Bad Apple!! Project

Date	@October 11, 2025
Select	Personal Project

Goal:

The goal of this project is to animate Bad Apple!! frame by frame using RF Signal plots. This will be done by using Python, and a bit of creativity.

https://www.youtube.com/watch?v=FtutLA63Cp8

Stage One:

Generate a sine wave.

```
import numpy as np
import matplotlib.pylab as plt

fs = 48000 # samples per second
f = 1000 # cycles per second
t = np.linspace(0, 1, int(fs), endpoint=False) # time vector
y = np.sin(2 * np.pi * f * t) # sampled sine wave

plt.plot(t[:200], y[:200])
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
```

```
plt.title('1 kHz Sine Wave sampled at 48kHz')
plt.show()
```

Term	Symbol	Meaning	Analogy / Notes
Signal	(y(t))	A varying quantity (e.g., voltage) over time	The thing we're measuring or transmitting
Amplitude	(A)	Instantaneous strength or height of the signal	"Volume" or "intensity"
Frequency	(f)	Cycles per second (Hz)	How fast the wave repeats
Period	(T = 1/f)	Duration of one full cycle	For 1 kHz, (T = 1 ms)
Sampling Rate	(f_s)	Samples taken per second (Hz)	"Camera frame rate" for signals
Samples per Cycle	(N = f_s / f)	Number of discrete points describing one wave	Determines smoothness of the digital curve
Time Step	(\Delta t = 1/f_s)	Time between samples	For 48 kHz → ≈ 20.83 µs
Nyquist Rate	(f_s > 2f)	Minimum rate to avoid aliasing	You must sample at > 2× the signal frequency
Phase	(\phi = 2\pif t)	Angle position within the wave cycle	Think of it as "where you are" on the sine curve

Stage Two: Amplitude Modulation (AM)

In stage two, I will…

Concept	Description
Modulating Signal	A second, slower waveform (or data array) that controls the amplitude.
Carrier Signal	The fast sine wave you already built (1 kHz).
Amplitude Modulation (AM)	Multiplying them: (s(t) = A(t)\sin(2\pi f_c t)). The amplitude of the carrier follows (A(t)).

Conceptual Background

What is Amplitude Modulation? Amplitude modulation is the act of using one signal to control the amplitude (height) of another signal. We start with…

- A **carrier wave**: high frequency sine wave (such as the 1kHz signal above)
- A **modulating wave**: lower frequency signal (e.g., voice, pixel brightness, or another sine)

Then we combine them together into one equation.

$$s(t) = [1 + m(t)] \cdot A_c sin(2\pi f_c t)$$

Where:

- s(t): modulated (final) signal
- m(t): modulating signal (normalized between -1 and 1)
- A_c: Carrier amplitude
- f_c: Carrier frequency

→ The modulating signal changes the amplitude of the carrier signal without altering its frequency.

3. Why It Matters

In the real world:

- **AM radio** sends sound by modulating amplitude of an RF carrier (hundreds of kHz).
- **Radar**, **optical**, and **communication systems** all rely on AM as a basic form of encoding.
- In your case, you'll use AM to **encode visual brightness** into an RF waveform.

So AM is both fundamental *and* practical — it's how you embed data into a wave.

Questions

▼ What is a carrier signal?

A carrier signal is a high-frequency sine wave that acts as the base or “transport” for carrying information.

▼ What is a modulating signal?

The modulating signal is known as the information signal as is often denoted as $m(t)$ that alters the amplitude of the carrier signal.

▼ What is a modulated signal?

The modulated signal is the final transmitted wave after combining the carrier and the modulating signal. It carries the information by changing the amplitude of the carrier according to $m(t)$.

Mathematically,

$$s(t) = [1 + m(t)]A_c sin(2\pi f_c t)$$

▼ Explain what an envelope is.

The envelope of a modulated signal can be thought of as the shape that outlines the smooth curves and outlines the peaks of the modulated wave. It follows the shape of the modulating signal.

▼ Why do we add 1 to $m(t)$

We add 1 to $m(t)$ to make sure that the carrier amplitude never goes negative, as $m(t)$ after normalization varies in the range of -1 to 1. This avoid phase inversion or signal distortion.

▼ What’s the general equation for amplitude modulation?

Where

- $m(t) =$ modulating signal
- $A_c =$ carrier amplitude
- $f_c =$ carrier frequency

- $t =$ time

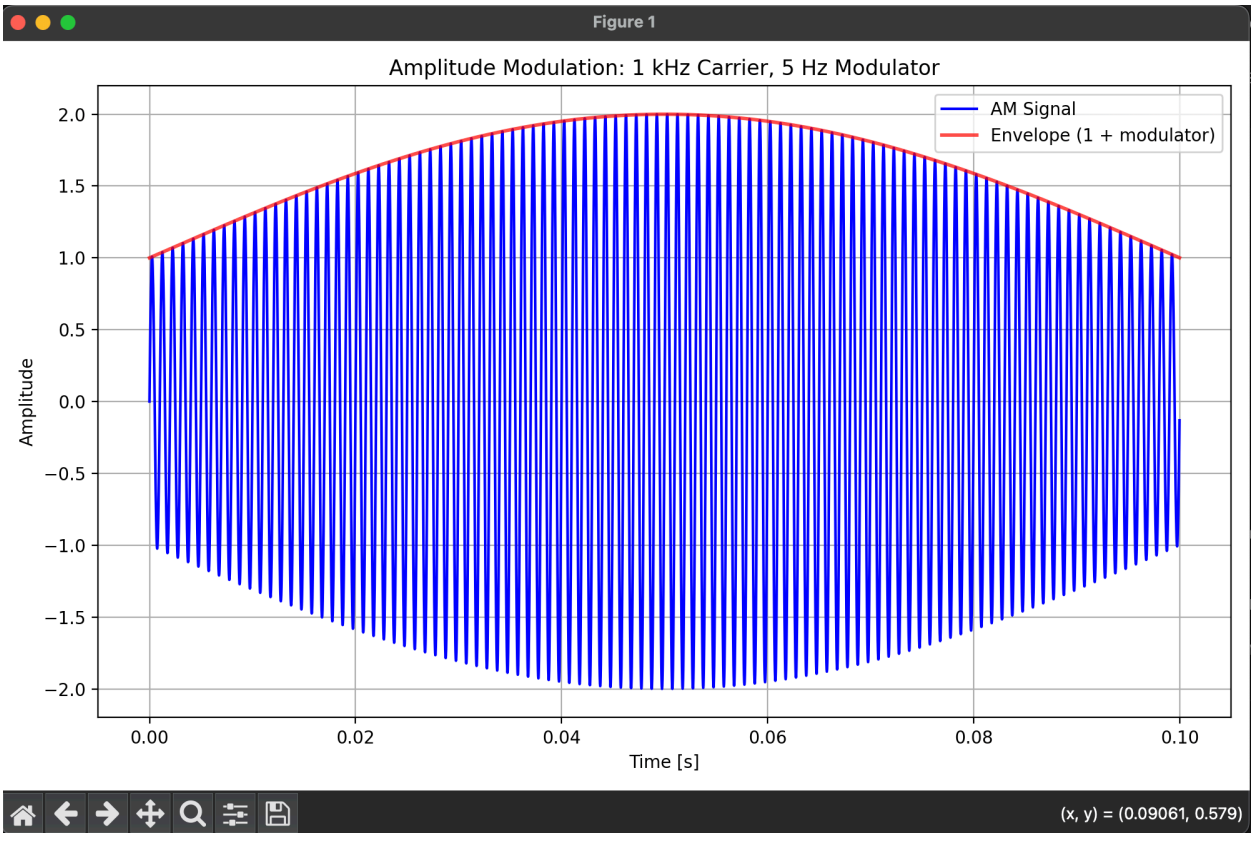
$$s(t) = [1 + m(t)]A_c sin(2\pi f_c t)$$

Code Implementation

Implement a 1 kHz carrier amplitude-modulated by a 5 Hz sine wave.

Then, analyze:

1. How the envelope follows the modulating wave
2. What happens when you increase the modulation frequency
3. What happens when you remove the `+1` (why it breaks symmetry)



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

# --- Signal parameters ---
fs = 48000           # Sampling rate (samples per second)
fc = 1000            # Carrier frequency (Hz)
fm = 5               # Modulating frequency (Hz)
duration = 1         # Signal length in seconds

# --- Time vector ---
t = np.linspace(0, duration, int(fs * duration), endpoint=False)

# --- Generate signals ---
carrier = np.sin(2 * np.pi * fc * t)        # High-frequency carrier
modulator = np.sin(2 * np.pi * fm * t)      # Slow modulating signal (controls a
mplitude)

# --- Perform amplitude modulation ---
# Add 1 so amplitude never goes negative (range 0 → 2)
modulated = (1 + modulator) * carrier

# --- Plot results ---
plt.figure(figsize=(10, 6))              # Create a new figure, 10×6 inches
plt.plot(t[:4800], modulated[:4800],
        color='blue', label='AM Signal')  # Plot blue AM waveform
plt.plot(t[:4800], 1 + modulator[:4800],
        color='red', label='Envelope',
        linewidth=2, alpha=0.7)          # Plot red envelope line on top
plt.title('Amplitude Modulation: 1 kHz Carrier, 5 Hz Modulator')  # Graph title
plt.xlabel('Time [s]')                   # X-axis label
plt.ylabel('Amplitude')                  # Y-axis label
plt.legend()                             # Show labels for lines
plt.grid(True)                           # Add grid lines
```

```
plt.tight_layout()                # Auto-adjust spacing
plt.show()                        # Render the figure
```

The envelope follows the modulated signal in almost a wide downward parabola shape. Then we increase the modulated frequency, the signal gets squished more and more as more cycles are happening. When we remove the + 1, the envelope doesn't cover the peaks, and the signal sits below the middle.

Stage Three: Image to Amplitude Modulator

We've already applied amplitude modulation where $m(t)$ was a sine wave, but this time, it will come from an image. These images will be the frames from the Bad Apple!! music video.

- Every pixel's brightness will become a modulation amplitude.
- Dark pixel → low amplitude
- Light pixel → high amplitude

The Mapping Process

Step	Operation	Result
a. Load image	Convert to grayscale (brightness only).	2-D array of intensities 0–255
b. Flatten to 1-D	Convert rows → a single list of pixel values.	1-D signal
c. Normalize	Divide by 255 → values between 0 and 1.	m(t)m(t)
d. Multiply with carrier	(1 + modulator) * carrier	Amplitude-modulated waveform

2. Why This Works

Let us think of the **carrier** as the transmission wave and the **pixels** as the information.

As the carrier's amplitude changes, you're literally encoding the image's brightness into an RF signal, exactly what amplitude modulation does in analog systems.

Resources

https://en.wikipedia.org/wiki/Digital_image

https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.spectrogram.html

https://pillow.readthedocs.io/en/stable/

→ For this code, the following image was used.

Complete Code

```
# --- Imports ---
import numpy as np
from PIL import Image, ImageOps
import matplotlib.pyplot as plt

# --- 1. Load image and convert to grayscale ---
im = Image.open("image.png")                # Replace with your image path
grayscale = ImageOps.grayscale(im)          # Convert to grayscale (0–255)
grayscale.show()                            # preview grayscale image

# --- 2. Convert to NumPy array, flatten, and normalize ---
img_array = np.array(grayscale)             # Convert to 2-D NumPy array
flattened_img = img_array.flatten()         # Flatten to 1-D
normalized_img = flattened_img.astype(np.float32) / 255.0  # Normalize to [0,
1]

# --- 3. Create a 1 kHz carrier of the same length ---
fs = 48000                                  # Sampling rate
fc = 1000                                   # Carrier frequency (1 kHz)
t = np.arange(len(normalized_img)) / fs      # Time vector (seconds)
carrier = np.sin(2 * np.pi * fc * t)        # Carrier sine wave

# --- 4. Perform amplitude modulation using image brightness ---
am_signal = (normalized_img + 1) * carrier    # AM waveform

# --- 5. Plot a short segment to visualize the modulation ---
plt.figure(figsize=(10, 5))
plt.plot(t[:2000], am_signal[:2000], color='blue')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.title('Image-Encoded Amplitude-Modulated Signal (1 kHz Carrier)')
plt.grid(True)
plt.tight_layout()
plt.show()
```

Result

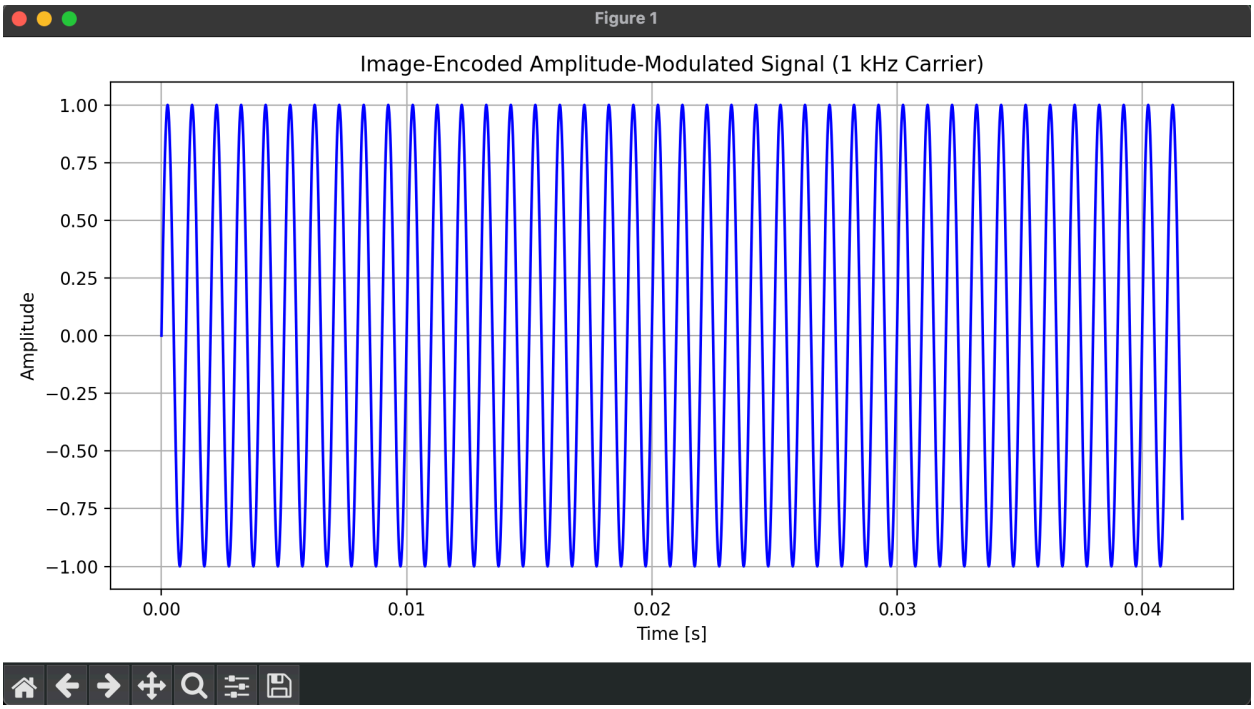
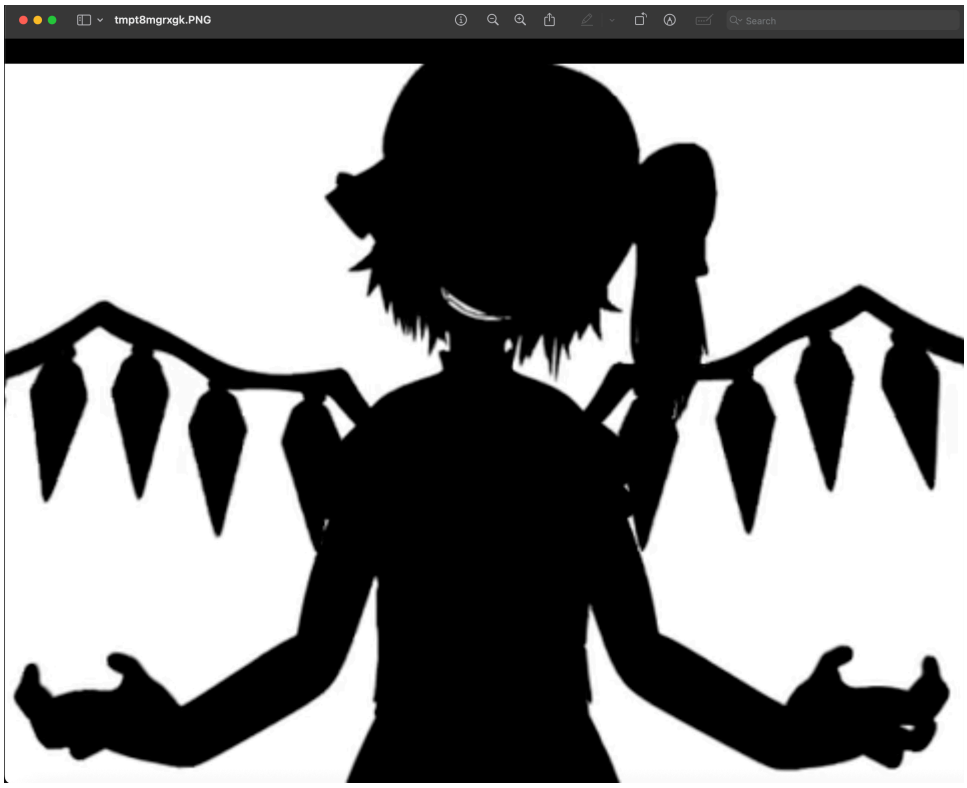


Figure 1
Image-Encoded Amplitude-Modulated Signal (1 kHz Carrier)
Amplitude
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.00
0.01
0.02
0.03
0.04
Time [s]


tmpt8mgrxgk.PNG

A note on why we use np.arange() instead of np.linspace().

You may have noticed that here, instead of using the regular np.linspace() we have used np.arange(). This is because when working with data from things such as images, it is often not as predictable as set time intervals or ranges, such as -1 to 1. This means, if two images are 4000 and 6000 respectively in size, then we would need to handle those cases individually with np.linspace().

Why 48000Hz? Why 1kHz?

Why do we use the sampling rate of 48000Hz and the frequency of 1 kHz. We do this as these are standard and extremely easy to work with.

Summary

In this stage, I learned how to convert an image into an amplitude-modulated (AM) signal.

I used a grayscale image, flattened its pixels into a one-dimensional array, and normalized their brightness values to the range [0, 1].

These values became the *modulating signal* $m(t)$, which controlled the amplitude of a continuous 1 kHz *carrier* sine wave.

Process

1. **Load & preprocess image** – convert to grayscale, flatten, and normalize pixel values.
2. **Generate carrier** – create a 1 kHz sine wave sampled at 48 kHz.
3. **Amplitude modulation** – multiply the carrier by (1+m(t))(1+m(t)) so the carrier's height follows image brightness.
4. **Visualize** – plot a short segment of the resulting waveform.

Key Concepts

- **Sampling rate (48 kHz):** number of digital samples per second; defines temporal resolution.
- **Carrier frequency (1 kHz):** base oscillation frequency of the sine wave.

- **Normalization:** maps pixel intensity to [0, 1] for clean amplitude scaling.
- **Amplitude Modulation:** carrier amplitude varies according to the modulating signal.
- **Why “+1”:** keeps amplitudes positive so the carrier never inverts.

Takeaway

This stage produced a one-dimensional RF-style signal whose amplitude encodes the brightness of each pixel.

The image information isn’t visible in the time-domain plot but will appear when analyzed in the **frequency domain**using a spectrogram (next stage).

Stage 4: Spectrogram Visualization

Time vs Frequency Domain

Our am_signal is currently in the time domain where each point corresponds to the signal’s amplitude at one moment in time. However, what we usual see in radio, audio, or visual spectrograms is how energy changes across frequencies over time.

A spectrogram converts this 1D signal into a 2D map showing

- X-axis: time
- Y-axis: frequency
- Color: signal strength (amplitude/power)

2. How It Works (Conceptually)

To build a spectrogram:

1. Split the signal into small overlapping chunks (windows).
2. Run a **Fast Fourier Transform (FFT)** on each chunk.
3. Stack all the frequency spectra side-by-side over time.

This gives you a 2D array a kind of “frequency movie” where the brightness shows energy at each frequency band.

🧾 Docs & References

Topic	Resource
SciPy Spectrogram	https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.spectrogram.html
Short-Time Fourier Transform (STFT)	Wikipedia: STFT
Matplotlib Colormaps	https://matplotlib.org/stable/tutorials/colors/colormaps.html
Spectrogram Visualization Tips	Try `inferno`, `viridis`, or `gray` colormaps — good contrast for amplitude data

Stage 4: Spectrogram Fundamentals

The spectrogram works by

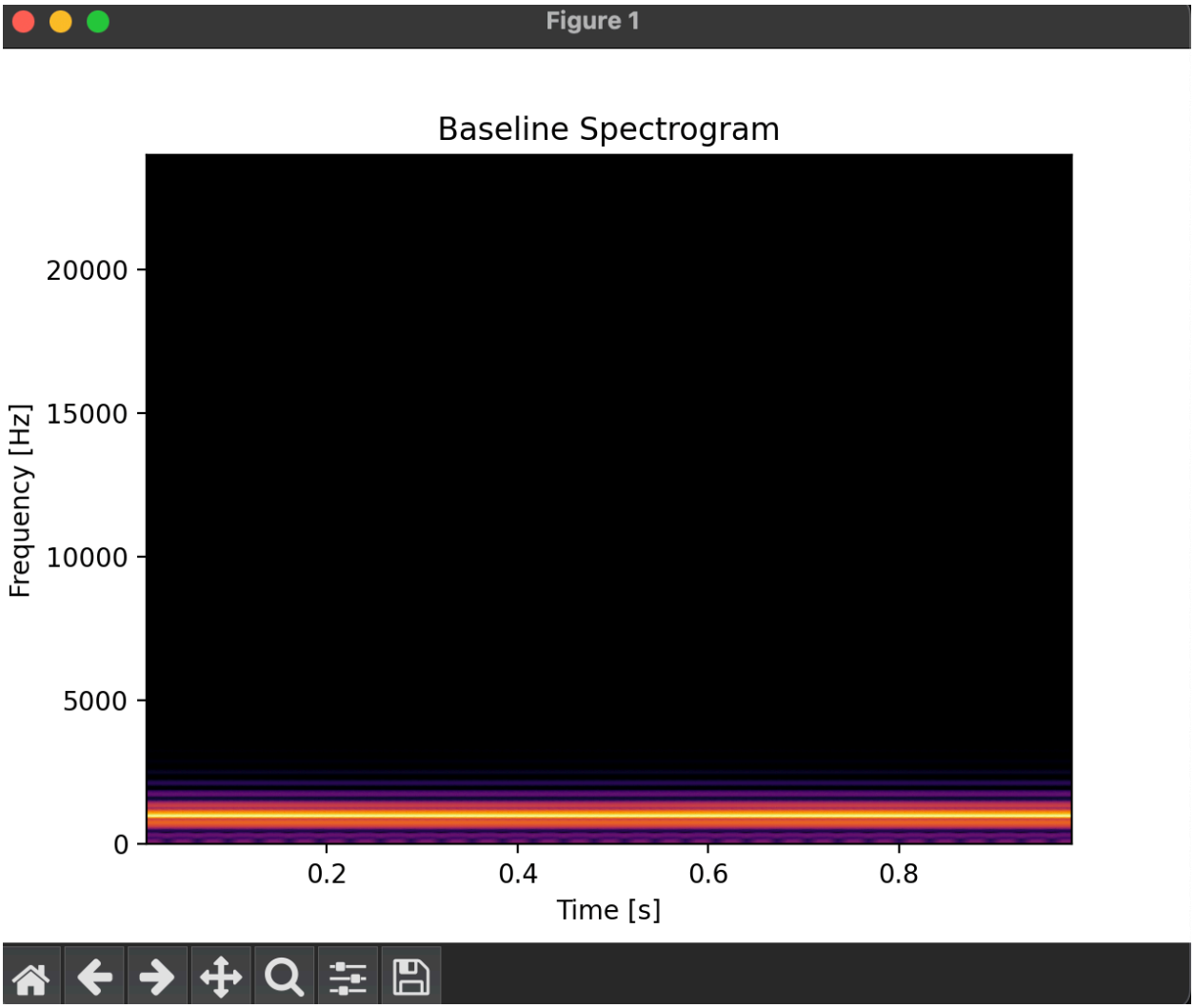
1. Taking small chunks (segments) of the signal
2. Computing an FFT on each chunk
3. Then moving a bit forward (overlap) and repeating

Resources:

https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.spectrogram.html

```
import numpy as np, matplotlib.pyplot as plt
from scipy import signal

fs = 48000
t = np.linspace(0, 1, fs, endpoint=False)
```

```
sig = np.sin(2*np.pi*1000*t)
f, tt, Sxx = signal.spectrogram(sig, fs, nperseg=1024, noverlap=512)
plt.pcolormesh(tt, f, 10*np.log10(Sxx+1e-10), shading="gouraud", cmap="infer
no")
plt.xlabel("Time [s]"); plt.ylabel("Frequency [Hz]")
plt.title("Baseline Spectrogram"); plt.show()
```



What does time here mean?

- Time shows how the signal changes over time. Each position on the horizontal axis corresponds to a moment in the signal, be that seconds or milliseconds.
- As you move rightward, you're looking at later parts of the sound or RF signal.

What is frequency here?

- Frequency shows the pitch or rate of oscilation of a component in the signal, measured in Hertz (Hz).
- Low frequencies such as bass tones or low RF bands are near the bottom.
- High frequencies such as high pitched noises or high RF bands are near the top.
- For RF: It could range from MHz to GHz depending on the system.

What is Amplitude in this context?

- Amplitude basically shows the strength or power of the signal at some given point.
- Brighter or warmer colors like yellow or red indicate higher amplitude, and thus a stronger signal.
- Darker or cooler colors like blue or black mean a weaker signal (low amplitude).

Summary

Property	Represented By	Meaning
Time	X-axis	When the signal occurs
Frequency	Y-axis	What frequencies are present
Amplitude (Intensity)	Color	How strong each frequency is

SciPy (signal.spectrogram)

The core call

```
f, tt, Sxx = signal.spectrogram(x, fs)
```

- f: Array of frequencies (y-axis)
- tt: Array of time bins (x-axis)
- Sxx: The spectrogram matrix where each cell shows signal power for a given amplitude and time.

Inputs

- x: Your signal, a 1-D NumPy array of amplitude values (e.g. from a waveform)
- fs: Sampling frequency in Hz (samples per second).
- window: The window function (default ‘hann’).
- nperseg: Number of samples per segment (FFT window length). Affects frequency resolution.
- noverlap: Number of samples to overlap between segments (smoother time transitions).
- nfft: Number of FFT points. If larger than nperseg, zero-pads the segment.
- scaling: ‘density’ (power spectral density) or ‘spectrum’ (power spectrum).
- mode: ‘psd’ default, ‘magnitude’, ‘complex’, or ‘angle’

Trade-off

Larger `nperseg`	Smaller `nperseg`
🔹 Better **frequency** resolution	🔹 Better **time** resolution
🔹 Poorer **time** resolution	🔹 Poorer **frequency** resolution

- You can’t maximize both — it’s a **time–frequency trade-off**.

Typical starting points

Use case	Recommended
Speech/music	`nperseg=1024`, `noverlap=512`
RF/steady signals	`nperseg=2048+`, `noverlap≈75%`
Short events	`nperseg=256`, `noverlap≈50%`

Quick intuition

- **nperseg** = how wide each analysis window is.

- **noverlap** = how much those windows overlap.
- Try different values and visualize — **larger** `nperseg` **= sharper frequency bands, blurrier time**.

Time vs Frequency Trade Off

A spectrogram uses short-time FFTs, meaning the signal is split into short windows and analyzed separately. This creates a trade-off between Time and Frequency. For example.

- A larger window captures more cycles which results in better frequency precision but worse time precision (can't tell exactly when changes happen).
- A smaller window computes fewer cycles, which results in better time precision but worse frequency precision (frequency bins become coarser).

🕑 Time Precision (Time Resolution)

- **Definition:** How accurately the spectrogram can tell *when* a change in the signal happens.
- **High time precision** → you can pinpoint the exact **moment** a frequency or event occurs.
- **Low time precision** → frequency changes appear *blurred across time*.

🧠 **Example:**

If you have two short tones — one at 1.0s and another at 1.1s —

a **small window** (e.g., `nperseg=128`) can show both as separate moments,

but a **large window** (e.g., `nperseg=1024`) will merge them together.

🎵 Frequency Precision (Frequency Resolution)

- **Definition:** How accurately the spectrogram can distinguish between **different frequencies**.
- **High frequency precision** → narrow, well-separated frequency peaks.
- **Low frequency precision** → peaks look broad or blended together.

🧠 **Example:**

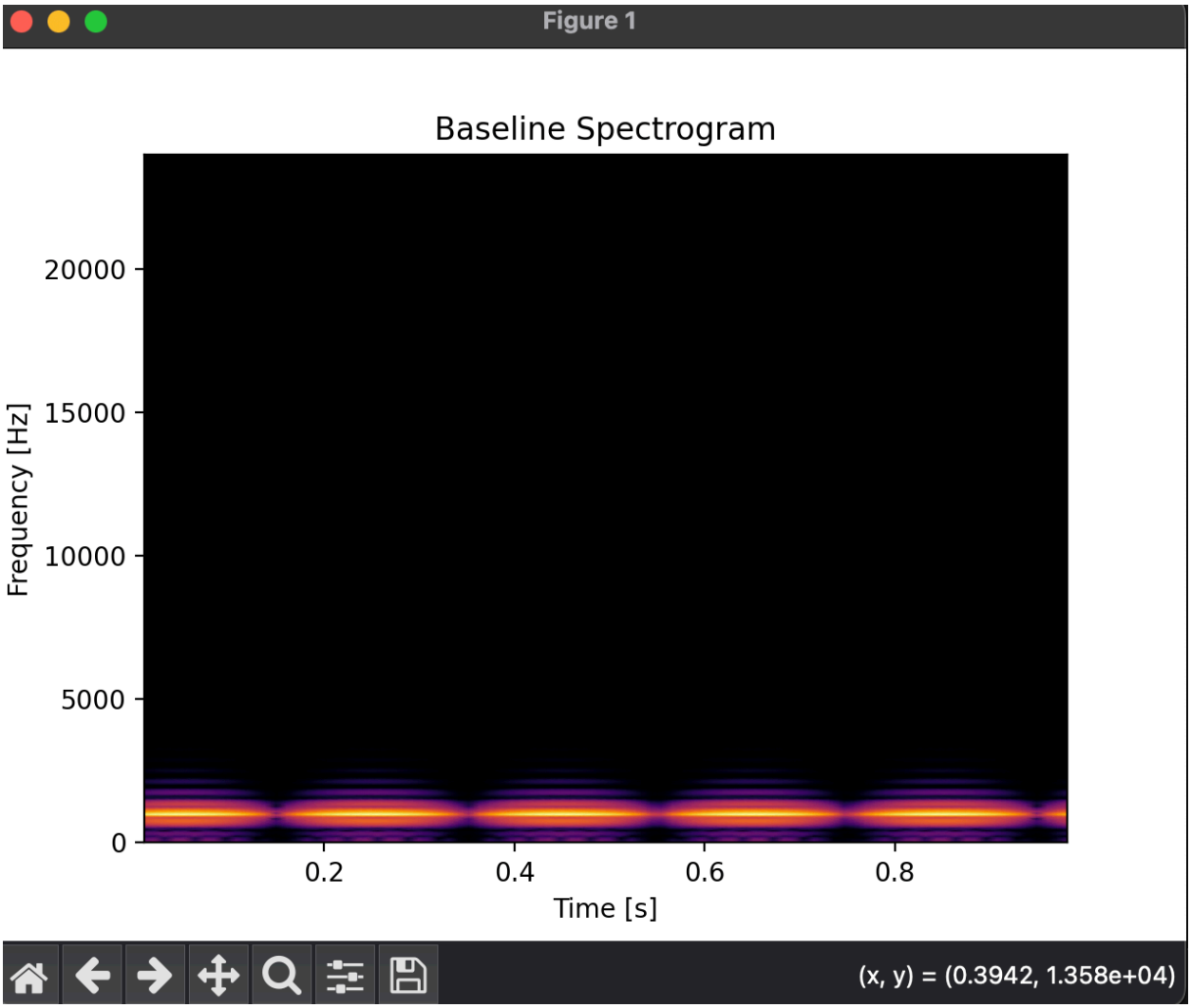
If you have two tones at 100 Hz and 110 Hz:

a **large window** (longer segment, more cycles) can clearly separate them,

but a **small window** can't — they blur into one wider band.

> **Spectral Leakage!**
>
> **Spectral leakage** happens when a signal segment **doesn't fit perfectly** into the FFT window — meaning the signal's cycles are **cut off** at the edges.
>
> When this happens, the FFT assumes the segment **repeats infinitely**, so those sharp edges look like sudden jumps or discontinuities.
>
> These discontinuities introduce **extra frequency components** that weren't actually in the original signal —
>
> so instead of a clean, sharp spike at one frequency, the energy **"leaks"** into nearby frequency bins.
>
> ---
>
> ## Example
>
> Imagine analyzing a 100 Hz sine wave sampled for 0.1 s:
>
> - If that 0.1 s segment contains an **integer number of cycles**, the FFT shows one clean line at 100 Hz.
> - If the segment cuts the wave mid-cycle, the FFT sees an abrupt jump at the window edges →
>
> the result spreads energy into frequencies like 90 Hz, 110 Hz, etc. — this is **leakage**.
>
> Visually, instead of a thin vertical line in the spectrogram, you'd see a **smeared band** across nearby frequencies.

Graphing an AM Signal




```
import numpy as np, matplotlib.pyplot as plt
from scipy import signal

f_c = 1000
f_m = 5

fs = 48000
t = np.linspace(0, 1, fs, endpoint=False)

carrier = np.sin(2 * np.pi * f_c * t)
modulator = 0.5 * (1 + np.sin(2 * np.pi * f_m * t))
sig = carrier * modulator
```

```
plt.figure(figsize=(10,4))
plt.plot(t[:2000], sig[:2000], color='blue')
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
plt.grid(True)
plt.tight_layout()
plt.show()


f, tt, Sxx = signal.spectrogram(sig, fs, nperseg=1024, noverlap=512)
plt.pcolormesh(tt, f, 10*np.log10(Sxx+1e-10), shading="gouraud", cmap="infer
no")
plt.xlabel("Time [s]"); plt.ylabel("Frequency [Hz]")
plt.title("Baseline Spectrogram"); plt.show()
```

Carrier

- The **bright horizontal band** (the most intense line) in the spectrogram = the **carrier frequency** (your 1 kHz tone).
- It stays constant in time, because it’s a continuous sine wave that doesn’t change frequency — it just *gets louder and quieter* when modulated.

Sidebands (modulators)

- The **slightly fainter, symmetric lines above and below** the carrier are your **sidebands** — energy created by the *modulation*.

🧠 Frequency resolution = fs / nperseg

When you run an FFT, you're splitting the frequency axis into discrete "bins."

If:

- `fs = 48000 Hz` (your sample rate)
- `nperseg = 1024` (window length, samples per FFT)

Then:

$$\text{Resolution per bin} = \frac{fs}{nperseg} = \frac{48000}{1024} \approx 46.9 \text{ Hz/bin}.$$

That means:

- The FFT divides the spectrum into 1024 slices, each ≈46.9 Hz wide.
- Frequencies that are *closer together than that* (like 995 Hz, 1000 Hz, 1005 Hz) will **fall into the same bin** — they blur together as one bright line.

📈 How to improve resolution

You need **narrower bins**, i.e., more points in each FFT.

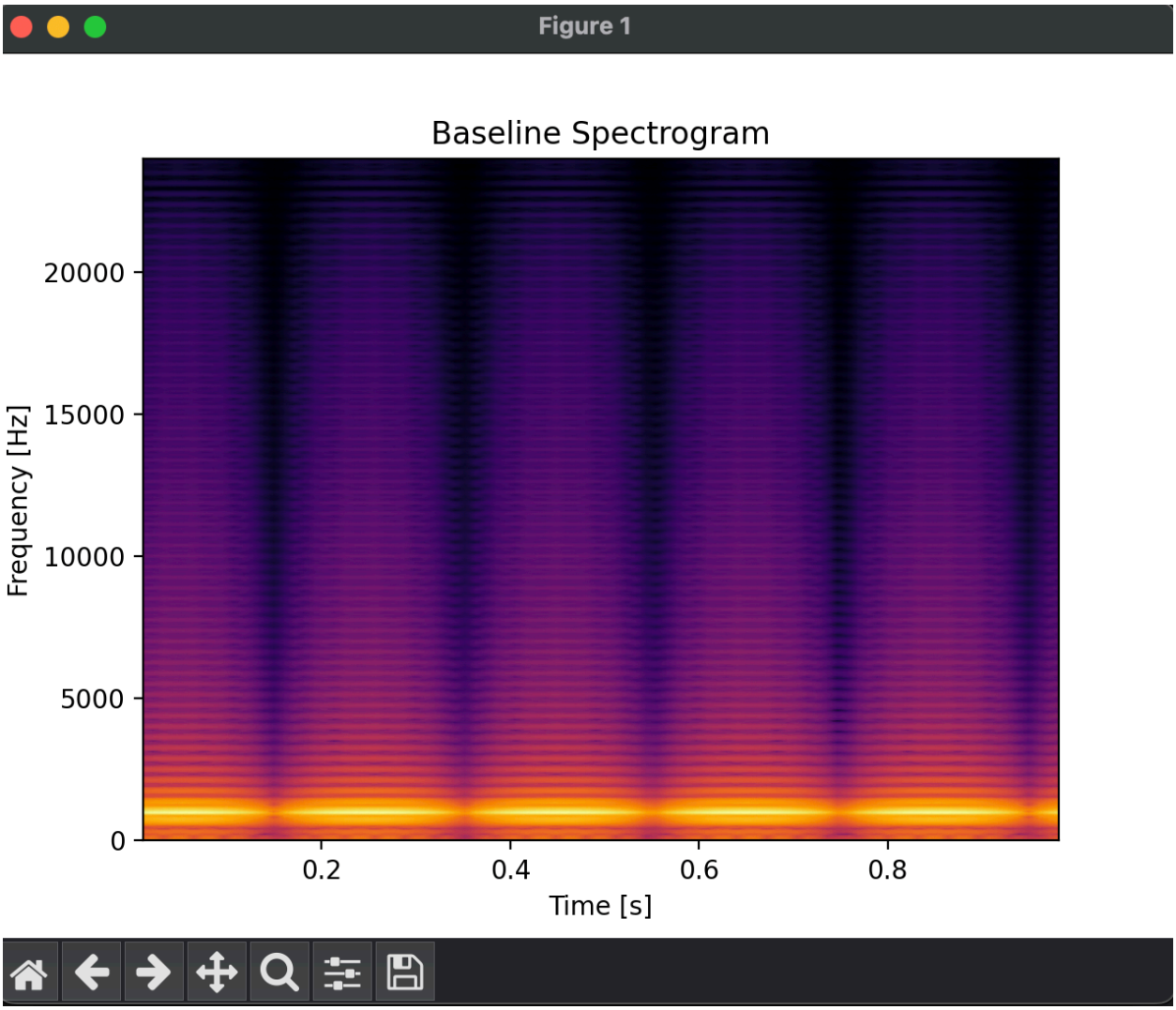
To do that, increase `nperseg`:

nperseg	Resolution (fs/nperseg)	Result
1024	46.9 Hz	too coarse — sidebands merge
8192	5.86 Hz	just about enough to separate ±5 Hz sidebands
16384	2.93 Hz	crisp separation of all three lines

Same graph but with Magnitude mode

It returns the **magnitude of the complex FFT values**, i.e. the **absolute amplitude** of each frequency component.

- It shows **how strong the signal is** at each frequency (not squared, not normalized).
- Units are **arbitrary amplitude units** — not physically meaningful (e.g. not Watts or Volts²).
- Useful for **visual comparison or qualitative analysis**, like spotting where energy is.

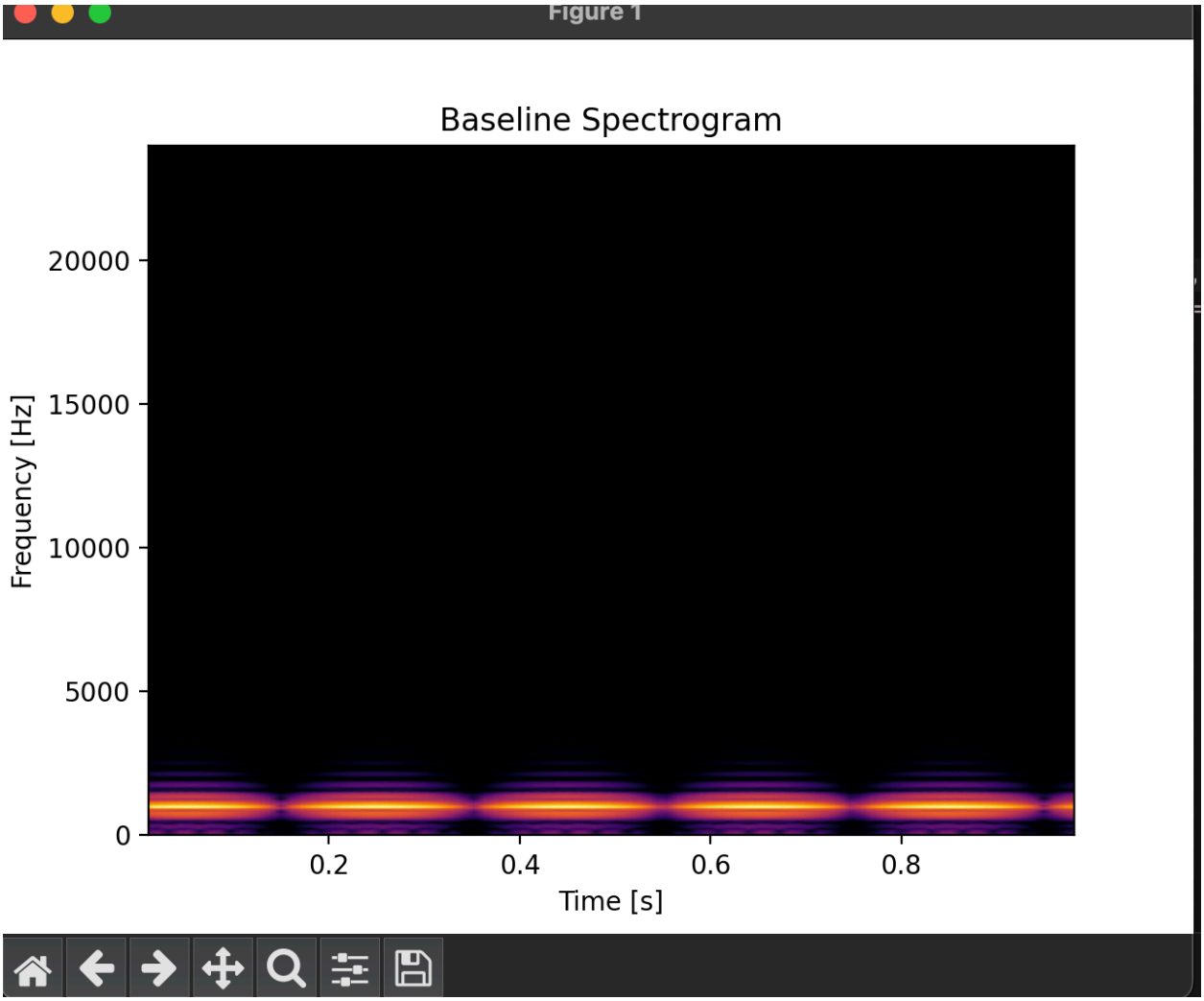


PSD Mode

It returns the **power per unit frequency (density)** for each frequency bin.

Meaning:

- It shows **how much signal power is present per Hz**.
- Units: typically V2/HzV2/Hz or W/HzW/Hz.
- It's physically meaningful — used when you want *true energy content* or *power comparisons*.



Adding decibel scaling

> Decibel Scaling: Decibel (dB) scaling is **a logarithmic scale used to express the ratio of two values, commonly sound intensity or power**. Instead of a linear scale, it uses logarithms to make it easier to represent a wide range of values with manageable numbers, so a 10 dB increase represents a 10-fold increase in power. Say for example a strong carrier has the power 1.0. Well, what if weak noise floor is something like 10^{-6}? Thats a million to one ratio, which is practically impossible to visualize on a linear scale.

So we compress that range **logarithmically** using **decibel scaling**.

For power (like PSD mode):

$$S_{xx,\mathrm{dB}} = 10\log_{10}(S_{xx})$$

Because decibels for power are defined as:

$$\mathrm{dB} = 10\log_{10}\left(\frac{P}{P_{\mathrm{ref}}}\right)$$

where P_{ref} is a reference power (often 1).

For amplitude (like magnitude mode):

$$S_{xx,\mathrm{dB}} = 20\log_{10}(|X|)$$

because power = amplitude², so the factor of 2 turns into ×20 instead of ×10.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

# --- Signal setup ---
f_c = 1000      # Carrier frequency
f_m = 5         # Modulation frequency
fs = 48000      # Sample rate
t = np.linspace(0, 1, fs, endpoint=False)

carrier = np.sin(2 * np.pi * f_c * t)
modulator = 0.5 * (1 + np.sin(2 * np.pi * f_m * t))
sig = carrier * modulator

# --- Time-domain plot ---
plt.figure(figsize=(10, 4))
plt.plot(t[:2000], sig[:2000], color='blue')
plt.xlabel("Time [s]")
```

```
plt.ylabel("Amplitude")
plt.title("Amplitude-Modulated Signal (Time Domain)")
plt.grid(True)
plt.tight_layout()
plt.show()

# --- Spectrogram ---
f, tt, Sxx = signal.spectrogram(sig, fs, nperseg=1024, noverlap=512, mode='ps
d')

# --- Decibel scaling ---
Sxx_dB = 10 * np.log10(Sxx + 1e-10)  # convert power to dB, avoid log(0)

# --- Plot the spectrogram in dB ---
plt.figure(figsize=(10, 5))
plt.pcolormesh(tt, f, Sxx_dB, shading="gouraud", cmap="inferno")
plt.xlabel("Time [s]")
plt.ylabel("Frequency [Hz]")
plt.title("Spectrogram (Power Spectral Density, dB scale)")
plt.colorbar(label="Power [dB]")
plt.tight_layout()
plt.show()
```

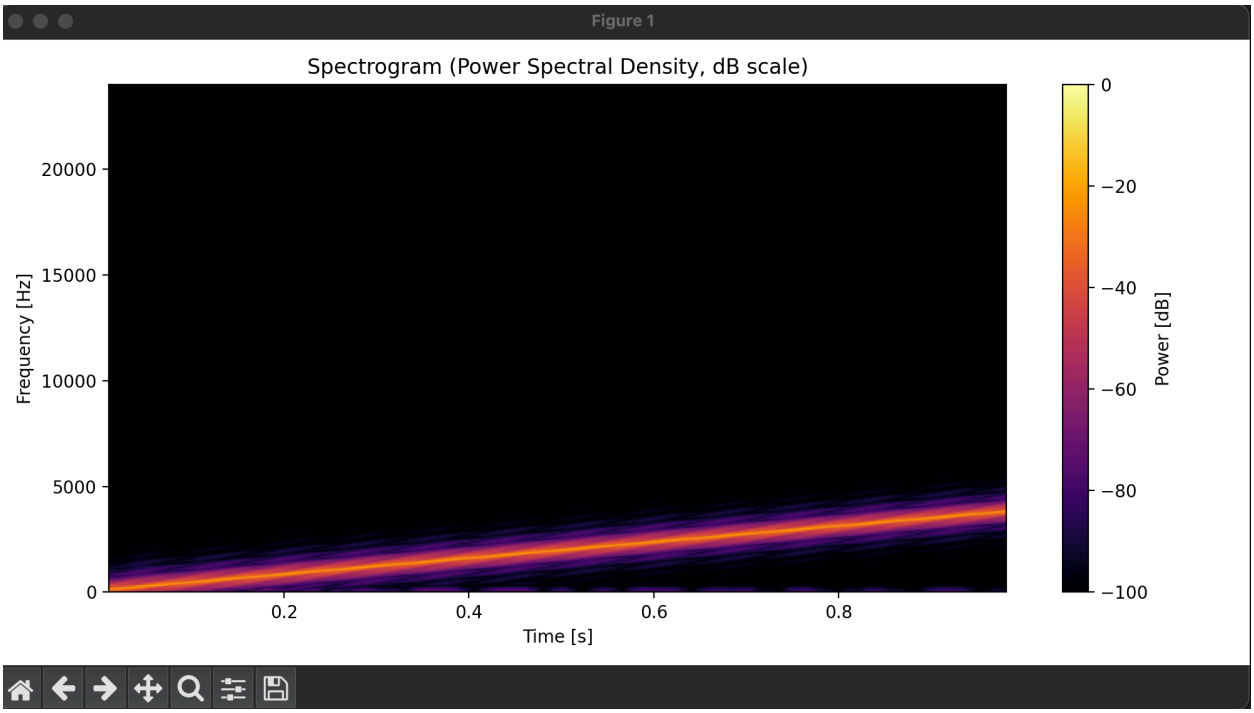
How do we plot for high or low-energy detail?

Use magnitude for high energy (where energy in concentrated), and use PSD mode for signal strengths quantitatively.

Chirp Signal

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
```

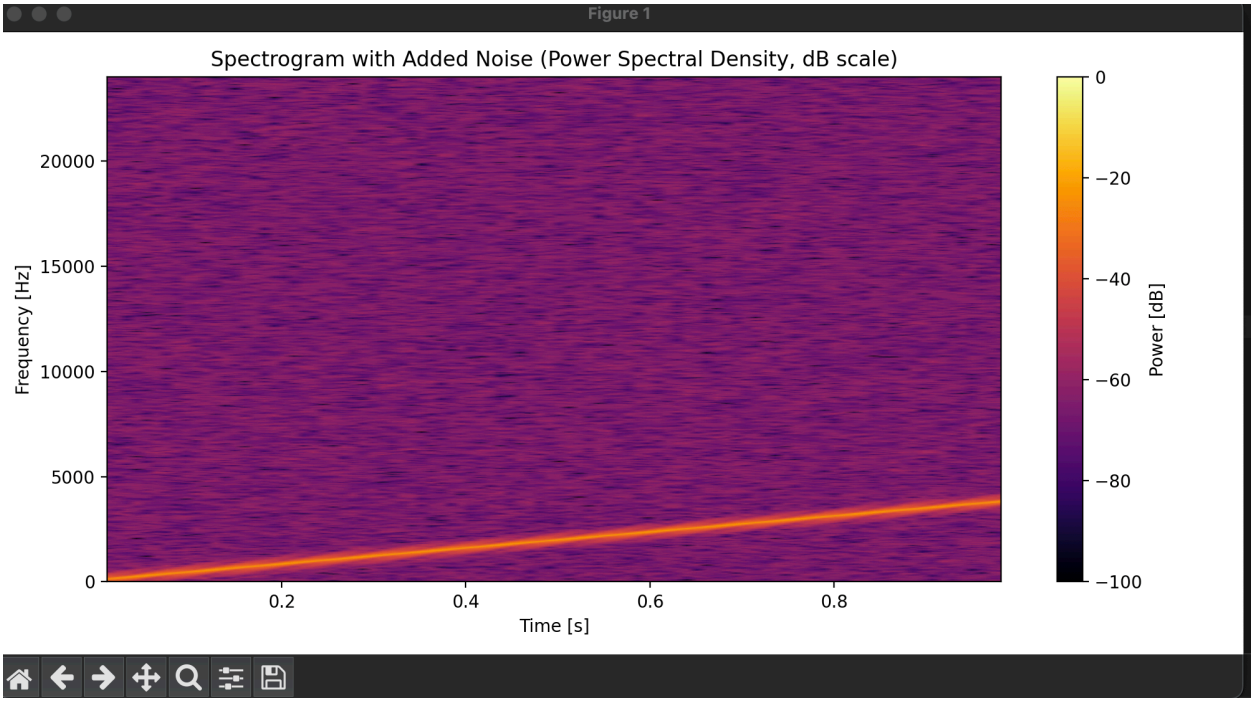
```
# --- Chirp Signal Setup ---
fs = 48000
t = np.linspace(0, 1, fs, endpoint=False)
sig = np.sin(2 * np.pi * (100 + 1900*t) * t)  # 100 → 2000 Hz chirp

# --- Spectrogram ---
f, tt, Sxx = signal.spectrogram(sig, fs, nperseg=1024, noverlap=512, mode='ps
d')

# Convert to dB
Sxx_dB = 10 * np.log10(Sxx + 1e-10)

# --- Plot and experiment ---
plt.figure(figsize=(10, 5))
plt.pcolormesh(tt, f, Sxx_dB, shading="gouraud", cmap="inferno",
               vmin=-100, vmax=0)  # ←—— experiment with these
plt.title("Spectrogram (Power Spectral Density, dB scale)")
plt.xlabel("Time [s]")
plt.ylabel("Frequency [Hz]")
plt.colorbar(label="Power [dB]")
plt.tight_layout()
plt.show()
```



Noisy Signal

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

# --- Chirp Signal Setup ---
fs = 48000
t = np.linspace(0, 1, fs, endpoint=False)
sig = np.sin(2 * np.pi * (100 + 1900*t) * t)  # chirp: 100 Hz → 2000 Hz

# --- Add Gaussian Noise ---
sig_noisy = sig + 0.1 * np.random.randn(len(t))  # 0.1 controls noise level

# --- Compute Spectrogram ---
f, tt, Sxx = signal.spectrogram(sig_noisy, fs, nperseg=1024, noverlap=512, mo
de='psd')

# --- Convert to Decibels ---
Sxx_dB = 10 * np.log10(Sxx + 1e-10)
```

```
# --- Plot Spectrogram ---
plt.figure(figsize=(10, 5))
plt.pcolormesh(tt, f, Sxx_dB, shading="gouraud", cmap="inferno", vmin=-100,
vmax=0)
plt.title("Spectrogram with Added Noise (Power Spectral Density, dB scale)")
plt.xlabel("Time [s]")
plt.ylabel("Frequency [Hz]")
plt.colorbar(label="Power [dB]")
plt.tight_layout()
plt.show()
```



Stage 5: Spectrogram Visualization and Animation

This stage converts each processed grayscale image frame into an **audio waveform** and then visualizes it as a **time-frequency spectrogram**, bridging visual and acoustic domains.

The resulting `.wav` file encodes brightness → amplitude mappings and frequency-based row assignments, allowing the original image to reappear when plotted back as a spectrogram.

Pipeline Summary

1. **Preprocessing**
   - Convert image → grayscale and normalize intensity `[0 – 1]`.
   - Flip vertically so **higher image rows = higher frequencies**.
   - Resize to a manageable resolution `(256 × 128)` for speed vs. detail.
2. **Signal Mapping**
   - Define frequency range: `100 Hz – 12 kHz`.
   - Each image row maps linearly to a frequency in this range.
   - Each image column corresponds to a short time slice (`duration_per_col = 0.05 s`).
3. **Synthesis**
   - For every column, generate a mini-frame by summing sine waves
   - Concatenate all column-frames → full continuous signal.
4. **Export & Normalization**
   - Normalize amplitude to ±1.
   - Write output as **16-bit PCM WAV** (`image_audio.wav`).
5. **Spectrogram Generation**
   - Compute spectrogram via `scipy.signal.spectrogram`.
   - Parameters:
     - `nperseg = 512` → balanced time/frequency resolution.
     - `noverlap = 128` → smooth transition between windows.
     - `scaling = "density"` for perceptually even color scaling.
6. **Visualization**
   - Use `plt.pcolormesh` to render in grayscale (`cmap="gray"`).

- Add log-scaled power (`10 × log₁₀(Sxx + 1e-10)`) for visual contrast.
- Label axes: **Time [s]** vs. **Frequency [Hz]**,

  with colorbar = **Power [dB]**.

Key Concepts

- **Amplitude ↔ Brightness:** pixel intensity modulates sine amplitude.
- **Frequency ↔ Vertical Position:** higher pixel rows = higher frequencies.
- **Time ↔ Horizontal Position:** columns determine signal duration sequence.
- **Spectrogram Reconstruction:** plotting recovers the original image pattern.

Next Steps

- Automate frame iteration across all *Bad Apple* frames for full animation.
- Experiment with:
  - `duration_per_col` (temporal speed)
  - `low/high` frequency mapping (spectral bandwidth)
  - color inversion or colormap variations for aesthetics.
- Optionally, overlay animation frames or export as a `.mp4` using `matplotlib.animation` or FFmpeg.

This stage converts each **video frame** of *Bad Apple!!* into a **time–frequency spectrogram**, transforming the visual domain into sound.

Each frame's brightness maps to amplitude, and vertical position maps to frequency — when plotted, the image re-emerges acoustically.

Overview

Concept	Mapping
Amplitude	Pixel brightness → sine wave amplitude
Frequency	Image row → frequency (100 Hz – 12 kHz)

Concept	Mapping
Time	Image column → short time slice (0.05 s)
Spectrogram	Reconstructed image pattern

Processing Pipeline

Frame Extraction

Uses **FFmpeg** to extract all frames from `badapple.mp4` into a `/frames` folder.

```
ffmpeg -i badapple.mp4 frames/frame_%04d.png
```

Image Preprocessing

- Convert each frame → grayscale
- Normalize pixel intensity `[0–1]`
- Flip vertically so higher image rows = higher frequencies
- Resize to `(256 × 128)` for performance balance

Signal Synthesis

Each column of pixels becomes a brief time slice:

$x(t) = \sum_{r} a_{r,c} \sin(2\pi f_r t)$

$x(t) = r\sum a_{r,c} \sin(2\pi f_r t)$

where:

- $a_{r,c}$ = pixel brightness (amplitude)
- f_r = frequency assigned to row r

All column signals concatenate into a continuous waveform representing the frame.

Spectrogram Generation

Compute spectrograms using **SciPy**:

```
f_vals, t_spec, Sxx = signal.spectrogram(
    signal_out,
    fs=48000,
    nperseg=512,
    noverlap=128,
    scaling="density"
)
```

Parameters:

- `nperseg=512` → balanced time/frequency detail
- `noverlap=128` → smoother transitions
- `scaling="density"` → even perceptual scaling

Visualization

Render with **Matplotlib**:

```
plt.pcolormesh(t_spec, f_vals, 10 * np.log10(Sxx + 1e-10),
               cmap="gray", shading="gouraud")
plt.xlabel("Time [s]")
plt.ylabel("Frequency [Hz]")
plt.colorbar(label="Power [dB]")
```

💻 Full Script: `run_batch.py`

```
import os, sys, subprocess, glob
import numpy as np

# --- make matplotlib save even in headless shells ---
import matplotlib
matplotlib.use("Agg")
```

```
import matplotlib.pyplot as plt

from PIL import Image, ImageOps
from scipy import signal

VIDEO_PATH = "badapple.mp4"
FRAME_DIR = "frames"
SPEC_DIR  = "output_spectrograms"

os.makedirs(FRAME_DIR, exist_ok=True)
os.makedirs(SPEC_DIR,  exist_ok=True)

# 1) Extract frames with FFmpeg
print("Extracting frames with FFmpeg...")
subprocess.run([
    "ffmpeg", "-hide_banner", "-loglevel", "error", "-y",
    "-i", VIDEO_PATH, os.path.join(FRAME_DIR, "frame_%04d.png")
], check=True)
print("✓ Frames extracted.")

# 2) Parameters
fs = 48000                      # Sampling rate
low, high = 100, 12000          # Frequency mapping range (Hz)
duration_per_col = 0.05          # Seconds per image column

frame_paths = sorted(glob.glob(os.path.join(FRAME_DIR, "frame_*.png")))
if not frame_paths:
    print("No frames found in ./frames — check VIDEO_PATH and permission
s.")
    sys.exit(1)

# 3) Frame loop
for idx, frame_path in enumerate(frame_paths, 1):
    try:
        # --- Preprocess image ---
        img = Image.open(frame_path)
```

```
    gray = ImageOps.grayscale(img).resize((256, 128))
    data = np.flipud(np.array(gray).astype(np.float32) / 255.0)

    rows, cols = data.shape
    freqs = np.linspace(low, high, rows)
    samples_per_col = int(duration_per_col * fs)
    t = np.linspace(0, duration_per_col, samples_per_col, endpoint=False)

    # --- Encode signal ---
    signal_out = np.zeros(0, dtype=np.float32)
    for col in range(cols):
        frame_sig = np.zeros(samples_per_col, dtype=np.float32)
        for r, f_hz in enumerate(freqs):
            amp = data[r, col]
            if amp != 0.0:
                frame_sig += amp * np.sin(2 * np.pi * f_hz * t)
        signal_out = np.concatenate((signal_out, frame_sig), axis=0)

    # --- Generate spectrogram ---
    f_vals, t_spec, Sxx = signal.spectrogram(
        signal_out, fs, nperseg=512, noverlap=128, scaling="density"
    )

    # --- Plot & save ---
    plt.figure(figsize=(10, 6))
    plt.pcolormesh(t_spec, f_vals, 10 * np.log10(Sxx + 1e-10),
                   cmap="gray", shading="gouraud")
    plt.title("Bad Apple!! — Spectrogram Encoding")
    plt.xlabel("Time [s]")
    plt.ylabel("Frequency [Hz]")
    plt.ylim([low, high])
    plt.colorbar(label="Power [dB]")
    plt.tight_layout()

    out_png = os.path.join(SPEC_DIR, f"spec_{idx:04d}.png")
    plt.savefig(out_png, dpi=120)
```

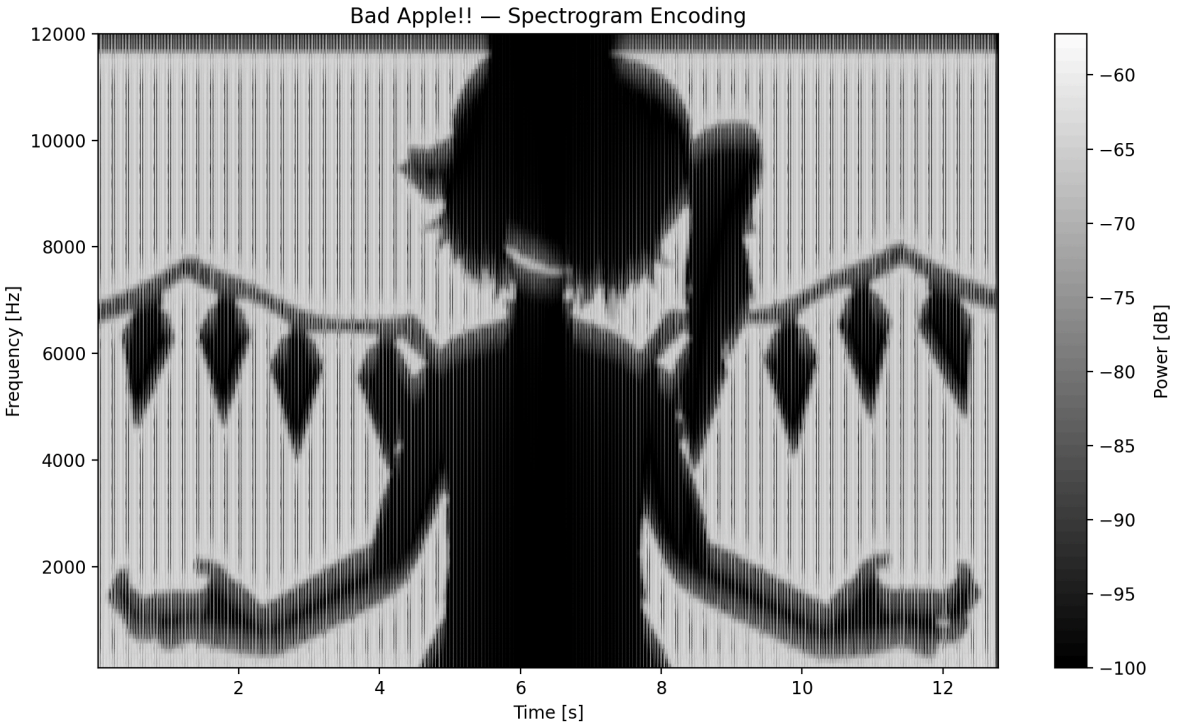
```
        plt.close()

        if idx % 50 == 0:
            print(f"Saved {out_png}")

    except Exception as e:
        print(f"[ERROR] Frame {idx} ({frame_path}) failed: {e}")
        raise

print(f"Done! Spectrograms saved in: {os.path.abspath(SPEC_DIR)}")
```

Notes

- **Headless compatibility:** `matplotlib.use("Agg")` allows rendering without a GUI.
- **Normalization:** Pixel intensities are in `[0–1]`, mapped directly to amplitude.
- **Performance:** For faster batch runs, use fewer frequencies or smaller frame sizes.
- **Automation:** You can recombine all spectrogram frames into an `.mp4`:

  ```
  ffmpeg -framerate 30 -i output_spectrograms/spec_%04d.png -c:v libx264 -pix_fmt yuv420p badapple_spec.mp4
  ```

🎬 Example Output


Bad Apple!! — Spectrogram Encoding
Frequency [Hz]
12000
10000
8000
6000
4000
2000
2
4
6
8
10
12
Time [s]
−60
−65
−70
−75
−80
−85
−90
−95
−100
Power [dB]