NumPy as Synth Engine

March 2026

11 min read • 2,365 words

There are zero audio files in PyTheory. Zero samples. Zero recordings. Zero WAVs, MP3s, OGGs, or any other format you could name. Not one byte of pre-recorded sound exists anywhere in the repository.

Every sound you hear — every plucked guitar string, every tabla stroke, every djembe slap, every organ chord — is computed at runtime. From math. Sine waves, noise generators, filters, and envelopes, all the way down. NumPy arrays are the synth engine.

And somehow, the results sound... real?

I still can't quite believe this works. I'm going to walk you through it, not as an expert in DSP (I am definitely not that), but as someone who kept adding features to a music theory library and accidentally ended up building physical models of goatskin membranes in Python.

Karplus-Strong: The Oldest Trick

The algorithm that started everything was invented in 1983 by Kevin Karplus and Alex Strong. It generates plucked string sounds, and it is almost offensively simple.

Here's the entire idea:

1. Fill a buffer with random noise. Make the buffer exactly one pitch period long (for 440 Hz at 44,100 samples/second, that's 100 samples).
2. Loop through the buffer. For each sample, average it with the next sample.
3. That's it. That's the algorithm.

```
import numpy

SAMPLE_RATE = 44_100

def plucked_string(hz, n_samples=SAMPLE_RATE):
    """Karplus-Strong: a buffer of noise that averages itself into music."""
    period = int(SAMPLE_RATE / hz)
    buf = numpy.random.uniform(-1.0, 1.0, period)

    out = numpy.zeros(n_samples)
    for i in range(n_samples):
        out[i] = buf[i % period]
        next_idx = (i + 1) % period
        buf[i % period] = 0.5 * (buf[i % period] + buf[next_idx]) * 0.999

    return out
```

The averaging acts as a lowpass filter, gradually removing high-frequency harmonics from the noise. This is exactly what happens to a real vibrating string — it starts with a bright, complex attack and then the higher harmonics die off faster than the lower ones, leaving a warm, decaying tone. The `0.999` decay factor controls how long the string rings. Higher means more sustain. Lower means more damping.

That's a plucked string. From a loop and an averaging operation.

But here's where it gets really fun. The same algorithm, with different parameters, produces completely different instruments.

Acoustic guitar adds body resonance. A real guitar body has resonant frequencies — the air cavity vibrates around 110 Hz, the top plate around 250 Hz, the back around 500 Hz. So we run the Karplus-Strong output through three bandpass filters at those frequencies and mix them back in:

```
# Body resonance — three formant peaks modeling the guitar body
for center, bw, gain in [(110, 60, 0.4), (250, 80, 0.3), (500, 120, 0.2)]:
    bp, ap = scipy.signal.butter(2, [center - bw, center + bw],
                                 btype="band", fs=SAMPLE_RATE)
    resonances += scipy.signal.lfilter(bp, ap, out) * gain
```

Electric guitar takes a different approach. Instead of body resonance, it simulates a magnetic pickup. A pickup at one-quarter of the string length cancels the 4th harmonic and boosts the 2nd, creating that characteristic electric guitar midrange honk. This is modeled as a comb filter — subtract a delayed copy of the signal from itself:

```
# Magnetic pickup at 1/4 string length — cancels 4th harmonic
pickup_pos = period // 4
pickup = numpy.zeros(n_samples)
pickup[pickup_pos:] = out[:-pickup_pos]
out = out - pickup * 0.3
```

Ukulele has a smaller body (resonances at 350, 700, and 1200 Hz instead of the guitar's lower frequencies) and uses softer initial noise to simulate nylon strings instead of steel. Same core algorithm. Completely different instrument.

I find this genuinely beautiful. The string physics are universal — noise into a delay line with averaging. What makes an acoustic guitar sound different from an electric guitar sound different from a ukulele is entirely about what happens around the string: the body, the pickup, the material. And all of that is captured in a handful of filter parameters.

The Tabla: Where I Lost My Mind

The Karplus-Strong stuff was cool but still felt like a known technique. The tabla is where things got wild.

A tabla is actually two drums — the dayan (a small wooden drum played with the right hand) and the bayan (a larger copper drum played with the left). Each drum produces multiple distinct sounds depending on where and how you strike it. The syahi, a circular patch of black paste on the drumhead, is what gives the tabla its characteristic ringing harmonics. These are sounds that have been refined over centuries by Indian classical musicians, and they are nothing like Western percussion.

I modeled six strokes. Each one is built from multiple physical components layered together.

Na — a sharp rim strike on the dayan. This layers four distinct physical elements: a goatskin membrane thump (random noise bandpass-filtered between 200 and 800 Hz), a wooden shell resonance (a sine wave at 800 Hz with fast decay), a syahi ring (three harmonics at 330, 680, and 1050 Hz, each with independent decay rates), and a finger attack transient (a burst of noise that dies in milliseconds):

```
def _synth_tabla_na(n_samples):
    t = numpy.arange(n_samples, dtype=numpy.float32) / SAMPLE_RATE

    # Goatskin membrane thump — bandpass filtered noise
    thump_raw = noise(int(SAMPLE_RATE * 0.05))
    bl, al = scipy.signal.butter(2, [200, 800], btype="band", fs=SAMPLE_RATE)
    thump = scipy.signal.lfilter(bl, al, thump_raw)
    thump *= exp_decay(len(thump), 40) * 0.8

    # Wooden shell resonance
    wood = numpy.sin(2 * numpy.pi * 800 * t[:len(thump)]) * exp_decay(len(thump), 50) *
    thump += wood

    # Syahi ring — three harmonics with independent decay
    ring = numpy.sin(2 * numpy.pi * 330 * t) * exp_decay(n_samples, 9) * 0.6
    ring2 = numpy.sin(2 * numpy.pi * 680 * t) * exp_decay(n_samples, 12) * 0.3
    ring3 = numpy.sin(2 * numpy.pi * 1050 * t) * exp_decay(n_samples, 16) * 0.15

    # Sharp finger attack
    click = noise(100) * exp_decay(100, 250) * 0.7

    # Layer everything
    result = ring + ring2 + ring3
    result[:len(thump)] += thump
    result[:100] += click
    return numpy.tanh(result * 1.4)
```

Tin — the open ring. Fuller membrane, longer singing ring, more sustain. The same architecture as Na but with different filter ranges and slower decay rates, because the full open stroke lets the membrane vibrate freely.

Ge — the deep bayan. This is where I started laughing at my own code. Five physical components: a goatskin membrane thud (bandpass noise at 40-250 Hz, much lower than the dayan), a copper shell resonance (120 Hz sine), a pitch-sweeping body that starts at 55 Hz and exponentially rises to 155 Hz (because the hand pressing the membrane changes the effective pitch — a technique unique to tabla), a sub-frequency boom at 40 Hz from the large cavity, and a palm attack transient.

```
# Pitch sweep body — hand modulates the membrane
freq = 55 + 100 * numpy.exp(-10 * t)
phase = 2 * numpy.pi * numpy.cumsum(freq) / SAMPLE_RATE
body = numpy.sin(phase) * exp_decay(n_samples, 5) * 0.7
```

That exponential pitch sweep is such a small piece of code for what it represents — the physical interaction between a musician's palm and a goatskin membrane stretched over a copper shell. Three lines.

Dha — both drums simultaneously. Na and Ge summed together and clipped through `tanh`. The simplest stroke in concept, the most complex in output.

Tit — the rapid-fire finger tap. Sixty milliseconds max. Mostly attack, barely any ring. For the fast rhythmic patterns — tiri-kita, taka-dina — where each stroke is a percussive whisper.

Ke — the muted bayan slap. Dead thud, no ring, eighty milliseconds. Just the palm killing the membrane immediately after striking it.

Here's what gets me: each stroke models the actual physics. The goatskin membrane is bandpass-filtered noise because that's what a vibrating membrane sounds like — broadband excitation filtered by the membrane's own resonant properties. The wooden shell versus the copper shell is just different resonance frequencies. The syahi creates those characteristic harmonic overtones because the added mass of the black paste changes the membrane's vibrational modes, producing specific harmonic ratios. These aren't approximations of recordings — they're approximations of physics.

The Djembe and Cross-Choking

The djembe gets three strokes: bass (center palm hit with goblet body resonance at 65 Hz and a sub at 45 Hz), tone (edge strike with clear ring at 250 and 500 Hz), and slap (spread fingers producing a sharp pop at 900, 1600, and 2400 Hz with a high-pass transient).

But the really cool part is the cross-choke system. When you hit a djembe slap, it automatically dampens any ringing bass or tone. Because that's what happens on a real djembe — your hand is on the skin, so any previous vibration gets killed when the new strike lands.

```
_CHOKE_GROUPS = {
    # Djembe — any strike dampens the others
    DJEMBE_BASS: (DJEMBE_TONE, DJEMBE_SLAP),
    DJEMBE_TONE: (DJEMBE_BASS, DJEMBE_SLAP),
    DJEMBE_SLAP: (DJEMBE_BASS, DJEMBE_TONE),
    # Hi-hats — closed chokes open
    CLOSED_HAT: (OPEN_HAT,),
    PEDAL_HAT: (OPEN_HAT,),
    # Cajon — slap dampens bass ring
    CAJON_SLAP: (CAJON_BASS,),
    # Doumbek — tek dampens dum
    DOUMBEK_TEK: (DOUMBEK_DUM,),
    DOUMBEK_KA: (DOUMBEK_DUM,),
}
```

The implementation is almost comically simple. When a new hit lands on a choke target, apply a 2-4 millisecond fade-out to whatever was ringing before it:

```
fade_len = min(int(SAMPLE_RATE * 0.004), start - prev_start)
fade = numpy.linspace(1.0, 0.0, fade_len)
part_stereo[start - fade_len : start] *= fade
```

A linear fade of a few milliseconds. That's what makes the difference between "synthesized drum sounds playing simultaneously" and "someone actually playing a drum." Without cross-choking, a fast bass-slap-bass pattern on a djembe sounds like three separate events. With it, each new hit silences the last, exactly the way a physical instrument behaves. Same for hi-hats — a closed hat chokes an open hat because your foot is pressing the cymbals together. Same physics, same simple fade.

The Hammond Organ

After all that physical modeling complexity, the Hammond organ is almost a palate cleanser. A real Hammond B3 has nine drawbars, each adding a sine wave at a specific harmonic. Pull them all out and you get that warm, round, unmistakably organ sound.

The entire implementation:

```
def hammond_wave(hz, peak, n_samples):
    t = numpy.arange(n_samples, dtype=numpy.float64) / SAMPLE_RATE

    wave = (
        numpy.sin(2 * numpy.pi * hz * t) * 1.0          # 16' fundamental
        + numpy.sin(2 * numpy.pi * hz * 2 * t) * 0.8    # 8'
        + numpy.sin(2 * numpy.pi * hz * 3 * t) * 0.6    # 5 1/3'
        + numpy.sin(2 * numpy.pi * hz * 4 * t) * 0.5    # 4'
        + numpy.sin(2 * numpy.pi * hz * 5 * t) * 0.3    # 2 2/3'
        + numpy.sin(2 * numpy.pi * hz * 6 * t) * 0.25   # 2'
        + numpy.sin(2 * numpy.pi * hz * 8 * t) * 0.15   # 1 3/5'
    )

    wave /= 3.5
    return (peak * wave).astype(numpy.int16)
```

Seven sine waves summed together. That's a Hammond organ. The specific ratios of the harmonics — which ones are louder, which are softer — determine the character. The `1.0, 0.8, 0.6, 0.5, 0.3, 0.25, 0.15` weighting curve is what makes it sound warm and round instead of bright and buzzy. Change those numbers and you get a completely different registration, from gospel to jazz to rock.

Additive synthesis is the simplest form of sound design. You're literally just adding waves together. And yet the result is immediately, unmistakably recognizable. Play a chord through that function and anyone over the age of thirty will say "that's an organ."

The Absurdity

Let me zoom out for a moment.

All of this is Python. Not C, not C++, not Rust, not some specialized DSP framework. Python. The same language people use to parse CSV files and train machine learning models and build web scrapers.

NumPy is the synth engine. `scipy.signal.butter` is the filter designer. `scipy.signal.lfilter` is the filter. `numpy.sin` is the oscillator. `numpy.random.uniform` is the noise generator. `numpy.tanh` is the soft clipper. These are general-purpose numerical computing tools being used to physically model instruments from around the world.

The tabla code is approximating the interaction between a human palm and a goatskin membrane stretched over a copper shell, using the same library that people use for statistical analysis. The Karplus-Strong algorithm is simulating steel strings vibrating over magnetic pickups using the same array operations you'd use to clean a spreadsheet.

I find this genuinely hilarious. And deeply satisfying. And a little bit awe-inspiring, if I'm honest.

There's a version of this project where I'd use a C extension for the DSP, or pull in a proper audio synthesis framework, or just use sample libraries like every other sensible person. It would probably sound better. It would certainly be faster. But it wouldn't have this quality that I keep coming back to — the fact that you can read the code and see the physics. The bandpass filter from 200 to 800 Hz is the goatskin membrane. The exponential pitch sweep is the hand pressing the drumhead. The comb filter is the magnetic pickup. The code doesn't just produce the sound; it explains why the sound exists.

Historical Tunings

Here's something you can't do with sample libraries: play a Karplus-Strong guitar string in Pythagorean temperament. Or a tabla in meantone. Or a Hammond organ tuned to just intonation.

Soundfonts are recordings of real instruments at specific pitches — locked to equal temperament, the modern Western tuning standard. If you want to hear what a harpsichord sounded like in 1685 (meantone) or how a raga sounds in its proper intonation (not the 12-TET approximation), you're out of luck. The samples were recorded at fixed frequencies. You can pitch-shift them, but that distorts the timbre.

When everything is computed from math, the frequency is just a parameter. Change it and the physics still work. A Karplus-Strong string at 440 Hz and a Karplus-Strong string at 438.07 Hz (Pythagorean A4) use the same algorithm — the delay line just gets slightly longer. The body resonance still works. The pickup simulation still works. The envelope still works. You're not stretching a recording; you're generating a new one.

This is one of those accidental advantages of doing synthesis from scratch. I didn't plan it. But now I can hear what a Renaissance lute actually sounded like, tuned the way its player would have tuned it — and that's really fun.

I've written about why PyTheory matters to me and how it grew into a mini DAW. The synthesis layer is really fun.
