Python音声処理入門（第2部）

音声の可視化とエフェクト処理

第2部の内容

- **音声の可視化**（波形・スペクトログラム）
- **matplotlibによる詳細な可視化技術**
- **Numpy配列操作による音声エフェクト**
- **配列インデックシングの実践的応用**
- **WAVファイル保存の内部メカニズム**

3. 音声の可視化：理論的背景

なぜ可視化が重要か？

直感的理解
数値データを視覚で把握

問題発見
ノイズ、クリッピング、無音部分の特定

エフェクト確認
処理前後の変化を比較

学習効果
音響現象を視覚的に学習

2つのアプローチ

- **時間軸表示**: 音声波形（振幅の時間変化）
- **周波数軸表示**: スペクトログラム（周波数成分の時間変化）

波形可視化の数学的基礎

時間軸の生成原理

```
# サンプル数から時間軸を計算
time = np.arange(0, DURATION, 1/FS)

# 実際の計算例（FS=44100Hz, DURATION=5秒）
# np.arange(0, 5, 1/44100)
# → [0, 0.0000227, 0.0000454, ..., 4.9999773]
# → 220500個の時間点
```

データポイントの対応関係

```
len(time) == len(myrecording)  # 必ずTrue

# 各インデックスでの対応
time[0] → myrecording[0]       # 0秒時点の振幅
time[1000] → myrecording[1000] # 約0.023秒時点の振幅
```

practice_2_visualize.py 詳細解説

```
import matplotlib.pyplot as plt

# 時間軸作成（重要なポイント）
time = np.arange(0, len(myrecording)/FS, 1/FS)

# より正確な時間軸作成方法
# データ長から逆算するため、浮動小数点誤差を回避
time = np.linspace(0, len(myrecording)/FS, len(myrecording))

# グラフ設定の詳細
plt.figure(figsize=(12, 4))    # 横長で波形を見やすく
plt.plot(time, myrecording)
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")        # 通常 -1.0 ~ 1.0 の範囲
plt.title("Waveform")
plt.grid(True, alpha=0.3)      # 薄いグリッドで読み取りやすく
plt.tight_layout()             # レイアウト自動調整
plt.show()
```

波形から読み取れる情報

振幅パターンの意味

```
大きな振幅 → 大きな音（loud）
小さな振幅 → 小さな音（quiet）
振幅 = 0   → 無音（silence）
```

時間的特徴の観察

- **アタック**: 音の立ち上がり（急峻な振幅変化）
- **ディケイ**: 音の減衰（徐々に小さくなる振幅）
- **サステイン**: 音の持続（一定レベルの振幅）
- **リリース**: 音の消失（振幅がゼロに向かう）

波形パターンによる音の種類識別

```
規則的な波形   → 楽器音、純音
不規則な波形   → 雑音、摩擦音
パルス状波形   → 破裂音、クリック音
```

スペクトログラム：周波数領域の可視化

スペクトログラムとは

「音声の周波数成分がどのように時間変化するかを表した2次元画像」

軸の意味

- **X軸**: 時間（秒）
- **Y軸**: 周波数（Hz）
- **色の濃淡**: その時刻・周波数での音の強さ（パワー）

```
from scipy.signal import spectrogram

# スペクトログラム計算
f, t, Sxx = spectrogram(
    myrecording,          # 音声データ
    fs=FS,                # サンプリング周波数
    window='hann',        # 窓関数（ハニング窓）
    nperseg=1024,         # フレーム長
    noverlap=512          # オーバーラップ長
)
```

スペクトログラムパラメータの詳細

窓関数の選択

'hann':　ハニング窓（一般的、サイドローブ小）
'hamming': ハミング窓（ハニングの改良版）
'blackman': ブラックマン窓（最もなめらか、分解能低）
'boxcar':　矩形窓（分解能高、漏れ大）

フレーム長（nperseg）の影響

小さい値（**256**）： 時間分解能高、周波数分解能低
中程度（**1024**）： バランス良好
大きい値（**4096**）：時間分解能低、周波数分解能高

オーバーラップの効果

```
オーバーラップ = nperseg // 2  # 50%が一般的
# → 時間変化をなめらかに追跡、計算量は増加
```

スペクトログラム表示の工夫

```
# dBスケール表示（対数スケール）
Sxx_db = 10 * np.log10(Sxx + 1e-10)  # ゼロ除算回避

# カラーマップとコントラスト調整
plt.figure(figsize=(12, 8))
plt.pcolormesh(t, f, Sxx_db,
               shading='gouraud',      # なめらかな表示
               cmap='viridis')         # 見やすいカラーマップ
plt.ylabel('Frequency [Hz]')
plt.xlabel('Time [s]')
plt.colorbar(label='Power [dB]')
plt.ylim([0, FS//2])                   # ナイキスト周波数まで表示
```

人間の知覚に合わせた表示

```
# 低周波数域を詳しく表示（対数周波数軸）
plt.yscale('log')
plt.ylim([20, FS//2])  # 可聴域のみ表示
```

4. Numpy配列による音声エフェクト

エフェクトの本質

「音声データ（数値配列）に対する数学的操作」

基本的な配列操作とその音響効果

```
配列 × スカラー    → 音量変更
配列 + 配列        → 音の合成
配列の順序変更     → 時間軸操作
配列の部分抽出     → 切り出し、ループ
```

エフェクト1: 音量制御の数学

```
# 基本の音量変更
louder = myrecording * 2.0     # 2倍の音量（+6dB）
quieter = myrecording * 0.5    # 半分の音量（-6dB）
```

dB（デシベル）との関係

```
# dB計算式: dB = 20 * log10(倍率)
import math

gain_2x = 20 * math.log10(2.0)    # 約+6.02dB
gain_half = 20 * math.log10(0.5)  # 約-6.02dB

# 実用的なdB指定関数
def apply_gain_db(audio, gain_db):
    gain_linear = 10 ** (gain_db / 20)
    return audio * gain_linear

# 使用例
louder_10db = apply_gain_db(myrecording, 10)   # +10dB
quieter_3db = apply_gain_db(myrecording, -3)   # -3dB
```

エフェクト2: 逆再生の配列操作

```
# 逆再生の実装
reversed_audio = myrecording[::-1]
```

スライシングの詳細

```
myrecording[start:stop:step]
[::-1] の意味:
start: 省略（最後から）
stop:  省略（最初まで）
step:  -1（1つずつ逆向き）
# 等価な書き方
reversed_audio = myrecording[len(myrecording)-1::-1]
```

部分的な逆再生

```
# 後半3秒だけ逆再生
FS = 44100
split_point = len(myrecording) - 3 * FS
normal_part = myrecording[:split_point]
reversed_part = myrecording[split_point:][::-1]
partial_reversed = np.concatenate([normal_part, reversed_part])
```

エフェクト3: 配列結合テクニック

```
# 1秒の無音を挟んで2回再生
silence_duration = 1.0  # 秒
silence_samples = int(silence_duration * FS)
# 無音配列の生成（次元を合わせる）
if myrecording.ndim == 1:
    silence = np.zeros(silence_samples, dtype=myrecording.dtype)
else:
    silence = np.zeros((silence_samples, myrecording.shape[1]),
                       dtype=myrecording.dtype)
# 結合
combined = np.concatenate([myrecording, silence, myrecording])
```

より高度な結合パターン

```
# フェードイン・フェードアウト付き結合
def create_fade(length, fade_type='in'):
    fade = np.linspace(0, 1, length) if fade_type == 'in' else np.linspace(1, 0, length)
    return fade

fade_samples = int(0.1 * FS)  # 0.1秒のフェード
fade_in = create_fade(fade_samples, 'in')
fade_out = create_fade(fade_samples, 'out')
# 適用
myrecording[:fade_samples] *= fade_in
myrecording[-fade_samples:] *= fade_out
```

エフェクト4: エコー（やまびこ）の詳細実装

```
def create_echo(audio, delay_sec, decay=0.5, FS=44100):
    delay_samples = int(delay_sec * FS)
    # 出力バッファ（元音声＋遅延分の長さ）
    output_length = len(audio) + delay_samples
    echo_audio = np.zeros(output_length, dtype=audio.dtype)
    # 元音声をコピー
    echo_audio[:len(audio)] = audio
    # 遅延音声を加算
    echo_audio[delay_samples:delay_samples+len(audio)] += audio * decay

    return echo_audio
```

多重エコーの実装

```
def multi_echo(audio, delays=[0.2, 0.4, 0.6], decays=[0.6, 0.4, 0.2], FS=44100):
    result = np.copy(audio)

    for delay, decay in zip(delays, decays):
        delay_samples = int(delay * FS)
        if len(result) > delay_samples:
            result[delay_samples:] += audio[:len(result)-delay_samples] * decay

    return result
```

音声エフェクトの実用的な考慮事項

クリッピング（音割れ）の防止

```
def prevent_clipping(audio, target_max=0.95):
    current_max = np.max(np.abs(audio))
    if current_max > target_max:
        # 正規化して音割れを防ぐ
        audio = audio * (target_max / current_max)
        print(f"クリッピング防止: {current_max:.3f} → {target_max:.3f}")
    return audio
# エフェクト適用後に必ずチェック
processed_audio = prevent_clipping(processed_audio)
```

ステレオ音声の処理

```
if myrecording.ndim == 2:  # ステレオの場合
    # 各チャンネルに個別にエフェクト適用
    left_channel = myrecording[:, 0]
    right_channel = myrecording[:, 1]
    # 左右で異なるエフェクト
    left_processed = left_channel * 1.2    # 左を少し大きく
    right_processed = right_channel * 0.8  # 右を少し小さく
    # ステレオ音声として再構成
    stereo_result = np.column_stack([left_processed, right_processed])
```

5. WAVファイル保存の内部メカニズム

WAVファイルフォーマットの構造

WAVファイル = RIFFヘッダ + フォーマットチャンク + データチャンク

ヘッダ情報の詳細

RIFF識別子:　"RIFF" (4 bytes)
ファイルサイズ: 全体サイズ-8 (4 bytes)
フォーマット:　"WAVE" (4 bytes)
フォーマット識別: "fmt " (4 bytes)
チャンクサイズ:　16 (4 bytes)
音声フォーマット: 1=PCM (2 bytes)
チャンネル数:　1 or 2 (2 bytes)
サンプリング周波数: 44100等 (4 bytes)
バイトレート:　FS×チャンネル×bit/8 (4 bytes)
ブロックサイズ:　チャンネル×bit/8 (2 bytes)
ビット深度:　16 or 24等 (2 bytes)
データ識別子:　"data" (4 bytes)
データサイズ:　実際の音声データサイズ (4 bytes)
音声データ:　PCMデータ本体

practice_4_wav.py の詳細解説

```
from scipy.io.wavfile import write, read
import numpy as np

# 録音（前半部分は同じ）
recording = sd.rec(int(DURATION * FS), samplerate=FS, channels=1)
sd.wait()
# データ型変換（重要！）
# sounddeviceの出力: float32 [-1.0, 1.0]
# WAVファイル標準: int16 [-32768, 32767]
recording_int16 = (recording * 32767).astype(np.int16)
# WAVファイル書き込み
write(FILENAME, FS, recording_int16)
```

データ型変換の必要性

```
# float32のまま保存した場合
write("float.wav", FS, recording)  # 一部ソフトで再生不可
# int16に変換して保存
recording_int16 = (recording * 32767).astype(np.int16)
write("int16.wav", FS, recording_int16)  # 汎用性が高い
```

WAVファイル読み書きの実践的テクニック

完全な読み書き関数の実装

```
def save_audio(filename, audio_data, sample_rate, bit_depth=16):
    """
    音声データをWAVファイルに保存（クリッピング対策付き）
    """
    # クリッピング防止
    max_val = np.max(np.abs(audio_data))
    if max_val > 1.0:
        audio_data = audio_data / max_val
        print(f"正規化しました: {max_val:.3f} → 1.0")

    # ビット深度に応じた変換
    if bit_depth == 16:
        audio_int = (audio_data * 32767).astype(np.int16)
    elif bit_depth == 24:
        audio_int = (audio_data * 8388607).astype(np.int32)
    else:
        raise ValueError("対応していないビット深度")

    write(filename, sample_rate, audio_int)
    print(f"{filename} に保存完了 ({bit_depth}bit)")
```

ファイルサイズの計算と最適化

ファイルサイズの予測

```python
def calculate_file_size(duration, sample_rate, channels, bit_depth):
    """
    WAVファイルのサイズを事前計算
    """
    samples_per_sec = sample_rate * channels
    bytes_per_sample = bit_depth // 8
    data_size = duration * samples_per_sec * bytes_per_sample
    header_size = 44  # WAVヘッダの標準サイズ
    total_size = data_size + header_size

    return {
        'data_bytes': int(data_size),
        'total_bytes': int(total_size),
        'total_mb': total_size / (1024 * 1024)
    }

# 使用例
size_info = calculate_file_size(
    duration=300,     # 5分
    sample_rate=44100,
    channels=2,       # ステレオ
    bit_depth=16
)
print(f"5分ステレオ録音: {size_info['total_mb']:.1f} MB")
```

実行方法とバリエーション

基本的な使用法

```
# デフォルト設定（5秒モノラル、44.1kHz）
python practice_4_wav.py

# ファイル名指定
python practice_4_wav.py -o my_recording.wav

# 録音時間指定
python practice_4_wav.py -d 10.0 -o long_recording.wav
```

高品質録音設定

```
# 高サンプリング周波数＋ステレオ
python practice_4_wav.py -r 48000 -c 2 -d 30 -o hq_stereo.wav

# 音声認識用（16kHzモノラル）
python practice_4_wav.py -r 16000 -c 1 -d 10 -o speech.wav
```

第2部まとめ

習得した技術

音声可視化
波形とスペクトログラムの生成・解釈

Numpy配列操作
エフェクト処理の数学的基礎

配列インデックシング
逆再生、部分抽出、結合

WAVファイル処理
内部構造の理解と実装

重要なポイント

- **音声 = 数値配列**: 全ての処理は配列操作として表現
- **可視化の力**: 数値だけでは見えない特徴を発見