本，代码如下：

```
#第 3 章/任务 2/featureextraction.py
import numpy as np
from sklearn import preprocessing
import python_speech_features as mfcc
```

```
def calculate_delta(array):
    """
    计算并返回 MFCC 增量特征
    (一阶差分参数 + 二阶差分参数) + 帧能量(此项可根据需求替换)
    """
    rows,cols = array.shape
    deltas = np.zeros((rows,20))
    N = 2
    for i in range(rows):
        index = []
        j = 1
        while j <= N:
            if i - j < 0:
              first = 0
            else:
              first = i - j
            if i + j > rows - 1:
                second = rows - 1
            else:
                second = i + j
            index.append((second, first))
            j += 1
        deltas[i] = (array[index[0][0]] - array[index[0][1]]
+ (2 * (array[index[1][0]] - array[index[1][1]]))) / 10
    return deltas

def extract_features(audio,rate):
    """提取音频的 20 维 MFCC 特征"""

mfcc_feature = mfcc.mfcc(audio,rate, 0.025, 0.01, 20,
nfft = 1200,
appendEnergy = True)

    mfcc_feature = preprocessing.scale(mfcc_feature)
    delta = calculate_delta(mfcc_feature)
    combined = np.hstack((mfcc_feature,delta))
    return combined
```

2）GMM-UBM 模型训练

在模型训练阶段会创建一个包含所有训练样本的特征向量的通用背景模型(Universal Background Mode,UBM),GMM-UBM 算法是一种基于概率模型的声纹识别方法。该算法通过训练大量背景数据来建立一个通用的语音模型,称为通用背景模型。UBM 可以看作对语音的表征,但它是从大量身份的混杂数据中训练而成的,不具备表征具体身份的能力,然后对于每个特定说话人,可以用其语音数据自适应地调整 UBM 参数,得到该说话人的个性化模型。

编写 GMM-UBM.py 脚本,代码如下:

```
#第 3 章/任务 2/GMM - UBM.py
import pickle
import numpy as np
from scipy.io.wavfile import read
from sklearn.mixture import GaussianMixture
from featureextraction import extract_features
import warnings
warnings.filterwarnings("ignore")

#训练样本路径
source = "../dataset/train/"

#模型保存路径
train_file = "../dataset/development_set_enroll.txt"
dest = '../Model/'
file_paths = open(train_file,'r')

count = 1

#所有人的特征
ubm_features = np.asarray(())

#获取每个声音的语音特征
features = np.asarray(())

for path in file_paths:
    path = path.strip()
    print(path)

    #读取音频
    sr,audio = read(source + path)

    #40 维 MFCC
    vector = extract_features(audio, sr)

    if features.size == 0:
        features = vector
        ubm_features = vector
    else:
        features = np.vstack((features, vector))
        ubm_features = np.vstack((ubm_features,vector))

    #为每个人生成 5 个模型
    if count == 5:
        gmm = GaussianMixture(n_components = 16,
max_iter = 200,
covariance_type = 'diag',
n_init = 3)
        gmm.fit(features)

        #保存高斯模型
```

```
            picklefile = path.split("-")[0]+".gmm"
            pickle.dump(gmm,open(dest+picklefile, 'wb'))
            print ('+ 说话人模型构建:', picklefile, "特征维度 = ", features.shape)
            features = np.asarray(())
            count = 0
        count = count + 1

UBM = GaussianMixture(n_components = 16,
max_iter = 200,
covariance_type = 'diag',
n_init = 3)
UBM.fit(ubm_features)

pickle.dump(UBM,open("../Model/UBM_MFCC_model.pkl", 'wb'))
```

终端运行脚本，查看音频设备编号，并修改 lab.py 中的 INPUT_DEVICE_INDEX 变量：

```
python get_index.py
```

在终端执行的命令如下：

```
python GMM-UBM.py
```

执行命令后的部分过程如图 3-13 所示。

```
说话者: ataмur.gmm 模型保存成功
说话者: ataru80.gmm 模型保存成功
说话者: atterer.gmm 模型保存成功
说话者: audioodyssey.gmm 模型保存成功
说话者: avsa242.gmm 模型保存成功
说话者: ax.gmm 模型保存成功
说话者: axllaruse.gmm 模型保存成功
说话者: azmisov.gmm 模型保存成功
说话者: B.gmm 模型保存成功
说话者: bachroxx.gmm 模型保存成功
说话者: bae.gmm 模型保存成功
说话者: Bahoke.gmm 模型保存成功
说话者: Bareford.gmm 模型保存成功
说话者: bart.gmm 模型保存成功
说话者: Bassel.gmm 模型保存成功
说话者: beady.gmm 模型保存成功
说话者: beez1717.gmm 模型保存成功
说话者: belmontguy.gmm 模型保存成功
```

图 3-13　GMM-UBM 训练过程

在 Model 目录下包含 34 个说话人的 GMM 模型和一个 UBM 模型。

3）说话人声音注册和识别

编写 lab.py 脚本，代码如下：

```
#第 3 章/任务 2/lab.py
#!usr/bin/python
# - * - coding: utf-8 - * -

import pyaudio
import wave
```

```
import os
import time

import pickle
import numpy as np
from scipy.io.wavfile import read
from sklearn.mixture import GaussianMixture
from featureextraction import extract_features

#录入声音
def reg_speaker(username, RECORED_SECONDS = 10):
    CHUNK = 1024
    RATE = 16000
    CHANNELS = 1
    FORMAT = pyaudio.paInt16
    Fpath = '../dataset/train/' + username + '/wav/'

    if not os.path.exists(fpath):
        os.makedirs(fpath)

    for i in range(1,6):

        fname = os.path.join(fpath, str(i) + '.wav')


        INPUT_DEVICE_INDEX = 0
        p = pyaudio.PyAudio()
        stream = p.open(format = FORMAT,
                        channels = CHANNELS,
                        rate = RATE,
                        input = True,
frames_per_buffer = CHUNK,
input_device_index = INPUT_DEVICE_INDEX)

        print("第{}次录音开始...".format(i))
        frames = []
        for j in range(0, int(16000/1024 * 10)):  #录音秒数
            data = stream.read(1024)
            frames.append(data)
        print("recording ok")

        stream.stop_stream()
        stream.close()
        p.terminate()

        wf = wave.open(fname, 'wb')
        wf.setnchannels(1)
        wf.setsampwidth(p.get_sample_size(pyaudio.paInt16))
        wf.setframerate(16000)
        wf.writeframes(b''.join(frames))
        wf.close()
```

```
            if i < 5:
                time.sleep(5)

    def train_speaker(username):
        fpath = '../dataset/train/' + username + '/wav/'

        if not os.path.exists(fpath):
            pass

        files = os.listdir(fpath) #得到文件夹下的所有文件名称
        features = np.asarray(())
        for fl in files:
            if os.path.splitext(fl)[-1] == ".wav":
                #读取音频
                sr,audio = read(os.path.join(fpath,fl))

                #获取 MFCC
                vector   = extract_features(audio,sr)

                if features.size == 0:
                    features = vector
                else:
                    features = np.vstack((features, vector))

    gmm = GaussianMixture(n_components = 16,
    max_iter = 200,
    covariance_type = 'diag',
    n_init = 3)
        gmm.fit(features)
        dest = '../Model/'
        #保存训练好的模型
        picklefile = username + ".gmm"
        pickle.dump(gmm,open(dest + picklefile, 'wb'))
        print ('+ 说话者建模完成:', picklefile, "相关数据点 = ", features.shape )

    def test_record(RECORED_SECONDS = 10):
        CHUNK = 1024
        RATE = 16000
        CHANNELS = 1
        FORMAT = pyaudio.paInt16
        Fpath = '../dataset/test/wav/'

        if not os.path.exists(fpath):
            os.makedirs(fpath)

    fname = os.path.join(fpath,time.strftime(
```

```
'%Y%m%d%H%M%S',
time.localtime(time.time())) + '.wav')

    INPUT_DEVICE_INDEX = 0
    p = pyaudio.PyAudio()
    stream = p.open(format = FORMAT,
                    channels = CHANNELS,
                    rate = RATE,
                    input = True,
frames_per_buffer = CHUNK,
input_device_index = INPUT_DEVICE_INDEX)

    print("录音开始...")
    frames = []
    for j in range(0, int(16000/1024 * 10)):  #录音秒数
        data = stream.read(1024)
        frames.append(data)
    print("录音完成")

    stream.stop_stream()
    stream.close()
    p.terminate()

    wf = wave.open(fname, 'wb')
    wf.setnchannels(1)
    wf.setsampwidth(p.get_sample_size(pyaudio.paInt16))
    wf.setframerate(16000)
    wf.writeframes(b''.join(frames))
    wf.close()

    return fname

def test_speaker(fname):
    modelpath = "../Model/"

    gmm_files = [os.path.join(modelpath,fn) for fn in os.listdir(modelpath) if fn.endswith('.gmm')]

    #加载高斯模型
    models = [pickle.load(open(fn,'rb')) for fn in gmm_files]
    UBM = pickle.load(open("../Model/UBM_MFCC_model.pkl",'rb'))
    speakers = [fn.split("/")[-1].split(".gmm")[0] for fn in gmm_files]
    sr,audio = read(fname)
    vector = extract_features(audio,sr)

    log_likelihood = np.zeros(len(models))

    for i in range(len(models)):
        gmm = models[i]  #逐模型进行比对
        scores = np.array(gmm.score(vector) - UBM.score(vector))
```

```
            log_likelihood[i] = scores.sum()

        winner = np.argmax(log_likelihood)
        print ("\t 识别为 - ", speakers[winner])

    if __name__ == "__main__":
        print('请输入对应数字')
        print('1.录入声音信息')
        print('2.声音识别')
        n = input('请输入')
        if int(n) == 1:
            uname = input('请输入姓名')
            reg_speaker(username = uname)
            train_speaker(uname)
        else:
            fn = t est_record()
            test_speaker(fn)
```

在终端执行的命令如下：

```
python lab.py
```

3.3　任务 3：实时声源定位

实时声源定位任务涉及使用话筒阵列技术来确定声音来源的具体位置，这在多种环境中都非常有用，例如在会议室中跟踪说话人的位置，或在安全系统中识别声音来源。此技术能够分析从不同方向接收的声音信号的时间差和强度差，以精确地定位声源。本任务将探讨如何利用话筒阵列捕捉声音信号，并通过声音信号处理算法实现对声源的实时定位。这不仅增强了环境感知能力，也为进一步的声音分析和处理提供了基础。

3.3.1　任务分析

实时声源定位是一个复杂的过程，涉及多种声音信号处理技术。该任务的成功执行依赖于对从不同话筒接收的声音信号进行精确分析，以识别声源的具体位置。这一任务的关键技术点包含话筒阵列的配置与使用、声音信号的到达时间差(Time Difference of Arrival，TDOA)分析、声强度和相位信息的利用等。

3.3.2　必备知识：声源定位

在三维空间中，除了时域、频域，还可以利用空域信息对信号进行处理，基于话筒阵列的远场语音识别场景，一些声源分离技术会用到声源方位信息。到达方向估计(Direction Of Arrival，DOA)技术并不仅限于单个目标源的定位，并且对于语音识别场景的声源目标是宽带信号。此外，定位出声源方向，还有益于产品的交互体验(寻向灯，以及电机转动姿态)。声源定位算法需要考虑稳健性和角分辨率两项指标。

1. 话筒阵列

话筒阵列是实现声源定位算法的基础。话筒阵列由若干话筒按一定布局排列构成。常见的话筒阵列中话筒的排列方式有线性排列、矩形排列、六边形排列等。话筒阵列在声源定位、声音去噪、语音提取等领域有着广泛应用。

2. 声源定位

假设声源距离话筒阵列较远,将声波看作平行波,声音传播的模型如图 3-14 所示。

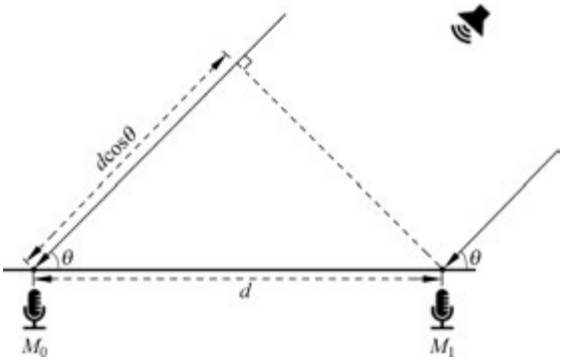

图 3-14 声音传播模型

两个话筒的间距为 d,声波到达的角度为 θ,则声源到两个话筒的距离差为 $d\cos(\theta)$,因此声波到达两个话筒的到达时间差(TDOA)为 $d\cos(\theta)/c$,其中为 c 声速。已知两个话筒的间距 d,以及声音到达两话筒的时间差 Δt,即可计算到达角度 θ。

利用互相关函数可以计算两个声音信号的到达时间差,假设话筒 M_0 接收的信号为 $x(t)$,话筒 M_1 接收的信号为 $y(t)$,声波到达 M_0 与 M_1 的时间差为 t_0,则 $x(t)$与 $y(t)$的互相关函数定义为

$$\phi_{xy}(t)=\int_{-\infty}^{+\infty}x(\tau)y(t+\tau)\,\mathrm{d}\tau=x(t)*y(-t) \tag{3-5}$$

式(3-5)中 * 表示卷积,t 表示时间差。将 $y(t)$的表达式代入,得

$$\phi_{xy}(t)=\int_{-\infty}^{+\infty}x(\tau)x(t+\tau-t_0)\,\mathrm{d}\tau \tag{3-6}$$

当 $t=t_0$ 时,$\phi_{xy}(t_0)$取得 $\phi_{xy}(t)$的最大值。

3.3.3 任务实施

1. 任务目标

- ❑ 熟悉四话筒阵列的基本构成。
- ❑ 能够使用四话筒阵列进行声源定位。
- ❑ 能够使用图形界面对声源定位进行可视化。

2. 实施环境

任务实施环境见表 3-3。

表 3-3　任务实施环境

硬　　件	软　　件	资　　源
高性能个人计算机 人工智能边缘计算实训设备 四话筒阵列	Debian/Ubuntu/Windows 10(或 11)/macOS Python 3.10 PyQt5==5.14.0 PyAudio==0.2.13 numpy==1.23.5	untitled.ui

3. 实施步骤

实时声源定位项目的目录结构如图 3-15 所示。

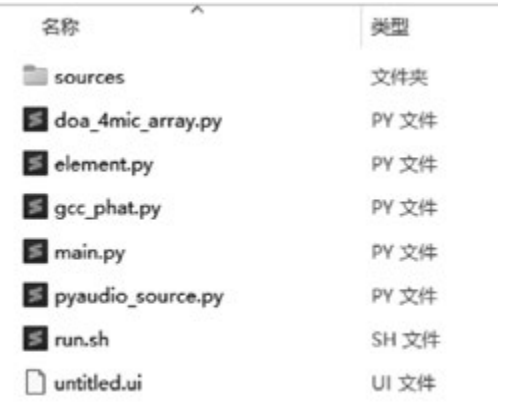

图 3-15　任务 3 项目的目录结构

实验步骤如下：

(1) 导入资源。

(2) 界面初始化。

(3) 实时声源定位。

1) 导入资源

编写 main.py 脚本，代码如下：

```
#第 3 章/任务 3/main.py
import sys
import os
from PyQt5 import uic
from PyQt5.Qt import *
from PyQt5.QtWidgets import *
import numpy as np
import time
from pyaudio_source import Source
from doa_4mic_array import DOA
#导入 UI 界面
ui_mainwindow, QtBaseClass = uic.loadUiType("untitled.ui")
```

2）界面初始化

初始化界面及控件，设置字体颜色，定义关闭事件函数和背景图片事件函数，代码如下：

```
#第 3 章/任务 3/main.py
class MainWindow(QMainWindow, ui_mainwindow):
    def __init__(self):
        QMainWindow.__init__(self)
        ui_mainwindow.__init__(self)
        self.setupUi(self)
        self.setWindowTitle("实时声源定位")
        self.label.setStyleSheet("color:white")
        self.label_2.setStyleSheet("color:white")
        self.label_3.setStyleSheet("color:white")
        self.label_4.setStyleSheet("color:white")
        self.label_6.setStyleSheet("color:white")
        self.label_7.setStyleSheet("color:white")
        QTimer.singleShot(0, self.location)
        self.CLOSE = False

    def __del__(self):
        print('del')

    def closeEvent(self, event):
        self.CLOSE = True
        print('closeEvent')
        QMainWindow.closeEvent(self, event)

    def paintEvent(self, event):
        painter = QPainter(self)
        painter.drawPixmap(self.rect(), QPixmap("./sources/1.png"))
```

3）实时声源定位

继续在 main.py 文件中编写代码。

创建 pyaudio_source.py 文件中的 Source 实例进行声音录制，录制采样率为 16 000，块大小为 320。创建 doa_4mic_array.py 文件中的 DOA 实例进行方向判断。最后将方向显示到界面上，并实时地进行方向定位，代码如下：

```
#第 3 章/任务 3/main.py
    def location(self):
        src = Source(rate=16000, frames_size=320)
        doa = DOA(rate=src.rate)
        src.link(doa)
        src.recursive_start()
        QApplication.processEvents()
        time.sleep(0.5)

        while True:
            if self.CLOSE == True:
                break
            try:
```

```
                    direction = doa.get_direction()
                    direction_int = np.ceil(direction)
                    if direction_int < 90 :
                        direction = 360 - direction_int
                    else :
                        direction = direction_int - 90
                    direction = 360 - direction
                    self.dial.setValue(int(direction))
                    QApplication.processEvents()
    # print('direction {}'.format(direction))
                    time.sleep(0.1)
                    self.repaint()
                except KeyboardInterrupt:
                    break
            src.recursive_stop()

    if __name__ == "__main__":
        app = QApplication(sys.argv)
        window = MainWindow()
        window.show()
        sys.exit(app.exec_())
```

首先在实验箱中找到四话筒阵列，然后插入实验箱主机，在终端执行的命令如下：

```
python main.py
```

程序执行后只需在不同的方位发出声音，就可以看到程序界面中的指针在不同方向上跳动，执行结果如图 3-16 所示。

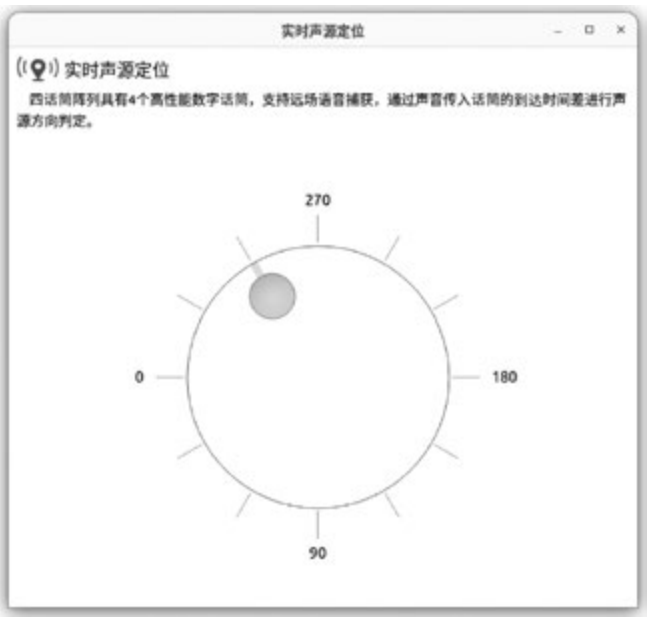

图 3-16　实时声源定位结果

项目总结

本章不仅阐述了声音信号的采集和处理方法，还介绍了如何进行语音特征提取和声纹识别，以及利用话筒阵列进行声源的实时追踪。通过这些学习，读者将能够在会议记录、安全监控及智能家居等多个领域中有效地应用声音跟踪技术，提高系统的性能和用户体验。本章旨在提供全面的理论支持和实践指导，帮助读者在声音处理领域取得实际进展和技术突破。