第 3 章 音频信号的获取

对音频信号进行分析处理，首先涉及的问题是音频信号从何而来。数字音频信号的来源大致有以下 3 种：

（1）模拟信号采样。

（2）音频或视频文件。

（3）计算机生成或合成。

下面分别介绍这几种获取方式。

3.1 采样与量化

采样过程是进行模拟信号数字化的第 1 步。采样就是按一定的时间间隔从模拟连续信号提取出一定数量的样本来，如图 3-1 所示。通过采样，连续的模拟信号被转换成离散的数字信号。

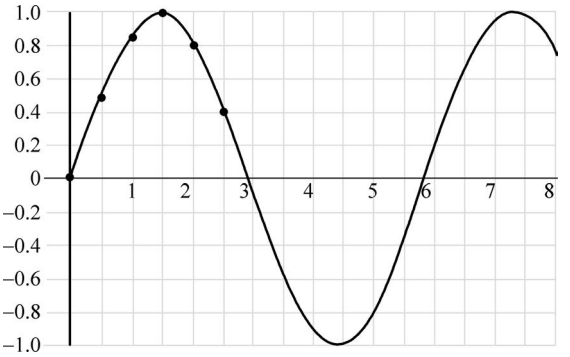


图 3-1 采样过程示意图

3.1.1 采样相关概念

下面用第 1 章中介绍的 Praat 对采样的相关概念进行介绍。用 Praat 打开一个音频文件后，可以查看如图 3-2 所示的信息，其中与采样有关的概念有采样率、采样周期等。

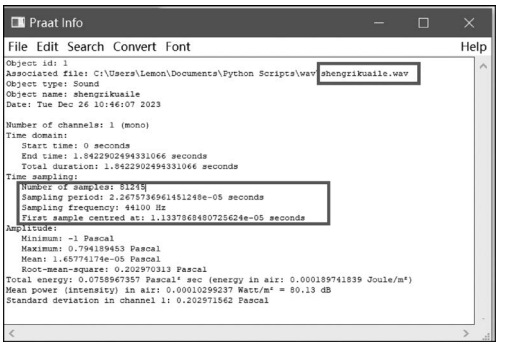


图 3-2　Praat 中有关采样的信息

1. 采样率

采样率(Sampling Rate)是指每秒对原始信号采样的次数。采样率越高，单位时间内的样本数据就越多，对信号波形的表示也越精确。不过，随着采样率的提高，计算机所需的存储空间会变大，处理速度则会降低，因此采样率并非越高越好。

2. 采样周期

两个相邻的采样点之间的间隔 T 称为采样周期(Sampling Period)，如图 3-3 所示。采样周期的倒数，称为采样频率(Sampling Frequency)。图中音频文件的采样频率为 44100Hz，采样周期则是 44100 的倒数，约为 2.2676×10^{-5}s。

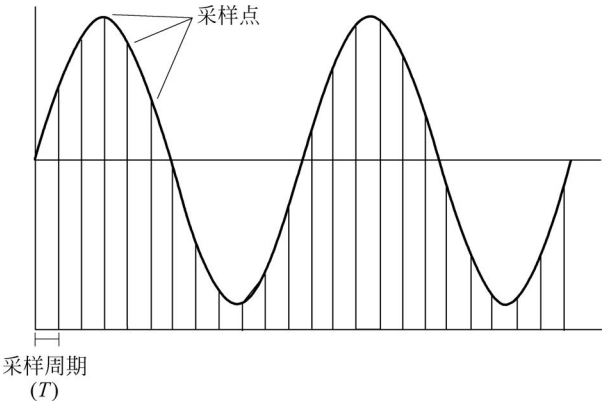


图 3-3　采样周期

3. 采样定理

根据香农采样定理(也称奈奎斯特采样定理)，为了不失真地恢复模拟信号，采样频率应该不小于模拟信号频谱中最高频率的 2 倍。采样率过低可能使信号失真，如图 3-4 所示，图中的原始信号经采样后频率发生了显著变化，这就是由采样率过低造成的。

人的听觉范围在 20Hz～20kHz，根据采样定理 40kHz 的采样率就能涵盖了人类听觉的全部范围，CD 采用了 44.1kHz 的采样率，从而保证了声音的纤毫毕现，而在语音识别系统中，一般使用 16kHz 的采样率，因为音素识别中有用的信息在 10kHz 以下，而且有效信

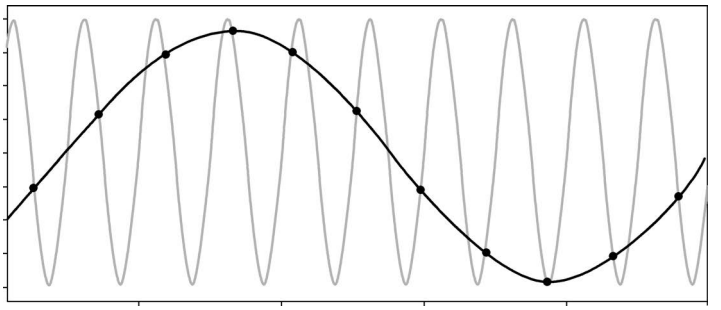


图 3-4　采样率过低引起失真

息集中在低频段，因此 16kHz 的采样率已经够用了。

4. 采样精度

采样精度是指存放一个采样值所使用的比特数。当采样精度为 8 位时，采样值可以有 256 个（$2^8=256$）；当采样精度为 16 位时，采样值可以有 65536 个（$2^{16}=65536$），考虑到采样值有正有负，实际的范围一般在－32768～32767。在大多数情况下，16 位的采样精度已经够用了。

假设采样率为 44100，采样精度为 16 位，1min、1h 的音频信息占用的存储空间计算如下：

```
16位=2字节
1min音频占用存储空间=2*44100*60/220=5.05MB
1h音频占用存储空间=5.05*60=303MB
```

上述计算是基于单声道的，如果是双声道，则占用空间还要翻一番，也就是说 CD 中每分钟的声音都要占用 10MB 的空间，可见要达到 CD 的高保真音质需要的存储空间是非常巨大的。

3.1.2　从话筒拾取信号

计算机和手机中都有录音的功能，可以用话筒拾取外部的声音信号并将其转换成数字音频信号。从本质上讲，这就是一个采样的过程，录制的声音信号已经从模拟信号转换成数字信号，便于保存和传输。

下面介绍如何用 Python 实现从话筒拾取声音信号，实现此功能需要用到 PyAudio 库。PyAudio 是一个 Python 的第三方音频库，可以在 Anaconda Prompt 中通过下列命令行安装。

```
pip install PyAudio
```

用 PyAudio 录制音频可以分成以下几步。

（1）创建 PyAudio 对象，典型的代码如下：

```
audio = pyaudio.PyAudio()
```

(2) 用 open()方法打开 stream 对象,其中可以指定采样格式、通道数、采样率等参数。

(3) 用 stream.read()方法读取数据流。

(4) 将数据保存为音频文件并关闭 stream 对象。

下面是一个具体的例子,代码如下:

```
#第 3 章/microphone_sound.py

import pyaudio
import wave

#设置录制参数
buffersize = 1024                    #数据块大小
channels = 2                         #声道数
rate = 16000                         #采样率
duration = 3                         #时长,单位为秒

#打开数据流
Format = pyaudio.paInt16             #采样格式:每样本 16 位
audio = pyaudio.PyAudio()            #创建 PyAudio 对象
stream = audio.open(format=Format, channels=channels, rate=rate,
                    input=True, frames_per_buffer=buffersize)

#录制话筒声音
print("开始录制...")
frames = []
for i in range(0, int(rate / buffersize *duration)):
   data = stream.read(buffersize)
   frames.append(data)
print("录制完成!")

#关闭数据流
stream.stop_stream()
stream.close()
audio.terminate()

#将录制的声音写入音频文件
filename = 'wav/output.wav'
f = wave.open(filename, 'wb')
f.setnchannels(channels)
f.setsampwidth(audio.get_sample_size(Format))
f.setframerate(rate)
f.writeframes(b''.join(frames))
f.close()

print("已保存为", filename)
```

上述程序运行后将输出如图 3-5 所示的信息,并生成一个名为 output.wav 的声音文件。

图 3-5 microphone_sound.py 运行结果

3.2 读取音频文件

在第 1 章中对常用的音频文件格式进行了介绍,从这些音频文件可以获得各种音频信号。不少 Python 包支持音频文件的读写,如 SciPy、Wave、Soundfile、Librosa 等,下面以 Librosa 为例进行介绍。

Librosa 中用来读取音频的是 load()函数,其原型如下:

```
librosa.load(path, *, sr=22050, mono=True, offset=0.0, duration=None, dtype =np.
float32, res_type ="soxr_hq") -> Tuple[np.ndarray, float]

【参数说明】
(1) path: 输入文件路径。
(2) sr: 目标采样率。
(3) mono:是否将信号转换成单声道。
(4) offset:开始读取时间,单位为秒。
(5) duration:读取时长。
(6) dtype:输出 y 的数据类型。
(7) res_type:重采样类型。

【返回值】
(1) y: 音频时间序列。
(2) sr: y 的采样率。
```

不过 Librosa 中并没有提供保存音频文件的函数,常用的解决方案是采用 Soundfile 中的 write()函数实现(Soundfile 是 Librosa 的依赖库之一),其原型如下:

```
soundfile.write(file, data, samplerate, subtype=None, endian=None, format=
None, closefd=True)

【参数说明】
(1) file: 写入的文件。
(2) data: 需要写入文件的音频数据,仅支持'float64'、'float32'、'int32'和'int16'类型。
(3) samplerate: 采样率。
(4) subtype: 子类型。
```

上述参数中的 subtype 可通过 default_subtype()函数获取其默认值,例如 WAV 格式的默认子类型为'PCM_16',MP3 格式的默认子类型为'MPEG_LAYER_III'。如需获取所有可选的子类型,可通过 available_subtypes()函数实现,以下列出的是其中的一部分:

```
{
'PCM_S8': 'Signed 8 bit PCM',
'PCM_16': 'Signed 16 bit PCM',
'PCM_24': 'Signed 24 bit PCM',
'PCM_32': 'Signed 32 bit PCM',
'PCM_U8': 'Unsigned 8 bit PCM',
'FLOAT': '32 bit float',
'DOUBLE': '64 bit float',
…
}
```

下面用一个简单的例子说明如何读写音频文件，代码如下：

```
#第 3 章/loadfile.py

import librosa
import soundfile as sf

#读取音频文件
y1, sr1 = librosa.load('wav/nihao.wav', sr = None)        #使用文件原采样率
y2, sr2 = librosa.load('wav/nihao.wav')                   #采用默认采样率
y3, sr3 = librosa.load('wav/nihao.wav', sr = 16000)       #指定采样率

#重采样后将数据写入音频文件
sf.write('wav/nihao_16k.wav', y3, sr3 )

print(y1.shape)
print(y2.shape)
print(y3.shape)

print(sr1)
print(sr2)
print(sr3)
```

程序运行后控制台输出结果如图 3-6 所示。由此可以看出，当将采样率设置为 None 和不设置采样率时结果是不一样的，前者采用的是文件的原采样率，而后者使用的是默认采样率 22050，因此，在使用 librosa.load()函数读取音频文件时一定要注意采样率 sr 的设置，否则结果可能和预想的不一样。此外，随着采样率的不同，返回的音频序列的数据长度也是不同的，这可以从 y1、y2、y3 的 shape 看出。

```
(35247,)
(17624,)
(12789,)
44100
22050
16000
```

图 3-6　loadfile.py 运行结果

在上述代码中读取的是完整的音频文件。有时音频文件很长,而有用的仅是其中的一小段,此时可以通过善用 librosa.load()函数中的 offset 和 duration 参数实现读取。

在对音频信号进行分析处理时,常常需要观察它的波形图。波形图的样例如图 3-7 所示,图中横轴表示时间,纵轴表示信号的幅度,波形图描绘的是信号随时间变化的情况。

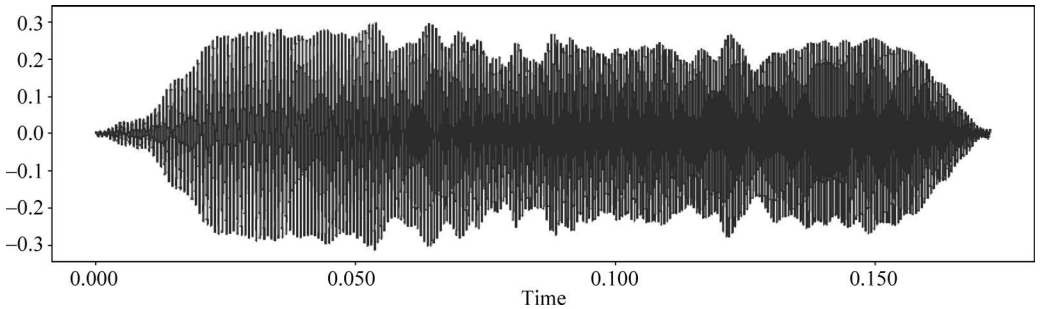


图 3-7 波形图的样例

在用 librosa.load()函数读取音频文件时,返回值 y 是音频的时间序列,可以用 Matplotlib 库中的 plot()函数绘制其波形图,但是用此方法绘制波形图并不方便,效果也不佳,为此 Librosa 专门提供了用于显示波形图的 waveshow()函数,其原型如下:

```
librosa.display.waveshow(y, *, sr=22050, max_points=11025, axis="time", offset
=0.0, marker="", where="post", label=None, transpose=False, ax=None, x_axis=
Deprecated(), data=None, **kwargs) -> AdaptiveWaveplot
```

【参数说明】

(1) y: 音频时间序列
(2) sr: y的采样率
(3) max_points: 绘制的最大样本数。
(4) axis: 横轴显示样式,可选参数如下。
◆'time': 显示为毫秒、秒、分或小时。
◆'h': 显示为小时、分或秒。
◆'m': 显示为分或秒。
◆'s': 显示为秒。
◆'ms': 显示为毫秒。
◆'lag': 类似'time'的显示法,但过中点后将以负数显示。
◆'lag_h':同'lag',但显示为小时。
◆'lag_m': 同'lag',但显示为分。
◆'lag_s': 同'lag',但显示为秒。
◆'lag_ms': 类似'lag',但显示为毫秒。
◆None, 'none'或'off': 标记将被隐藏。
(5) offset: 开始绘制波形图的起始位置,单位为秒。
(6) marker: 样本值标记方式,如'.'、','、'o'等,详见 matplotlib.markers。考虑到波形图一般非常密集,此处使用其默认值即可,不推荐其他选项。
(7) label: 图的标签。
(8) transpose: 转置标志(布尔值),如为 True,则纵向显示,否则横向显示。
(9) ax: 用于绘图的轴,可选 matplotlib.axes.Axes 或 None
(10) x_axis: 该参数在 0.10.0 版已过时,将在 1.0 版被移除。

下面的例子将读取音频文件的一小部分并绘制其波形图，代码如下：

```
#第 3 章/loadfile2.py

import librosa
import librosa.display
import matplotlib.pyplot as plt

#读取音频文件
y1, sr1 = librosa.load('wav/speech.wav', sr=None)
y2, sr2 = librosa.load('wav/speech.wav', offset=5.0, duration=3.0, sr=None)

#采样率等数据
print(y1.shape)
print(y2.shape)
print(sr1)
print(sr2)

#完整音频的波形图
plt.figure(figsize=(14,3))
plt.title('All')
plt.ylim(-0.8, 0.8)
librosa.display.waveshow(y1, sr=sr1)
plt.show()

#截取部分波形图
plt.figure(figsize=(14,3))
plt.title('Part')
plt.ylim(-0.8, 0.8)
librosa.display.waveshow(y2, sr=sr2)
plt.show()
```

程序读取的文件是长约 14s 的语音，第 1 次读取的是完整的音频，第 2 次则只读取了从第 5s 开始时长为 3s 的一小段音频，输出的数据如图 3-8 所示。由此可见，第 2 次读取的数据明显少于第 1 次。

```
(312691,)
(66150,)
22050
22050
```

图 3-8　loadfile2. py 运行结果

为了更直观地观察这两段音频的区别，程序还输出了它们的波形图，如图 3-9 所示。很明显，两段音频的时长不一致，其中第 2 段的波形与第 1 段第 5～8s 的波形是一致的。

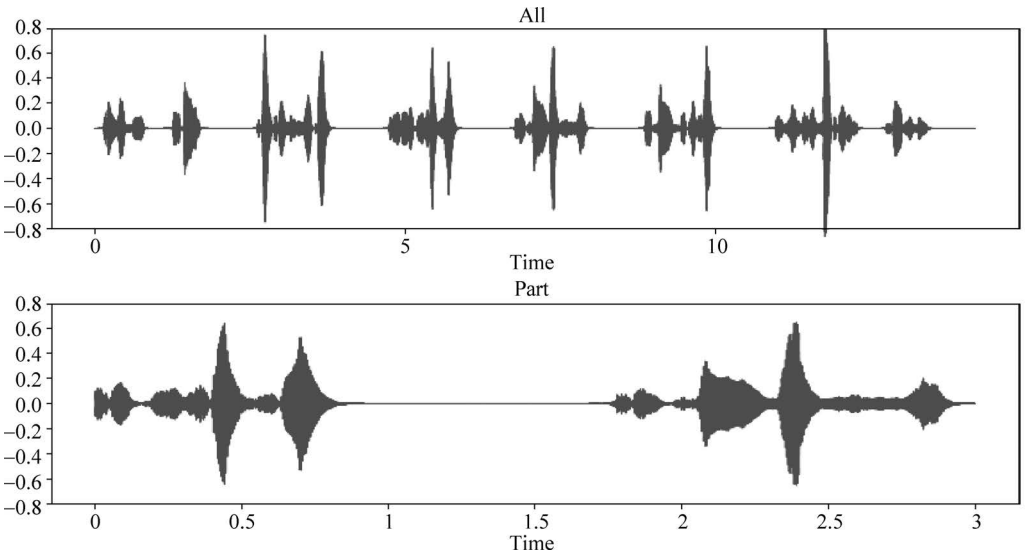


图 3-9　loadfile2. py 输出的波形图

3.3　从视频文件提取

有时，需要从视频文件中提取音频部分，此时需要用到 MoviePy。MoviePy 是一个用于视频编辑的 Python 库，可实现视频的基本操作（如剪切、连接）、视频合成、视频处理等功能。它可以读写 mp4、rm、avi、flv、rmvb 等常见的视频格式，也包括动图的 GIF 格式。

MoviePy 可以在 Anaconda Prompt 中通过下列命令行安装。

```
pip install moviepy
```

下面的例子用 MoviePy 读取一个视频文件并从中提取音频部分，代码如下：

```
#第 3 章/extract_audio.py

from moviepy.editor import VideoFileClip

#加载视频文件
video = VideoFileClip('wav/seagull.avi')
print('视频时长:', video.duration, '秒')
print('分辨率:', video.size)

#提取音频并保存为文件
audio = video.audio
audio.write_audiofile('wav/seagull.mp3')

#显示音频相关信息
```

```
print('音频时长:', audio.duration)
print('声道数:', audio.nchannels)
print('采样率:', audio.fps)
print('OK')
```

程序运行后输出结果如图 3-10 所示。此外，程序还将生成一个音频文件：seagull.mp3。

```
视频时长: 17.61 秒
分辨率: [1280, 720]
MoviePy - Writing audio in wav/seagull.mp3
                                                                MoviePy - Done.

音频时长: 17.61
声道数: 2
采样率: 44100
OK
```

图 3-10　extract_audio.py 运行结果

3.4　声音的合成

音频信号也可以通过计算机直接生成或合成。

3.4.1　纯音的生成

最简单的声音是纯音，纯音可以用正弦函数直接生成，也可以调用 Librosa 中的 tone() 函数生成，后者只需一行代码，其函数原型如下：

```
librosa.tone(frequency, *, sr=22050, length=None, duration=None, phi=None) ->
np.ndarray

【参数说明】
(1) frequency: 频率。
(2) sr: 输出信号的采样率。
(3) length: 输出信号的样本数。
(4) duration: 时长，单位为秒；如果 length 和 duration 均被定义，则 length 优先。
(5) phi: 相位偏移，单位为弧度。

【返回值】
tone_signal: 合成的正弦信号。
```

下面举例说明用正弦函数和 librosa.tone()函数生成纯音的过程，代码如下：

```
#第 3 章/sinewave.py

import librosa
import numpy as np
import matplotlib.pyplot as plt
```

```
#参数设置
N = 256                    #采样点数
sr = 200                   #采样频率
freq = 20                  #频率
fig, ax = plt.subplots(nrows=2, ncols=1)

#用正弦函数生成
t = np.arange(0, N/sr, 1/sr)
data = np.sin(2 *np.pi *freq *t)
ax[0].plot(t, data)

#调用 tone()函数
tone = librosa.tone(freq, sr=sr, length=N)
ax[1].plot(t, tone)
plt.legend()
plt.show()
```

程序的运行结果如图 3-11 所示，两种方式生成的波形图一模一样，不过如果将 data 数组和 tone 数组逐一进行比较，则可以发现个别数值会有细微的差别。

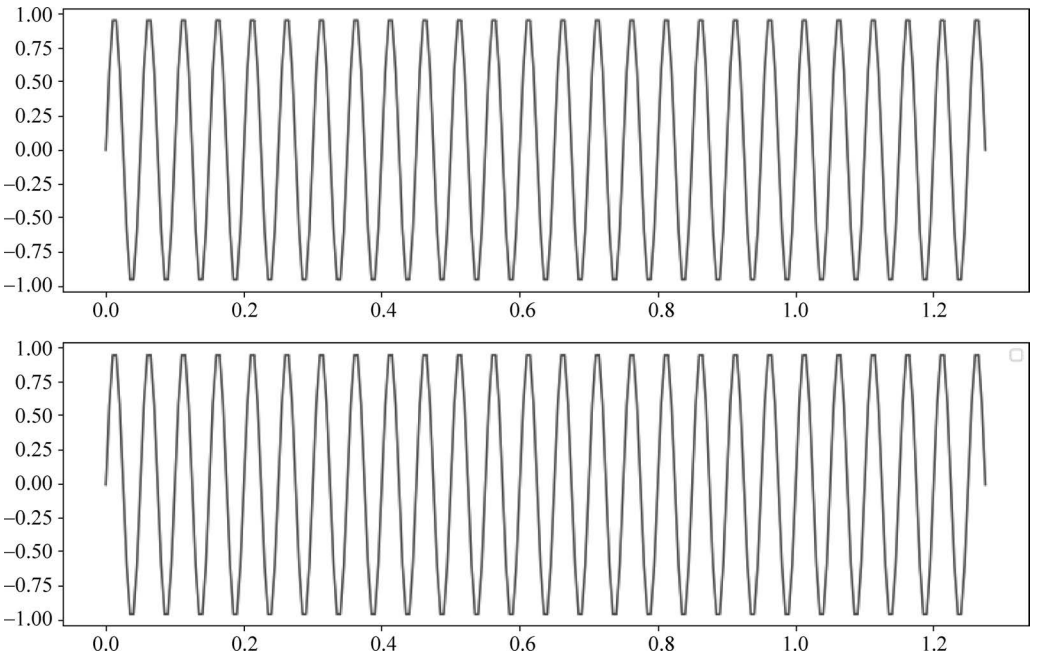


图 3-11　sinewave. py 运行结果

3.4.2　复合音的生成

如果需要生成正弦波叠加而成的复合音，则同样可以用上述两种方法相加生成，用正弦函数生成 20Hz 和 50Hz 的纯音叠加而成的复合音的代码如下：

```
data = np.sin(2 *np.pi *20 *t) + np.sin(2 *np.pi *50 *t)
```

用 tone()函数生成的代码如下：

```
tone = librosa.tone(20, sr=sr, length=N) + librosa.tone(50, sr=sr, length=N)
```

两者生成的图形也是一样的，如图 3-12 所示。

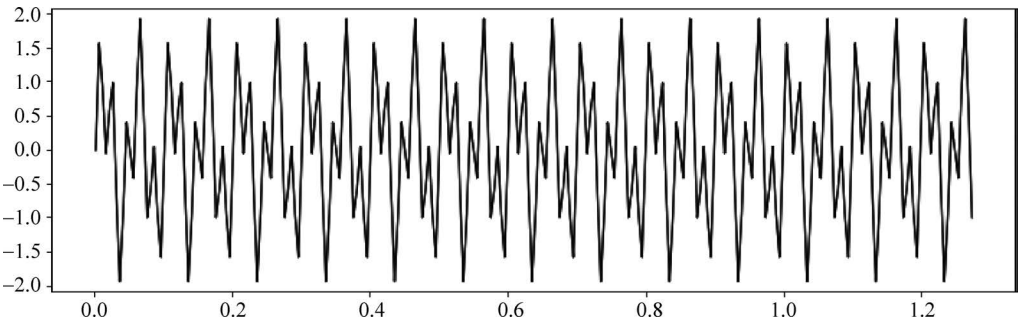


图 3-12　复合音的波形图

有时需要用代码产生一个啁啾信号，Librosa 中也提供了相应的函数，其原型如下：

```
librosa.chirp(*, fmin, fmax, sr: float=22050, length=None, duration=None,
linear=False, phi=None) -> np.ndarray

【参数说明】
(1) fmin：初始频率。
(2) fmax：最终频率。
(3) sr：输出信号的采样率。
(4) length：输出信号的样本数。
(5) duration：时长，单位为秒；如果 length 和 duration 均被定义，则 length 优先。
(6) linear：如果值为 True，则为线性扫频，否则为指数扫频。
(7) phi：相位偏移，单位为弧度。
```

以下是一个简单的例子，代码如下：

```
chirp = librosa.chirp(fmin=110, fmax=110*64, length=22050)
```

由于 linear 参数的默认值为 False，所以以上代码将输出一个指数扫频信号，其频谱图如图 3-13 所示(上半部分)；如果将 linear 设为 True，则输出一个线性扫频信号，频谱图见图 3-13 中下半部分。

3.4.3　音效的合成

前两节中声音是通过纯代码产生的，并没有任何外部的数据，这样产生的声音通常较为简单，而更为常用的场景是在外部数据的基础上加工合成一个新的声音或者音效，下面介绍几种常用的生成方法。

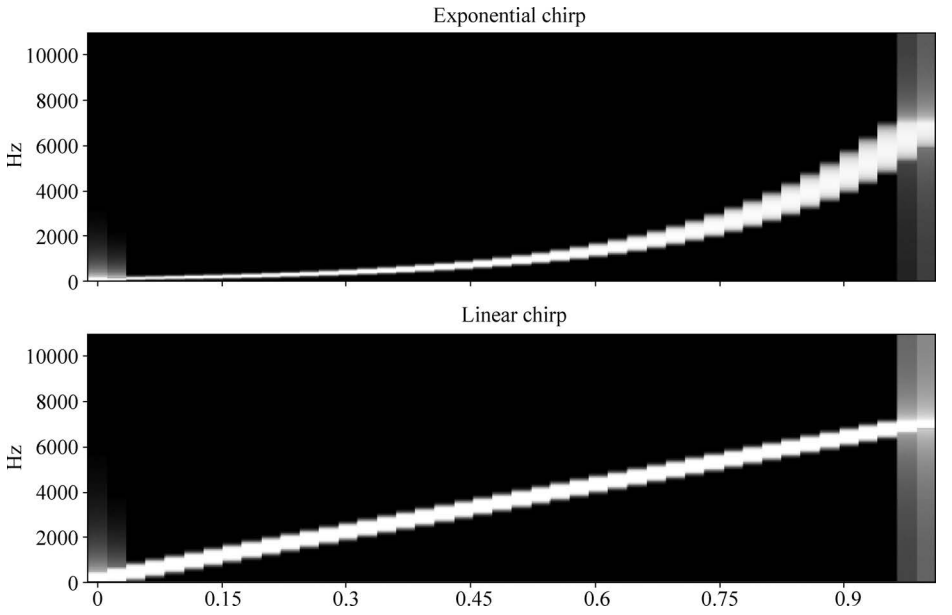


图 3-13　啁啾信号的频谱图

1. 淡入淡出

淡入淡出(Fade In/Fade Out)是一种常用的音频处理技术，通过淡入淡出处理可以使声音在开始时逐渐变强，而在结束时逐渐变弱，从而避免突兀的感觉。淡入淡出技术在音频编辑、影视制作、游戏开发等领域都有广泛应用。

淡入淡出的原理非常简单，它实际上是通过控制声音的振幅实现的，第三方库 Pydub 提供了实现淡入淡出的函数。

Pydub 可以在 Anaconda Prompt 中通过下列命令行安装。

```
pip install pydub
```

下面的程序对一段音乐分别进行了淡入和淡出处理，代码如下：

```
#第 3 章/fade_in_out.py

from pydub import AudioSegment

#读取音频文件
audio = AudioSegment.from_file('wav/rock.wav')

#淡入淡出处理(单位为毫秒)
fade_in = audio.fade_in(3000)                    #开头 3s 淡入
fade_out = audio.fade_out(3000)                  #最后 3s 淡出

#保存处理结果
fade_in.export('wav/fade_in.wav')
```

```
fade_out.export('wav/fade_out.wav')
print('OK')
```

淡入淡出处理的效果如图 3-14 所示，图中从上到下分别为原音频、淡入处理后的音频、淡出处理后的音频的波形图。

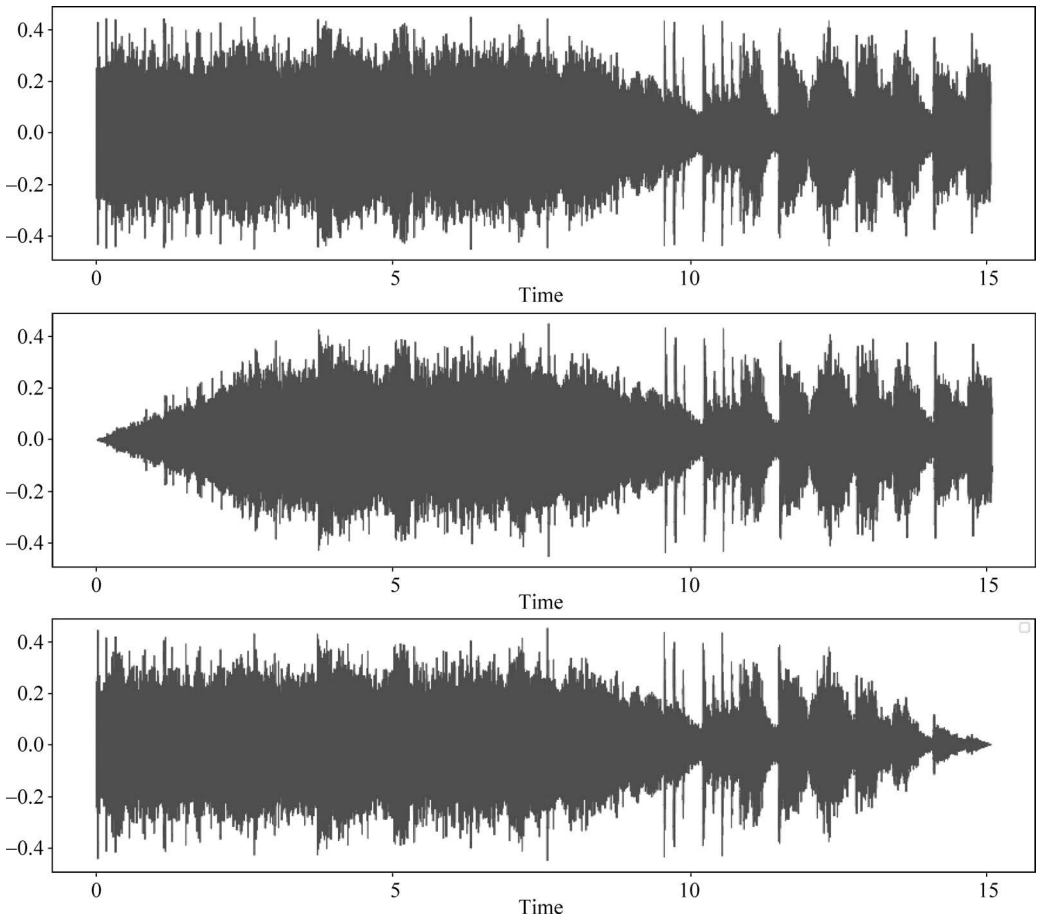


图 3-14　fade_in_out.py 运行结果

2. 变调变速

网络上有不少变声器，能实现男声变女声、大叔音变萝莉音等效果。变声器在语音聊天、直播、游戏、配音等领域被广为使用，本节将对此技术进行简单介绍。

变速变调可以分为变调不变速、变速不变调和变调又变速 3 种。

1）变调不变速

声音频率的高低叫作音调，而语音中的音调又和基频（F0）密切相关。一般来讲，女声的基频高于男声，童声的基频高于成人，而改变声音的频率就能将男声变女声或女声变男声。

对声音进行变调最简单的方法是进行重采样。假如一段音频正常播放时长为2s,而通过技术手段将音频压缩为1s,那么这段声音的频率无形之间就提高了一倍,音调自然也就变高了。同理,如果将这段音频拉长为4s,这样音调就变低了。

SciPy库中的resample()函数能够实现重采样,例如

```
scipy.signal.resample(x, num)
```

其中,x为需要重采样的信号,num为重采样后的信号长度。

下面用一个简单的例子说明通过重采样改变音调的方法,代码如下:

```
#第3章/resample.py

import librosa
from scipy import signal
import matplotlib.pyplot as plt
import soundfile as sf

#读取音频文件
y, sr = librosa.load('wav/shengrikuaile.wav', sr=16000)
n = len(y)                                  #原信号采样点数

#重采样
y2 = signal.resample(y, n//2)               #采样点减半
sf.write('wav/resampled.wav', y2, sr)

#绘制重采样前后的波形图
plt.figure(figsize=(15, 4))
librosa.display.waveshow(y, sr=sr, axis='time')

plt.figure(figsize=(15, 4))
librosa.display.waveshow(y2, sr=sr, axis='time')
```

上述代码对一段语音进行了重采样,重采样后的采样点减少了一半,因而频率提高了一倍。对比重采样前后的音频可以发现,原来的声音是一个正常的男声,重采样后声音变尖,颇像女声。重采样前后的波形图如图3-15所示,两者的波形完全一样,但重采样后时长缩短了。

2) 变速不变调

视频播放器中大多具有变速功能,例如1.5倍速、2倍速或者0.5倍速。

变速不变调的经典方法为TSM(Time-Scale Modificaiton)。该方法属于时域法,其基本思路是将音频切割成很小的等份,将每份都截去或者增添一小段数据,然后重新合成。

OLA(Overlap-and-add)是TSM中最为简单的算法,该算法原理如图3-16所示。

该算法可以分成5步,具体如下:

(1) 对音频进行分帧,取其中一帧 X_m,如图3-16(a)所示。

(2) 对 X_m 加汉宁窗,如图3-16(b)所示。

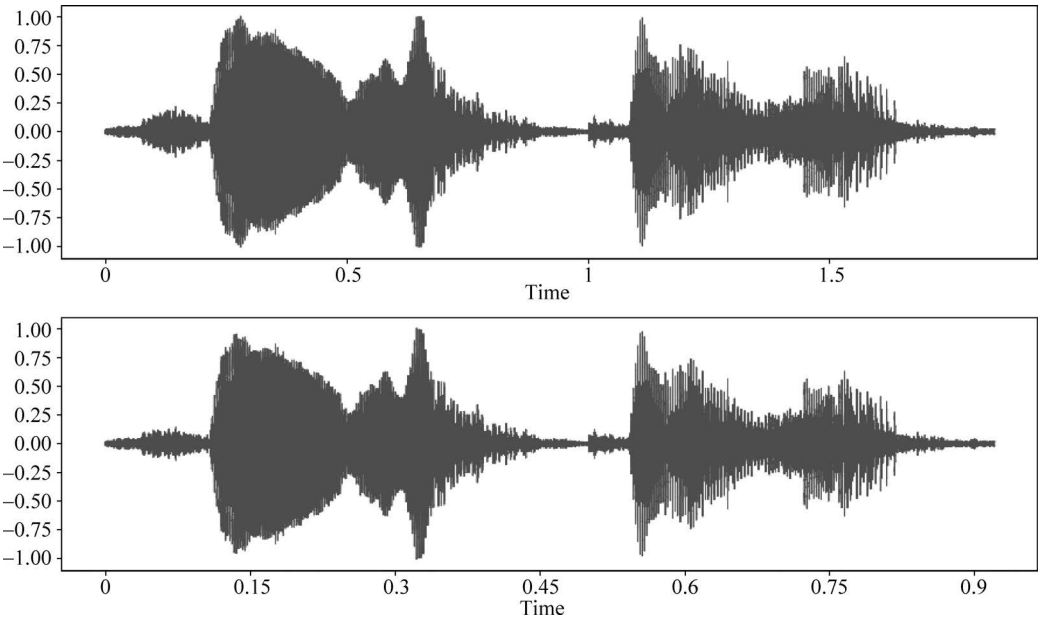


图 3-15　resample.py 运行结果

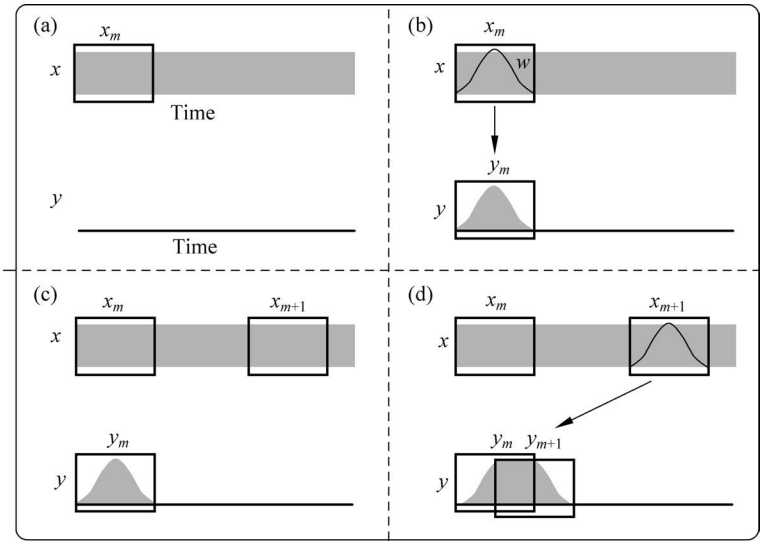


图 3-16　OLA 算法原理

(3) 从 X_m 出发，间隔 H_a 取下一帧 X_{m+1}，如图 3-16(c)所示。

(4) 对 X_{m+1} 加汉宁窗，并与前一帧 X_m 叠加，如图 3-16(d)所示。

(5) 重复此过程直至结束。

这种方法虽然简单，但由于前后两帧的波形可能并不连续，经 OLA 算法处理后的音频会有怪异的声响，如图 3-17 所示。

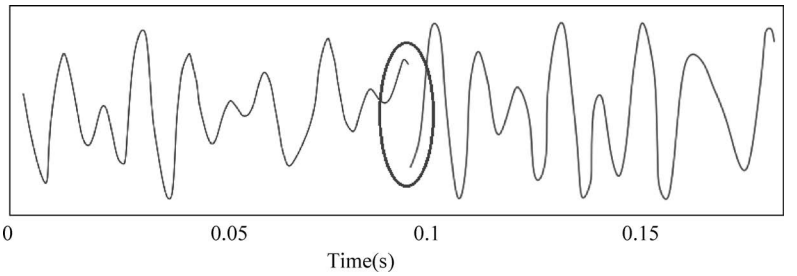
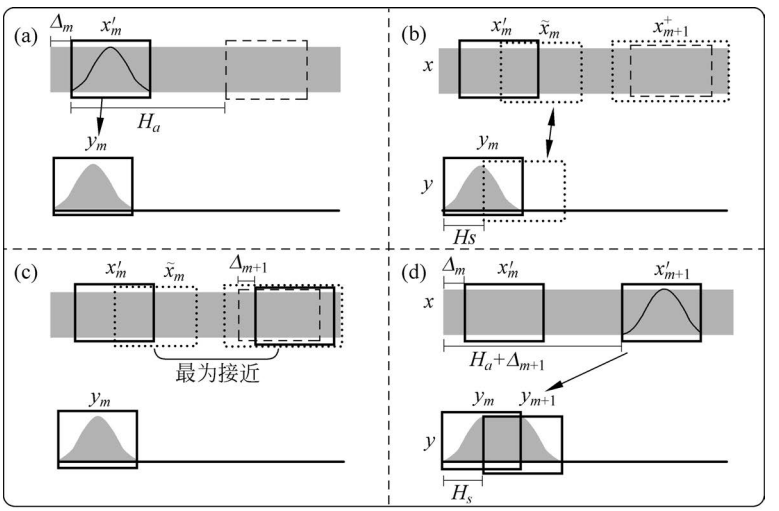


图 3-17　OLA 算法造成波形不连续

OLA 的改进算法有 SOLA、SOLA-FS、TD-PSOLA、WSOLA 等，其中较为知名的是波形相似叠加（Waveform Similarity Overlap-Add，WSOLA）算法。WSOLA 算法不是简单地将 X_{m+1} 与前一帧叠加，而是通过在一个区间内寻找与 $X_m^{\sim}$ 最相似的一帧进行叠加来解决波形不连续的问题，其原理如图 3-18 所示。



(a) 输入单频信号中取一帧x_m，将加窗处理后的x'_m复制到输出序列的y
(b,c)从扩展区域x^+_{m+1}中寻找与自然时间序列中的$x_m^{\sim}$最为接近的一帧
(d)将加窗处理后的帧x'_{m+1}复制到输出序列y

图 3-18　WSOLA 算法原理

对声音进行变速变调的实现较为理想的 Python 库仍然是 Pydub，以下程序用 Pydub 中的 speedup()函数和_spawn()函数进行变速变调，代码如下：

```
#第 3 章/change_voice.py

import pydub

#读取音频文件
audio = pydub.AudioSegment.from_wav('wav/shengrikuaile.wav')
```

```
print('原时长：', audio.duration_seconds, '秒')

#变速变调
speed_rate = 2.0                    #速度变化比率
pitch_rate = 1.5                    #声调变化比率
audio = audio.speedup(playback_speed=speed_rate, chunk_size=300)
print('变速后：', audio.duration_seconds, '秒')
new_rate = int(audio.frame_rate *pitch_rate)
audio = audio._spawn(audio.raw_data,
                     overrides={'frame_rate': new_rate})
print('最终时长：', audio.duration_seconds, '秒')

#保存变换后的音频文件
audio.export('wav/changed.wav')
print('OK')
```

程序的变声效果较为理想，不过无论是变调还是变速音频时长都有变化，如图 3-19 所示。

```
原时长：  1.8422902494331066 秒
变速后：  0.966984126984127 秒
最终时长：  0.6446560846560847 秒
```

图 3-19　change_voice. py 运行结果

此外，变调变速后的波形图与原来的波形图相比也发生了较大的变化，如图 3-20 所示。

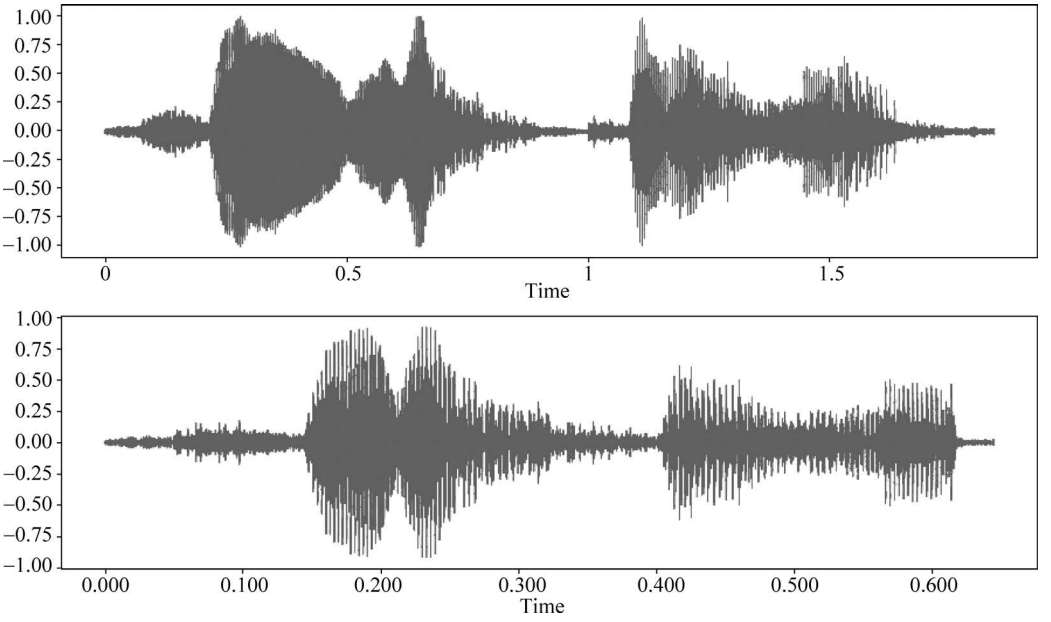


图 3-20　变速变调前后波形图对比

3. 滤波器

在对音频信号进行处理的过程中，经常需要进行滤波。例如，人说话时的基频一般在100～400Hz，提取基音时只需语音的低频成分，可以通过低通滤波实现；又如，某些噪声频率较低，可通过高通滤波去除。

滤波器可分为以下4类。

（1）高通滤波器：仅允许高频信号通过。

（2）低通滤波器：仅允许低频信号通过。

（3）带通滤波器：仅允许指定频率范围内的信号通过。

（4）带阻滤波器：阻止指定频率范围内的信号通过。

这些滤波器类型可以用有限脉冲响应（Finite Impulse Response，FIR）或无限脉冲响应（Infinite Impulse Response，IIR）滤波器实现。所谓脉冲响应（Impulse Response），是指滤波器在时域内的表现，滤波器通常具有较宽的频率响应，这对应于时域内的短时间脉冲，如图3-21所示。

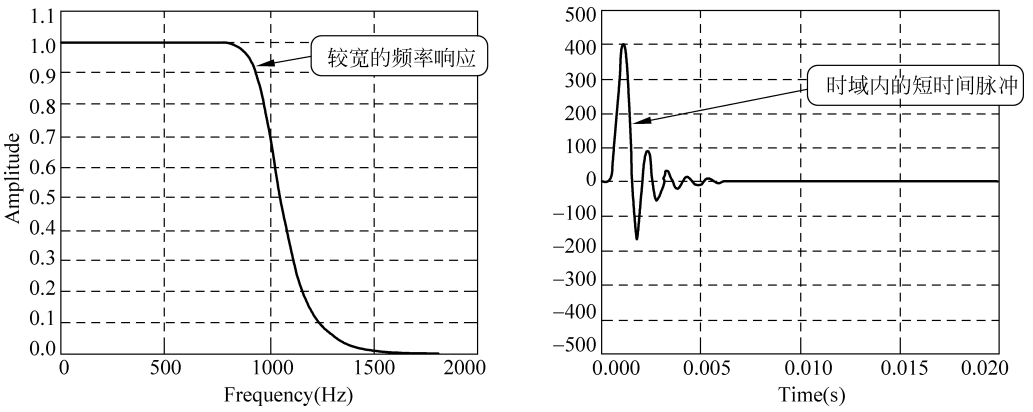


图3-21 低通滤波器的脉冲响应

滤波后的信号与原始信号相比会出现轻微的偏移或时间延迟，在这点上，FIR和IIR的表现并不相同。FIR滤波器在所有频率上具有相同的时延，而IIR滤波器的时延随频率的变化而变化，通常IIR滤波器中最大的时延出现在截止频率处。不过，通过“前向”和“后向”滤波，时延可以被消除，这就是零相位滤波（Zero Phase Filtering）。

理想滤波器要求在通带内的幅频特性为常数、相频特性的斜率为常值，在通带外的幅频特性为0，而这在物理上是无法实现的，因此，在实际应用中常采用适当逼近的方法，常用的逼近方法有以下几种。

（1）巴特沃斯逼近。

（2）切比雪夫逼近。

（3）贝塞尔逼近。

巴特沃斯（Butterworth）滤波器设计简单，性能上没有明显的缺点，因而得到了广泛应用，下面以巴特沃斯逼近为例进行介绍。

SciPy 中的 butter()函数可用来设计巴特沃斯滤波器，该函数的原型如下：

```
scipy.signal.butter(N, Wn, btype='low', analog=False, output='ba', fs=None)

【参数说明】
(1) N：过滤器的阶数。
(2) Wn：截止频率。对数字滤波器来讲，如果 fs 未指定，则 Wn 为归一化后频率，值为 0~1；对模拟滤波器来讲，Wn 是角频率(如 rad/s)。
(3) btype：过滤器类型，默认为低通滤波器，可选项如下。
◆'lowpass'：低通滤波器。
◆'highpass'：高通滤波器。
◆'bandpass'：带通滤波器。
◆'bandstop'：带阻滤波器。
(4) analog：True 为模拟滤波器，False 为数字滤波器。
(5) output：输出类型，可选项如下。
◆'ba'：分子分母。
◆'zpk'：零极点。
◆'sos'：二阶基本节。
(6) fs：数字系统的采样频率。

【返回值】
根据 output 类型不同返回值也不同，具体如下。
◆b,a：依次代表分子和分母多项式，仅当 output 为'ba'时。
◆z,p,k：依次代表零点、极点和增益，仅当 output 为'zpk'时。
◆sos：二阶基本节结构，仅当 output 为'sos'时。
```

具体用来滤波的是 lfilter()函数，其原型如下：

```
scipy.signal.lfilter(b, a, x, axis=-1, zi=None)

【参数说明】
(1) b：一维序列中的分子系数向量。
(2) a：一维序列中的分母系数向量。
(3) x：N 维的输入数组。
(4) axis：线性过滤的轴。
(5) zi：滤波延迟的初始条件。

【返回值】
(1) y：数字滤波后的输出。
(2) zf：如果 zi 为 None，则将不会有返回值，否则将保存最终滤波延迟值。
```

经 lfilter()函数滤波后的信号会产生一定的时延，如果需要保持相位不变，则可采用零相位滤波，在 SciPy 中可以用 filtfilt()函数实现，其原型如下：

```
scipy.signal.filtfilt(b, a, x, axis=-1, padtype='odd', padlen=None, method='pad',
irlen=None)
# 对信号进行两次线性数字滤波，一次向前，一次向后，滤波后相位保持不变

【参数说明】
```

(1) b：分子系数向量。
(2) a：分母系数向量。
(3) x：需要滤波的数据。
(4) axis：对 x 进行过滤的轴。
(5) padtype：填充类型，可选项如下。
◆'odd'：奇。
◆'even'：偶。
◆'constant'：常数。
◆None：不填充。
(6) padlen：填充长度。
(7) method：信号边缘的处理方式，可选项如下。
◆'pad'：填充法，填充方式由 padtype 和 padlen 决定，irlen 参数被忽略。
◆'gust'：Gustafsson 法，padtype 和 padlen 将被忽略。
(8) irlen：当 method 为'gust'时，该参数用于指定滤波器的脉冲响应的长度，当 irlen 为 None 时不会忽略任何部分的脉冲响应。对于长信号，指定 irlen 可显著提升滤波器的性能。

下面用一个具体的例子说明对信号进行滤波的方法，代码如下：

```
#第 3 章/filters.py

from scipy import signal
import numpy as np
import matplotlib.pyplot as plt

#构造信号 x0
t = np.linspace(-1, 1, 201)
x0 = (np.sin(2*np.pi *0.7*t *(1-t) + 2) + 0.3*np.sin(2*np.pi *1.5*t + 1)
      + 0.2*np.cos(2*np.pi *3.5*t))

#添加噪声
r = np.random.default_rng()
noise = r.standard_normal(len(t)) *0.1
noisy = x0 + noise

#低通滤波
b, a = signal.butter(3, 0.05)
zi = signal.lfilter_zi(b, a) #设置初始状态
z1, _ = signal.lfilter(b, a, noisy, zi=zi *noisy[0])
z2, _ = signal.lfilter(b, a, z1, zi=zi *z1[0])

#零相位滤波
y = signal.filtfilt(b, a, noisy)

#绘制滤波前后波形图
plt.figure
plt.grid(True)
plt.plot(t, noisy, 'b', alpha=0.6)
plt.plot(t, z1, 'r-.')
plt.plot(t, z2, 'r-')
```

```
    plt.plot(t, y, 'k--')
    plt.legend(('noisy', 'lfilter I', 'lfilter II', 'filtfilt'), loc='best')
    plt.show()
```

程序的运行结果如图 3-22 所示。注意调用 lfilter()函数产生的波形是有延迟的，而零相位滤波后相位不变，只是滤掉了高频部分。

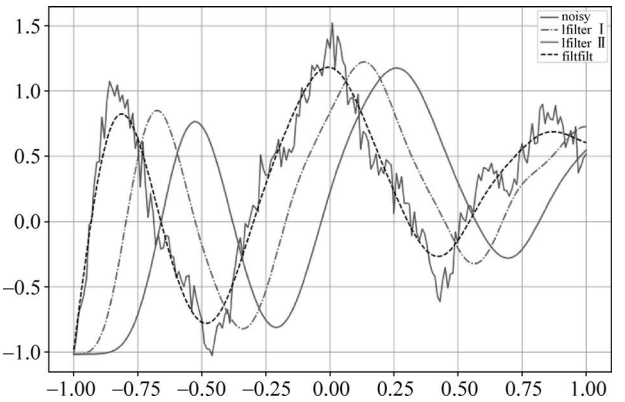


图 3-22　filters. py 运行结果