Day 1 · Audio feature explorer: from sound to the features every speech model eats

Goal in one line: understand what a speech file *is* (numbers + a sample rate) and turn it, step by step, into the log-mel spectrogram that Whisper, ECAPA-TDNN and most voice-conversion models actually receive.

What you will build: `describe(path)`, a feature-explorer function that prints the key numbers of any clip (duration, peak, dBFS, median pitch, high-frequency energy, log-mel shape) and draws a 3-row figure (waveform, spectrogram with pitch overlay, log-mel). You run it over 20 Khmer clips and compare phone vs studio and male vs female.

Time: ~3 h

By the end you can ...

- load any audio file as 16 kHz mono float32 in [-1, 1] and say what each of those words means
- read a waveform and a spectrogram the way you read a line chart
- explain sampling, Nyquist, aliasing, quantization and dBFS with real numbers
- compute an STFT, a mel filterbank, a log-mel spectrogram and MFCCs by hand, and match the librosa versions
- pull out pitch (F0) and formants (F1/F2) and say why male and female voices look different

How to read this notebook (for the team)

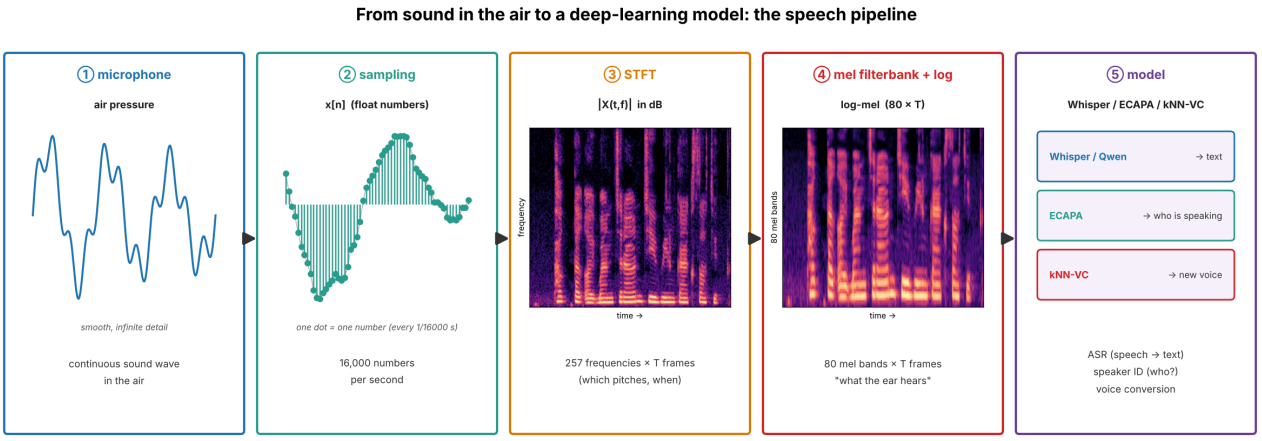
You see	What it is	Do you need it?
Purple heading	a new section and its explanation	**yes**: read the bold lines and the "Why it matters" line
Visualization Version	one picture that explains the whole section	**yes**: the first image of each section is the fastest way in
Additional reading	what the next code cell does and what its output means	optional: read just the **In short** lines if you are busy
code cell + plot	the real computation on our Khmer clips	run it, look at the plot

Fast path (≈ 30 min): for each section read the purple text, look at the Visualization Version image, then read only the "In short" lines. **Full path (≈ 3 h):** read everything and run every cell.

The big idea

```
microphone        sampling        STFT          mel filterbank + log        model
air pressure  ->  x[n] float  ->  |X(t,f)| dB ->  log-mel (80 x T)      ->  Whisper / ECAPA / kNN-VC
(continuous)      16000 per s     257 x T         "what the ear hears"       (ASR, speaker ID, voice
conversion)
```

Visualization Version



- A speech file is just a long list of numbers: how far the microphone membrane moved, measured 16 000 times per second. That list is honest but useless to look at. Speech lives in *frequencies*: which pitches are vibrating at each moment. So every speech model first chops the signal into short frames (25 ms), measures the energy at each frequency in each frame (the STFT), squeezes the frequency axis the way the ear does (mel), and takes a log so quiet and loud parts both become visible. The result is an image, 80 rows by T columns, and from then on speech is a computer-vision problem.

Additional reading

Today is about looking at that pipeline stage by stage on real Khmer speech, so that tomorrow when a model mis-hears a clip you can open the log-mel and *see* why.

Good to read

In our research. The Khmer speaker-anonymization project replaces a speaker's voice while keeping the words. Everything we will anonymize or evaluate flows through these features: ECAPA-TDNN (the attacker that tries to recognize the speaker) eats 80-bin log-mel; the ASR that checks the words are preserved eats log-mel; the McAdams anonymizer works on the LPC envelope from section 9; and pitch (F0) is both a strong speaker cue and a thing we must deliberately change. If you can read a spectrogram, you can read what an anonymizer did to a voice.

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short load the libraries, set the course constants and point to the 3 example clips.

- **Libraries:** numpy, pandas, soundfile (reads WAV files), librosa (audio features), scipy (signal tools), matplotlib (plots).

- **Helpers from `src/utils.py`:** `load` (read a clip as 16 kHz float), `show_wave`, `show_spec`, `play`, `f0_track`, `median_f0`, `rms_dbfs`, `stft_db`, `logmel`.
- **Constants used everywhere:** `SR = 16000` (samples per second), `N_FFT, HOP, WIN = 512, 160, 400` (25 ms window every 10 ms, explained in section 6).
- **`CLIPS`:** the 3 example recordings: studio female, phone female, phone male.

What the output means

In short it prints the library versions. If this cell fails, nothing below will run.

`librosa 1.0.0 | numpy 2.x`: the exact numbers don't matter, only that the imports worked.

```
import os, sys, warnings, io
os.environ.setdefault("OMP_NUM_THREADS", "4")   # keep BLAS from grabbing every core (set BEFORE numpy)
warnings.filterwarnings("ignore")
sys.path.insert(0, "..")                         # so `from src.utils import ...` works from notebooks/

from pathlib import Path
import numpy as np
import pandas as pd
import soundfile as sf
import scipy.signal, scipy.fft
import librosa, librosa.display
import matplotlib.pyplot as plt
from IPython.display import display

from src.utils import play, show_wave, show_spec, median_f0, f0_track, rms_dbfs, stft_db, logmel, load

np.random.seed(0)
SR = 16000                          # every clip in this course is 16 kHz mono
N_FFT, HOP, WIN = 512, 160, 400   # 25 ms Hann window every 10 ms (explained in section 6)
DATA = Path("../data")
OUT = Path("../outputs/day1"); OUT.mkdir(parents=True, exist_ok=True)

CLIPS = {"studio (female)": DATA / "audio/khmer_studio.wav",
         "phone (female)":  DATA / "audio/khmer_phone_female.wav",
         "phone (male)":    DATA / "audio/khmer_phone_male.wav"}
print("librosa", librosa.__version__, "| numpy", np.__version__)
```

```
librosa 1.0.0 | numpy 2.5.3
```

1. Audio is numbers + a sample rate

A microphone turns air pressure into a voltage. A sound card measures that voltage `fs` times per second (`fs` = *sample rate*, in Hz) and stores each measurement as a number. That is all a WAV file is: a header (sample rate, channels, bits) followed by the numbers.

Worked example. `khmer_studio.wav` is 3.15 s at 16 000 Hz, so it holds 3.15 × 16 000 = 50 475 numbers. Stored as 16-bit integers each number lies in [-32768, 32767]. Dividing by 32 768 maps them to the float range [-1, 1], which is the convention every library uses (`librosa.load` does this for you):

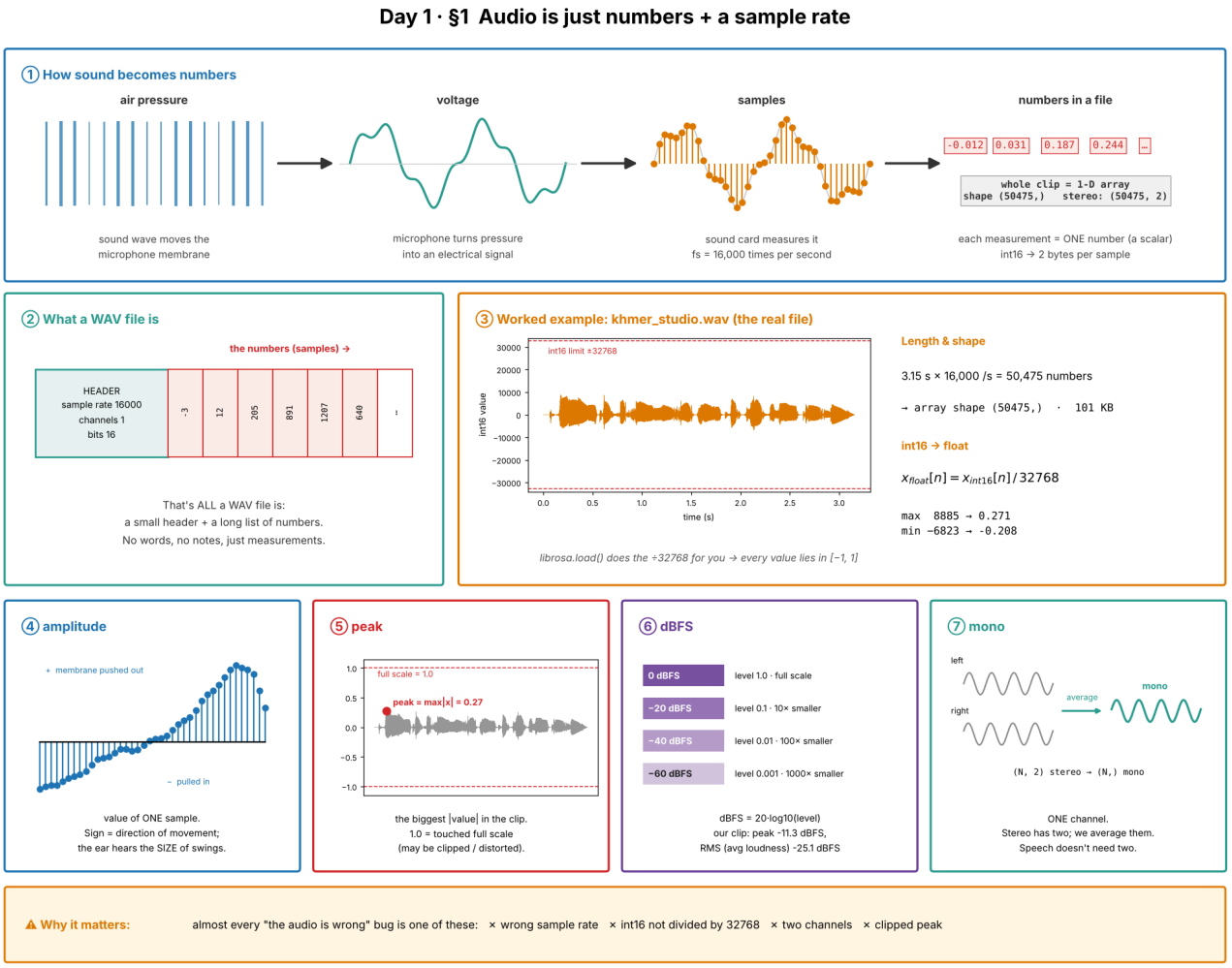
$$x_{\text{float}}[n] = \frac{x_{\text{int16}}[n]}{32768}$$

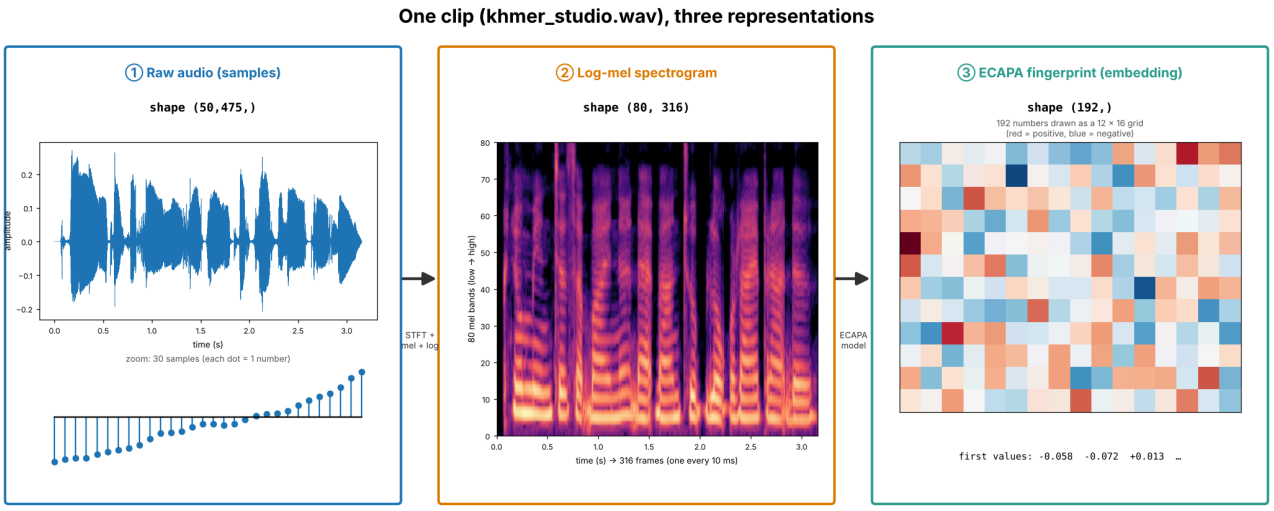
- **amplitude** = the value of one sample. Its sign is just the direction the membrane moved; the ear only hears the size of the swings.
- **peak** = max |x|. 1.0 means the recording touched full scale.
- **dBFS** (decibels relative to full scale) = 20 log10(level). 0 dBFS is full scale, -20 dBFS is 10x smaller, -40 dBFS is 100x smaller. We quote peak and RMS (average loudness) in dBFS.

- **mono** = one channel. Stereo files have two; we average them, speech does not need two.

Why it matters: every bug in speech ML that is "the audio is wrong" is one of these four numbers being wrong (wrong sample rate, int not divided by 32768, two channels, clipped peak).

Visualization Version





What each number means	air pressure at one instant (every 1/16,000 s)	energy in one pitch band at one 10 ms moment	a learned voice feature (no simple meaning)
Size grows with clip length?	YES (6 s → 96,000)	YES (more frames)	NO, always 192
Contains	everything: words, voice, noise	words + voice (phase removed)	mostly WHO is speaking
Can you play it as sound?	yes	roughly (needs a vocoder)	no
Used by	storage, playback (Day 1)	ASR models, input to ECAPA	speaker verification (ECAPA attacker, Day 4)

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short inspect the 3 main clips (studio, phone female, phone male) and **prove that the float values are just the int16 values ÷ 32768**.

1. **For each clip:**
   - `sf.info` reads **only the header**: what the file *claims* (sample rate, channels, format).
   - `load` reads the **actual audio** as floats in **[−1, 1]** at 16 kHz.
   - It computes **n_samples, duration, peak, peak dBFS, RMS dBFS** → one row per clip in a table.
2. **The check:** read the studio clip twice (as **raw int16** and as **float**), find the **loudest sample**, and compare.
3. Print the **first 8 raw samples**.

What the output means

In short the loudest sample is **8885 in the file → 0.2711 after loading** (exactly ÷ 32768), the clip is **not clipped**, and it **starts with near-silence**.

```
loudest sample #2855: int16 = 8885, float = 0.2711, 8885/32768 = 0.2711
```

- The loudest moment is sample **#2855** (≈ 0.18 s into the clip).
- Stored in the file as **8885** → after loading it's **0.2711**, and 8885 ÷ 32768 = **0.2711**.
- ✅ `librosa.load` just **divides by 32768**: same sound, different scale.

- 0.27 is far below 1.0 → the clip is **not clipped**.

`first 8 samples (int16): [1 1 1 1 -1 0 -1 0]`

- Values of **±1 out of ±32768** = basically **silence** (tiny background noise before the speaker starts).
- Normal: most recordings begin with a short moment of quiet.

The table (one row per clip), e.g. studio: **16000 Hz · 1 channel · WAV/PCM_16 · 50,475 samples · 3.155 s · peak 0.271 (−11.3 dBFS) · RMS −25.1 dBFS**

```
rows = []
for name, p in CLIPS.items():
    info = sf.info(str(p))                      # header only: what the file *claims*
    x = load(p)                                 # librosa.load(..., sr=16000) -> float32 in [-1, 1]
    rows.append(dict(clip=name, file=p.name, sr=info.samplerate, channels=info.channels,
                     format=f"{info.format}/{info.subtype}", n_samples=len(x),
                     duration_s=round(len(x) / SR, 3), peak=round(float(np.abs(x).max()), 3),
                     peak_dBFS=round(20 * np.log10(np.abs(x).max()), 1), rms_dBFS=round(rms_dbfs(x), 1)))
table = pd.DataFrame(rows)
display(table)

# Notice: the raw int16 values and the float values are the same numbers divided by 32768
raw, _ = sf.read(str(CLIPS["studio (female)"]), dtype="int16")
x = load(CLIPS["studio (female)"])
i = np.argmax(np.abs(raw))
print(f"loudest sample #{i}: int16 = {raw[i]},  float = {x[i]:.4f},  {raw[i]}/32768 = {raw[i]/32768:.4f}")
print("first 8 samples (int16):", raw[:8])
```

	clip	file	sr	channels	format	n_samples	duration_s	peak
0	studio (female)	khmer_studio.wav	16000	1	WAV/PCM_16	50475	3.155	0.27
1	phone (female)	khmer_phone_female.wav	16000	1	WAV/PCM_16	66480	4.155	0.536
2	phone (male)	khmer_phone_male.wav	16000	1	WAV/PCM_16	59920	3.745	0.332

```
loudest sample #2855: int16 = 8885,  float = 0.2711,  8885/32768 = 0.2711
first 8 samples (int16): [ 1  1  1  1 -1  0 -1  0]
```

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short play the 3 clips so you can hear them.

- `play(..., normalize=False)` keeps each clip's real loudness, so you hear the true differences between studio and phone recordings.

What the output means

In short three audio players: studio female, phone female, phone male.

- Listen for: the **studio** clip is clean and continuous; the **phone female** clip starts with almost a second of silence; the **phone male** voice is clearly lower (about one octave below the women).

```
# Listen. normalize=False in play() keeps the real loudness difference between clips.
for name, p in CLIPS.items():
    print(name)
    display(play(load(p)))
```

```
studio (female)
phone (female)
phone (male)
```

Additional Reading

In short all three files have the same format (16 kHz, mono, 16-bit) and almost the same loudness (≈ −25 dBFS); only the content differs.

How to read this table. All three files are 16 kHz, mono, 16-bit PCM (`WAV/PCM_16`), so the only differences are the *content*. The RMS levels are almost identical (-25.1, -26.3, -25.3 dBFS): this course's clips were loudness-normalized when the dataset was prepared, which is exactly what section 5 teaches you to do. The peaks differ (0.27 studio, 0.54 phone female, 0.33 phone male): the phone-female clip has one loud burst. Peak dBFS minus RMS dBFS (14 dB studio, 21 dB phone female) is the *crest factor*: speech is spiky, so the average is always far below the peak, and a clip with a few loud syllables has a bigger crest factor. The loudest studio sample is int16 8885 = 0.2711 after dividing by 32768, and the first samples are tiny integers (1, 1, 1, -1, 0 ...): near-silence.

2. Waveforms: reading amplitude over time

A waveform plot is just `x[n]` against time. Three things to read from it:

- **tall swings = loud** (a vowel), **flat near zero = pause**, so you can see word rhythm.
- **a thick fuzzy band that never reaches zero** = the *noise floor*: room hiss, phone compression, a fan. Studio audio has a thin line in the gaps; phone audio has a visible band.
- **a line that sits above or below zero** = *DC offset*, a hardware bias. Harmless to the ear, bad for a model. (Check: mean of x should be about 0.)

Zoomed in to 20 ms, a vowel is a repeating wiggle. The repetition period is the **pitch period**: the vocal folds snap shut once per period. If the pattern repeats every 5 ms the pitch (F0) is 1/0.005 = 200 Hz.

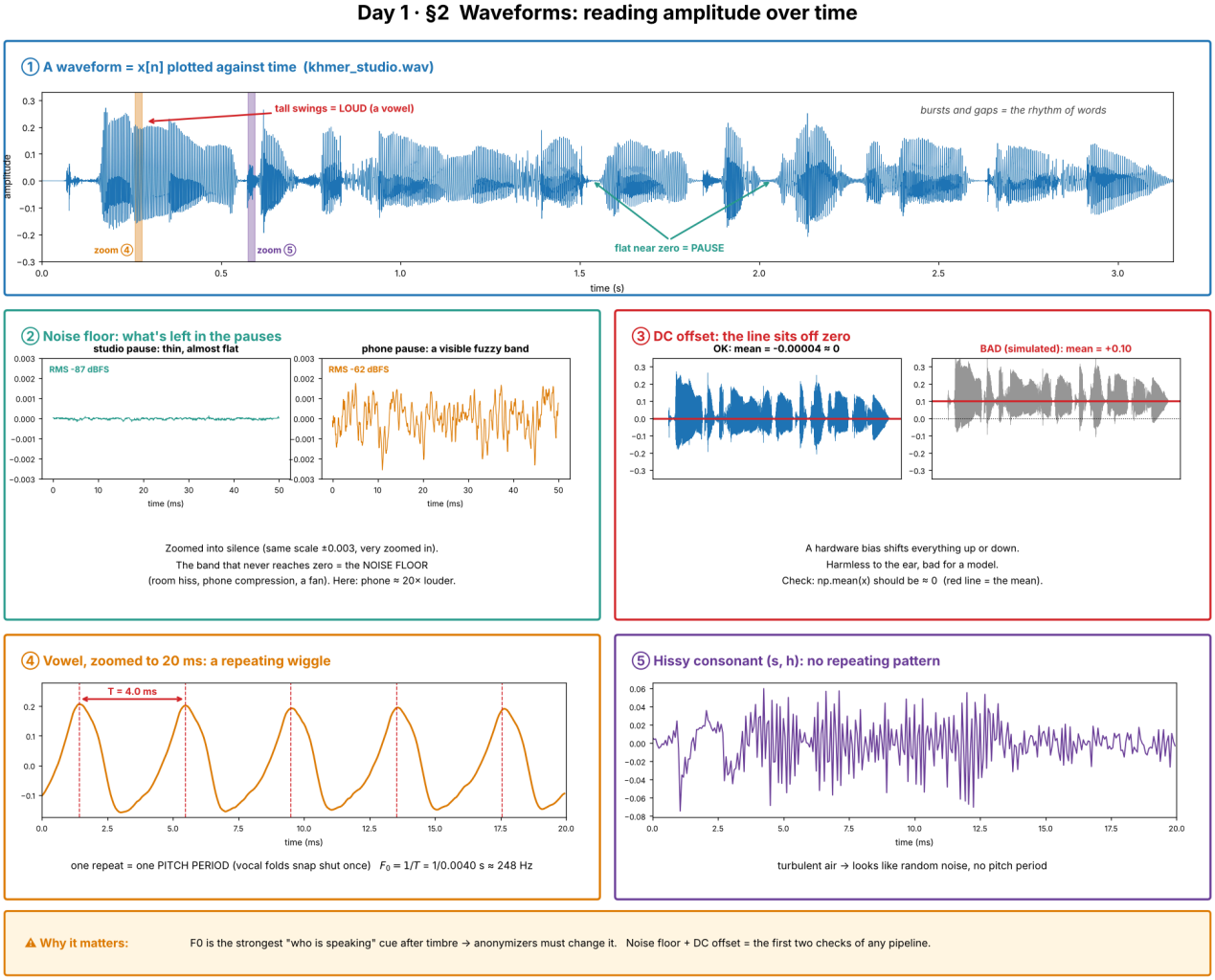
$$F_0 = \frac{1}{T_{\text{period}}}$$

A hissy consonant (s, h) has no repeating pattern: it is turbulent air, which looks like random noise.

Why it matters: F0 is the single strongest "who is speaking" cue after timbre, and anonymizers must change it; noise floor and DC offset are the first two things a feature pipeline should check.

Visualization Version

1st image is necessary , other images are additional reading. If you want to understand futher and you have time then you can , if you dont want to its alright.



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short plot the **3 clips' waveforms** one above the other, show each clip's **DC offset (mean)** and **loudness (RMS)**, and **estimate its noise floor**.

1. **For each clip** (studio female, phone female, phone male), `show_wave` draws the waveform. The title shows the **mean** (DC offset check) and the **RMS** loudness in dBFS.
2. **Noise-floor estimate:** cut the clip into **20 ms frames** (320 samples), compute each frame's loudness (RMS), average the **quietest 10 %** of frames, and draw it as a **dotted line** (±floor).

What the output means

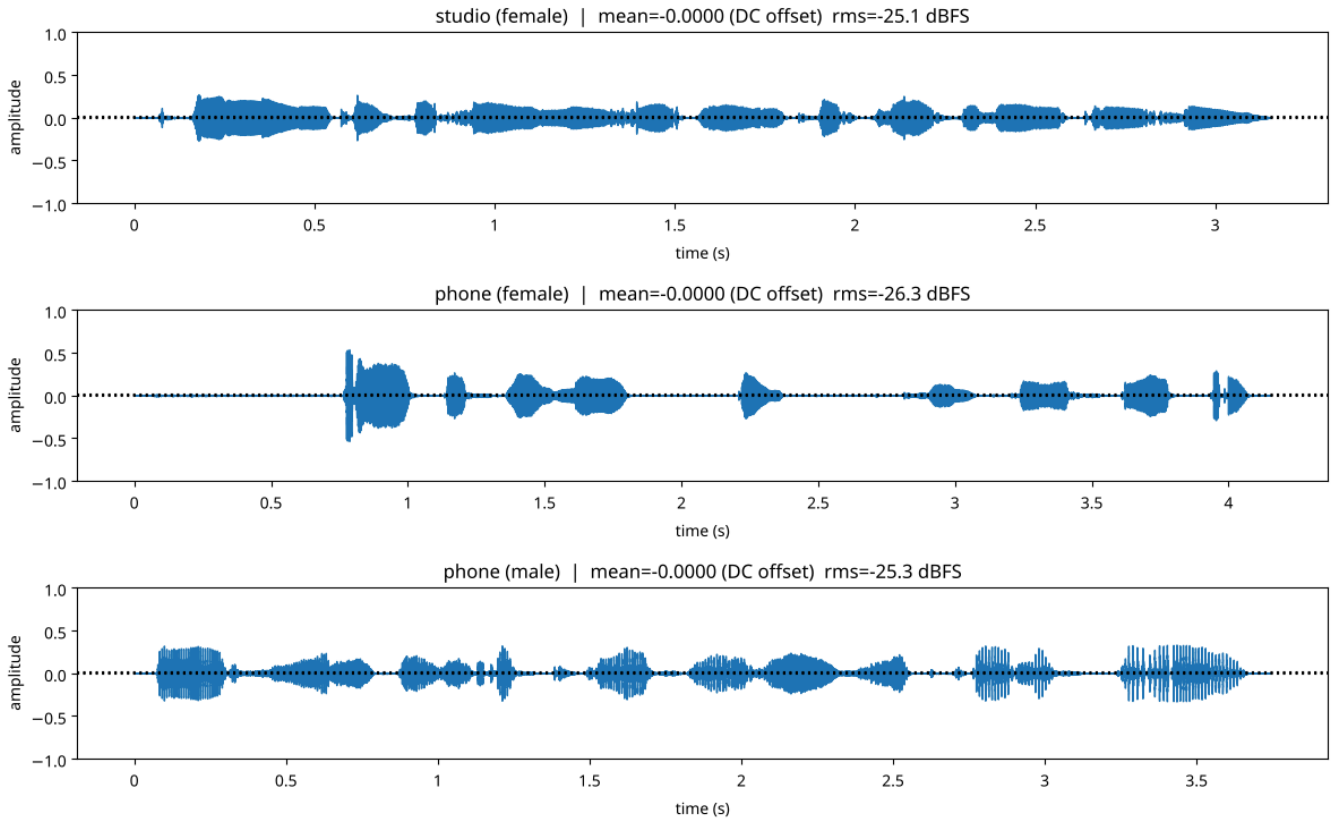
In short all three clips have **no DC offset** and about the **same loudness (≈ −25 dBFS)**. The phone clips' floor (**−62 to −64 dBFS**) is trustworthy; the studio clip's **−51 dBFS** is not, because it has no real pauses.

	Studio (female)	Phone (female)	Phone (male)
Rhythm	speech almost nonstop, 0.07 → 3.15 s	**0.75 s silence** at start + clear gaps	bursts with short gaps; a fine **"comb"** inside syllables
Mean (DC offset)	0.0000 ✅	0.0000 ✅	0.0000 ✅
RMS loudness	−25.1 dBFS	−26.3 dBFS	−25.3 dBFS
"Noise floor" estimate	−51 dBFS ⚠️	−64 dBFS	−62 dBFS

- **Each burst = a syllable**, each gap = a pause between words. All rows share the same −1 … +1 scale.
- **The male "comb":** low pitch → pitch pulses far apart (about one every **8 ms** ≈ 125 Hz), visible even at this zoom.
- **The noise-floor trap:** the studio clip's quietest frames still contain speech tails, so −51 dBFS is not its real floor. **A noise-floor estimate only works when the clip contains pauses.**
- On a dark theme the dotted line is invisible (it is drawn in black, `c="k"`).

```
fig, axes = plt.subplots(3, 1, figsize=(12, 7.5), sharex=False)
for ax, (name, p) in zip(axes, CLIPS.items()):
    x = load(p)
    show_wave(x, title=f"{name}  |  mean={x.mean():+.4f} (DC offset)  rms={rms_dbfs(x):.1f} dBFS", ax=ax)
    # dotted line = the RMS level of the quietest 10% of 20 ms frames (a cheap noise-floor estimate;
    # notice: it only measures the *noise* if the clip actually contains pauses)
    frames = librosa.util.frame(x, frame_length=320, hop_length=320)
    floor = np.sqrt((frames ** 2).mean(0)); floor = np.sort(floor)[: len(floor) // 10].mean()
    ax.axhline(floor, ls=":", c="k"); ax.axhline(-floor, ls=":", c="k")
    print(f"{name:16s} quietest 10% of frames ~ {20*np.log10(floor+1e-9):.0f} dBFS")
plt.tight_layout()
```

```
studio (female)  quietest 10% of frames ~ -51 dBFS
phone (female)   quietest 10% of frames ~ -64 dBFS
phone (male)     quietest 10% of frames ~ -62 dBFS
```



Additional Reading

In short each burst is a syllable; no clip has a DC offset; the noise-floor number is only meaningful for the phone clips, which have real pauses.

How to read this plot. Each burst of tall swings is a syllable; the narrow gaps between bursts are the pauses between words. The three clips have very different rhythm: the studio clip (top) is read without a single pause, speech from 0.07 s to 3.15 s, while the phone-female clip (middle) starts with 0.75 s of silence and has clear gaps at 1.05 s, 1.9-2.2 s and 2.4-2.8 s. The male clip (bottom) shows the pitch pulses as a fine comb inside each syllable because his pitch is low (one pulse every 8 ms is visible at this zoom). The "quietest 10 % of frames" number is -62 to -64 dBFS for the phone clips, whose gaps are real pauses, but -51 dBFS for the studio clip, whose quietest frames still contain speech tails: a noise-floor estimate only works when there is silence to measure. All three means are 0.0000, so there is no DC offset to fix. The amplitude axis is the same [-1, 1] on every row, so heights are directly comparable.

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short zoom into **20 ms of a vowel** and **20 ms of a hissy consonant** from the male clip, then measure the pitch by counting repetitions.

1. Compute the spectrogram and pick two moments automatically: the loudest **low-frequency-heavy** frame (a vowel) and the loudest **high-frequency-heavy** frame (a fricative like s/h).
2. Plot 20 ms (320 samples) of each.
3. **Autocorrelation** on the vowel: compare the window with shifted copies of itself; the shift where it matches best = one pitch period.

What the output means

In short the vowel repeats every **8.38 ms → F0 ≈ 119 Hz**, close to pyworld's whole-clip median of **126 Hz**; the consonant has no repeating pattern.

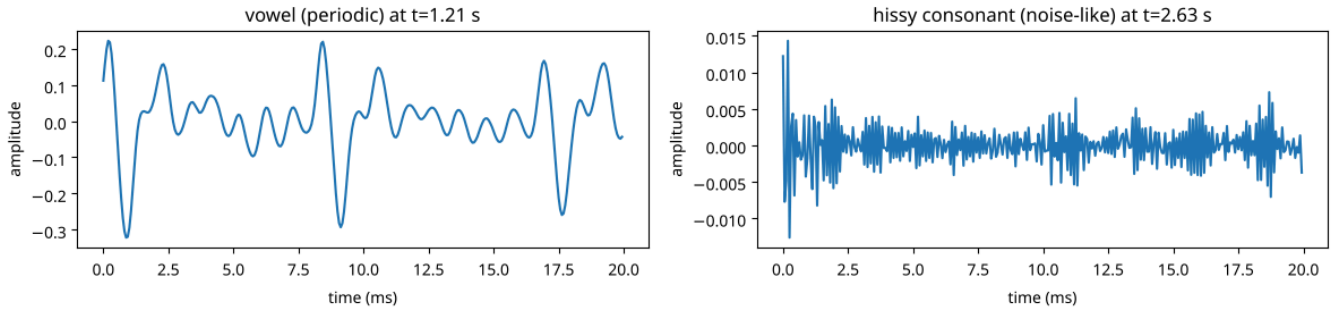
- `strongest repetition every 134 samples = 8.38 ms -> F0 ~ 119 Hz` : 134 / 16000 = 0.00838 s, and 1 / 0.00838 ≈ 119 Hz.
- `median F0 of the whole clip (pyworld): 126 Hz` : a typical male voice. The two agree because they measure the same thing two different ways (one frame vs the whole clip).

```
# Zoom: 20 ms of a vowel vs 20 ms of a hissy consonant, chosen automatically from the male phone clip.
x = load(CLIPS["phone (male)"])
S = stft_db(x)                                  # (257, frames), 10 ms per frame
hz = librosa.fft_frequencies(sr=SR, n_fft=N_FFT)
low = S[hz < 1000].mean(0); high = S[hz > 4000].mean(0)
loud = S.mean(0) > S.mean(0).max() - 25         # ignore quiet frames
t_vowel = np.argmax(np.where(loud, low - high, -1e9)) * HOP / SR     # most "low-heavy" loud frame
t_hiss  = np.argmax(np.where(loud, high - low, -1e9)) * HOP / SR     # most "high-heavy" loud frame

fig, axes = plt.subplots(1, 2, figsize=(12, 3))
for ax, t0, label in zip(axes, [t_vowel, t_hiss], ["vowel (periodic)", "hissy consonant (noise-like)"]):
    seg = x[int(t0 * SR): int(t0 * SR) + int(0.020 * SR)]            # 20 ms = 320 samples
    ax.plot(np.arange(len(seg)) / SR * 1000, seg)
    ax.set(title=f"{label} at t={t0:.2f} s", xlabel="time (ms)", ylabel="amplitude")
plt.tight_layout()

# Count the repetitions in the vowel window with autocorrelation -> pitch period
seg = x[int(t_vowel * SR): int(t_vowel * SR) + 320]
ac = np.correlate(seg, seg, "full")[319:]
lag = np.argmax(ac[40:200]) + 40                                     # search 80..400 Hz
print(f"vowel: strongest repetition every {lag} samples = {1000*lag/SR:.2f} ms  ->  F0 ~ {SR/lag:.0f} Hz")
print(f"median F0 of the whole clip (pyworld): {median_f0(x):.0f} Hz")
```

```
vowel: strongest repetition every 134 samples = 8.38 ms  ->  F0 ~ 119 Hz
median F0 of the whole clip (pyworld): 126 Hz
```



Additional Reading

In short left = a vowel, a repeating shape every 8.4 ms (one vocal-fold closure each); right = a fricative, small random wiggles with no repeat.

How to read this plot. Left (t = 1.21 s): a sharp downward spike at about 0.8 ms, 9.2 ms and 17.6 ms, with the same smaller wiggles after each: the pattern repeats every 8.4 ms, so 20 ms holds

about 2.4 pitch periods. 1/0.0084 s = 119 Hz, and pyworld's median over the whole clip is 126 Hz, a typical male voice. Each spike is one closure of the vocal folds. Right (t = 2.63 s): amplitude 10x smaller (note the y-axis, 0.015 instead of 0.3), a click at 0 ms and then fast random wiggles with no repeating shape: turbulent airflow, a fricative. If you can tell these two apart by eye, you already have the idea behind voiced/unvoiced detection.

3. Sampling, Nyquist and aliasing

Sampling at `fs` Hz can represent frequencies up to **fs/2** (the *Nyquist frequency*) and nothing above it. At 16 kHz that is 8 kHz, which covers almost all of speech (the "s" sound reaches higher, which is why 16 kHz speech sounds slightly dull compared with 48 kHz).

What happens to a frequency *above* Nyquist? It does not disappear: it folds back and shows up as a fake lower frequency, an **alias**:

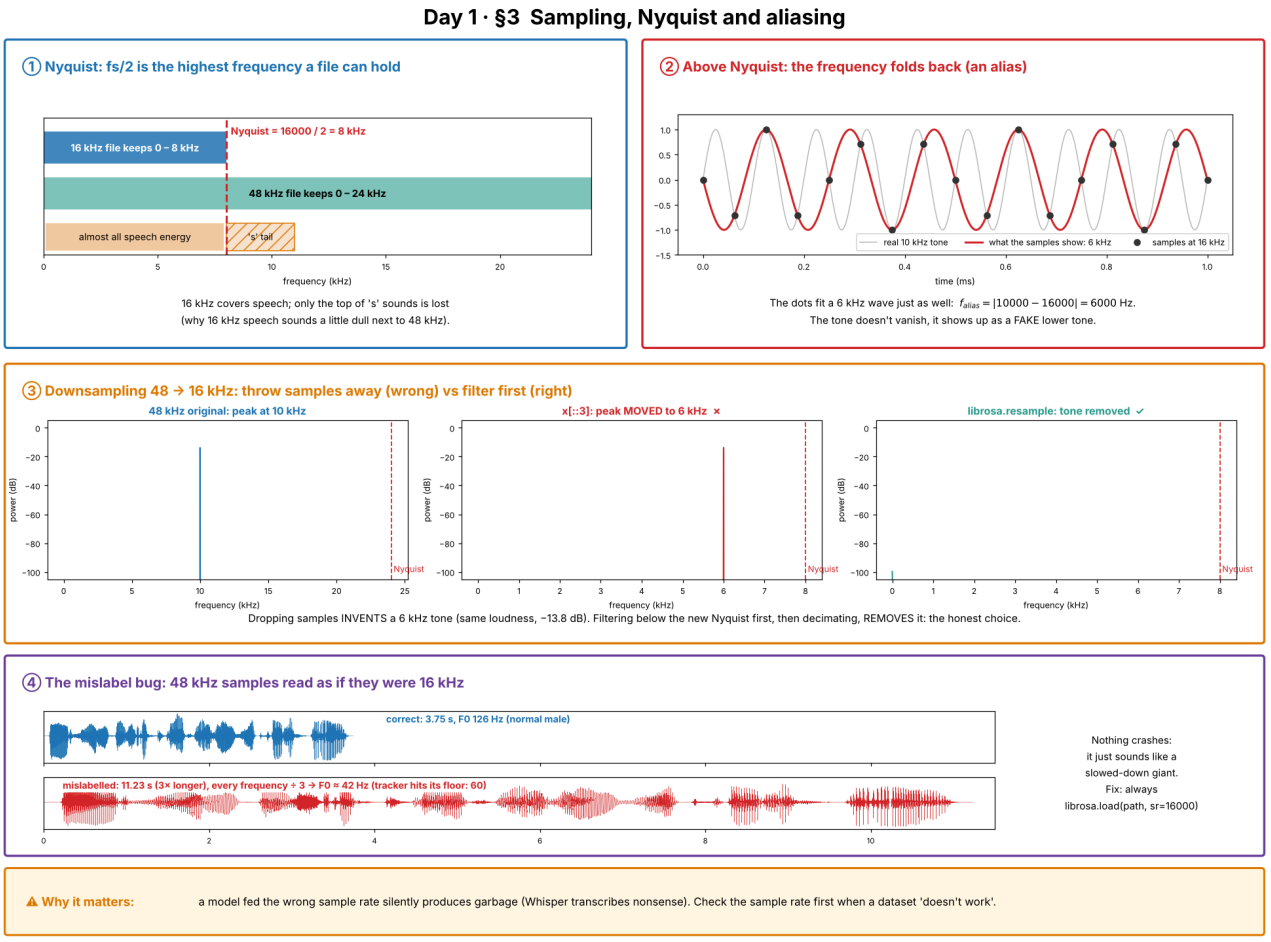
$$f_{\text{alias}} = |f - k \cdot f_s| \quad \text{for the integer } k \text{ that brings it below } f_s/2$$

Worked example. A 10 kHz tone sampled at 16 kHz: $|10\,000 - 16\,000|$ = 6 000 Hz. The tone *appears* at 6 kHz. So you must never downsample by just throwing samples away (`x[::3]`); you first low-pass filter below the new Nyquist, then decimate. `librosa.resample` does both.

A related, very common bug: a file recorded at 48 kHz but *labelled* 16 kHz. Nothing crashes; the audio simply plays 3x slower and every frequency, including the pitch, drops by 3x.

Why it matters: mismatched sample rates silently destroy models (a Whisper fed 48 kHz samples as if 16 kHz transcribes garbage) and are the first thing to check when a dataset "does not work".

Visualization Version



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short make a **10 kHz tone** at 48 kHz, then bring it down to 16 kHz in two ways: by **throwing samples away** (`x[::3]`) and by **proper resampling**.

- `bad = tone48[::3]` : keep every 3rd sample, **no filter**.
- `good = librosa.resample(...)` : low-pass filter below 8 kHz first, then decimate.
- Plot the frequency content (periodogram) of the original, the bad and the good version, with the Nyquist line in red.

What the output means

In short throwing samples away moves the tone from **10 kHz to a fake 6 kHz** at the same loudness; proper resampling **removes** it.

version	peak at	level	meaning

48 kHz original	10000 Hz	−13.8 dB	the real tone
`x[::3]`	**6000 Hz**	−13.8 dB	❌ an **alias**: a tone that was never there
`librosa.resample`	(none)	−99.7 dB	✅ only the noise floor is left: the tone can't exist in a 16 kHz file

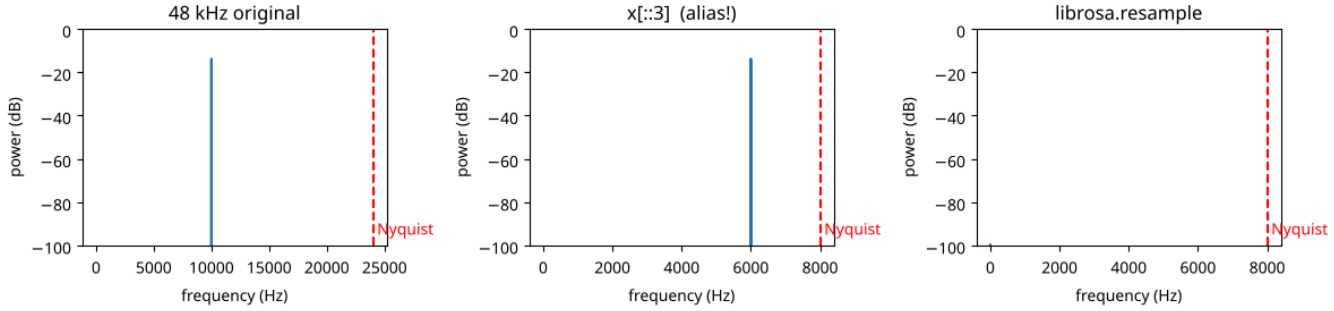
```
# A 10 kHz tone sampled at 48 kHz, then brought to 16 kHz in two ways.
fs_hi = 48000
t = np.arange(int(0.5 * fs_hi)) / fs_hi
tone48 = 0.5 * np.sin(2 * np.pi * 10000 * t).astype(np.float32)

bad  = tone48[::3]                                  # just drop 2 of every 3 samples: NO anti-alias filter
good = librosa.resample(tone48, orig_sr=fs_hi, target_sr=SR)   # low-pass at 8 kHz first, then decimate

def periodogram_db(x, sr):
    f, P = scipy.signal.periodogram(x, sr, window="hann")
    return f, 10 * np.log10(P + 1e-12)

fig, axes = plt.subplots(1, 3, figsize=(12, 3))
for ax, (lab, sig, sr) in zip(axes, [("48 kHz original", tone48, fs_hi), ("x[::3]  (alias!)", bad, SR),
("librosa.resample", good, SR)]):
    f, P = periodogram_db(sig, sr)
    ax.plot(f, P); ax.set(title=lab, xlabel="frequency (Hz)", ylabel="power (dB)", ylim=(-100, 0))
    ax.axvline(sr / 2, c="r", ls="--"); ax.text(sr / 2, -95, " Nyquist", c="r")
    print(f"{lab:20s} peak at {f[np.argmax(P)]:6.0f} Hz, level {P.max():6.1f} dB")
plt.tight_layout()
```

```
48 kHz original      peak at  10000 Hz, level  -13.8 dB
x[::3]  (alias!)     peak at   6000 Hz, level  -13.8 dB
librosa.resample     peak at      0 Hz, level  -99.7 dB
```



Additional Reading

In short dropping samples invents a 6 kHz tone; filtering then decimating removes it, which is the honest result.

How to read this plot. Left: one spike at 10 000 Hz, the Nyquist line is at 24 000 Hz, all is well. Middle: the new Nyquist line is at 8 000 Hz and the spike has moved to 6 000 Hz, a tone that was never in the signal; it is exactly as loud as the original (-13.8 dB in both). Right: proper resampling first removed everything above 8 kHz, so the plot is just the noise floor around -100 dB: the tone is gone, correctly, because a 16 kHz file cannot hold it. The lesson: dropping samples *invents* content; filtering then decimating *removes* content, which is the honest choice.

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short repeat the aliasing test on **real speech**: add a 10 kHz whistle to a 48 kHz version of the studio clip, then downsample it both ways.

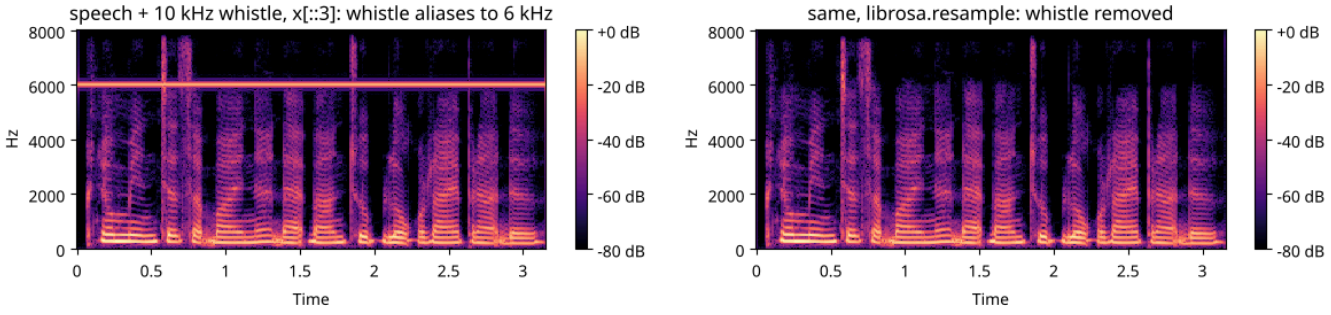
- Left spectrogram: `x48[::3]` (no filter). Right spectrogram: `librosa.resample` (filter first).

What the output means

In short on the left a **bright horizontal line at 6 kHz** runs through the speech (the aliased whistle); on the right it is **gone**.

- No text output, only the two spectrograms. Same lesson as above, now on a spectrogram you'll read every day.

```
# Same alias on a spectrogram, and on real speech: resample the studio clip 16->48 kHz, then slice vs resample back.
x = load(CLIPS["studio (female)"])
x48 = librosa.resample(x, orig_sr=SR, target_sr=48000)
# add a 10 kHz whistle to the 48 kHz version so there is something above 8 kHz to alias
x48 = x48 + 0.05 * np.sin(2 * np.pi * 10000 * np.arange(len(x48)) / 48000).astype(np.float32)
fig, axes = plt.subplots(1, 2, figsize=(12, 3))
show_spec(x48[::3], title="speech + 10 kHz whistle, x[::3]: whistle aliases to 6 kHz", ax=axes[0])
show_spec(librosa.resample(x48, orig_sr=48000, target_sr=SR), title="same, librosa.resample: whistle removed", ax=axes[1])
plt.tight_layout()
```



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short simulate the **sample-rate mislabel bug**: a correct 48 kHz file read as if it were 16 kHz.

- Make a proper 48 kHz copy of the male clip, then analyse and play its samples at 16 kHz.

What the output means

In short the clip becomes **3× longer (3.75 s → 11.23 s)** and every frequency drops 3×; the pitch tracker reports **60 Hz** instead of 126 Hz.

- `correct: 59920 samples / 16000 = 3.75 s, median F0 = 126 Hz`
- `mislabel: 179760 samples / 16000 = 11.23 s, median F0 = 60 Hz` : the true value would be 126 / 3 = 42 Hz, but the tracker can't go below ~50 Hz, so it locks onto a wrong harmonic. **A pitch at the tracker's floor is itself a warning sign.**
- The two players: the mislabelled one sounds like a slowed-down giant.

```
# The sample-rate mislabel bug: 48 kHz samples played/analysed as if they were 16 kHz.
x = load(CLIPS["phone (male)"])
x48 = librosa.resample(x, orig_sr=SR, target_sr=48000)     # a correct 48 kHz version of the same clip
print(f"correct  : {len(x)} samples / 16000 = {len(x)/SR:.2f} s, median F0 = {median_f0(x):.0f} Hz")
print(f"mislabel : {len(x48)} samples / 16000 = {len(x48)/SR:.2f} s, median F0 = {median_f0(x48, SR):.0f} Hz   <- 3x
longer, F0/3")
print("listen: the 'mislabelled' version sounds like a slowed-down giant")
display(play(x)); display(play(x48, SR))
```

```
correct  : 59920 samples / 16000 = 3.75 s, median F0 = 126 Hz
mislabel : 179760 samples / 16000 = 11.23 s, median F0 = 60 Hz   <- 3x longer, F0/3
listen: the 'mislabelled' version sounds like a slowed-down giant
```

Additional Reading

In short reading 48 kHz samples as 16 kHz stretches time 3× and divides pitch by 3; always load with `librosa.load(path, sr=16000)`.

How to read this. Treating 48 000 samples per second as 16 000 stretches the clip from 3.75 s to 11.23 s and divides every frequency by 3: the male pitch should drop from 126 Hz to 42 Hz, below any human voice. `median_f0` reports 60 Hz rather than 42 because the tracker has a 50 Hz floor and starts locking onto the wrong harmonic; a pitch tracker returning values at its floor is itself a warning sign of a rate mismatch. The fix is one line, `librosa.load(path, sr=16000)`, which reads the header rate and resamples; the bug only happens when code reads raw samples and *assumes* the rate. That is also why `load()` in `src/utils.py` always passes `sr=SR`.

4. Quantization: how many bits per sample

Each sample is stored with a fixed number of bits. 16 bits gives 65 536 possible levels, 8 bits gives 256, 4 bits gives 16. Rounding a sample to the nearest level adds a small error, **quantization noise**. Each extra bit halves the step size, which halves the noise, which is 6 dB:

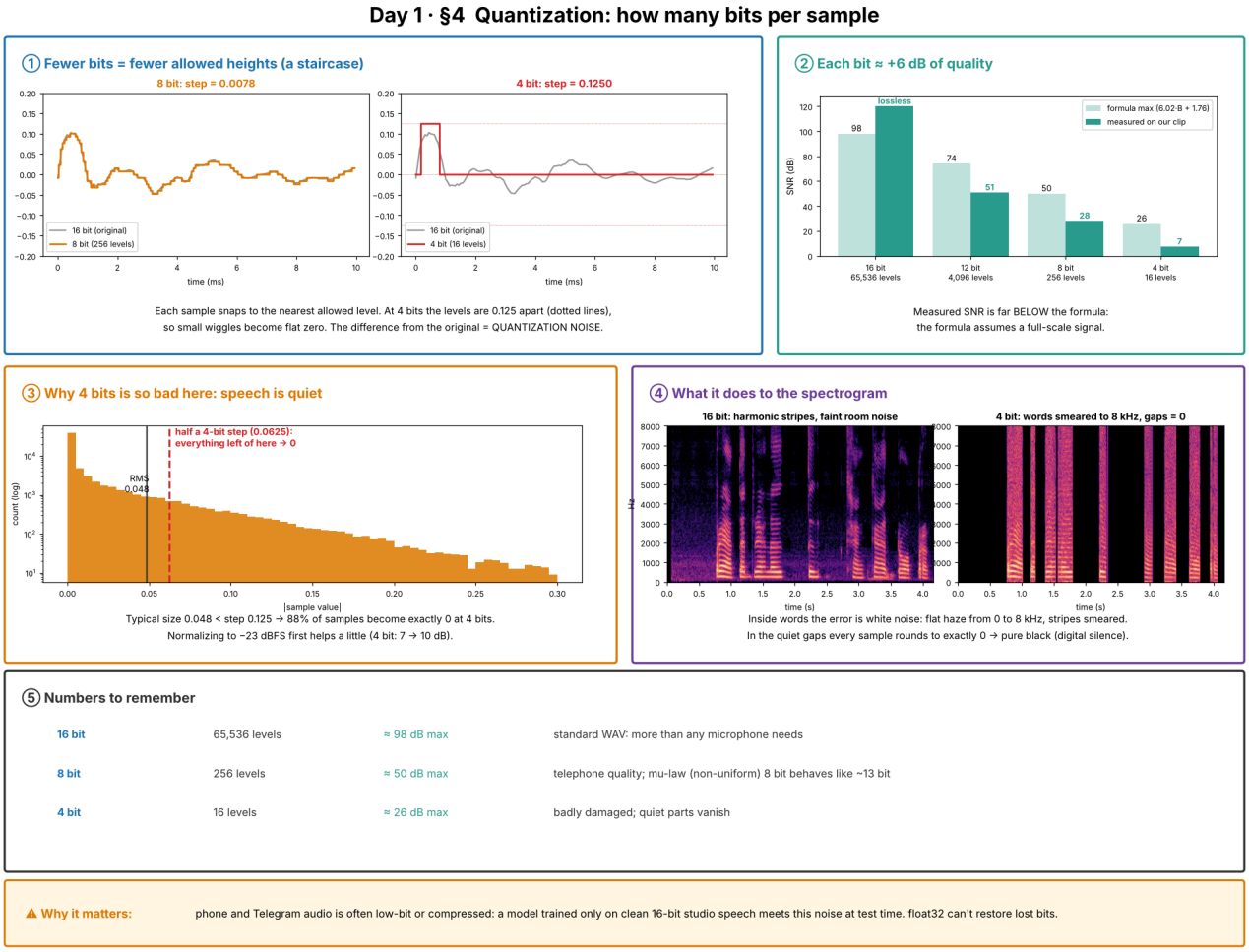
$$\mathrm{SNR}_{\max} \approx 6.02 \cdot B + 1.76 \ \mathrm{dB}$$

Worked example. 16 bits: about 98 dB SNR (far more than any microphone). 8 bits: about 50 dB. 4 bits: 26 dB. Those are *maximum* values for a full-scale signal. Our speech has an RMS of only 0.05 (-26 dBFS), so the 4-bit step of 0.125 is bigger than most of the samples: they snap to zero, and the measured SNR is a few dB, not 26. The gap between the formula and the measurement is the reason loudness normalization (section 5) comes *before* any lossy step.

$$\mathrm{SNR} = 10 \log_{10} \frac{\sum x^2}{\sum (x - \hat{x})^2}$$

Why it matters: telephone and Telegram audio is often 8-bit mu-law or heavily compressed; a model trained only on clean 16-bit studio speech meets this noise at test time. Also, float32 features are not "more precision for free": the information was fixed when the sample was quantized.

Visualization Version



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short re-store the phone-female clip with **fewer bits** (16, 12, 8, 4) and measure how much quality is lost (SNR).

- `requantize(x, bits)` : snap every sample to the nearest of 2^bits allowed levels.
- `snr_db(x, y)` : signal power ÷ error power, in dB (higher = better).
- Cross-check against soundfile's own 8-bit writer, then try normalizing to −23 dBFS before quantizing.

What the output means

In short measured SNR is **far below** the textbook maximum (4 bit: **7 dB** vs 26 dB) because our speech is quiet: **88 %** of samples round to exactly 0 at 4 bits.

bits	levels	step	measured SNR	formula max

16	65,536	0.00003	lossless	98 dB
12	4,096	0.00049	50.8 dB	74 dB
8	256	0.0078	28.0 dB	50 dB
4	16	0.125	7.3 dB	26 dB

- `RMS amplitude = 0.048, 4-bit step = 0.125 -> 88% of samples become exactly 0` : a typical sample is smaller than half a step.
- Normalizing to −23 dBFS first helps a little (8 bit: 28 → 31 dB; 4 bit: 7 → 10 dB).
- The 8-bit cross-check differs by exactly one step (0.0078): soundfile truncates, our function rounds. Same levels.

```
x = load(CLIPS["phone (female)"])

def requantize(x, bits):
    # Snap each sample to the nearest of 2**bits levels in [-1, 1) (a mid-tread, unbiased quantizer).
    step = 2.0 / 2 ** bits
    return (np.round(x / step) * step).astype(np.float32)

def snr_db(x, y):
    return 10 * np.log10(np.sum(x ** 2) / (np.sum((x - y) ** 2) + 1e-20))

# Cross-check against soundfile's own 8-bit unsigned PCM writer, in memory. libsndfile truncates (floor)
# instead of rounding, so the two differ by at most one step (0.0078); the levels are the same.
buf = io.BytesIO(); sf.write(buf, x, SR, format="WAV", subtype="PCM_U8"); buf.seek(0)
x8_sf, _ = sf.read(buf, dtype="float32")
x8, x4 = requantize(x, 8), requantize(x, 4)
print(f"8-bit via soundfile vs manual rounding: max |diff| = {np.abs(x8_sf - x8).max():.4f} (= one 8-bit step: floor vs
round)")
for b, y in [(16, requantize(x, 16)), (12, requantize(x, 12)), (8, x8), (4, x4)]:
    s_ = snr_db(x, y)
    print(f"{b:2d} bits: {2**b:6d} levels, step {2/2**b:.5f}, measured SNR = {'lossless' if s_ > 150 else f'{s_:5.1f} dB'}
(formula max {6.02*b+1.76:.0f} dB)")
print(f"RMS amplitude of this clip = {np.sqrt(np.mean(x**2)):.3f}, 4-bit step = 0.125  ->  {100*np.mean(x4 == 0):.0f}% of
samples become exactly 0 at 4 bits")
# Using the full range matters: normalize to -23 dBFS first, then quantize
xn = x * 10 ** ((-23 - rms_dbfs(x)) / 20)
print(f"after -23 dBFS normalization:  8 bits SNR = {snr_db(xn, requantize(xn, 8)):.1f} dB,  4 bits SNR = {snr_db(xn,
requantize(xn, 4)):.1f} dB")
print("\nlisten: 16 bit, 8 bit, 4 bit")
display(play(x)); display(play(x8)); display(play(x4))
```

```
8-bit via soundfile vs manual rounding: max |diff| = 0.0078 (= one 8-bit step: floor vs round)
16 bits:  65536 levels, step 0.00003, measured SNR = lossless   (formula max 98 dB)
12 bits:   4096 levels, step 0.00049, measured SNR =  50.8 dB   (formula max 74 dB)
 8 bits:    256 levels, step 0.00781, measured SNR =  28.0 dB   (formula max 50 dB)
 4 bits:     16 levels, step 0.12500, measured SNR =   7.3 dB   (formula max 26 dB)
RMS amplitude of this clip = 0.048, 4-bit step = 0.125  ->  88% of samples become exactly 0 at 4 bits
after -23 dBFS normalization:  8 bits SNR = 30.8 dB,  4 bits SNR = 10.1 dB

listen: 16 bit, 8 bit, 4 bit
```

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short plot 10 ms of the 16-bit vs 4-bit signal, and the spectrogram of the 4-bit version.

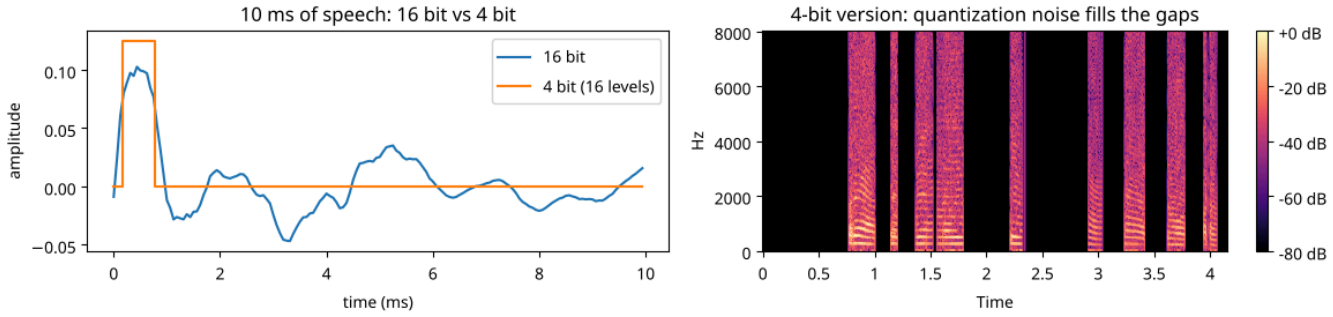
- Left: the smooth original with the 4-bit **staircase** on top. Right: spectrogram of the 4-bit clip.

What the output means

In short the 4-bit staircase can only sit at 16 heights, so small wiggles become **flat zero**; in the spectrogram the words fill with **haze up to 8 kHz**.

- No text output. See the How-to-read notes below.

```
fig, axes = plt.subplots(1, 2, figsize=(12, 3))
seg = slice(int(1.0 * SR), int(1.0 * SR) + 160)     # 10 ms somewhere in the middle
tt = np.arange(160) / SR * 1000
axes[0].plot(tt, x[seg], label="16 bit"); axes[0].step(tt, x4[seg], where="mid", label="4 bit (16 levels)")
axes[0].set(title="10 ms of speech: 16 bit vs 4 bit", xlabel="time (ms)", ylabel="amplitude"); axes[0].legend()
show_spec(x4, title="4-bit version: quantization noise fills the gaps", ax=axes[1])
plt.tight_layout()
```



Additional Reading

In short fewer bits = a coarser staircase = quantization noise (white, so it shows as even haze); quiet speech suffers most.

How to read this plot. Left: the smooth 16-bit curve and the 4-bit staircase on top of it. The staircase can only sit at 16 heights (step 0.125), so a wiggle of size 0.05 becomes flat zero and only the biggest peaks reach one step: that difference is the quantization noise. Right: compare with the spectrogram of the same clip in section 6; here the dark gaps between words have filled with a uniform purple haze from the bottom to 8 kHz, and the harmonic stripes are smeared. Quantization noise is white (flat across frequency), which is why it shows as even haze rather than stripes. The numbers printed above: 16 bits is lossless here (the file already is 16 bit), 12 bits gives 51 dB, 8 bits 28 dB and 4 bits only 7 dB, far below the 74 / 50 / 26 dB maxima, because the clip's RMS amplitude is 0.048 and 88 % of the samples round to exactly zero at 4 bits. Normalizing to -23 dBFS first buys about 3 dB; the real fix for low bit depths is a non-uniform quantizer (mu-law, used by telephones), which is 8 bits that behave like 13.

5. Loudness: peak, RMS, clipping, normalization, trimming

Two different "how loud" numbers:

- **peak dBFS** = 20 log10(max|x|). Tells you how close you are to clipping. 0 dBFS = the ceiling.
- **RMS dBFS** = 20 log10(sqrt(mean(x^2))). Tells you the average energy, close to perceived loudness. Speech normally sits at -20 to -30 dBFS RMS.

$$\mathrm{RMS} = \sqrt{\frac{1}{N}\sum_{n} x[n]^2}, \qquad \mathrm{dBFS} = 20\log_{10}(\mathrm{RMS})$$

Clipping. If a sample would exceed 1.0 it is cut off at 1.0: the wave tops become flat. That flat top is a sharp corner, and sharp corners contain every frequency, so clipping sprays energy over the whole spectrogram (it sounds crunchy). Clipping cannot be undone.

Normalization. Models should not care whether a phone was held near or far. The usual fix is to scale the clip so its RMS is a fixed value, e.g. -23 dBFS (a broadcasting standard; -20 to -25 is common). Scale factor: gain = 10^((target - current)/20). Peak normalization (make max|x| = 1) is worse: one loud click sets the gain for the whole clip.

Trimming. Leading and trailing silence contains no speech, costs compute, and dilutes pooled features (a speaker embedding is an average over frames; silence frames pull it toward "nobody"). `librosa.effects.trim(x, top_db=30)` cuts everything more than 30 dB below the loudest frame at both ends.

Why it matters: ECAPA, Whisper and kNN-VC all see normalized, trimmed audio in training; feed them a -45 dBFS clip with 2 s of silence and quality drops without any error message.

Visualization Version



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short measure the studio clip's loudness, then demo **clipping**, **normalization** and **trimming** on it.

1. **Original:** peak dBFS, RMS dBFS, and crest factor (peak − RMS).
2. **Clipping:** multiply by 5 and cut at ±1.
3. **Normalization:** one gain so the RMS becomes −23 dBFS.
4. **Trimming:** add 0.5 s of faint noise at both ends, then cut it off with `librosa.effects.trim(top_db=30)`.

What the output means

In short ×5 clips only **0.3 %** of samples yet damages the sound; normalization needs a gain of only **×1.27**; trim removes the added **0.95 s** of padding almost exactly.

- `original: peak -11.3 dBFS, RMS -25.1 dBFS, crest factor 13.8 dB`: speech is spiky; the average sits far below the peak.
- `clipped: peak 0.0 dBFS, RMS -11.2 dBFS, 0.3% of samples sit on the rail`: very few samples hit ±1, but the flat tops are audible.
- `normalize: gain x1.27 -> RMS -23.0 dBFS, peak -9.2 dBFS (still below 0: safe)`.
- `trim: padded 4.15 s -> kept 0.51 s .. 3.71 s = 3.20 s`: the original clip itself loses only 0.03 s, because it was already tightly cut.

```
x = load(CLIPS["studio (female)"])
print(f"original : peak {20*np.log10(np.abs(x).max()):6.1f} dBFS, RMS {rms_dbfs(x):6.1f} dBFS, crest factor
{20*np.log10(np.abs(x).max()) - rms_dbfs(x):.1f} dB")

# Clipping demo: multiply by 5, cut at +-1. Peak is now exactly 0 dBFS but the signal is damaged.
clipped = np.clip(5 * x, -1, 1)
print(f"clipped  : peak {20*np.log10(np.abs(clipped).max()):6.1f} dBFS, RMS {rms_dbfs(clipped):6.1f} dBFS, "
      f"{100*np.mean(np.abs(clipped) >= 0.999):.1f}% of samples sit on the rail")

# Loudness normalization to -23 dBFS RMS: a single gain for the whole clip.
TARGET = -23.0
gain = 10 ** ((TARGET - rms_dbfs(x)) / 20)
x_norm = x * gain
print(f"normalize: gain x{gain:.2f}  -> RMS {rms_dbfs(x_norm):6.1f} dBFS, peak {20*np.log10(np.abs(x_norm).max()):6.1f}
dBFS  (still below 0: safe)")

# Trimming leading/trailing silence. The studio clip is already tightly cut, so first add 0.5 s of faint
# room noise (-60 dBFS) at both ends, the way a real recording session leaves it, then trim it back off.
pad = (0.001 * np.random.randn(int(0.5 * SR))).astype(np.float32)
x_padded = np.concatenate([pad, x_norm, pad])
x_trim, (i0, i1) = librosa.effects.trim(x_padded, top_db=30)
print(f"trim     : padded {len(x_padded)/SR:.2f} s -> kept {i0/SR:.2f} s .. {i1/SR:.2f} s = {len(x_trim)/SR:.2f} s "
      f"({(len(x_padded)-len(x_trim))/SR:.2f} s removed; the original clip itself loses only "
      f"{(len(x_norm) - len(librosa.effects.trim(x_norm, top_db=30)[0]))/SR:.2f} s)")
print("\nlisten: original, clipped, normalized+trimmed")
display(play(x)); display(play(clipped)); display(play(x_trim))
```

```
original : peak  -11.3 dBFS, RMS  -25.1 dBFS, crest factor 13.8 dB
clipped  : peak    0.0 dBFS, RMS  -11.2 dBFS, 0.3% of samples sit on the rail
normalize: gain x1.27  -> RMS  -23.0 dBFS, peak   -9.2 dBFS  (still below 0: safe)
trim     : padded 4.15 s -> kept 0.51 s .. 3.71 s = 3.20 s (0.95 s removed; the original clip itself loses only 0.03 s)

listen: original, clipped, normalized+trimmed
```

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short draw the original, clipped, normalized+padded (with trim points) and clipped-spectrogram versions, plus a zoom on one clipped peak.

- 2 × 2 grid of plots, then a 10 ms zoom showing the curve the signal *wanted* to follow (dotted) vs the flat roof it got.

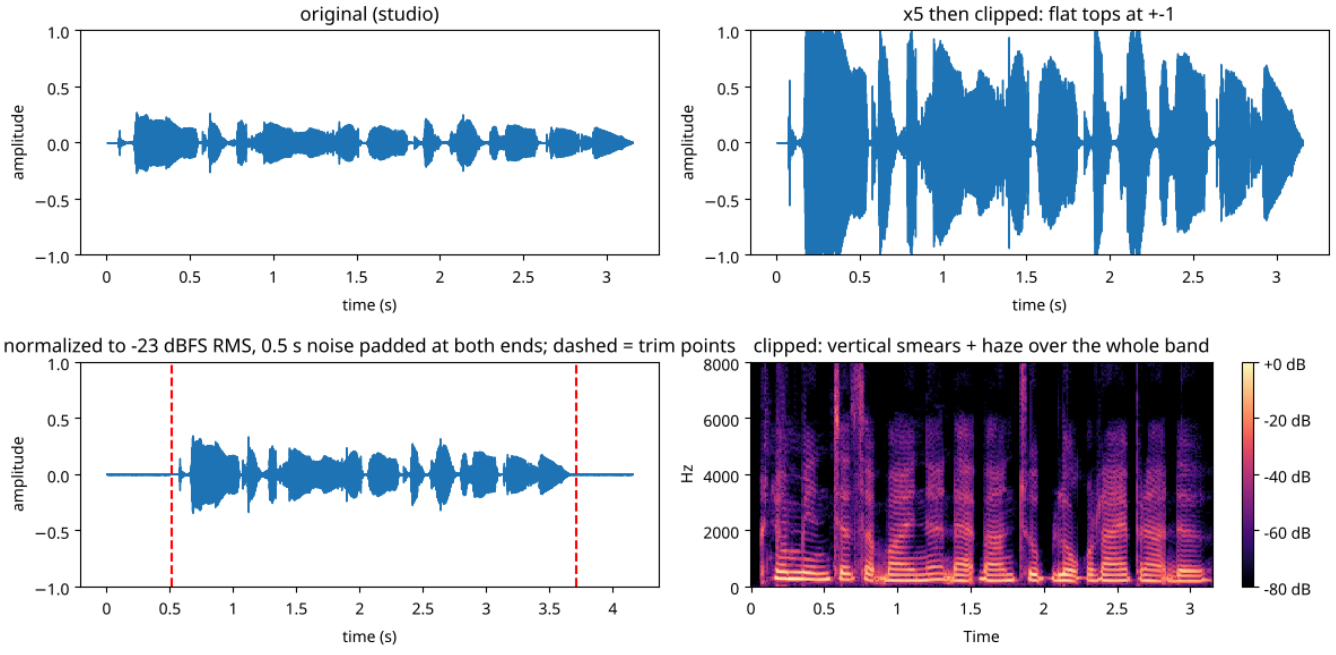
What the output means

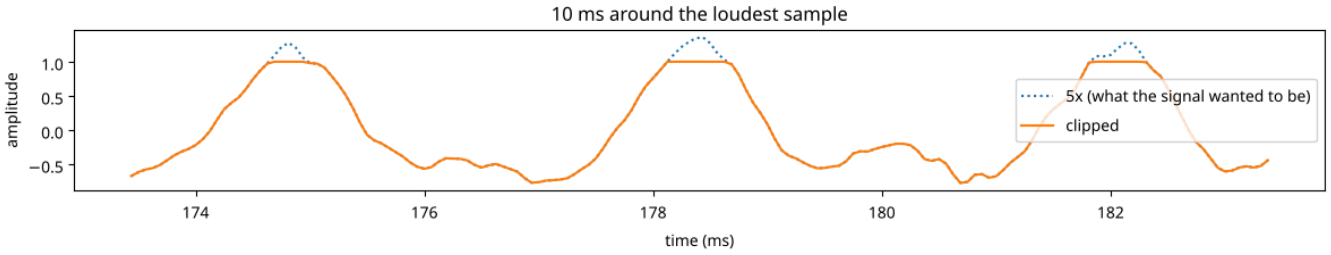
In short clipping = **flat tops** in the waveform and **bright vertical bars to 8 kHz** in the spectrogram; the red dashed lines show where trim cuts.

- No text output. See the How-to-read notes below.

```
fig, axes = plt.subplots(2, 2, figsize=(12, 6))
show_wave(x, title="original (studio)", ax=axes[0, 0])
show_wave(clipped, title="x5 then clipped: flat tops at +-1", ax=axes[0, 1])
show_wave(x_padded, title=f"normalized to {TARGET:.0f} dBFS RMS, 0.5 s noise padded at both ends; dashed = trim points",
ax=axes[1, 0])
axes[1, 0].axvline(i0 / SR, c="r", ls="--"); axes[1, 0].axvline(i1 / SR, c="r", ls="--")
show_spec(clipped, title="clipped: vertical smears + haze over the whole band", ax=axes[1, 1])
plt.tight_layout()

# zoom on a clipped peak
peak_i = int(np.argmax(np.abs(x)))
fig, ax = plt.subplots(figsize=(12, 2.5))
ax.plot(np.arange(peak_i - 80, peak_i + 80) / SR * 1000, 5 * x[peak_i - 80: peak_i + 80], ":", label="5x (what the signal
wanted to be)")
ax.plot(np.arange(peak_i - 80, peak_i + 80) / SR * 1000, clipped[peak_i - 80: peak_i + 80], label="clipped")
ax.set(title="10 ms around the loudest sample", xlabel="time (ms)", ylabel="amplitude"); ax.legend(); plt.tight_layout()
```





Additional Reading

In short clipping can't be undone and sprays energy across all frequencies; normalize by RMS, and trim the silent ends.

How to read this plot. Top-left is the raw studio clip, peak 0.27 (-11.3 dBFS), RMS -25.1 dBFS. Top-right: after x5 the loud syllables at 0.2-0.5 s, 1.0 s and 2.1 s are sliced flat at +1 and -1; only 0.3 % of samples sit on the rail, yet the RMS jumped to -11.2 dBFS and the sound is audibly crunchy. Bottom-left: normalization to -23 dBFS only needed a gain of x1.27 (same shape, slightly taller, peak -9.2 dBFS, still safe), and the red dashed lines show that `trim` cut the 0.5 s of padded noise off each end almost exactly (it keeps 0.51 s to 3.71 s of the 4.15 s padded clip) while removing only 0.03 s from the original clip, which was already tight. Bottom-right: the clipped clip's spectrogram has bright vertical bars reaching to 8 kHz exactly where the top-right plot has flat tops; that is the "every frequency" energy of a sharp corner. The zoomed last plot shows the dotted curve the signal wanted to follow and the flat roof it got.

6. From one spectrum to the spectrogram (STFT)

One frame. Take 25 ms of speech (400 samples at 16 kHz). Speech is *quasi-stationary*: over 25 ms the mouth has not moved much, so the frame is almost a fixed sound. The FFT measures how much of each frequency is in it:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, w[n]\, e^{-2\pi i k n / N}, \qquad f_k = k \cdot \frac{f_s}{N}$$

`w[n]` is a **window**. Without one (a rectangular window) the frame starts and stops abruptly and those two edges smear every sharp peak into wide skirts, **spectral leakage**. A **Hann window** fades the frame in and out and keeps the peaks sharp.

Many frames. Slide the frame forward by a hop of 10 ms (160 samples) and stack the spectra as columns: that is the **short-time Fourier transform (STFT)** and its magnitude in dB is the **spectrogram**. Our parameters:

- `win_length = 400` (25 ms): how much signal each column sees.
- `hop_length = 160` (10 ms): time between columns, so 100 columns per second.
- `n_fft = 512` : FFT size (the 400-sample window is zero-padded to 512); gives `512/2 + 1 = 257` frequency bins, each `16000/512 = 31.25 Hz` wide, from 0 to 8000 Hz.

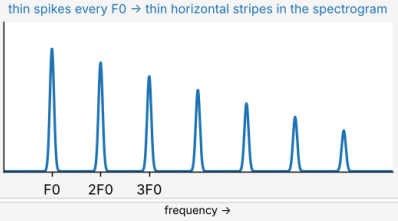
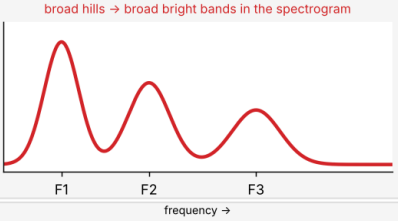
Trade-off. A longer window gives finer frequency resolution (bins are fs/N wide) but blurs time; a shorter window is sharp in time but smears frequency. 25 ms is the speech compromise: long enough to see pitch harmonics, short enough to catch a 50 ms consonant.

Why it matters: the spectrogram *is* the input image for everything downstream; every parameter here (frame size, hop) becomes the model's time resolution.

Visualization Version

1st image is the summary. The 2nd shows how one spectrum becomes one column of the spectrogram.



	HARMONICS the source: vocal-fold buzz	FORMANTS the filter: mouth resonances
Names	1F0, 2F0, 3F0, ... (also written H1, H2, H3)	F1, F2, F3
Where	exact multiples of F0 125, 250, 375, 500 Hz ...	wherever the mouth resonates e.g. 500, 1500, 2500 Hz (not multiples)
Made by	the vocal folds snapping open and shut	the throat, tongue, jaw and lips (the shape of the tube)
Look like	thin spikes every F0 → thin horizontal stripes in the spectrogram 	broad hills → broad bright bands in the spectrogram 
Tell you	pitch (and partly who)	the vowel (and partly who)

real vowel spectrum = harmonic spikes (positions from F0) × formant envelope (heights from the mouth)

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short take **one 25 ms frame** of a vowel, compute its spectrum **with and without a Hann window**, and read the pitch off the harmonics.

- Pick the loudest low-frequency frame of the studio clip (a vowel) at 0.94 s.
- FFT of the raw frame (rectangular window) and of the frame × Hann window.
- Find the peaks below 1500 Hz and measure the spacing between them.

What the output means

In short with Hann the peaks are clean needles at **219, 406, 625, 812 ... Hz**, about **203 Hz apart** = the pitch F0 (pyworld agrees: 203 Hz).

- `Peaks (Hz): [219 406 625 812 1031 1250 1438]` : these are the **harmonics** (1×, 2×, 3× ... F0).
- `mean spacing between peaks = 203 Hz ~ F0` : the gap between harmonics *is* the pitch.
- `frequency bin width = 31.25 Hz, bins = 257` : 16000 / 512 = 31.25 Hz per bin; 512/2 + 1 = 257 bins from 0 to 8 kHz.

```
x = load(CLIPS["studio (female)"])
S = stft_db(x)
hz = librosa.fft_frequencies(sr=SR, n_fft=N_FFT)
# pick the loudest low-frequency frame of the clip as "a vowel frame"
fr = int(np.argmax(S[hz < 1000].mean(0)))
t0 = fr * HOP
```

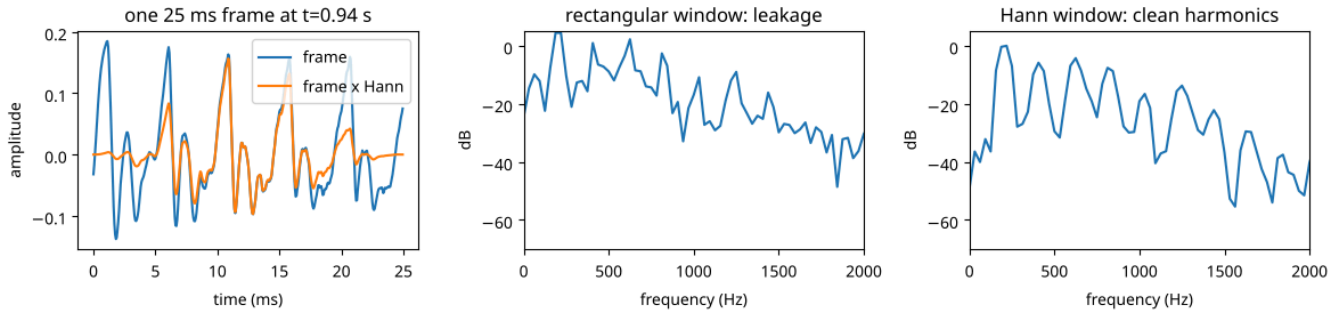
```
frame = x[t0: t0 + WIN]
hann = scipy.signal.get_window("hann", WIN)

spec_rect = np.abs(np.fft.rfft(frame, N_FFT))
spec_hann = np.abs(np.fft.rfft(frame * hann, N_FFT))
f = np.fft.rfftfreq(N_FFT, 1 / SR)

fig, axes = plt.subplots(1, 3, figsize=(12, 3))
axes[0].plot(np.arange(WIN) / SR * 1000, frame, label="frame"); axes[0].plot(np.arange(WIN) / SR * 1000, frame * hann,
label="frame x Hann")
axes[0].set(title=f"one 25 ms frame at t={t0/SR:.2f} s", xlabel="time (ms)", ylabel="amplitude"); axes[0].legend()
for ax, sp, lab in [(axes[1], spec_rect, "rectangular window: leakage"), (axes[2], spec_hann, "Hann window: clean
harmonics")]:
    ax.plot(f, 20 * np.log10(sp / spec_hann.max() + 1e-6)); ax.set(title=lab, xlabel="frequency (Hz)", ylabel="dB", xlim=
(0, 2000), ylim=(-70, 5))
plt.tight_layout()

# The harmonic spacing in the Hann spectrum is the pitch
peaks, _ = scipy.signal.find_peaks(20 * np.log10(spec_hann + 1e-6), distance=4, prominence=8)
peaks = peaks[f[peaks] < 1500]
print(f"frame at {t0/SR:.2f} s. Peaks (Hz): {np.round(f[peaks]).astype(int)}")
print(f"mean spacing between peaks = {np.mean(np.diff(f[peaks])):.0f} Hz  ~  F0.  pyworld median F0 of clip =
{median_f0(x):.0f} Hz")
print(f"frequency bin width = {SR/N_FFT:.2f} Hz, bins = {len(f)}")
```

```
frame at 0.94 s. Peaks (Hz): [ 219  406  625  812 1031 1250 1438]
mean spacing between peaks = 203 Hz  ~  F0.  pyworld median F0 of clip = 203 Hz
frequency bin width = 31.25 Hz, bins = 257
```



Additional Reading

In short the Hann window turns muddy peaks into clean harmonic needles; their spacing is the pitch.

How to read this plot. Left: 25 ms of a vowel; the orange version has been multiplied by the Hann window so it starts and ends at zero. Middle (rectangular window): there are peaks, but each peak has wide skirts and the valleys between them only drop 10-20 dB, so the picture is muddy. Right (Hann): the same peaks, now clean needles with 20-30 dB valleys, at 219, 406, 625, 812, 1031, 1250, 1438 Hz. They are evenly spaced about 203 Hz apart, and that spacing is the pitch (pyworld agrees: median F0 of this clip = 203 Hz): the peaks are the **harmonics**, at 1x, 2x, 3x ... F0. The first three harmonics (200-650 Hz) are the tallest: that hill is the first formant (section 9). The peak positions are multiples of the 31.25 Hz bin width, which is why 203 × 2 = 406 and not 405.

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short compute the full **STFT** of the studio clip and draw the spectrograms of all 3 clips.

- `librosa.stft(..., n_fft=512, hop_length=160, win_length=400, window='hann')` → a complex matrix of shape (bins, frames).

What the output means

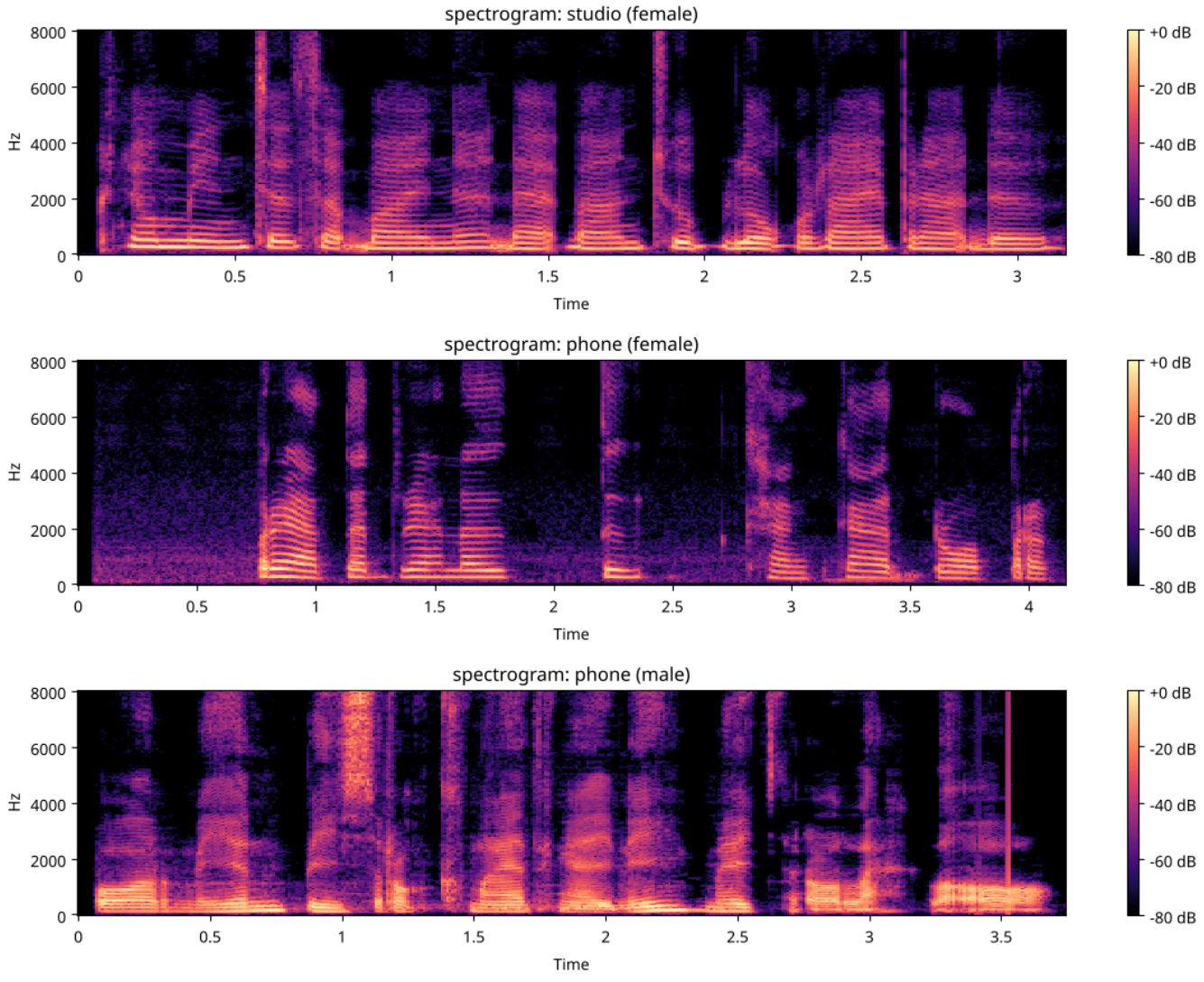
In short the 3.15 s studio clip becomes a **(257, 316)** grid: 257 frequency bins × 316 frames (one every 10 ms).

- `STFT (257, 316) (bins x frames); 1 frame per 10 ms, bin width 31.25 Hz`.
- `expected frames = 1 + len(x)//hop = 316`: librosa pads so the first frame is centred at t = 0.
- The 3 spectrograms are explained in detail just below: **this is the one skill to take from today.**

```
# From one frame to all frames: the STFT.
D = librosa.stft(x, n_fft=N_FFT, hop_length=HOP, win_length=WIN, window="hann")   # complex (257, T)
S_db = librosa.amplitude_to_db(np.abs(D), ref=np.max)                              # magnitude in dB, max = 0
print(f"signal {len(x)} samples = {len(x)/SR:.2f} s  ->  STFT {D.shape}  (bins x frames);  1 frame per {HOP/SR*1000:.0f}
ms, bin width {SR/N_FFT:.2f} Hz")
print(f"expected frames = 1 + len(x)//hop = {1 + len(x)//HOP}  (librosa pads so the first frame is centred at t=0)")

fig, axes = plt.subplots(3, 1, figsize=(12, 9))
for ax, (name, p) in zip(axes, CLIPS.items()):
    show_spec(load(p), title=f"spectrogram: {name}", ax=ax)
plt.tight_layout()
```

```
signal 50475 samples = 3.15 s  ->  STFT (257, 316)  (bins x frames);  1 frame per 10 ms, bin width 31.25 Hz
expected frames = 1 + len(x)//hop = 316  (librosa pads so the first frame is centred at t=0)
```



Additional Reading

In short stacked horizontal stripes = harmonics (pitch), broad bright bands = formants (vowel), vertical smears = fricatives, haze in gaps = noise.

How to read this plot (the one skill to take from today). Time runs left to right, frequency bottom to top (0 to 8 kHz), colour is energy in dB (bright yellow = 0 dB = loudest point in the clip, black = 80 dB below it).

- **Horizontal stripes low down, stacked like a comb**: pitch harmonics of a voiced sound (vowels, nasals). The gap between stripes is F0. The two female clips (rows 1-2) have stripes about 200-245 Hz apart, so a coarse comb that is easy to count (in row 2 at 0.8-1.0 s you can count 8 stripes below 2 kHz); the male clip (row 3) has stripes about 125 Hz apart, a finer comb that looks almost solid below 1 kHz. Where the stripes bend (row 2 at 1.4-1.8 s, rising then falling) the speaker's pitch is moving: intonation.
- **Bright hills or bands across several harmonics** (one around 300-800 Hz, one around 1-2.5 kHz): formants, the resonances of the mouth shape. They move as the vowel changes; in row 3 the band at 1-1.5 kHz around 3.3-3.6 s is a different vowel from the one at 0.1-0.3 s with its energy at 500-1000 Hz.
- **Vertical smears reaching to the top with no stripes**: fricatives (s, h, ch): noise, no pitch. Row 2 at 0.75 s is a textbook one (bright from 2 to 8 kHz right before the first vowel); row 3 at 1.0-1.2 s and 2.65 s are others. The thin bright vertical line at 3.52 s in row 3 is a click, not speech.
- **Uniform haze in the gaps between words**: background noise. The studio clip (row 1) has no gaps at all; the phone-female clip (row 2) shows it best: the first 0.75 s is a purple haze below about 1.5 kHz, the phone's low-frequency room noise. The studio harmonics stay visible up to 6 kHz; the phone clips lose their stripes above about 4 kHz, where only noise remains.

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short draw the same 1 s of the male clip with **10 ms, 25 ms and 100 ms** windows.

- Only the window length changes; everything else is the same.

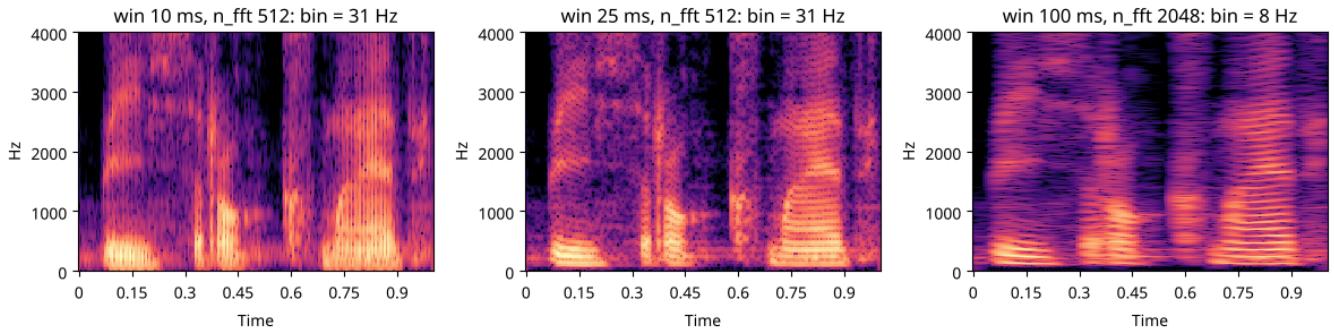
What the output means

In short short windows are sharp in **time**, long windows are sharp in **frequency**; **25 ms** shows both, which is why it's the default.

- No text output. See the How-to-read notes below.

```
# Time-frequency trade-off: same 1 s of the male clip with 10 ms, 25 ms and 100 ms windows.
x = load(CLIPS["phone (male)"])
```

```
seg = x[int(0.8 * SR): int(1.8 * SR)]
fig, axes = plt.subplots(1, 3, figsize=(12, 3.2))
for ax, win_ms in zip(axes, [10, 25, 100]):
    win = int(win_ms / 1000 * SR); nfft = max(512, 2 ** int(np.ceil(np.log2(win))))
    S = librosa.amplitude_to_db(np.abs(librosa.stft(seg, n_fft=nfft, hop_length=80, win_length=win, window="hann")),
ref=np.max)
    img = librosa.display.specshow(S, sr=SR, hop_length=80, x_axis="time", y_axis="hz", ax=ax, cmap="magma", vmin=-80,
vmax=0)
    ax.set(title=f"win {win_ms} ms, n_fft {nfft}: bin = {SR/nfft:.0f} Hz", ylim=(0, 4000))
plt.tight_layout()
```



Additional Reading

In short 10 ms shows pitch pulses but no stripes; 100 ms shows razor stripes but smears onsets; 25 ms is the compromise.

How to read this plot. Left (10 ms window): the individual pitch pulses are visible as thin vertical lines, every 8 ms or so, but there are no harmonic stripes: the window is shorter than one pitch period, so frequency is blurred (each bin is 31 Hz but the effective resolution is about 100 Hz). Right (100 ms window): the harmonics are razor-sharp horizontal lines, but every consonant onset is smeared over 100 ms, and fast pitch changes turn into stairs. Middle (25 ms): both stripes and onsets visible. This is the compromise every speech front-end makes, and it is why `win_length=400` is the default for the rest of the course.

7. Mel and log-mel: what the ear (and the model) hears

The STFT has 257 bins spaced evenly 31 Hz apart. The ear is not even: it can tell 100 Hz from 200 Hz easily but 7000 Hz from 7100 Hz barely. The **mel scale** stretches the low frequencies and squashes the high ones so that equal steps in mel feel like equal steps in pitch:

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right)$$

Worked example. 0 → 0 mel, 1000 Hz → 1000 mel, 4000 Hz → 2146 mel, 8000 Hz → 2840 mel. So the first 1 kHz gets 1000 mel and the last 4 kHz gets only 700 mel. We place 80 triangular filters evenly on the mel axis, multiply the power spectrum by each triangle and sum: 257 bins become 80 **mel bands**, narrow at the bottom, wide at the top.

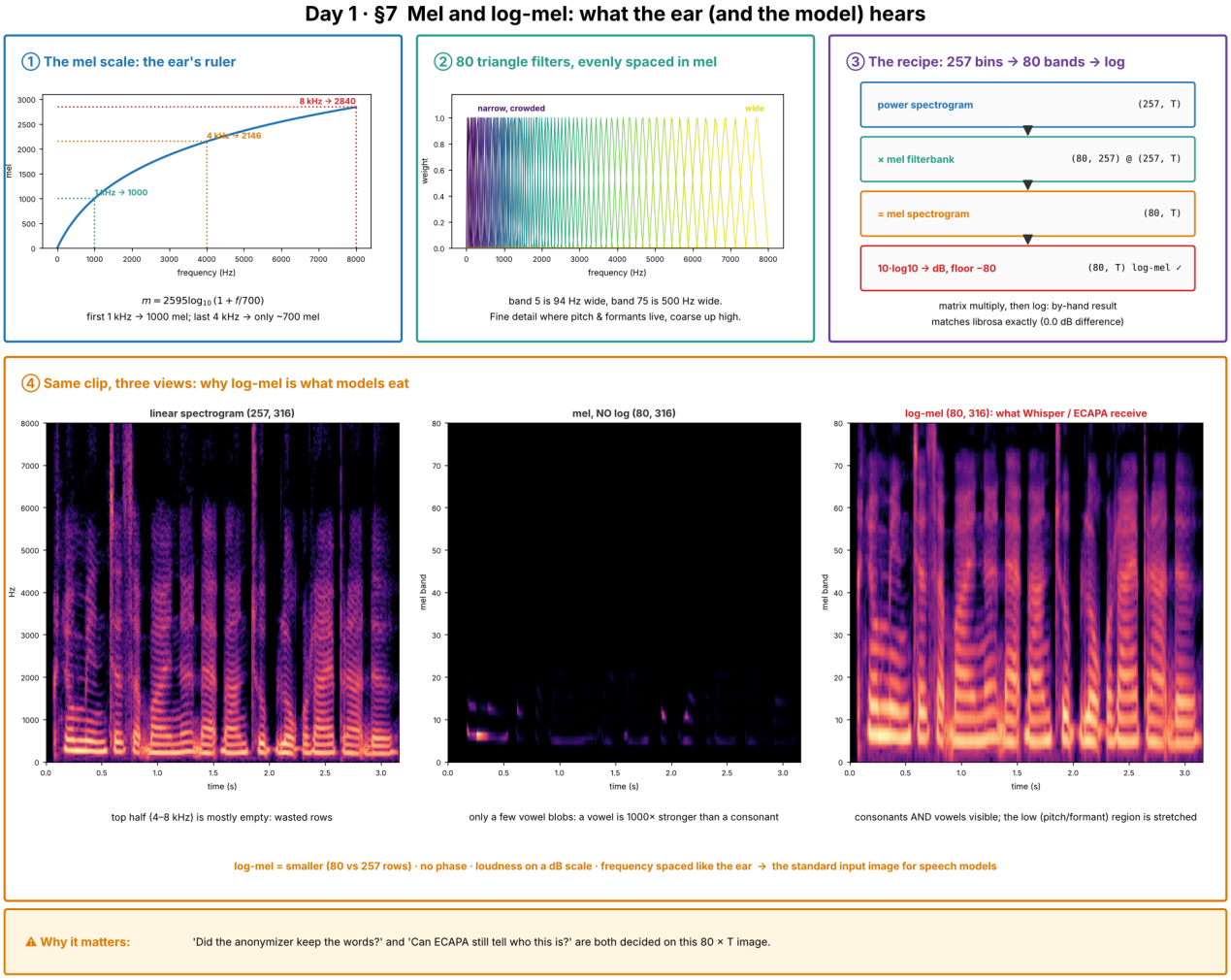
Log. Loudness is also perceived logarithmically. Taking 10 log10 (dB) compresses the range so a quiet consonant at -50 dB is as visible as a loud vowel at 0 dB. Without the log, a network only sees the vowels.

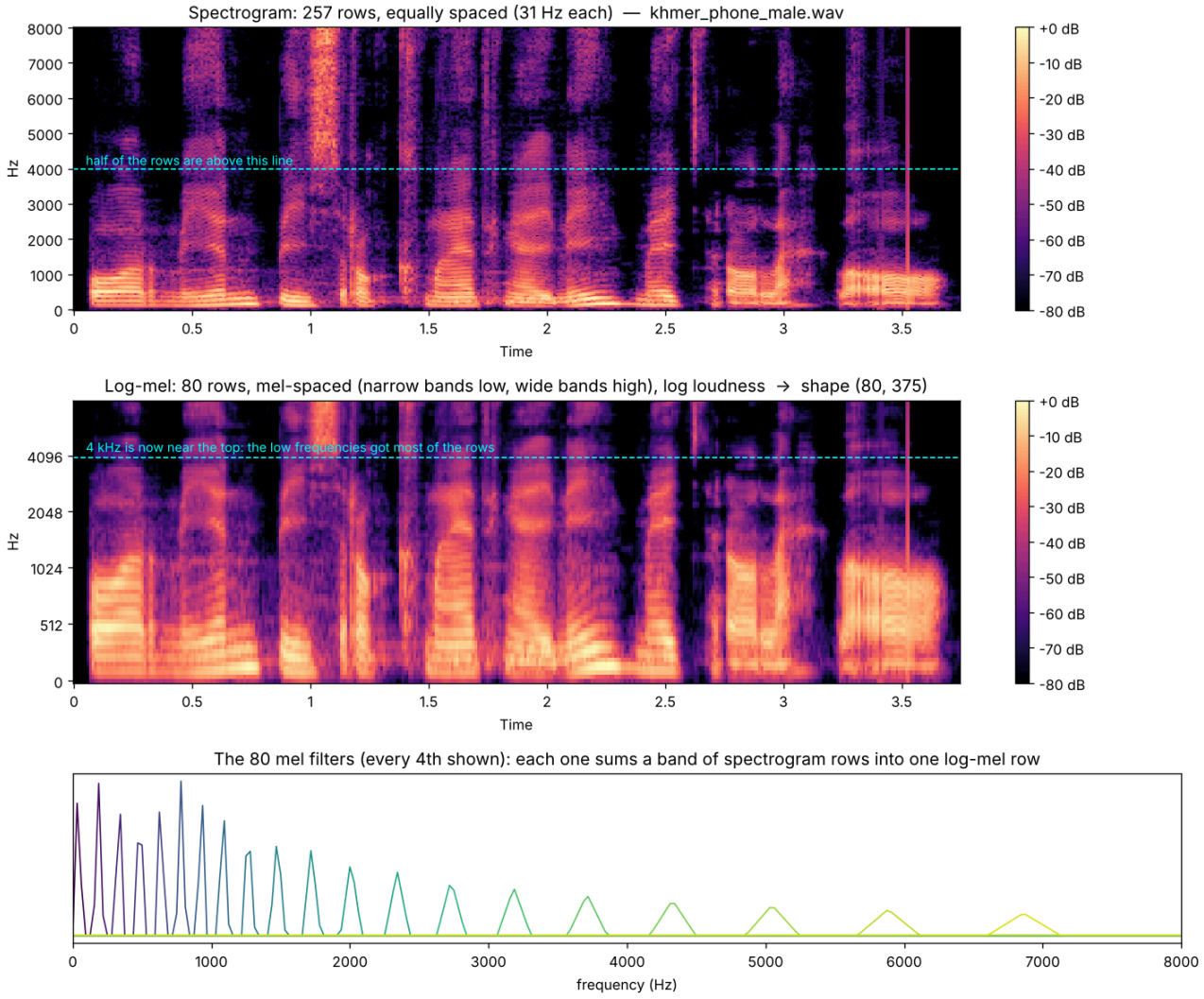
The result, the **log-mel spectrogram** (80 x T, one column per 10 ms), is the input of Whisper, ECAPA-TDNN, HiFi-GAN and most TTS systems. It is smaller than the STFT (80 vs 257 rows), has no phase (models do not need it to recognise content or speaker), and matches human perception.

Why it matters: when we ask "did the anonymizer keep the words?" or "can ECAPA still tell who this is?", the answer is decided on this 80 x T image.

Visualization Version

1st image is the summary. The 2nd compares the 257-row spectrogram with the 80-row log-mel on the male clip.





Additional reading: you can read all of it or just read the "In short"

Log-mel in short

In short log-mel is the spectrogram redrawn the way the ear hears it: an **80 × T** grid, one column every 10 ms.

Log-mel is the spectrogram, redrawn the way your ear hears it. It's an 80 × T grid: 80 frequency bands, one column every 10 ms. How it's made, in three steps on top of the STFT:

- Spectrogram: 257 evenly spaced frequency bins per frame.
- Mel filters: squash those 257 bins into 80 bands, narrow at low frequencies, wide at high ones, because the ear resolves low frequencies finely and high ones coarsely.
- Log: take the logarithm of each value (decibels), because the ear judges loudness by ratios. Why it matters: it's the standard input to speech models, including Whisper and ECAPA. It's also what your Day 2 CNN reads, treated like an 80-row image. Smaller than the raw spectrogram, and shaped to emphasize what humans hear in speech.

Why not feed the model the raw spectrogram?

In short wrong emphasis, wrong loudness scale, and too big. Log-mel fixes all three.

Three problems with feeding a model the raw spectrogram:

- Wrong emphasis. The raw spectrogram gives equal space to every 31 Hz slice, so half its rows cover 4–8 kHz, where speech has almost no information (only about 2 % of the energy in our clips). Mel compresses the top and spreads out the bottom, so the rows go where the vowels, pitch and formants are.
- Wrong loudness scale. Raw power values span a huge range: a vowel can be a million times stronger than a soft consonant. A neural net struggles with that. The log squeezes it into roughly −80 to 0 dB, where a quiet “s” and a loud “ah” are both visible and a 2× change in volume becomes a small constant shift instead of a rescale of everything.
- Too big. 257 rows → 80 rows is three times less data, with nothing useful lost. Cheaper to train, less for the model to overfit on. The one problem it doesn’t solve: it’s still a fixed, hand-designed front-end. WavLM and HuBERT skip it and learn their own features from the raw waveform, but most speech models (Whisper, ECAPA) still use log-mel.

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short build the **80 mel filters** and draw them, plus the mel-scale curve.

- `librosa.filters.mel(sr=16000, n_fft=512, n_mels=80)` → a (80, 257) matrix: one triangle per row.
- Print the centre frequency of a few bands and the width of a low vs a high band.

What the output means

In short low bands are **narrow (94 Hz)**, high bands are **wide (500 Hz)**: fine detail where pitch and formants live, coarse detail up high.

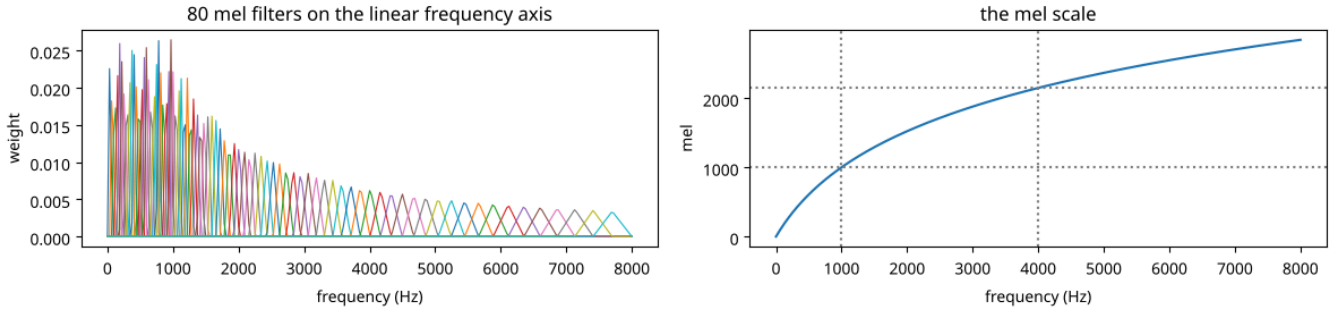
- `filterbank shape (80, 257)` : 80 filters, each covering the 257 STFT bins.
- `centre frequency of mel band 0, 10, 20, 40, 60, 79 (Hz): [37 410 782 1722 3711 7699]` : the first 20 bands cover 0–800 Hz, the last 20 cover 3.7–8 kHz.
- `width of band 5 vs band 75: 94 Hz vs 500 Hz` .

```
mel_fb = librosa.filters.mel(sr=SR, n_fft=N_FFT, n_mels=80)        # (80, 257): one triangle per row
hz = librosa.fft_frequencies(sr=SR, n_fft=N_FFT)
centers = librosa.mel_frequencies(n_mels=80 + 2, fmin=0, fmax=SR / 2)[1:-1]
print("filterbank shape", mel_fb.shape)
print("centre frequency of mel band  0, 10, 20, 40, 60, 79 (Hz):", np.round(centers[[0, 10, 20, 40, 60, 79]]).astype(int))
print("width of band 5 vs band 75:", f"{np.sum(mel_fb[5] > 0) * SR/N_FFT:.0f} Hz vs {np.sum(mel_fb[75] > 0) *
SR/N_FFT:.0f} Hz")

fig, axes = plt.subplots(1, 2, figsize=(12, 3))
for row in mel_fb:
    axes[0].plot(hz, row, lw=0.8)
axes[0].set(title="80 mel filters on the linear frequency axis", xlabel="frequency (Hz)", ylabel="weight")
f_axis = np.linspace(0, 8000, 500)
axes[1].plot(f_axis, 2595 * np.log10(1 + f_axis / 700)); axes[1].set(title="the mel scale", xlabel="frequency (Hz)",
ylabel="mel")
for fq in [1000, 4000]:
```

```
    axes[1].axvline(fq, ls=":", c="gray"); axes[1].axhline(2595 * np.log10(1 + fq / 700), ls=":", c="gray")
plt.tight_layout()
```

```
filterbank shape (80, 257)
centre frequency of mel band  0, 10, 20, 40, 60, 79 (Hz): [  37  410  782 1722 3711 7699]
width of band 5 vs band 75: 94 Hz vs 500 Hz
```



Additional Reading

In short the triangles are thin and crowded at low frequencies and wide at high ones, exactly like the mel curve.

How to read this plot. Left: every coloured triangle is one mel band. Near 0-1000 Hz they are thin and crowded (only 1-2 FFT bins wide, which is why some look like spikes); near 8000 Hz each is several hundred Hz wide. Right: the mel curve is steep at the start and flattens out; the dotted lines show that 1 kHz sits at 1000 mel but 4 kHz only at about 2150 mel. Together they say: the model gets fine detail where the pitch harmonics and formants live, and a coarse summary where fricatives live. The printed widths make the same point in numbers: band 5 is 94 Hz wide, band 75 is 500 Hz wide.

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short compute the **log-mel by hand** (power spectrogram → × mel filters → log) and check it equals librosa's version; then show 3 views of the same clip.

- `mel_power = mel_fb @ P` : (80, 257) @ (257, T) = (80, T).
- `10 * log10(...)` , set the max to 0 dB, floor at −80 dB.
- Plot: linear spectrogram, mel **without** log, and log-mel.

What the output means

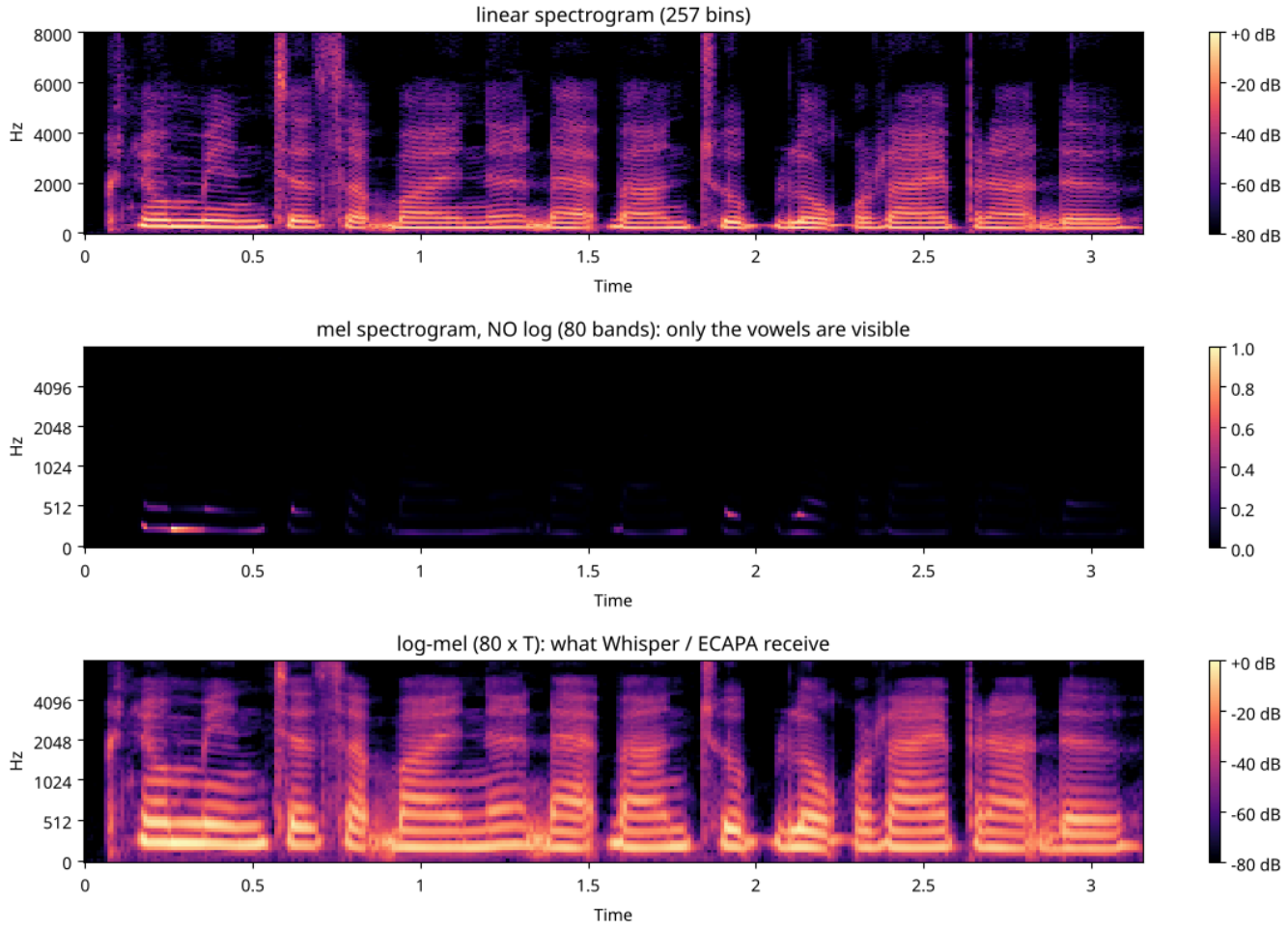
In short the by-hand log-mel matches librosa **exactly (0.0000 dB difference)**; without the log only the loud vowels are visible.

- `STFT power (257, 316) -> mel (80, 316) -> log-mel (80, 316)` : 257 rows become 80, time is unchanged.
- `max |hand - librosa| = 0.0000 dB` : the recipe really is just a matrix multiply and a log.

```
x = load(CLIPS["studio (female)"])
P = np.abs(librosa.stft(x, n_fft=N_FFT, hop_length=HOP, win_length=WIN, window="hann")) ** 2   # power spectrogram (257,
T)
mel_power = mel_fb @ P                                                                          # (80, 257) @ (257, T) =
(80, T)
logmel_hand = 10 * np.log10(np.maximum(mel_power, 1e-10)); logmel_hand -= logmel_hand.max()     # dB, max 0
logmel_hand = np.maximum(logmel_hand, -80)                                                      # floor at -80 dB
(librosa's top_db)
logmel_lib = logmel(x)                                                                          # src.utils helper (same
recipe)
print(f"STFT power {P.shape} -> mel {mel_power.shape} -> log-mel {logmel_hand.shape};  max |hand - librosa| =
{np.abs(logmel_hand - logmel_lib).max():.4f} dB")

fig, axes = plt.subplots(3, 1, figsize=(12, 8))
show_spec(x, title="linear spectrogram (257 bins)", ax=axes[0])
img = librosa.display.specshow(mel_power / mel_power.max(), sr=SR, hop_length=HOP, x_axis="time", y_axis="mel",
ax=axes[1], cmap="magma")
axes[1].set(title="mel spectrogram, NO log (80 bands): only the vowels are visible"); plt.colorbar(img, ax=axes[1])
show_spec(x, title="log-mel (80 x T): what Whisper / ECAPA receive", ax=axes[2], mel=True)
plt.tight_layout()
```

```
STFT power (257, 316) -> mel (80, 316) -> log-mel (80, 316);  max |hand - librosa| = 0.0000 dB
```



Additional Reading

In short linear spectrogram wastes rows on empty high frequencies; mel without log hides consonants; log-mel shows everything at a sensible size.

How to read this plot. Top: the linear spectrogram, most of the picture (4-8 kHz) is nearly empty. Middle: the mel spectrogram *without* the log: a few bright blobs where the vowels are and black everywhere else, because a vowel is 1000x more powerful than a fricative and a linear colour scale cannot show both. Bottom: the log-mel; the y-axis is now mel-spaced (notice the tick labels bunch up toward the top: 512, 1024, 2048, 4096 take the same vertical space), the consonants

and even the noise floor are visible, and the harmonics in the low bands are stretched out. The code confirms the by-hand version (matrix multiply, 10 log10, floor at -80 dB) equals the library version exactly (max difference 0.0 dB). The printed centre frequencies (37, 410, 782, 1722, 3711, 7699 Hz for bands 0, 10, 20, 40, 60, 79) show the stretch: the first 20 bands cover 0-800 Hz, the last 20 cover 3.7-8 kHz.

8. MFCC by hand (and why it is now legacy)

Before neural networks, the standard speech feature was the **MFCC** (mel-frequency cepstral coefficient). Recipe: take the log-mel column (80 values), apply a DCT (discrete cosine transform, a cousin of the FFT that works on real numbers) and keep only the first 13 coefficients:

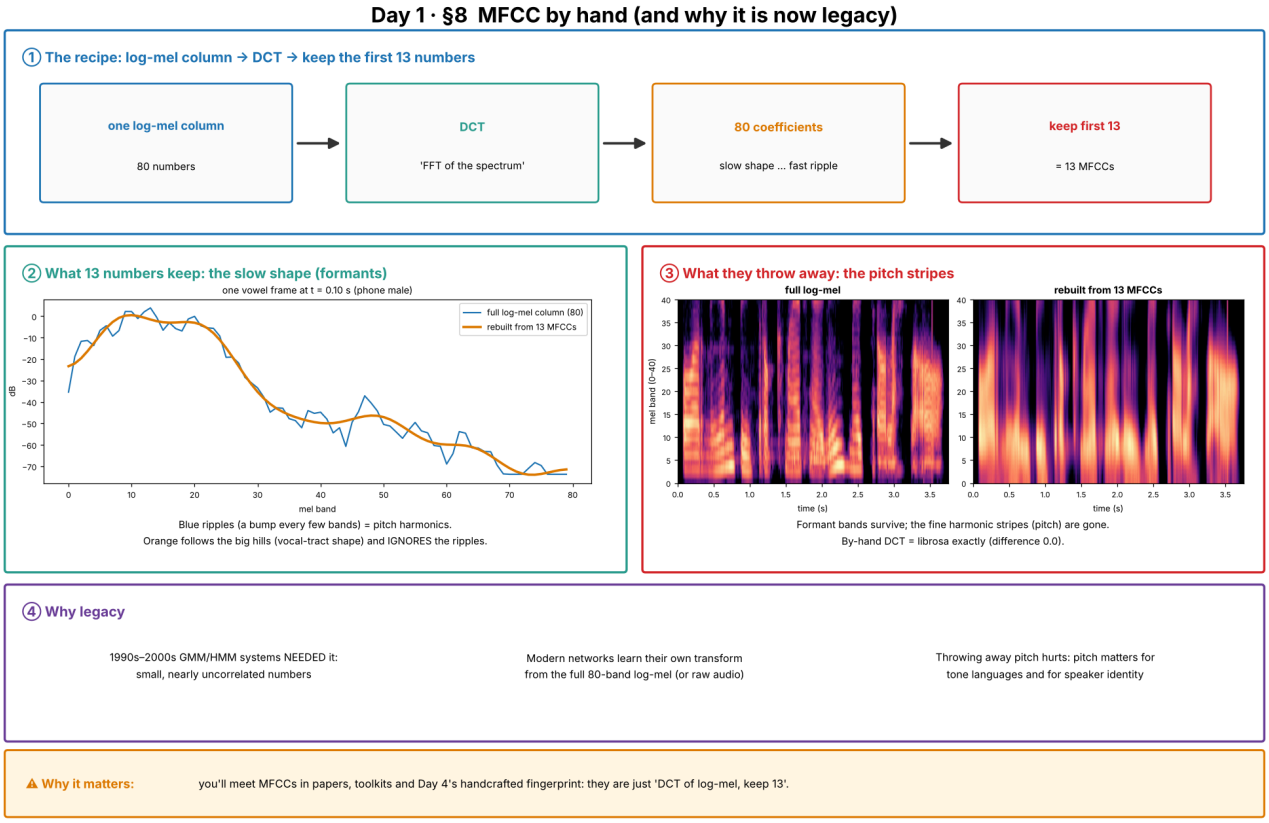
$$c_k = \sum_{m=0}^{M-1} \log S_m \cos\left[\frac{\pi k}{M}\left(m + \tfrac{1}{2}\right)\right], \quad k = 0 \ldots 12$$

What this does: the log-mel column is a wiggly curve with a slow *shape* (the formants = the vocal tract) plus fast ripples (the pitch harmonics). A DCT is like an FFT *of the spectrum*: the first few coefficients describe the slow shape, the later ones the fast ripple. Keeping 13 keeps the vocal-tract shape and throws away pitch. The coefficients are also nearly uncorrelated, which GMM/HMM systems of the 1990s-2000s needed.

Why legacy: a neural network can learn whatever transform of the log-mel it wants, and throwing away 67 of 80 numbers (including pitch, which matters for tone languages and for speaker identity) is a loss it does not need. Modern models take the 80-bin log-mel directly, or raw waveform. MFCCs survive in lightweight keyword spotting and in old pipelines, and `c_0` is still a handy energy feature.

Why it matters: you will meet MFCCs in papers and in evaluation toolkits; knowing they are "DCT of log-mel, keep 13" means you can read them without fear.

Visualization Version



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short compute **13 MFCCs** by hand (log-mel → DCT → keep 13), check against librosa, then rebuild the log-mel from only those 13 numbers.

- `scipy.fft.dct(L, type=2, norm='ortho')[:13]` : the DCT of each log-mel column, first 13 rows kept.
- `idct` with the other 67 rows set to zero → what 13 numbers can still describe.

What the output means

In short the by-hand MFCCs equal librosa's **exactly**; 13 numbers keep the **formant hills** but lose the **pitch stripes**.

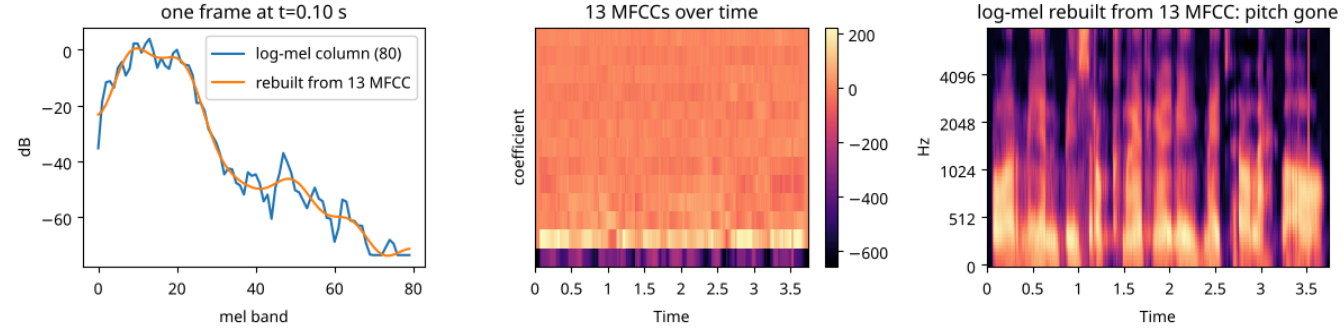
- `log-mel (80, 375) -> MFCC (13, 375); max |hand - librosa| = 0.00e+00` : 80 numbers per frame become 13.

```
x = load(CLIPS["phone (male)"])
# librosa's recipe: mel power -> power_to_db with ref=1.0 (NOT ref=max) -> DCT type 2, orthonormal -> first 13 rows
M = librosa.feature.melspectrogram(y=x, sr=SR, n_fft=N_FFT, hop_length=HOP, win_length=WIN, n_mels=80, power=2.0)
L = librosa.power_to_db(M, ref=1.0)                                          # (80, T) in dB
mfcc_hand = scipy.fft.dct(L, type=2, axis=0, norm="ortho")[:13]               # (13, T)
```

```
mfcc_lib = librosa.feature.mfcc(y=x, sr=SR, n_mfcc=13, n_fft=N_FFT, hop_length=HOP, win_length=WIN, n_mels=80)
print(f"log-mel {L.shape} -> MFCC {mfcc_hand.shape};  max |hand - librosa| = {np.abs(mfcc_hand - mfcc_lib).max():.2e}")

# Reconstruct the log-mel from only 13 coefficients to see what was thrown away
L_from13 = scipy.fft.idct(np.pad(mfcc_hand, ((0, 80 - 13), (0, 0))), type=2, axis=0, norm="ortho")
fig, axes = plt.subplots(1, 3, figsize=(12, 3.2))
fr = int(np.argmax(L[:20].mean(0)))                                          # a loud voiced frame
axes[0].plot(L[:, fr], label="log-mel column (80)"); axes[0].plot(L_from13[:, fr], label="rebuilt from 13 MFCC")
axes[0].set(title=f"one frame at t={fr*HOP/SR:.2f} s", xlabel="mel band", ylabel="dB"); axes[0].legend()
img = librosa.display.specshow(mfcc_lib, sr=SR, hop_length=HOP, x_axis="time", ax=axes[1], cmap="magma");
axes[1].set(title="13 MFCCs over time", ylabel="coefficient"); plt.colorbar(img, ax=axes[1])
img = librosa.display.specshow(L_from13 - L_from13.max(), sr=SR, hop_length=HOP, x_axis="time", y_axis="mel", ax=axes[2],
cmap="magma", vmin=-80, vmax=0); axes[2].set(title="log-mel rebuilt from 13 MFCC: pitch gone")
plt.tight_layout()
```

```
log-mel (80, 375) -> MFCC (13, 375);  max |hand - librosa| = 0.00e+00
```



Additional Reading

In short 13 MFCCs keep the smooth vocal-tract shape and throw away the pitch harmonics.

How to read this plot. Left: the blue curve is one log-mel column of a vowel; its ripples (a bump every few bands in the low bands) are the pitch harmonics; the orange curve rebuilt from 13 MFCCs follows the big hills (the formants) and ignores the ripples. Middle: the 13 coefficients over time; row 0 is energy (bright on vowels, dark on pauses), rows 1-12 change with the vowel. Right: the log-mel rebuilt from those 13 numbers; compare it with the full log-mel: the formant bands are still there, the fine harmonic stripes are gone. That is exactly what MFCCs keep and what they discard. The by-hand DCT matches librosa to floating-point precision.

9. Pitch (F0) and formants: a quick look

Two numbers describe a voiced sound:

- **F0, the pitch**: how fast the vocal folds vibrate. Adult males 85-155 Hz, adult females 165-255 Hz (typical ranges, individuals overlap). It is set by the size of the larynx, so it is a strong gender and speaker cue. `f0_track` (WORLD's DIO + StoneMask from `pyworld`) returns one F0 value per 10 ms, 0 where the frame is unvoiced.

F0 is the physical measurement: how many times per second the vocal folds vibrate, in Hz. It's what median_f0 and the pitch tracker compute.

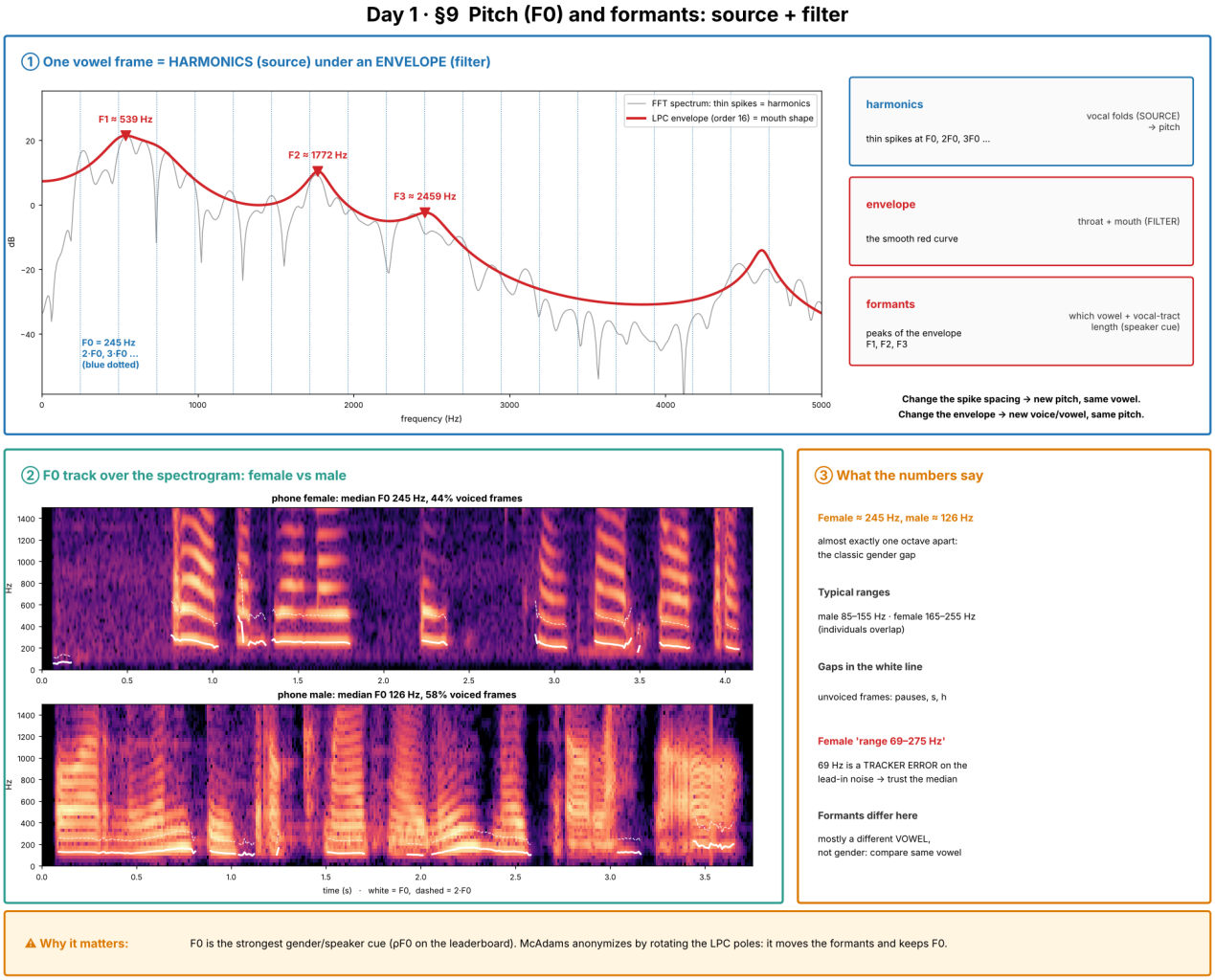
- **Formants F1, F2, F3**: the resonance peaks of the vocal tract (throat + mouth), the *hills* under the harmonics. F1 and F2 identify the vowel (for "ii" F1 is low and F2 high; for "aa" both are mid); their absolute positions scale with vocal-tract length, another speaker cue.

LPC (linear prediction) fits an all-pole filter to one frame: it predicts each sample from the previous `p` samples (here `p = 16`). The frequency response of that filter is a smooth envelope over the spectrum whose peaks are the formants:

$$\hat{x}[n] = -\sum_{k=1}^{p} a_k x[n-k], \qquad H(f) = \frac{1}{|A(e^{2\pi i f/f_s})|}$$

Why it matters: the McAdams anonymizer in this course works by rotating the poles of exactly this LPC filter, which shifts the formants while keeping F0, so section 9 is the inside of one of our baselines. And F0 is what the `rho_f0` column of the leaderboard measures.

Visualization Version



Additional reading: you can read all of it or just read the "In short"

In short LPC predicts each sample from the previous ones; the prediction filter is a model of the mouth, and its peaks are the formants.

LPC

LPC in short: predict each audio sample from the previous 16 samples (order 16 in the code below). The 16 weights that do this best describe a smooth filter, and that filter is a model of the mouth. Plot it and you get the smooth spectral envelope; its hills are the formants.

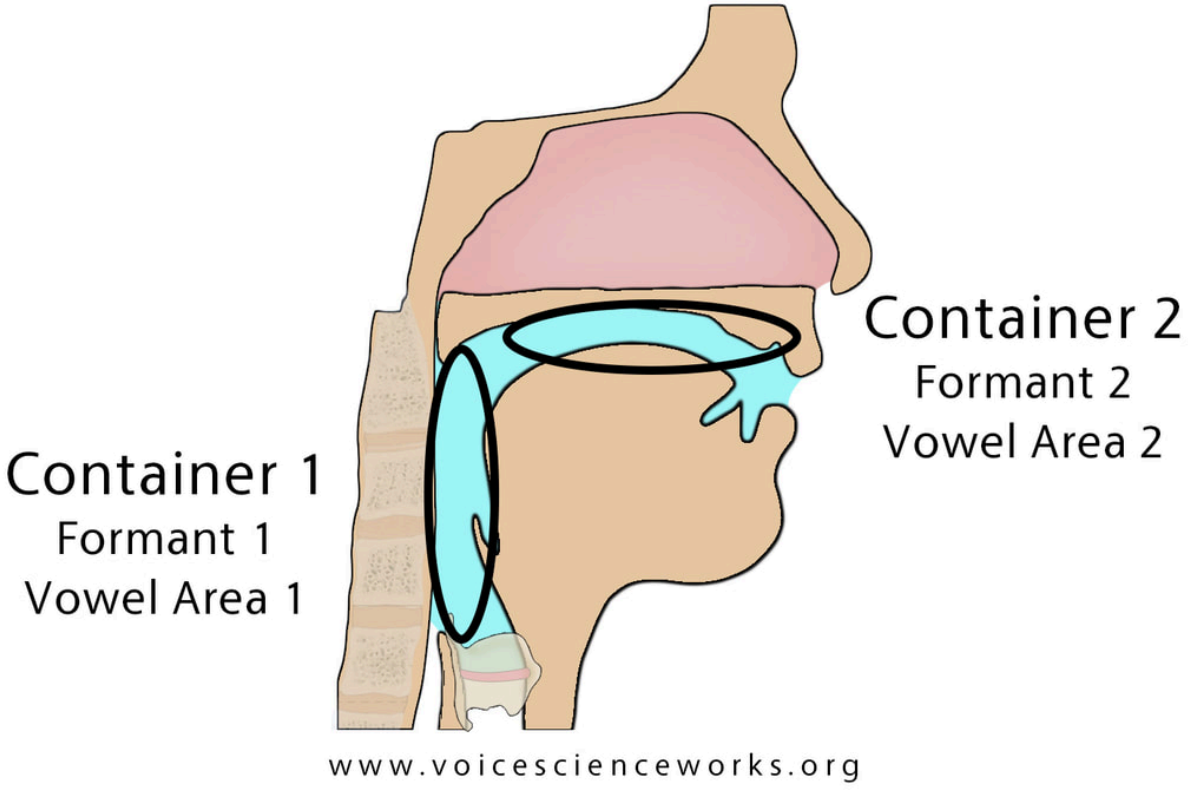
What it gives you:

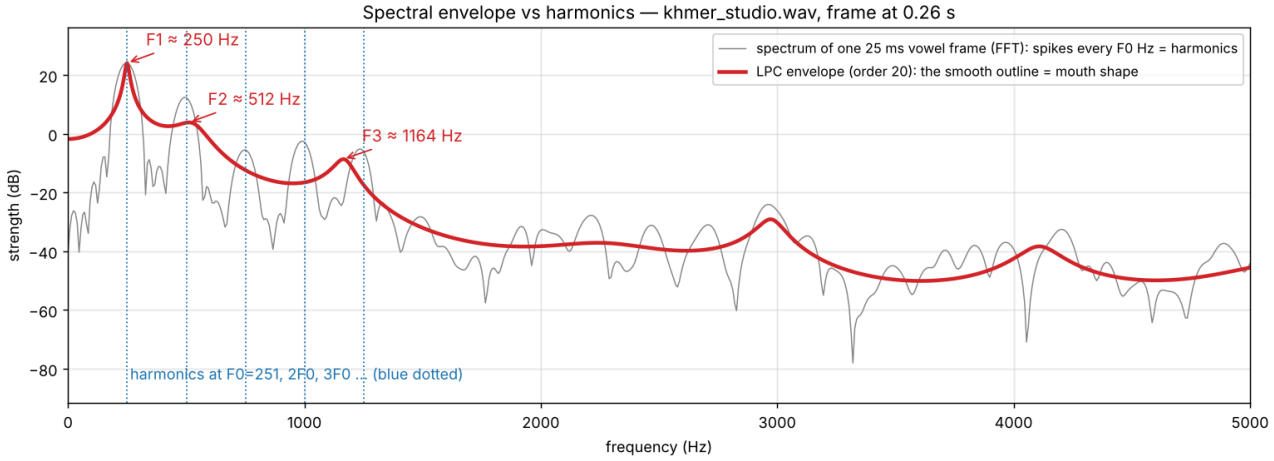
Envelope / filter = mouth shape (F1, F2, F3) Residual (what it couldn't predict) = the vocal-fold buzz (F0) Used in Day 1 to find formants, and in McAdams (Day 6) to move them.

We use LPC to find formants:

It's the standard way: run LPC on a 25 ms voiced frame, plot the envelope, and the peaks are F1, F2, F3. Praat does exactly this under the hood. Two alternatives you'll meet: Reading the spectrogram by eye: the broad bright bands are the formants. Fine for understanding, not for numbers. Neural models (WavLM, ECAPA) never compute formants explicitly. They learn whatever they need from log-mel or raw audio, where the formants are implicit in the envelope. So LPC = the classical, explicit formant finder, and it's the one McAdams builds on.

LPC is the tool for finding and editing formants explicitly, and you need it when you want to work with formants. A neural model doesn't ask for it.





Reading the envelope picture

In short grey spikes = harmonics (source), red curve = envelope (filter), peaks of the red curve = formants.

- Grey jagged line = the FFT spectrum. 1025 dots (one strength value per frequency slot) joined into one grey line = one spectrum, for that one 25 ms frame.
- The spikes at the blue dotted lines are the harmonics: 251 Hz (F0), 502, 753, 1004… That's the vocal-fold buzz, spaced exactly F0 apart.
- Red smooth curve = the LPC envelope. It traces over the tops of the spikes and ignores the gaps between them. That's the mouth shape. Hills in the red curve = formants: F1 ≈ 250 Hz, F2 ≈ 512 Hz, F3 ≈ 1164 Hz, then smaller ones near 3000 and 4100 Hz.

Two things to notice:

- The envelope and the harmonics are independent layers. If this speaker said the same vowel at a lower pitch, the grey spikes would move closer together, but the red curve would stay the same.
- F1 and F2 are close together and low, which means a vowel with a fairly closed mouth, probably something like "o" or "u". (Fair warning: here F1 sits right on the first harmonic, so LPC's peak may be partly riding on the harmonic. That's a known limitation when F0 is high.)
- envelope = the whole smooth curve (the mouth's shape)
- formants = the peaks of that curve (F1, F2, F3)
- harmonics = the thin spikes under the curve (F0, 2F0, 3F0)

so harmonic the spike of the actual line which is the frequency itself, draw out a smooth line of the harmonic and then the formant is the peak of the envelope

Harmonics = the thin spikes in the measured spectrum, at F0, 2F0, 3F0…

Envelope = the smooth outline over those spikes.

Formants = the peaks of that envelope.

Where harmonics come from

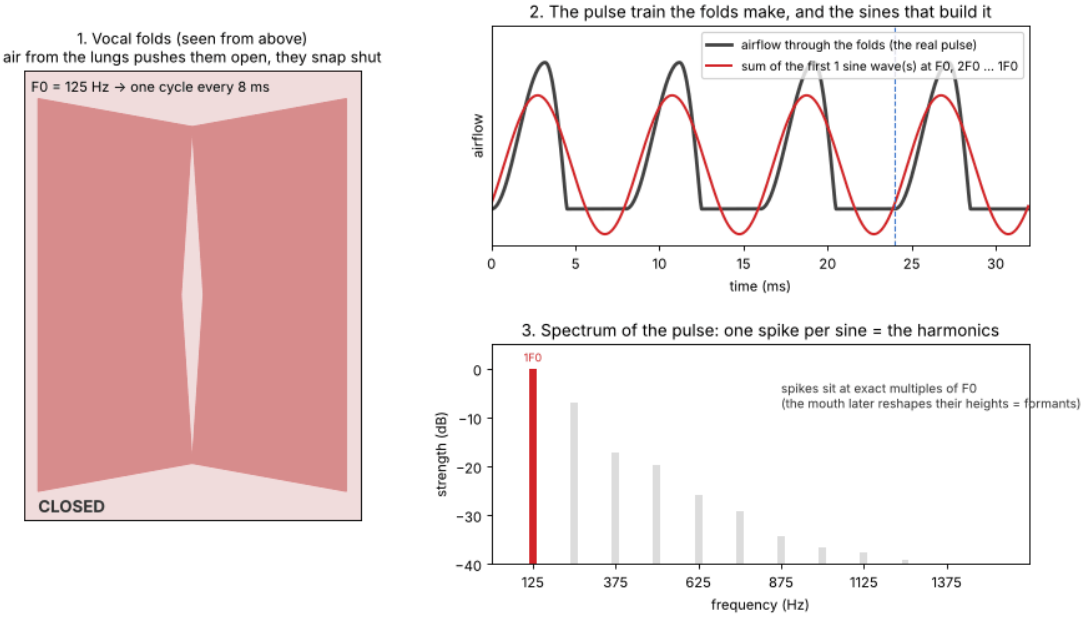
In short the vocal folds make a repeating buzz; any repeating pulse is a sum of sines at F0, 2·F0, 3·F0 …; each sine is one harmonic spike.

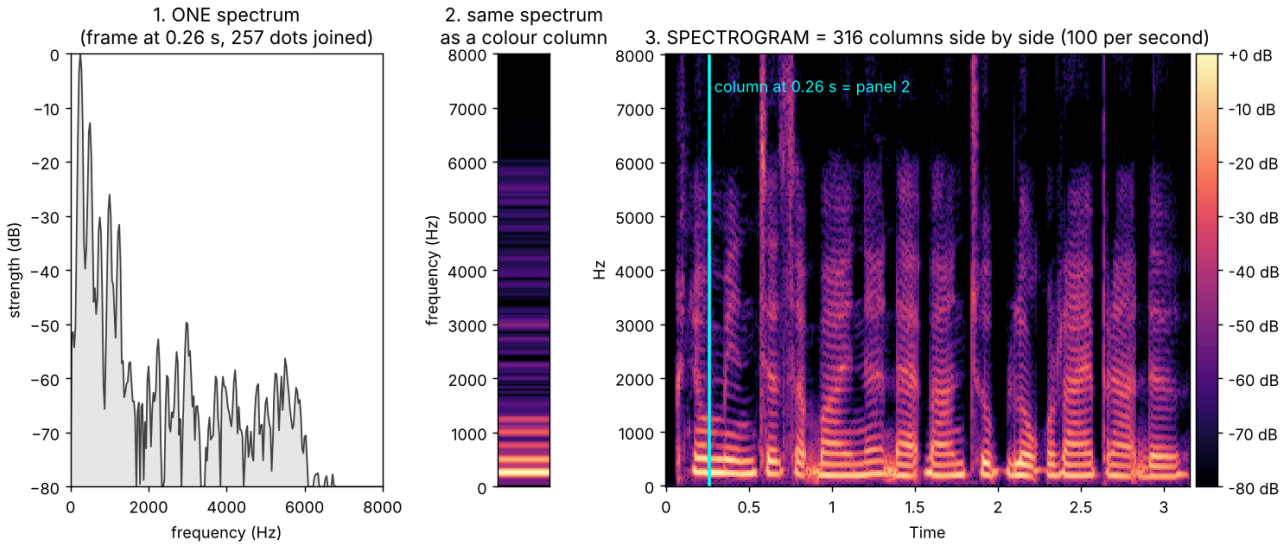
A harmonic spike is one of the sharp peaks in a vowel's spectrum, and they sit at exact multiples of the pitch: F0, 2×F0, 3×F0, 4×F0… Where they come from: the vocal folds don't produce a pure sine wave. They snap open and shut in a sharp, repeating pulse, like a buzz. Any repeating pulse is made of a sine at the repeat rate (F0) plus sines at every whole multiple of it. Those multiples are the harmonics, and each one shows up as a spike in the spectrum.

In your picture: F0 was 251 Hz, so the spikes land at 251, 502, 753, 1004, 1255 Hz… The spacing between neighbouring spikes is always F0. That's why you can read pitch off a spectrogram: closely packed stripes = low voice, widely spaced stripes = high voice. Two checks:

- Fricatives like "s" have no repeating pulse, just noise, so no spikes, just a rough smear.
- The heights of the spikes are set by the formants (the envelope). The positions are set by F0. That's the source–filter split in one sentence.

A harmonic is a sine wave at F0 × n, where n = 1, 2, 3… The multiplication is of the frequency (where the spike sits), not of the spike's height.





Additional reading: you can read all of it or just read the "In short"

What this cell does

In short track **F0 every 10 ms** for the two phone clips and draw it on top of their spectrograms (zoomed to 0–2 kHz).

- `f0_track` (pyworld) gives one F0 per frame, 0 where unvoiced. Solid white = F0, dashed white = 2·F0.

What the output means

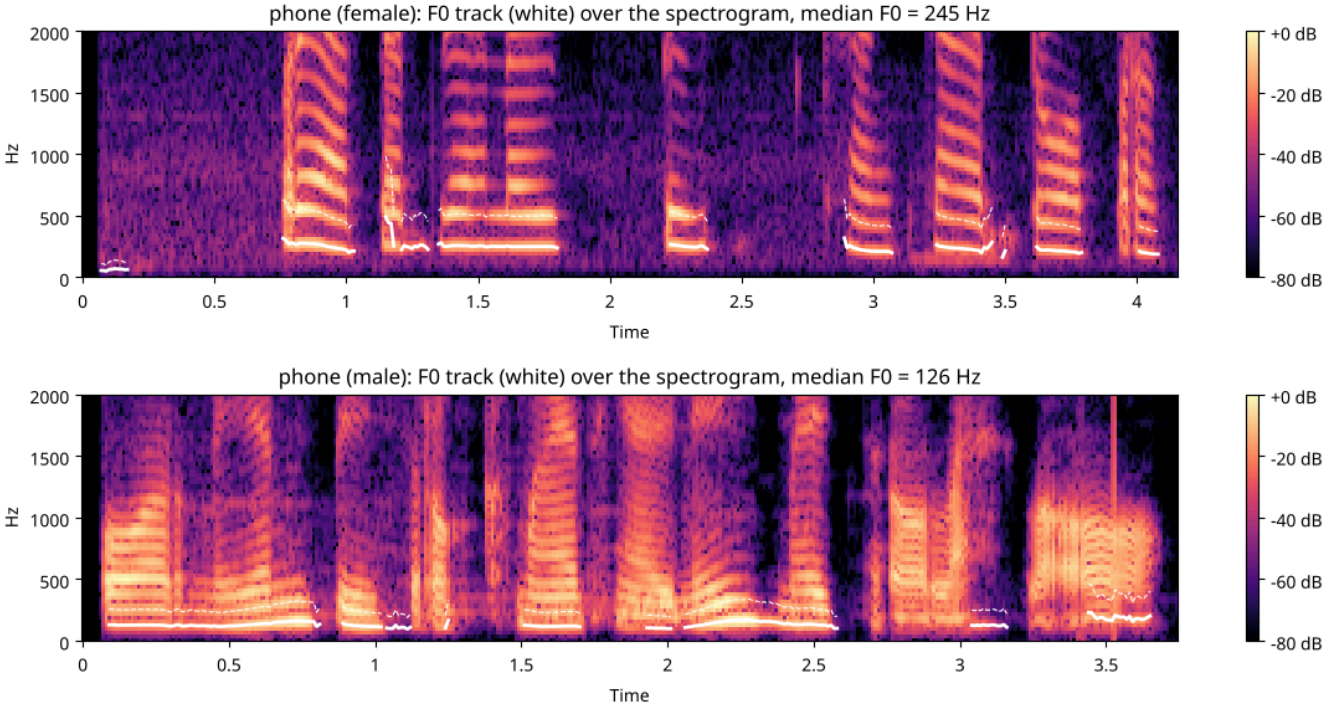
In short female median **245 Hz**, male **126 Hz** (one octave apart); the female 'range' starts at **69 Hz** only because of a tracker error on the lead-in noise.

clip	voiced frames	median F0	5–95 % range
phone female	44 %	245 Hz	69–275 Hz ⚠️ (69 is a tracker error)
phone male	58 %	126 Hz	101–186 Hz

- If the white line sits on the lowest bright stripe, the tracker is right.
- Trust the **median**; the range is where tracker mistakes show up.

```
fig, axes = plt.subplots(2, 1, figsize=(12, 6))
for ax, name in zip(axes, ["phone (female)", "phone (male)"]):
    x = load(CLIPS[name])
    f0, t = f0_track(x)
    show_spec(x, title=f"{name}: F0 track (white) over the spectrogram, median F0 = {median_f0(x):.0f} Hz", ax=ax)
    ax.plot(t, np.where(f0 > 0, f0, np.nan), c="w", lw=2)
    ax.plot(t, np.where(f0 > 0, 2 * f0, np.nan), c="w", lw=0.8, ls="--")   # 2nd harmonic for reference
    ax.set(ylim=(0, 2000))
    v = f0[f0 > 0]
    print(f"{name:15s}: {100*len(v)/len(f0):.0f}% voiced frames, F0 median {np.median(v):.0f} Hz, range {np.percentile(v,
5):.0f}-{np.percentile(v, 95):.0f} Hz")
plt.tight_layout()
```

```
phone (female) : 44% voiced frames, F0 median 245 Hz, range 69-275 Hz
phone (male)   : 58% voiced frames, F0 median 126 Hz, range 101-186 Hz
```



Additional Reading

In short the white F0 line sits on the lowest harmonic stripe: female ≈ 245 Hz, male ≈ 126 Hz; one short line near 70 Hz at the start is a tracker error.

How to read this plot. The y-axis is zoomed to 0-2000 Hz so the harmonic comb is big. The solid white line sits exactly on the lowest bright stripe (the first harmonic = F0) and the dashed line on the second stripe (2 x F0): if the tracker was wrong the line would jump to a different stripe. The female track (top) sits at 245 Hz and barely moves (a flat, read-aloud intonation), the male (bottom) at 126 Hz with a range of 101-186 Hz. Gaps in the line are unvoiced frames (fricatives, pauses): 44 % of the female clip's frames are voiced, 58 % of the male's. Look at the very start of the top panel: at 0.07-0.15 s the tracker draws a short line near 70 Hz on what is only background noise. That one error is why the female 5th percentile prints as 69 Hz; a pitch tracker should always be read together with a voicing or energy check. Above the comb, the bright hills at about 500-1000 Hz and 1500-2500 Hz are F1 and F2, which the next cell measures on one frame.

Additional reading: you can read all of it or just read the "In short"

In short % voiced mostly reflects silence and consonants; the median is the speaker's typical pitch; the range is where tracker errors show up.

Additional info: the three F0 numbers

1. % voiced frames The tracker checks every 10 ms frame: is there a repeating vocal-fold cycle here? If yes, it's "voiced" and gets an F0 value; if no (silence, breath, hissy consonants like s/h), it gets 0. Female: 44 % of frames voiced. Remember this clip has about 0.8 s of silence

at the start, which drags the number down. Her speech itself is more voiced than 44 % suggests. Male: 58 % voiced. His clip starts almost immediately, so more of it is speech. So this number mostly reflects how much silence and how many unvoiced consonants the clip contains, not the speaker's voice.

"Repeating vocal-fold cycle" means the folds are vibrating: open, snap shut, open, snap shut, over and over at a steady rate. Each open-shut is one cycle, and the number of cycles per second is F0.

2. F0 median The middle value of all the voiced frames: the speaker's typical pitch. Female 245 Hz, male 126 Hz: almost exactly an octave apart (2×126 = 252). That's the classic gender gap. It's also the strongest single speaker clue the attacker gets, which is why same-gender trials are the honest EER test.
3. F0 range The lowest and highest F0 found. The code probably uses something like the 5th–95th percentile to avoid outliers. Male 101–186 Hz: a normal speaking range, about 0.9 octave, the melody of a sentence. Female 69–275 Hz: the top (275) is fine, but 69 Hz is impossible for her. That's well below any female voice. It's a tracker error: on the lead-in noise before she speaks, pyworld latched onto something and reported a low "pitch". You can see it on her spectrogram as the frames near the start that have no real harmonics. The Day 1 notebook flags exactly this.

The lessons The median is trustworthy; the range is where tracker mistakes show up. Use the median or a trimmed percentile range for research numbers. Pitch trackers make octave errors (halving or doubling) and spurious detections in noise. Trim silence first (librosa.effects.trim) and treat extreme values with suspicion. For your project: ρF0 compares the original and anonymized F0 tracks frame by frame, so a tracker glitch like that 69 Hz pollutes the correlation. That's why the Day 6 rho_f0 throws out frames more than an octave from the clip's median.

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short find the **formants** of one vowel frame per speaker with an **LPC envelope** (order 16).

- Pick the loudest low-frequency frame (a vowel), compute its FFT (the harmonics) and the LPC envelope (the smooth outline).
- The first 3 peaks of the envelope between 150 and 5000 Hz = F1, F2, F3.

What the output means

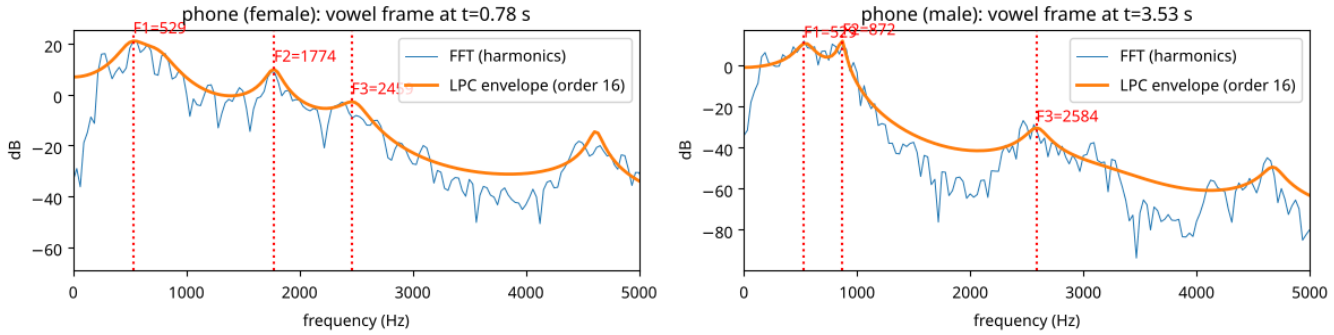
In short both frames have **F1 ≈ 529 Hz**; F2 is **1774 Hz (female) vs 872 Hz (male)**, but that's mostly because the frames are **different vowels**, not because of gender.

- `phone (female): formants F1..F3 = [ 529 1774 2459] Hz`
- `phone (male): formants F1..F3 = [ 529 872 2584] Hz`
- To compare vocal tracts fairly you must compare the **same vowel**.

```python
# LPC envelope of one vowel frame (order 16) -> formants
def lpc_envelope(frame, order=16, n_fft=N_FFT):
    a = librosa.lpc(frame * scipy.signal.get_window("hann", len(frame)), order=order)   # a[0] = 1
    w, h = scipy.signal.freqz([1.0], a, worN=n_fft // 2 + 1, fs=SR)                      # H = 1 / A
    return w, 20 * np.log10(np.abs(h) + 1e-9)

fig, axes = plt.subplots(1, 2, figsize=(12, 3.2))
for ax, name in zip(axes, ["phone (female)", "phone (male)"]):
    x = load(CLIPS[name])
    S = stft_db(x); hz = librosa.fft_frequencies(sr=SR, n_fft=N_FFT)
    fr = int(np.argmax(S[hz < 1000].mean(0)))                    # loudest low-frequency frame = a vowel
    frame = x[fr * HOP: fr * HOP + WIN]
    spec = 20 * np.log10(np.abs(np.fft.rfft(frame * scipy.signal.get_window("hann", WIN), N_FFT)) + 1e-9)
    w, env = lpc_envelope(frame)
    env += spec.max() - env.max()                                 # align levels for the plot
    pk, _ = scipy.signal.find_peaks(env, distance=8)
    pk = pk[(w[pk] > 150) & (w[pk] < 5000)][:3]
    ax.plot(hz, spec, lw=0.7, label="FFT (harmonics)"); ax.plot(w, env, lw=2, label="LPC envelope (order 16)")
    for i, p in enumerate(pk):
        ax.axvline(w[p], c="r", ls=":"); ax.text(w[p], env[p] + 3, f"F{i+1}={w[p]:.0f}", c="r")
    ax.set(title=f"{name}: vowel frame at t={fr*HOP/SR:.2f} s", xlabel="frequency (Hz)", ylabel="dB", xlim=(0, 5000));
ax.legend(loc="upper right")
    print(f"{name:15s}: formants F1..F3 = {np.round(w[pk]).astype(int)} Hz")
plt.tight_layout()
```

```
phone (female) : formants F1..F3 = [ 529 1774 2459] Hz
phone (male)   : formants F1..F3 = [ 529  872 2584] Hz
```



Additional Reading

In short thin blue line = harmonics (source), thick orange line = LPC envelope (filter), red marks = formants.

How to read this plot. The thin blue line is the FFT of one vowel frame: a picket fence of harmonics. The thick orange line is the LPC envelope: a smooth curve that hugs the tops of the pickets. Its peaks, marked in red, are the formants. The pickets are the *source* (vocal folds, F0 = picket spacing) and the envelope is the *filter* (vocal tract, formants); this is the source-filter model, and it is why you can change one without the other: a voice conversion that shifts the orange curve but keeps the pickets changes "who" without changing "what". The frames picked here give F1 = 529 Hz for both speakers, F2 = 1774 Hz (female) vs 872 Hz (male) and F3 around 2500 Hz. The F2 difference is mostly a *different vowel* (the argmax lands on different words), not gender; to compare vocal tracts you would compare the same vowel, which is exactly what section 9's follow-up in the anonymization days does with McAdams-shifted formants.

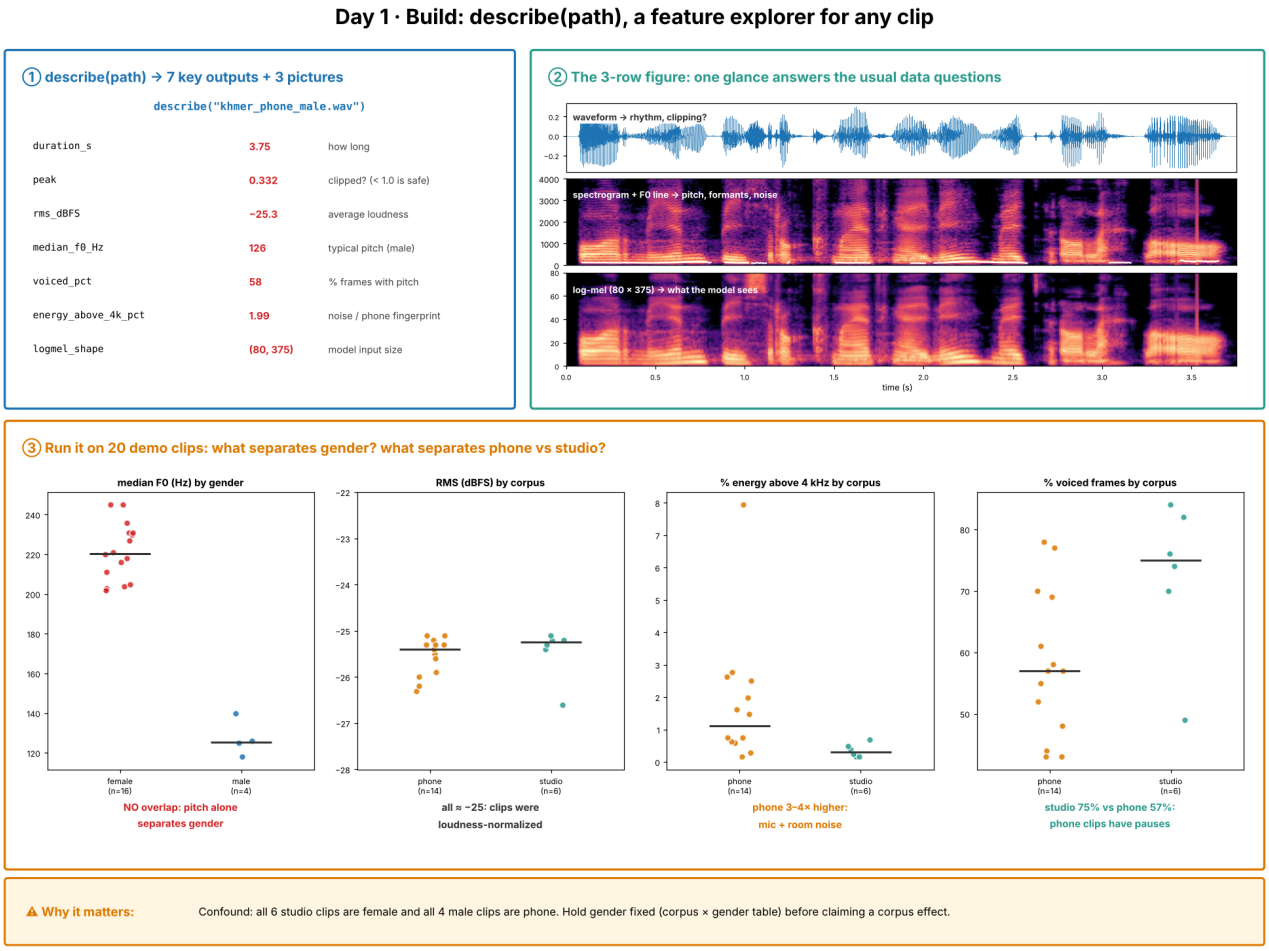
Build: a feature explorer function

Everything above, in one function. `describe(path)` loads a clip, computes six numbers and (optionally) draws the three views we now know how to read. Then we run it on the 20 demo clips (6 studio female, 10 phone female, 4 phone male, per `demo_clips.csv`) and ask two questions

with a table: *does pitch separate the genders?* and *does loudness / high-frequency energy separate phone from studio?*

The "% energy above 4 kHz" number is new: it is the share of the total STFT power that sits in bins above 4000 Hz. Band-limited phone codecs and noisy phone mics move this number in opposite directions, so it is a cheap corpus fingerprint.

Visualization Version



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short define `describe(path)`: one function that computes a clip's key numbers and (optionally) draws the 3-row figure; run it on the male clip.

- Numbers: duration, peak, RMS dBFS, median F0, % voiced frames, % energy above 4 kHz, log-mel shape.
- Figure: waveform / spectrogram + F0 line (0–4 kHz) / log-mel.

What the output means

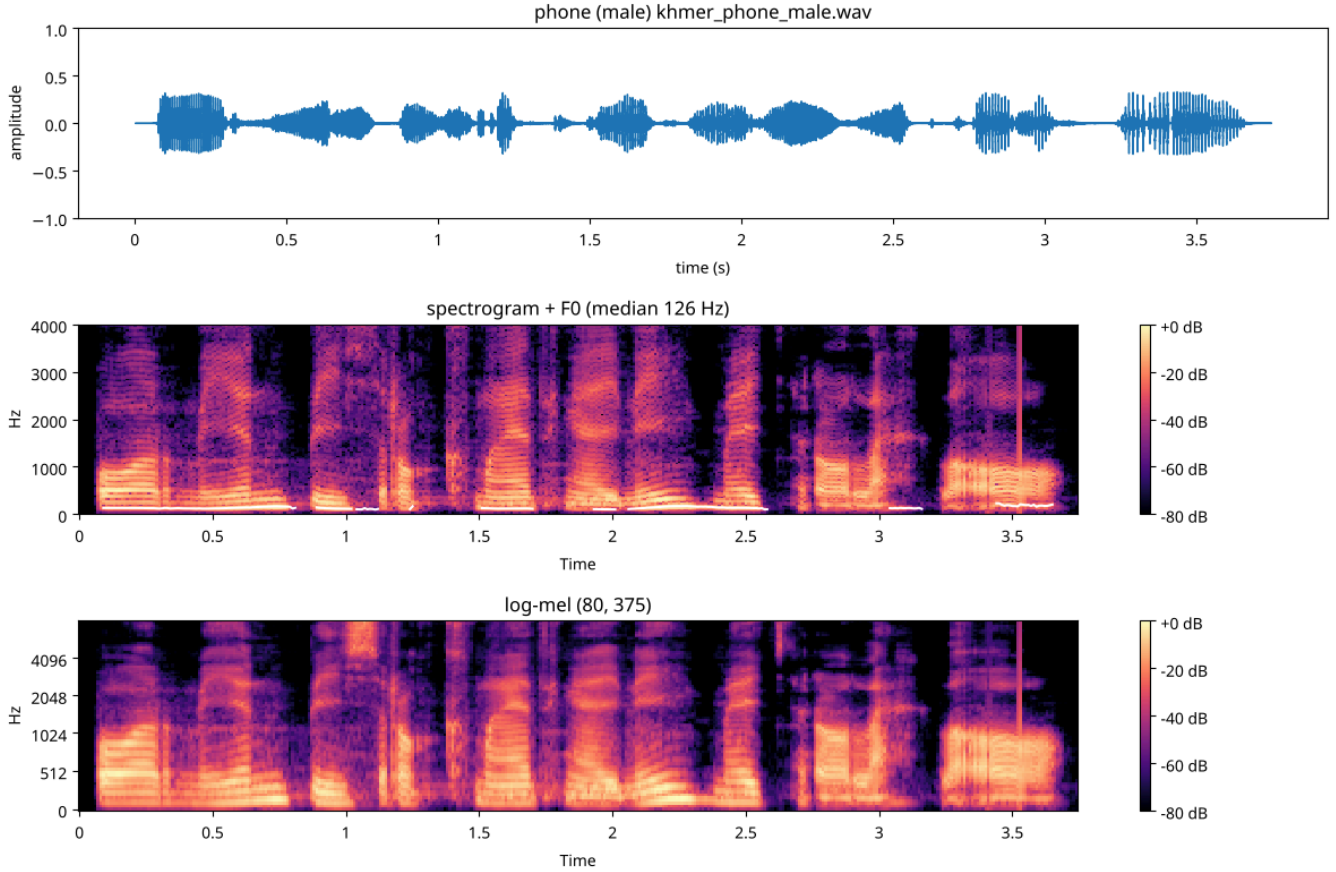
In short the male clip is **3.75 s**, not clipped (peak 0.33), normal loudness (−25.3 dBFS), pitch **126 Hz**, 58 % voiced, **2 %** energy above 4 kHz, log-mel **(80, 375)**.

- Each number is explained in the next four cells (`rms_dBFS` , `voiced_pct` , `energy_above_4k_pct` , `logmel_shape`).

```
def describe(path, plot=True, title=None):
    '''Key numbers of one clip (+ a 3-row figure). Everything at 16 kHz mono float32.'''
    x = load(path)
    P = np.abs(librosa.stft(x, n_fft=N_FFT, hop_length=HOP, win_length=WIN, window="hann")) ** 2
    hz = librosa.fft_frequencies(sr=SR, n_fft=N_FFT)
    LM = logmel(x)
    f0, t = f0_track(x)
    info = dict(file=Path(path).name,
                duration_s=round(len(x) / SR, 2),
                peak=round(float(np.abs(x).max()), 3),
                rms_dBFS=round(rms_dbfs(x), 1),
                median_f0_Hz=round(float(np.median(f0[f0 > 0])), 0) if (f0 > 0).any() else np.nan,
                voiced_pct=round(100 * float((f0 > 0).mean()), 0),
                energy_above_4k_pct=round(100 * float(P[hz > 4000].sum() / P.sum()), 2),
                logmel_shape=LM.shape)
    if plot:
        fig, axes = plt.subplots(3, 1, figsize=(12, 8))
        show_wave(x, title=title or info["file"], ax=axes[0])
        show_spec(x, title=f"spectrogram + F0 (median {info['median_f0_Hz']:.0f} Hz)", ax=axes[1])
        axes[1].plot(t, np.where(f0 > 0, f0, np.nan), c="w", lw=1.5)
        axes[1].set(ylim=(0, 4000))      # zoom to 0-4 kHz so the F0 line (100-300 Hz) is readable
        show_spec(x, title=f"log-mel {LM.shape}", ax=axes[2], mel=True)
        plt.tight_layout()
    return info

d = describe(CLIPS["phone (male)"], title="phone (male) khmer_phone_male.wav")
print(pd.Series(d).to_string())
```

```
file                  khmer_phone_male.wav
duration_s                            3.75
peak                                 0.332
rms_dBFS                             -25.3
median_f0_Hz                         126.0
voiced_pct                            58.0
energy_above_4k_pct                   1.99
logmel_shape                   (80, 375)
```



Additional Reading

In short row 1 answers 'clipped?', row 2 'what pitch, any noise?', row 3 'what does the model see?'.

How to read this plot. Row 1: the waveform, for rhythm and loudness (peak 0.33, RMS -25.3 dBFS, no clipping). Row 2: the spectrogram zoomed to 0-4 kHz with the white F0 line at 126 Hz hugging the lowest harmonic, for pitch, formants and noise; 58 % of frames are voiced; the fricative at 1.0-1.2 s and the click at 3.52 s stand out as vertical features. Row 3: the log-mel (80 × 375, i.e. 3.75 s at 100 frames/s), the model's view: the bottom 1 kHz takes about a third of the height, so the harmonics and formants that were squeezed into the lowest sixth of row 2 now fill the picture, while the 2-8 kHz region above is compressed into the top half. Only 2 % of this clip's energy is above 4 kHz. One glance at the three rows answers the usual data questions: is it clipped (row 1), what is the pitch and is there a noise floor (row 2), is the band-limited region empty (row 3, top).

Additional reading: you can read all of it or just read the "In short"

In short rms_dBFS = average loudness on a scale where 0 is the maximum; −25 is normal speech.

rms_dBFS

rms_dBFS is the clip’s average loudness, written in decibels relative to the loudest possible level.

Two parts to the name:

RMS (root mean square) = the typical amplitude of the clip:

square every sample (so negative swings count too),take the mean,take the square root. For this clip that gives about 0.054.

dBFS (decibels relative to full scale) = that number converted to a log scale where 1.0 (the ceiling, “full scale”) is 0 dB:

20 · log10(0.054) ≈ −25.3 dBFS So −25.3 means “the typical amplitude is about 1/18 of full scale.”

How to read the number:

0 dBFS = as loud as the file can hold. Real speech is always negative. Every −6 dB halves the amplitude: −6 ≈ 0.5, −12 ≈ 0.25, −20 = 0.1, −26 = 0.05.

Normal speech recordings: −20 to −30. Below −40 is very quiet; a file at −60 is probably near-silence or a mistake. What it’s for:

A sanity check: all your clips sit at −25 to −26, so the corpus was loudness-normalized and no clip is accidentally quiet or hot. Models like ECAPA and Whisper expect speech at a normal level; a clip at −50 dBFS gets worse results even though nothing crashes. It's the "average" partner of peak: peak says whether the recording was damaged (clipping), RMS says how loud it sounds.

Overall: rms_dBFS = how loud the clip is on average, on a scale where 0 is the maximum and more negative is quieter. −25 is normal speech. You check it so every clip feeds the models at a sensible, consistent level.

Additional reading: you can read all of it or just read the "In short"

In short voiced_pct = % of frames with a pitch; mostly tells you how much silence and how many s/h sounds the clip has.

voiced_pct = the percentage of 10 ms frames where the vocal folds were vibrating, i.e. frames with a pitch.

For the male clip: 375 frames, 58 % voiced → about 217 frames have an F0, the other 158 are silence, breaths or unvoiced consonants (s, h, k…).

What it tells you:

Mostly how much of the clip is speech vs pause. The female clip's 44 % is lower largely because of its 0.8 s of silence at the start.

A bit about the sound content: Khmer sentences with many fricatives score lower.

A sanity flag: near 0 % means no speech was found (wrong file, pure noise, or the tracker failed); near 100 % is suspicious too.

In the project it's useful because ρF0 and the pitch statistics are computed over voiced frames only. If an anonymizer leaves very few voiced frames (ASR→TTS can change this), those numbers rest on little data.

Additional reading: you can read all of it or just read the "In short"

In short energy_above_4k_pct = how much of the energy is in the top half (4–8 kHz); phone noise pushes it up.

energy_above_4k_pct = what share of the clip's total energy sits above 4 kHz, in the upper half of the 0–8 kHz band.

Additional reading: you can read all of it or just read the "In short"

In short logmel_shape = (80 bands, frames); a quick check that the model input has the right size.

logmel_shape = the size of the log-mel grid, (mel bands, frames). For the male clip: (80, 375).

- 80 = the number of mel frequency bands (rows). Fixed by the setup; it's what Whisper and ECAPA-style models expect.
- 375 = the number of time frames (columns). One every 10 ms, so 3.75 s × 100 = 375.
- It's there as a check that the feature matrix is what the model expects: 80 rows (not 257, which would mean you forgot the mel step, and not 13, which is MFCC) and a frame count that matches the duration. If the frames came out as ~1100, you'd know the clip was still at 48 kHz.

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short run `describe` on the **20 demo clips** (no plots), collect the numbers in one table and save it to `outputs/day1/features.csv`.

- 20 clips = 6 studio female + 10 phone female + 4 phone male.

What the output means

In short a 20-row table, one row per clip, saved to CSV for the comparison in the next cell.

- Columns: clip id, speaker, gender, corpus, then the 7 `describe` outputs.

```
# Run over the 20 demo clips (no plots) and build the table.
meta = pd.read_csv(DATA / "meta/demo_clips.csv")
rows = []
for _, r in meta.iterrows():
    d = describe(DATA / "audio/demo" / f"{r.utt_id}.wav", plot=False)
    d.update(utt_id=r.utt_id, speaker=r.speaker, gender=r.gender, corpus=r.corpus)
    rows.append(d)
feats = pd.DataFrame(rows)[["utt_id", "speaker", "gender", "corpus", "duration_s", "peak", "rms_dBFS", "median_f0_Hz",
"voiced_pct", "energy_above_4k_pct", "logmel_shape"]]
feats.to_csv(OUT / "features.csv", index=False)
print("saved", OUT / "features.csv")
display(feats)
```

```
saved ../outputs/day1/features.csv
```

	utt_id	speaker	gender	corpus	duration_s	peak	rms_dBFS	median
0	slr42_7548_3412151647	slr42-7548	female	studio (OpenSLR 42)	6.26	0.252	-25.2	
1	slr42_7548_0402083719	slr42-7548	female	studio (OpenSLR 42)	3.35	0.275	-25.4	

2	slr42_3154_4104924772	slr42-3154	female	studio (OpenSLR 42)	3.15	0.271	-25.1
3	slr42_3154_0657701350	slr42-3154	female	studio (OpenSLR 42)	3.42	0.223	-26.6
4	slr42_6438_0808051780	slr42-6438	female	studio (OpenSLR 42)	3.35	0.397	-25.2
5	slr42_6438_4294471887	slr42-6438	female	studio (OpenSLR 42)	3.85	0.271	-25.3
6	iany_ab586c1d_52008df7	iany-s-ab586c1d	female	phone (iAny)	4.16	0.536	-26.3
7	iany_ab586c1d_c1396ba8	iany-s-ab586c1d	female	phone (iAny)	3.90	0.640	-26.0
8	iany_b643351e_2d6423e4	iany-s-b643351e	female	phone (iAny)	3.17	0.366	-25.5
9	iany_b643351e_3f003edd	iany-s-b643351e	female	phone (iAny)	3.79	0.239	-25.4
10	iany_32bb69d3_2c02ab33	iany-s-32bb69d3	female	phone (iAny)	3.06	0.344	-25.2
11	iany_32bb69d3_df7ee2c9	iany-s-32bb69d3	female	phone (iAny)	3.55	0.403	-25.3
12	iany_7c7ad67c_31719ef4	iany-s-7c7ad67c	female	phone (iAny)	3.31	0.544	-25.1
13	iany_7c7ad67c_d933c231	iany-s-7c7ad67c	female	phone (iAny)	3.39	0.408	-25.3
14	iany_f925f041_569e84d2	iany-s-f925f041	female	phone (iAny)	3.20	0.379	-25.6
15	iany_f925f041_a7527a1d	iany-s-f925f041	female	phone (iAny)	3.73	0.397	-25.4
16	iany_ad89c739_3dee3a99	iany-s-ad89c739	male	phone (iAny)	3.75	0.332	-25.3
17	iany_ad89c739_5e3f67dd	iany-s-ad89c739	male	phone (iAny)	3.23	0.346	-25.1
18	iany_ea2f2c04_8406c570	iany-s-ea2f2c04	male	phone (iAny)	3.97	0.345	-26.2
19	iany_ea2f2c04_8ff0fe03	iany-s-ea2f2c04	male	phone (iAny)	3.60	0.382	-25.9

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short summarise the 20 clips **by gender**, **by corpus**, and **by corpus × gender**, and draw 3 scatter plots.

- Median of each number per group, plus a count of clips per group.

What the output means

In short **pitch separates gender perfectly** (female 220 Hz vs male 126 Hz, no overlap); **loudness separates nothing** (all ≈ −25 dBFS); **phone clips have 3–4× more energy above 4 kHz** and fewer voiced frames.

group	median F0	RMS	% voiced	% above 4 kHz
female (16)	220 Hz	−25.4	69.5	0.6
male (4)	126 Hz	−25.6	57	2.3
phone (14)	217 Hz	−25.4	57	1.1
studio (6)	212 Hz	−25.2	75	0.3

- **Confound:** all studio clips are female and all male clips are phone, so check the `corpus × gender` table before claiming a corpus effect.

```
num = ["duration_s", "peak", "rms_dBFS", "median_f0_Hz", "voiced_pct", "energy_above_4k_pct"]
print("by gender:"); display(feats.groupby("gender")[num].median().round(1))
print("by corpus:"); display(feats.groupby("corpus")[num].median().round(1))
print("by corpus x gender (n clips):"); display(feats.groupby(["corpus", "gender"])[num].agg(["median",
"count"]).round(1).iloc[:, [0, 1, 2, 4, 6, 8, 10, 11]])

fig, axes = plt.subplots(1, 3, figsize=(12, 3.2))
for ax, col, by in zip(axes, ["median_f0_Hz", "rms_dBFS", "energy_above_4k_pct"], ["gender", "corpus", "corpus"]):
    for g, sub in feats.groupby(by):
        ax.scatter(np.random.uniform(-0.1, 0.1, len(sub)) + (0 if g == sorted(feats[by].unique())[0] else 1), sub[col],
label=g, alpha=0.8)
    ax.set(title=f"{col} by {by}", xticks=[0, 1], xticklabels=sorted(feats[by].unique()), ylabel=col); ax.legend()
plt.tight_layout()
```

```
by gender:
```

gender	duration_s	peak	rms_dBFS	median_f0_Hz	voiced_pct	energy_above_4k_pct
female	3.4	0.4	-25.4	220.5	69.5	0.6
male	3.7	0.3	-25.6	125.5	57.0	2.3

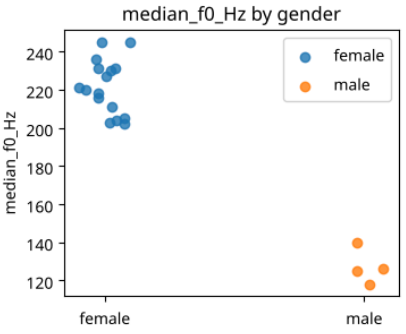
```
by corpus:
```

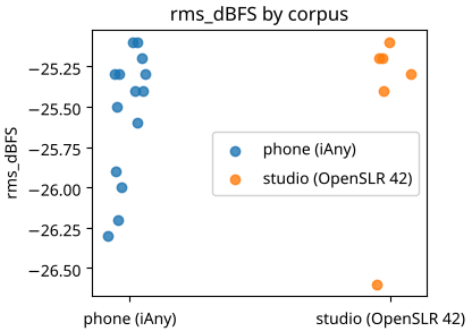
corpus	duration_s	peak	rms_dBFS	median_f0_Hz	voiced_pct	energy_above_4k_pct
phone (iAny)	3.6	0.4	-25.4	217.0	57.0	1.1
studio (OpenSLR 42)	3.4	0.3	-25.2	212.5	75.0	0.3

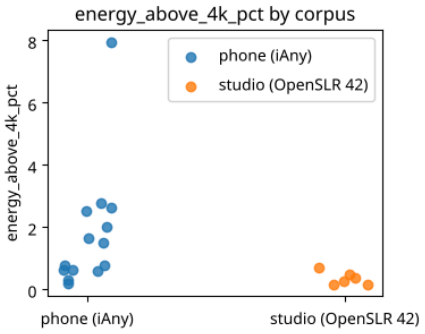
```
by corpus x gender (n clips):
```

		duration_s		peak	rms_dBFS	median_f0_Hz	voiced_pct	energy_above_4
		median	count	median	median	median	median	median
corpus	gender							

phone (iAny)	**female**	3.5	10	0.4	-25.4	223.5	58.0	0.8
	male	3.7	4	0.3	-25.6	125.5	57.0	2.3
studio (OpenSLR 42)	**female**	3.4	6	0.3	-25.2	212.5	75.0	0.3







Additional Reading

In short pitch alone separates gender here; loudness was normalized so it separates nothing; high-frequency energy and % voiced separate phone from studio.

Interpretation. (Numbers from the tables above.) *Gender:* median F0 of the 16 female clips is 220 Hz (every clip between 202 and 245 Hz) and of the 4 male clips 125 Hz (118-140 Hz), with no overlap at all, so in this set pitch alone separates gender; that is why an anonymizer that leaves F0 untouched leaks gender, and why `rho_f0` is on the leaderboard. *Loudness:* RMS is -25 to -26 dBFS for every single clip, phone or studio, and peaks are 0.22-0.64: the clips were loudness-normalized when the dataset was built, so RMS tells you nothing about corpus here (on raw data it usually does). *What does separate the corpora:* the studio clips are 75 % voiced frames vs 57 % for phone, because the OpenSLR readers speak continuously while the iAny phone clips contain pauses and lead-in silence; and the phone clips put 3-4x more of their energy above 4 kHz (median 1.1 % vs 0.3 %, up to 7.9 % for one clip), not because the speakers hiss more but because the phone mic and room add broadband noise that shows up as haze in the spectrogram, while the studio clips are nearly silent above the speech. Note the confound: all 6 studio clips are female and all 4 male clips are phone, so "studio vs phone" and "female vs male" are partly the same split in this small set (the male clips also have the highest above-4 kHz share, 2.3 %); a fair comparison should hold gender fixed, which the `corpus x gender` table does, and this point returns on evaluation day.