Python Programming for Audio

Essential Tools and Techniques for Audio Processing

Xingjian Du (x.du@rochester.edu)

September 9, 2025

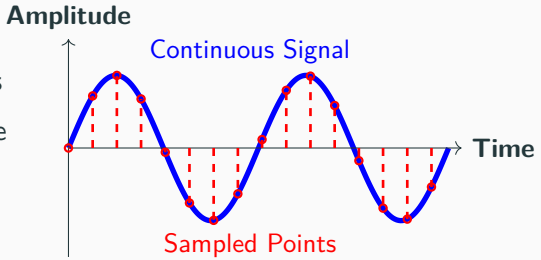
ECE 477: Computer Audition
University of Rochester
Electrical and Computer Engineering Department

1. **Audio Fundamentals**
   - Digital audio representation
   - Sampling theory and quantization
   - Fourier Transform and STFT
   - Mel-frequency analysis
2. **Audio Codecs**
   - Lossless vs. lossy compression
   - Common audio formats
3. **Python Audio Libraries**
   - librosa, scipy.signal, numpy, matplotlib
4. **Audio Processing Tools**
   - SoX and FFmpeg integration
5. **Essential Audio Parameters**
   - Sample rates, bit depths, compression ratios

Audio Fundamentals

Waveforms as Arrays

- Audio signals are continuous analog waveforms
- Digitization converts to discrete samples
- Python represents audio as NumPy arrays
- Each sample = amplitude at discrete time



Key Concepts

- Sampling rate (Hz)
- Bit depth (bits per sample)
- Channels (mono, stereo, multichannel)

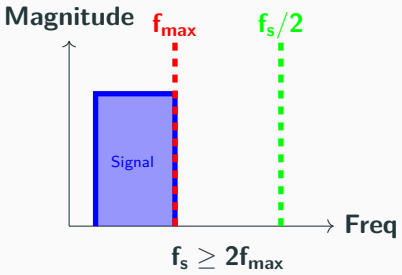
```python
import librosa
import numpy as np
import matplotlib.pyplot as plt

# Load audio file
audio, sr = librosa.load('audio_file.wav', sr=None)

# Display basic information
print(f"Sample rate: {sr} Hz")
print(f"Duration: {len(audio)/sr:.2f} seconds")
print(f"Shape: {audio.shape}")

# Plot waveform
plt.figure(figsize=(10, 4))
time = np.linspace(0, len(audio)/sr, len(audio))
plt.plot(time, audio)
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.title('Audio Waveform')
plt.show()
```

Nyquist-Shannon Theorem

- Sampling rate $\geq 2 \times$ highest frequency
- Prevents aliasing artifacts
- Common rates: 44.1kHz, 48kHz, 96kHz

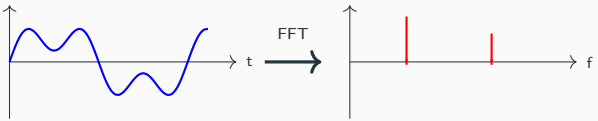
Quantization

- Maps continuous amplitudes to discrete levels
- Bit depth determines resolution
- 16-bit: 65,536 levels
- 24-bit: 16,777,216 levels



Time $\leftrightarrow$ Frequency Domain

- Decomposes signal into frequency components
- FFT: Fast Fourier Transform algorithm
- Reveals spectral content
- Essential for audio analysis



Applications

- Spectral analysis
- Filtering
- Audio effects
- Compression

```python
import numpy as np, matplotlib.pyplot as plt
from scipy.fft import fft, fftfreq

# Generate sample signal
fs = 44100; t = np.linspace(0, 1, fs)
signal = np.sin(2*np.pi*440*t) + 0.5*np.sin(2*np.pi*880*t)

# Compute FFT and plot
fft_result = fft(signal)
frequencies = fftfreq(len(signal), 1/fs)
plt.figure(figsize=(10, 4))
plt.plot(frequencies[:len(frequencies)//2],
         np.abs(fft_result)[:len(fft_result)//2])
plt.xlabel('Frequency (Hz)'); plt.ylabel('Magnitude')
plt.xlim(0, 2000); plt.title('Frequency Spectrum'); plt.show()
```

Audio Codecs

Lossless

- Perfect reconstruction
- Larger files
- Professional use
- WAV, FLAC, ALAC

Lossy

- Irreversible loss
- Smaller files
- Consumer use
- MP3, AAC, OGG

Format	Type	Quality	Size
WAV	Lossless	Perfect	Large
FLAC	Lossless	Perfect	Medium
MP3	Lossy	Good	Small
AAC	Lossy	Better	Smaller

Python Audio Libraries

Key Features:

- Comprehensive audio loading and processing
- Spectral analysis (STFT, CQT, Mel spectrograms)
- Feature extraction (MFCCs, chroma, tempo)
- Beat tracking and rhythm analysis
- Audio effects and transformations

Installation: `pip install librosa`

Common Use Cases:

- Music information retrieval
- Audio visualization
- Preprocessing for machine learning
- Audio analysis and research

```python
import librosa
import numpy as np

# Load audio
y, sr = librosa.load('music.wav', sr=22050)

# Extract features
# MFCCs (Mel-frequency cepstral coefficients)
mfccs = librosa.feature.mfcc(y=y, sr=sr,
                             n_mfcc=13)

# Chroma features
chroma = librosa.feature.chroma(y=y, sr=sr)

# Spectral centroid
spectral_centroids = \
    librosa.feature.spectral_centroid(y=y, sr=sr)

# Zero crossing rate
zcr = librosa.feature.zero_crossing_rate(y)

# Tempo and beat tracking
tempo, beats = librosa.beat.beat_track(y=y, sr=sr)

print(f"Tempo: {tempo:.2f} BPM")
print(f"MFCCs shape: {mfccs.shape}")
print(f"Chroma shape: {chroma.shape}")
```

Flexible Loading Parameters:

- `sr=None` preserves original sample rate
- `mono=False` preserves stereo/multichannel
- `offset, duration` for partial loading
- `dtype` controls precision (float32/float64)

```
# Load with custom parameters
y, sr = librosa.load('audio.wav', sr=22050, mono=True,
                     offset=10.0, duration=30.0)
# Load stereo preserving channels
y_stereo, sr = librosa.load('stereo.wav', sr=None, mono=False)
# Load with specific precision
y_hq, sr = librosa.load('audio.wav', dtype=np.float64)
```

Key Spectral Analysis Functions:

- `librosa.stft()`: Short-Time Fourier Transform
- `spectral_centroid()`: Brightness measure
- `spectral_rolloff()`: High-frequency content
- `zero_crossing_rate()`: Temporal texture

```
# Compute spectral features
S = librosa.stft(y, hop_length=512)
centroid = librosa.feature.spectral_centroid(y=y, sr=sr)
rolloff = librosa.feature.spectral_rolloff(y=y, sr=sr, roll_percent=0.85)
zcr = librosa.feature.zero_crossing_rate(y)
# Display shapes
print(f"STFT shape: {S.shape}")
print(f"Centroid shape: {centroid.shape}")
```

Mel-Scale Analysis:

- Perceptually relevant frequency representation
- Common in speech/music analysis
- `n_mels` controls frequency resolution
- `fmax` sets maximum frequency

```
# Mel spectrogram and MFCCs
mel_spec = librosa.feature.melspectrogram(y=y, sr=sr,
                                          n_mels=128, fmax=8000)
mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)
# Delta features for temporal dynamics
delta_mfccs = librosa.feature.delta(mfccs)
delta2_mfccs = librosa.feature.delta(mfccs, order=2)
print(f"Mel spectrogram shape: {mel_spec.shape}")
```

Source Separation Techniques:

- Separates harmonic and percussive components
- Useful for instrument analysis
- `margin` controls separation strength
- Can process magnitude spectrogram directly

```
# Harmonic-percussive separation
y_harmonic, y_percussive = librosa.effects.hpss(y)
# On spectrogram domain
D = librosa.amplitude_to_db(np.abs(librosa.stft(y)))
D_harmonic, D_percussive = librosa.decompose.hpss(D,
                                                  margin=(1.0,5.0))
# Play separated components
print(f"Original length: {len(y)}")
print(f"Harmonic length: {len(y_harmonic)}")
```

Pitch-Related Features:

- `chroma`: Pitch class profiles
- `piptrack()`: Pitch tracking
- `tonnetz`: Tonal centroid features
- Useful for music information retrieval

```
# Pitch and chroma features
chroma = librosa.feature.chroma_stft(y=y, sr=sr)
pitches, magnitudes = librosa.piptrack(y=y, sr=sr, threshold=0.1)
tonnetz = librosa.feature.tonnetz(y=y, sr=sr)
# Constant-Q chroma for better frequency resolution
chroma_cqt = librosa.feature.chroma_cqt(y=y, sr=sr)
print(f"Chroma shape: {chroma.shape}")
```

Rhythm Analysis:

- `beat_track()`: Beat detection
- `tempo()`: Tempo estimation
- `onset_detect()`: Note onset detection
- Dynamic programming for robust tracking

```
# Tempo and beat analysis
tempo, beats = librosa.beat.beat_track(y=y, sr=sr)
onsets = librosa.onset.onset_detect(y=y, sr=sr, units='time')
# Multiple tempo hypotheses
tempo_multi = librosa.beat.tempo(y=y, sr=sr, aggregate=None)
beat_times = librosa.frames_to_time(beats, sr=sr)
print(f"Detected tempo: {tempo:.1f} BPM")
print(f"Number of beats: {len(beats)}")
```

Built-in Audio Effects:

- `time_stretch()`: Change speed without pitch
- `pitch_shift()`: Change pitch without speed
- `preemphasis()`: High-frequency boost
- `trim()`: Remove silence automatically

```python
# Audio effects
y_fast = librosa.effects.time_stretch(y, rate=1.5)
y_pitched = librosa.effects.pitch_shift(y, sr=sr, n_steps=4)
y_trimmed, idx = librosa.effects.trim(y, top_db=20)
y_preemph = librosa.effects.preemphasis(y, coef=0.97)
print(f"Original: {len(y)}, Trimmed: {len(y_trimmed)}")
```

Matrix Factorization Techniques:

- `nn_filter()`: Non-negative filtering
- Integration with scikit-learn NMF
- Useful for source separation
- Can discover latent patterns

```
# Spectral decomposition
S_full, phase = librosa.magphase(librosa.stft(y))
S_filter = librosa.decompose.nn_filter(S_full,
                                       aggregate=np.median,
                                       metric='cosine')
S_foreground = S_full - S_filter
# Reconstruct audio from filtered spectrogram
y_foreground = librosa.istft(S_foreground * phase)
```

Built-in Display Functions:

- `librosa.display.specshow()`: Spectrograms
- `librosa.display.waveshow()`: Waveforms
- Automatic frequency/time axis labeling
- Colormap and scaling options

```
# Visualization
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 8))
plt.subplot(3, 1, 1)
librosa.display.waveshow(y, sr=sr)
plt.subplot(3, 1, 2)
librosa.display.specshow(librosa.amplitude_to_db(S),
                         sr=sr, x_axis='time', y_axis='hz')
plt.subplot(3, 1, 3)
librosa.display.specshow(mfccs, sr=sr, x_axis='time')
plt.tight_layout()
```

Common Workflow Patterns:

- Feature extraction pipelines
- Real-time processing with frames
- Integration with machine learning
- Batch processing multiple files

```python
# Feature extraction pipeline
def extract_features(audio_path):
    y, sr = librosa.load(audio_path, sr=22050)
    features = np.hstack([
        np.mean(librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13), axis=1),
        np.mean(librosa.feature.spectral_centroid(y=y, sr=sr), axis=1),
        np.mean(librosa.feature.zero_crossing_rate(y), axis=1)
    ])
    return features

# Batch processing
features_list = [extract_features(f) for f in audio_files]
```

scipy.signal:

- Signal processing utilities (filters, windows)
- Fourier transforms and convolution
- Filter design and analysis

numpy:

- Fundamental array operations
- Mathematical functions
- Broadcasting and vectorization

matplotlib:

- Audio visualization and plotting
- Spectrograms and waveform displays
- Publication-quality figures

Audio Processing Tools

Sound eXchange (SoX) Features:

- Command-line audio processing
- Format conversion
- Audio effects and filters
- Batch processing capabilities
- Cross-platform availability

```
# Format conversion
sox input.wav output.mp3
# Apply effects
sox input.wav output.wav reverb 0.8 0.8 0.8 0.5
# Change sample rate
sox input.wav -r 44100 output.wav
# Trim audio
sox input.wav output.wav trim 10 30
```

FFmpeg Capabilities:

- Audio/video encoding and decoding
- Format conversion and transcoding
- Streaming and filtering
- Metadata manipulation

```
# Extract audio from video
ffmpeg -i video.mp4 -vn -acodec copy audio.mp3
# Convert with quality
ffmpeg -i input.wav -b:a 320k output.mp3
# Mix audio files
ffmpeg -i audio1.wav -i audio2.wav -filter_complex amix output.wav
# Apply filters
ffmpeg -i input.wav -af "highpass=f=200" output.wav
# Change sample rate
ffmpeg -i input.wav -ar 44100 output.wav
```

Subprocess Integration Pattern:

- Use `subprocess.run()` for command execution
- Capture output and error streams
- Check return codes for success/failure
- Handle file paths and arguments properly

```python
import subprocess

def convert_with_sox(input_file, output_file, effect=None):
    cmd = ['sox', input_file, output_file]
    if effect:
        cmd.extend(effect.split())
    result = subprocess.run(cmd, capture_output=True, text=True)
    return result.returncode == 0

def extract_audio_ffmpeg(video_file, audio_file):
    cmd = ['ffmpeg', '-i', video_file, '-vn', '-acodec', 'libmp3lame',
           '-ab', '192k', audio_file]
    result = subprocess.run(cmd, capture_output=True, text=True)
    return result.returncode == 0

# Example usage
convert_with_sox('input.wav', 'output.wav', 'reverb 0.5')
extract_audio_ffmpeg('video.mp4', 'audio.mp3')
```

Essential Audio Parameters

Key Digital Audio Concepts:

- **Sample Rate**: How often audio is measured (Hz)
- **Bit Depth**: Precision of each audio sample
- **Nyquist Frequency**: Maximum frequency = Sample Rate / 2
- **Dynamic Range**: Difference between loudest and quietest sounds
- **Bit Rate**: Amount of data per second (for compressed audio)

Why These Matter:

- Higher sample rates capture more high-frequency detail
- Higher bit depths provide better signal-to-noise ratio
- Proper parameters prevent aliasing and quantization noise
- Understanding these helps choose appropriate formats

Standard Sample Rates:

- 44.1 kHz: CD Audio (standard music quality)
- 48 kHz: Professional audio (video/film standard)
- 96 kHz: High-resolution audio (audiophile applications)

Nyquist Frequency Examples:

- 44.1 kHz $\rightarrow$ 22.05 kHz maximum frequency
- 48 kHz $\rightarrow$ 24 kHz maximum frequency
- Human hearing: 20 Hz to 20 kHz

Common librosa STFT Settings:

- n_fft=2048, hop_length=512 (default)
- n_fft=1024, hop_length=256 (faster)
- n_fft=4096, hop_length=1024 (detailed)

Mel Spectrogram Defaults:

- n_mels=128 (standard)
- n_mels=80 (speech applications)
- fmax=8000 Hz (speech) or None (music)

MFCC Parameters:

- n_mfcc=13 (standard speech)
- n_mfcc=20-40 (music applications)

Music & Audio Content:

- Bass: 60-250 Hz
- Midrange: 250 Hz - 4 kHz (most musical content)
- Treble: 4-20 kHz (harmonics, cymbals, air)
- Sub-bass: 20-60 Hz (kick drums, low synths)

Speech:

- Fundamental frequency: 80-300 Hz (male), 165-265 Hz (female)
- Formants: 300-3400 Hz (intelligibility)
- Consonants: 2-8 kHz (clarity, sibilants)

MIDI Note Ranges:

- C0: 16.35 Hz, C4 (middle C): 261.6 Hz, C8: 4186 Hz
- Piano: A0 (27.5 Hz) to C8 (4186 Hz)
- Human voice: 80-1100 Hz (fundamentals)

Summary and Resources

Audio Fundamentals:

- Digital audio = arrays of samples
- Sampling rate and bit depth determine quality
- FFT reveals frequency content, STFT shows time evolution
- Mel scale provides perceptually relevant analysis

Python Ecosystem:

- Librosa for comprehensive audio analysis
- SciPy for signal processing and filtering
- NumPy for array operations
- Matplotlib for visualization

Practical Tools:

- SoX and FFmpeg for command-line processing
- Choose appropriate sample rates and bit depths
- Understand codec trade-offs (quality vs. size)

Thank You!

Questions and Discussion

ECE 477: Computer Audition
University of Rochester - ECE Department

```
python -c "import librosa; print('Happy audio processing!')"
```