Audio DSP Fundamentals

A Hands-On Learning Guide for Live Coders

Learn the building blocks of sound synthesis and audio processing through practical coding exercises

RECOMMENDED PROGRAMMING LANGUAGE

Python

Why Python?

- NumPy for efficient array operations on audio samples
- scipy.io.wavfile for reading/writing WAV files
- matplotlib for visualizing waveforms and spectra
- Simple syntax lets you focus on DSP concepts
- Excellent offline documentation available

December 2024 - January 2025

Contents

Part 1: Foundations

The WAV File Format

Before diving into DSP, you need to understand how digital audio is stored. The WAV format is one of the simplest uncompressed audio formats, making it perfect for learning. Understanding it from first principles will give you insight into how all digital audio works.

What is Digital Audio?

Sound is a continuous wave of pressure changes in air. To store it digitally, we take measurements (samples) of the wave's amplitude at regular intervals. The sample rate tells us how many measurements per second (e.g., 44100 Hz means 44,100 samples per second). The bit depth tells us the precision of each measurement (e.g., 16-bit means values from -32768 to +32767).

WAV File Structure

A WAV file consists of chunks of data, each with a header. The main structure is:

Offset	Size	Field	Description
0	4	ChunkID	"RIFF" (ASCII)
4	4	ChunkSize	File size minus 8 bytes
8	4	Format	"WAVE" (ASCII)
12	4	Subchunk1ID	"fmt " (ASCII, note space)
16	4	Subchunk1Size	16 for PCM
20	2	AudioFormat	1 for PCM (uncompressed)
22	2	NumChannels	1 = mono, 2 = stereo
24	4	SampleRate	e.g., 44100
28	4	ByteRate	SampleRate × NumChannels × BitsPerSample/8
32	2	BlockAlign	NumChannels × BitsPerSample/8
34	2	BitsPerSample	8, 16, 24, or 32
36	4	Subchunk2ID	"data" (ASCII)
40	4	Subchunk2Size	NumSamples × NumChannels × BitsPerSample/8
44	...	Data	The actual audio samples

Key Concepts

Little-Endian: Multi-byte values are stored with the least significant byte first. For example, the number 44100 (0x0000AC44) is stored as bytes: 44 AC 00 00.

Sample Values: For 16-bit audio, samples are signed integers from -32768 to +32767. A value of 0 represents silence, while maximum values represent the loudest sound.

Stereo Interleaving: For stereo files, samples alternate: L1 R1 L2 R2 L3 R3...

Exercise 1: Generate a Pure Sine Wave

Concept

The sine wave is the fundamental building block of all sound. Every complex sound can be decomposed into a sum of sine waves (this is Fourier's theorem). A sine wave represents the purest tone possible—a single frequency with no harmonics.

The mathematical formula for a sine wave is: $y(t) = A \times \sin(2\pi ft + \phi)$, where A is amplitude, f is frequency in Hz, t is time in seconds, and ϕ is phase offset.

Goal

Create an array of floating-point samples representing a sine wave at a given frequency. This array will be the foundation for all subsequent exercises.

Task Description

Write a function with the following specifications:

Input Parameters	
`frequency`	The pitch of the tone in Hz (e.g., 440 for A4)
`duration`	Length of the sound in seconds
`sample_rate`	Samples per second (default: 44100)
`amplitude`	Peak amplitude from 0.0 to 1.0 (default: 0.5)
Output	
`samples`	A NumPy array of float values between -amplitude and +amplitude

Algorithm Steps

1. Calculate the total number of samples: num_samples = duration × sample_rate

2. Create a time array from 0 to duration with num_samples points

3. Apply the sine formula to each time point

4. Scale by amplitude

Hints

- *Use numpy.linspace(0, duration, num_samples, endpoint=False) to create evenly spaced time values*
- *Use numpy.sin() which operates on entire arrays efficiently*
- *Remember: 2π radians = one complete cycle, so at frequency f, you complete f cycles per second*
- *Test with 440 Hz (A4 note) for 1 second—you should get exactly 44100 samples*

Verification

Plot your waveform using matplotlib. For a 440 Hz wave, you should see about 44 complete cycles in a 0.1-second window. The wave should smoothly oscillate between -amplitude and +amplitude.

Exercise 2: Build a WAV File Writer

Concept

Understanding file formats at the byte level is fundamental to audio programming. This exercise teaches you binary file I/O and data representation. You'll write raw bytes in the exact format that audio software expects.

Goal

Write a function that takes a NumPy array of audio samples (floating-point, -1.0 to 1.0) and saves it as a valid 16-bit mono WAV file that any audio player can open.

Task Description

Implement a WAV writer with these specifications:

Input Parameters	
`filename`	Path to the output .wav file
`samples`	NumPy array of floats from -1.0 to 1.0
`sample_rate`	Samples per second (default: 44100)
Requirements	
Format	16-bit PCM, mono, little-endian
Clipping	Values outside [-1, 1] should be clipped

Algorithm Steps

1. Open file in binary write mode ('wb')

2. Write RIFF header: 'RIFF' + file_size + 'WAVE'

3. Write fmt chunk with audio parameters

4. Write data chunk header: 'data' + data_size

5. Convert float samples to 16-bit integers: int_sample = float_sample × 32767

6. Write each sample as 2 bytes (little-endian)

Hints

- *Use Python's struct module: struct.pack('<I', value) for 4-byte little-endian unsigned int*
- *Use struct.pack('<h', value) for 2-byte little-endian signed int (for samples)*
- *Clip samples: numpy.clip(samples, -1.0, 1.0) before conversion*
- *The file size in the header is total file size minus 8 bytes*
- *Convert array to int16: (samples * 32767).astype(numpy.int16)*

Verification

Use your sine wave function from Exercise 1 to generate a 440 Hz tone, then save it with your WAV writer. Open the file in any audio player—you should hear a clean A4 note. Also try opening in a hex editor to verify the header structure matches the specification.

Part 2: Basic Waveforms

Beyond the sine wave, synthesizers use several classic waveforms. Each has a distinct timbre (tonal quality) because of its harmonic content. Understanding these waveforms is essential for subtractive synthesis—the technique used in most analog synthesizers.

Exercise 3: Square Wave from Harmonics

Concept

A square wave alternates between two values (high and low) with instant transitions. It has a hollow, woody sound and contains only odd harmonics (1f, 3f, 5f, 7f...). The amplitude of each harmonic is 1/n where n is the harmonic number.

Fourier series for square wave: $y(t) = (4/\pi) \times [\sin(2\pi ft) + \sin(6\pi ft)/3 + \sin(10\pi ft)/5 + ...]$

Goal

Build a square wave by adding sine waves together (additive synthesis). This demonstrates that complex waves are sums of simple sines and introduces you to harmonic series.

Task Description

Create two functions:

1. **Naive square wave:** Simply output +amplitude when $\sin(2\pi ft) > 0$, else -amplitude

2. **Band-limited square wave:** Sum the first N odd harmonics using the Fourier series

Hints

- *Start with num_harmonics=10 and increase to hear the difference*
- *The naive approach causes aliasing (unwanted high-frequency artifacts)*
- *Odd harmonics only: 1, 3, 5, 7... (use range(1, 2*num_harmonics, 2))*
- *Listen to both versions—the naive one sounds harsher/buzzier*

Verification

Plot both waveforms. The band-limited version should have slight ripples at the transitions (Gibbs phenomenon) while the naive version has perfectly sharp edges. Compare the sound.

Exercise 4: Sawtooth Wave

Concept

The sawtooth wave ramps up (or down) linearly then resets instantly. It's the richest-sounding classic waveform because it contains ALL harmonics (both odd and even). This makes it bright and buzzy—perfect for brass sounds, leads, and bass.

Fourier series: $y(t) = (2/\pi) \times [\sin(2\pi ft) - \sin(4\pi ft)/2 + \sin(6\pi ft)/3 - \sin(8\pi ft)/4 + ...]$

Goal

Generate a sawtooth wave using both the naive (direct ramp) and band-limited (Fourier series) methods.

Task Description

Implement:

1. **Naive sawtooth:** For each period, linearly ramp from -1 to +1. Use modulo arithmetic: saw(t) = 2 × (t × frequency mod 1) - 1

2. **Band-limited sawtooth:** Sum harmonics with alternating signs and decreasing amplitudes

Hints

- *For naive version: (2 * ((t * frequency) % 1)) - 1*
- *Harmonics: 1, 2, 3, 4, 5... with amplitudes: 1, 1/2, 1/3, 1/4...*
- *Alternating sign: (-1)^(n+1) gives +, -, +, -, ...*
- *Compare the brightness of sawtooth vs square wave—can you hear more harmonics?*

Exercise 5: Triangle Wave

Concept

The triangle wave ramps up and down linearly without flat segments. Like the square wave, it contains only odd harmonics, but they roll off much faster (amplitude is $1/n^2$). This gives it a softer, more mellow sound—often described as flute-like.

Goal

Generate a triangle wave. This is often the most challenging basic waveform to implement efficiently.

Task Description

Two approaches:

1. **Direct calculation:** Use abs(sawtooth(t)) × 2 - 1 (creates triangle from sawtooth)

2. **Integration:** Triangle is the integral of square wave—implement numeric integration

Hints

- *Simplest: triangle(t) = 2 × abs(2 × ((t × freq + 0.25) % 1) - 1) - 1*
- *Phase shift of 0.25 (quarter period) aligns the peaks nicely*
- *For Fourier series: only odd harmonics, amplitude = $(8/\pi^2) \times (1/n^2)$, alternating signs*

Part 3: Amplitude & Envelopes

Exercise 6: ADSR Envelope

Concept

Real sounds don't just turn on and off—they evolve over time. The ADSR envelope is the classic model for this evolution: Attack (how fast sound reaches peak), Decay (drop to sustain level), Sustain (level held while key is pressed), Release (fade after key release).

Goal

Create a function that generates an envelope curve, then apply it to a waveform by multiplying.

Task Description

Parameters (all in seconds except sustain_level):

Parameter	Description	Typical Value
`attack`	Time to reach peak	0.01 - 0.5s
`decay`	Time to fall to sustain	0.1 - 0.5s
`sustain_level`	Amplitude during sustain (0-1)	0.5 - 0.8
`release`	Time to fade to zero	0.1 - 1.0s
`note_duration`	Total time key is held	varies

Hints

- *Create arrays for each segment and concatenate: numpy.concatenate([attack, decay, sustain, release])*
- *Attack: numpy.linspace(0, 1, attack_samples)*
- *Decay: numpy.linspace(1, sustain_level, decay_samples)*
- *For musical sounds, try attack=0.01, decay=0.1, sustain=0.7, release=0.3*

Exercise 7: Tremolo Effect

Concept

Tremolo is a periodic variation in amplitude—the volume wobbles up and down. It's created by multiplying the audio signal by a low-frequency oscillator (LFO). This is different from vibrato, which modulates pitch.

Goal

Implement a tremolo effect using an LFO (low-frequency oscillator) to modulate amplitude.

Task Description

Create a function that applies tremolo to any input signal. Parameters: LFO rate (Hz), depth (0-1 where 1 means full volume swing from 0 to 1).

Formula: output = input × (1 - depth/2 + (depth/2) × sin(2π × rate × t))

Hints

- *Typical LFO rates: 2-10 Hz (faster feels frantic, slower is more subtle)*
- *The formula keeps amplitude between (1-depth) and 1.0*
- *Try applying to a sustained organ-like tone (sine wave with long ADSR)*

Part 4: Filters

Filters are the heart of subtractive synthesis. They shape the harmonic content of sounds by amplifying or attenuating certain frequencies. This section introduces the simplest digital filters that you can implement with just a few lines of code.

Exercise 8: Moving Average (Low-Pass)

Concept

The simplest low-pass filter: average the current sample with the previous N samples. This smooths out rapid changes (high frequencies) while letting slow changes (low frequencies) through. It's the digital equivalent of RC circuit smoothing.

Goal

Implement a moving average filter and observe how it affects different waveforms.

Task Description

For each output sample: y[n] = (x[n] + x[n-1] + x[n-2] + ... + x[n-N+1]) / N

Where N is the window size (number of samples to average).

Hints

- *Use numpy.convolve(signal, numpy.ones(N)/N, mode='same') for efficiency*
- *Or implement sample-by-sample with a for loop to understand the process*
- *Apply to a square wave—watch the sharp edges become rounded*
- *Larger N = more smoothing = lower cutoff frequency*

Verification

Apply to a sawtooth wave with N=10, 50, 100. Plot before and after. Listen to how the bright buzz becomes progressively duller. You're removing harmonics!

Exercise 9: First-Order Low-Pass Filter

Concept

The exponential moving average (EMA) or first-order IIR low-pass filter uses feedback: each output depends on the previous output. This creates a more musical roll-off than the simple moving average and is computationally cheaper.

The formula is: $y[n] = \alpha \times x[n] + (1 - \alpha) \times y[n-1]$

Where α (alpha) controls the cutoff frequency: smaller α = lower cutoff = more filtering.

Goal

Implement this filter and explore how α affects the sound.

Task Description

Process the signal sample-by-sample:

1. Initialize y_prev = 0

2. For each sample x[n]: $y[n] = \alpha \times x[n] + (1 - \alpha) \times y\_prev$

3. Update y_prev = y[n]

Computing α from cutoff frequency:

$\alpha = 1 - e^{(-2\pi \times cutoff\_freq / sample\_rate)}$

Hints

- *Try cutoff frequencies of 500 Hz, 1000 Hz, 5000 Hz on a sawtooth wave*
- *This is an IIR (Infinite Impulse Response) filter—it has memory*
- *At $\alpha = 1$, no filtering occurs (output = input)*
- *At $\alpha = 0$, output is stuck at initial value (complete filtering)*

Exercise 10: First-Order High-Pass Filter

Concept

A high-pass filter removes low frequencies while preserving highs—the opposite of low-pass. It's essential for removing unwanted rumble or creating thin, cutting sounds. Mathematically, it's the complement of a low-pass filter.

Goal

Derive and implement a high-pass filter from the low-pass filter you already built.

Task Description

Key insight: High-pass = Original signal - Low-passed signal

Alternative direct formula: $y[n] = \alpha \times (y[n-1] + x[n] - x[n-1])$

Hints

- *The easiest approach: run your low-pass filter, then subtract from original*
- *For the direct formula, α is calculated the same way as for low-pass*
- *Test on a mix of low sine (100 Hz) + high sine (2000 Hz)—the HPF should remove the low one*

Part 5: Delay & Modulation

Exercise 11: Simple Delay/Echo

Concept

Delay effects store audio in a buffer and play it back later, mixing it with the original. A single delay creates an echo. Multiple delays create more complex effects. Adding feedback (sending output back to input) creates repeating echoes.

Goal

Implement a delay line with controllable delay time, feedback, and wet/dry mix.

Task Description

Parameter	Description
`delay_time`	Delay in seconds (e.g., 0.25 for quarter-second echo)
`feedback`	0-1, how much delayed signal feeds back (0.5 = half volume each repeat)
`wet_dry`	0-1, mix of effect (wet) vs original (dry)

Algorithm

1. Create a delay buffer of size = delay_time × sample_rate

2. For each sample: read from buffer, write input + feedback × buffer_value to buffer

3. Output = dry × input + wet × buffer_output

Hints

- *Use a circular buffer (modulo indexing) for efficiency*
- *Start without feedback (feedback=0) to test basic delay*
- *Keep feedback below 1.0 or the echoes will grow infinitely loud!*

Exercise 12: Ring Modulation

Concept

Ring modulation multiplies two signals together. Unlike tremolo (which uses a unipolar LFO), ring mod uses a full bipolar signal, creating sum and difference frequencies. This produces the classic "robot voice" or metallic bell sounds.

When you multiply sin(A) × sin(B), you get frequencies at (A+B) and (A-B), not A or B!

Goal

Implement ring modulation and explore how carrier frequency affects the output timbre.

Task Description

output[n] = input[n] × carrier[n], where carrier is typically a sine wave.

Experiment with different carrier frequencies: 100 Hz, 440 Hz, 1000 Hz.

Hints

- *Use your sine wave generator to create the carrier*
- *Try ring modulating a vocal recording for the robot voice effect*
- *Ring mod with a harmonic frequency (e.g., 2× the fundamental) creates octaves*

Part 6: Frequency Domain

Exercise 13: Discrete Fourier Transform

Concept

The DFT transforms a time-domain signal into frequency-domain representation. It answers: "What frequencies are present in this signal, and how strong is each one?" This is the foundation of spectral analysis, EQ, and many DSP algorithms.

Formula: $X[k] = \Sigma(n=0 \text{ to } N-1)\ x[n] \times e^{(-j2\pi kn/N)}$

Where X[k] is the complex amplitude at frequency bin k, and N is the number of samples.

Goal

Implement the DFT from scratch (not using FFT) to understand the math. Then compare with numpy.fft.fft().

Task Description

1. Implement the naive $O(N^2)$ DFT using the formula above

2. Return magnitude (absolute value) and phase (angle) of each bin

3. Create a function to convert bin index to frequency in Hz

Hints

- *Use numpy.exp() for e^(complex) calculations*
- *j = 1j in Python (the imaginary unit)*
- *Frequency of bin k = k × sample_rate / N*
- *Only the first N/2 bins are useful (the rest are mirror images for real signals)*
- *This is slow! Use numpy.fft.fft() for real work, but implementing it teaches the concept*

Exercise 14: Spectrum Analyzer

Concept

A spectrum analyzer visualizes the frequency content of audio in real-time (or near real-time). You've seen these as the bouncing bars in music visualizers. This exercise brings together everything: reading audio, computing FFT, and displaying results.

Goal

Build a simple spectrum analyzer that reads a WAV file and plots its frequency content over time.

Task Description

1. Read a WAV file (use scipy.io.wavfile or your own reader)

2. Divide into overlapping chunks (e.g., 1024 samples, 50% overlap)

3. Apply a window function (Hanning) to each chunk

4. Compute FFT of each chunk

5. Convert magnitude to decibels: dB = 20 × log10(magnitude)

6. Plot as a spectrogram using matplotlib's imshow or pcolormesh

Hints

- *Windowing prevents spectral leakage: numpy.hanning(chunk_size)*
- *matplotlib.pyplot.specgram() does steps 2-6 automatically—use it to verify your implementation*
- *Use a logarithmic frequency axis for a more musical view*
- *Test with your generated waveforms—square wave should show only odd harmonics!*

Appendix A: WAV Format Reference

Complete Header Structure (44 bytes)

This is the minimum header for a standard 16-bit PCM WAV file. Copy this reference for Exercise 2.

Bytes	Value	Field	Notes
0-3	52 49 46 46	ChunkID	"RIFF" in ASCII
4-7	(file size - 8)	ChunkSize	Little-endian 32-bit
8-11	57 41 56 45	Format	"WAVE" in ASCII
12-15	66 6D 74 20	Subchunk1ID	"fmt " (with space)
16-19	10 00 00 00	Subchunk1Size	16 for PCM
20-21	01 00	AudioFormat	1 = PCM
22-23	01 00	NumChannels	1 = mono, 2 = stereo
24-27	44 AC 00 00	SampleRate	44100 in little-endian
28-31	88 58 01 00	ByteRate	SampleRate × BlockAlign
32-33	02 00	BlockAlign	NumChannels × BitsPerSample/8
34-35	10 00	BitsPerSample	16 bits
36-39	64 61 74 61	Subchunk2ID	"data" in ASCII
40-43	(data size)	Subchunk2Size	NumSamples × BlockAlign
44+	...	Data	Raw audio samples

Sample Representation

16-bit samples are signed integers: -32768 to +32767. In memory:

Float Value	Integer Value	Bytes (Little-Endian)
+1.0 (max positive)	+32767	FF 7F
0.0 (silence)	0	00 00
-1.0 (max negative)	-32768	00 80
+0.5	+16383	FF 3F
-0.5	-16384	00 C0

Appendix B: Offline Setup Guide

Prepare this environment BEFORE your flight or trip to patchy internet areas.

Required Python Packages

Install while you have internet:

```
pip install numpy scipy matplotlib
```

Download Offline Documentation

Download these for offline reference (search for 'download offline docs'):

• NumPy documentation (PDF available)

• Matplotlib documentation (downloadable)

• Python official tutorial

Test Your Setup

Run this before leaving internet:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile

# Test 1: Generate sine wave
t = np.linspace(0, 1, 44100)
wave = np.sin(2 * np.pi * 440 * t)
print("Sine wave shape:", wave.shape)

# Test 2: Plot works
plt.figure()
plt.plot(t[:1000], wave[:1000])
plt.title("If you see this, matplotlib works!")
plt.savefig("test_plot.png")
plt.close()
print("Plot saved successfully")

# Test 3: WAV writing works
wave_int = (wave * 32767).astype(np.int16)
wavfile.write("test.wav", 44100, wave_int)
print("WAV file created successfully")

print("\nAll tests passed! You're ready for offline coding.")
```

Suggested Project Structure

```
audio_dsp_learning/
■■■ exercises/
■   ■■■ ex01_sine.py
■   ■■■ ex02_wav_writer.py
■   ■■■ ex03_square.py
■   ■■■ ...
■■■ outputs/
```

```
■   ■■■ test_sine.wav
■   ■■■ ...
■■■ plots/
■   ■■■ ...
■■■ utils/
    ■■■ wav_utils.py
    ■■■ dsp_utils.py
```

Appendix C: Quick Reference Card

Essential NumPy for Audio DSP

Operation	Code
Create time array	`t = np.linspace(0, duration, num_samples, endpoint=False)`
Sine wave	`np.sin(2 * np.pi * freq * t)`
Clip to range	`np.clip(signal, -1.0, 1.0)`
Convert to int16	`(signal * 32767).astype(np.int16)`
Concatenate arrays	`np.concatenate([arr1, arr2, arr3])`
Element-wise multiply	`signal1 * signal2`
FFT	`np.fft.fft(signal)`
FFT frequencies	`np.fft.fftfreq(n, 1/sample_rate)`
Magnitude	`np.abs(fft_result)`
Phase	`np.angle(fft_result)`
Hanning window	`np.hanning(window_size)`
Convolve (filter)	`np.convolve(signal, kernel, mode='same')`

Common Frequencies (Musical Notes)

Note	Frequency (Hz)	Note	Frequency (Hz)
C4 (Middle C)	261.63	A4 (Tuning)	440.00
D4	293.66	B4	493.88
E4	329.63	C5	523.25
F4	349.23	D5	587.33
G4	392.00	E5	659.25

Decibel Conversions

Linear to dB: dB = 20 × log■■(amplitude)
dB to Linear: amplitude = 10^(dB/20)

dB	Amplitude	Perception
0 dB	1.0	Reference level
-6 dB	0.5	Half amplitude
-12 dB	0.25	Quarter amplitude
-20 dB	0.1	One-tenth
-40 dB	0.01	Very quiet

-60 dB	0.001	Nearly silent

Happy coding! Remember: the best way to learn DSP is to listen to your code. Generate sounds, modify them, and trust your ears. ■