Chapter 12: Introduction to Image Analysis and Audio Signal Processing

School of International Liberal Studies
Waseda University
Introductory Data Science

Introduction to Image Analysis and Audio Signal Processing

Learning Objectives

- **Digital Images:**
  - *Represent Images:* Pixels, channels, resolution.
  - *Preprocess Images:* Resize, normalize, grayscale conversion.
  - *Detect Edges:* Sobel, Canny filters.
  - *Apply Basic Filters:* Blurring, sharpening, embossing.
- **Digital Sounds:**
  - *Understand Sound:* Amplitude, frequency, period, phase.
  - *Digitize Sound:* Analog to digital conversion.
  - *Interpret Spectral Data:* Time vs. frequency domains.

What is an Image?

- An **image** is a 2D grid of **pixels** representing visual data.
- Each **pixel** contains intensity or color information.
- Images can be grayscale (single-channel) or colored (RGB).
- *Mathematically*, an image is a map:

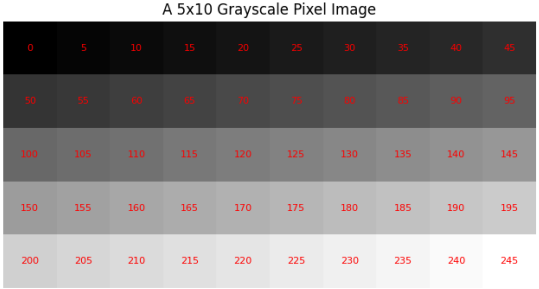
$$f(x, y) : [a, b] \times [c, d] \to [0, 255]$$

What are Pixels and Resolution

- **Pixel:** Smallest unit of an image.
  - *Grayscale:* Intensity from 0 (black) to 255 (white)
  - *RGB Color:* Combination of Red, Green, Blue intensities
- **Resolution:** Number of pixels (width $\times$ height), e.g., 1920×1080.

higher resolution $\rightarrow$ more detail

A Grayscale Image



Pixel intensity values range from 0 (black) to 255 (white)

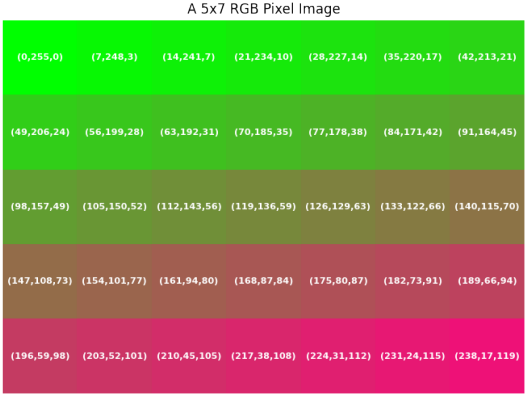
RGB Color Image Representation

In RGB images, each pixel is a vector of RGB intensities:

$$f(x,y) = \begin{bmatrix} R(x,y) \\ G(x,y) \\ B(x,y) \end{bmatrix}$$

- Combines Red, Green, Blue to form full color.
- Each channel ranges from 0–255.

An RGB Image



Each pixel combines RGB intensities to produce colors.

Example Color and Grayscale Images from `skimage`

- Two 512×512 images of astronaut Eileen Collins
- Grayscale derived by removing RGB channels, retaining intensity.
- Grayscale images reduce data complexity, useful in analysis.
- Source: `skimage` Python library, popular for image processing tasks:
  - Segmentation
  - Filtering
  - Feature Extraction

Grayscale Image (Astronaut Eileen Collins)

Color Image (Astronaut Eileen Collins)

Image Preprocessing

- **Objective:** prepares image data for machine learning and computer vision tasks
- **Common preprocessing steps:**
  - *resizing*
  - *normalization*
  - *augmentation*

Image Dimensions vs. Resolution

- **Dimensions (image size):**
  - Refer to the *width* and *height* of an image in *pixels*
  - Example: An image sized 1920×1080 has 1920 pixels in width and 1080 in height
- **Resolution:**
  - Refers to the *density of pixels* in a physical space
  - Measured in *PPI (pixels per inch)* or *DPI (dots per inch)*
  - Higher resolution = more pixels per inch $\rightarrow$ *sharper and more detailed prints*

Image Resizing: What & Why

- **What is Image Resizing?** To change an image's dimensions (width $\times$ height in pixels) May or may not affect the *resolution* (PPI or DPI) *Upscaling* can introduce blur or distortion
  Downscaling may lead to loss of fine detail
- **Why resize an image?** Ensure *consistent input size* for machine learning models *Reduce file size* for storage efficiency or faster computation

Interpolation Methods

- When resizing, pixel values must be estimated:
  - **Downscaling** $\rightarrow$ average pixels over regions.
  - **Upscaling** $\rightarrow$ insert new pixels based on estimation.
- Common methods:
  - **Nearest-Neighbor** – Fast but pixelated (jagged or blocky)
  - **Bilinear** – Smooths edges but can blur
  - **Bicubic** – Smoothest, uses more neighboring pixels

Interpolation Methods Compared

- **Nearest-Neighbor Interpolation**
  - Uses the *closest existing pixel* for the new pixel
  - Fast and simple; but can be *blocky*
- **Bilinear Interpolation**
  - Takes the **average of the 4 nearest neighbors**
  - smoother results than nearest-neighbor; may cause *blurring*
- **Bicubic Interpolation**
  - Uses a *4×4 grid (16 neighbors)* and cubic polynomials
  - Produces *visually smooth and refined* images

Low Resolution Versions of Image

(a) Reduced to 256x256 pixels

(b) Reduced to 128x128 pixels

(c) Reduced to 64x64 pixels

All resized from 512×512 image using `cv2.INTER_AREA`

Enlarged Reduced Images with Interpolation Methods

- (a): Reduced 64×64 grayscale version
- (b): Enlarged with **nearest-neighbor** → *pixelated*
- (c): Enlarged with **bilinear** → smoother but *blurred*
- (d): Enlarged with **bicubic** → *refined*, smooth results

What is Image Filtering

- **Image filtering:** modifies pixel values
  - to enhance features or
  - reduce noise
- **Common goals:**
  - *sharpening*
  - *smoothing*
  - *edge detection*
- **Implementation:** via *2D convolution* using small matrices

Convolution by Example (1/5)

- I: original image; K: kernel for *vertical edge detection*

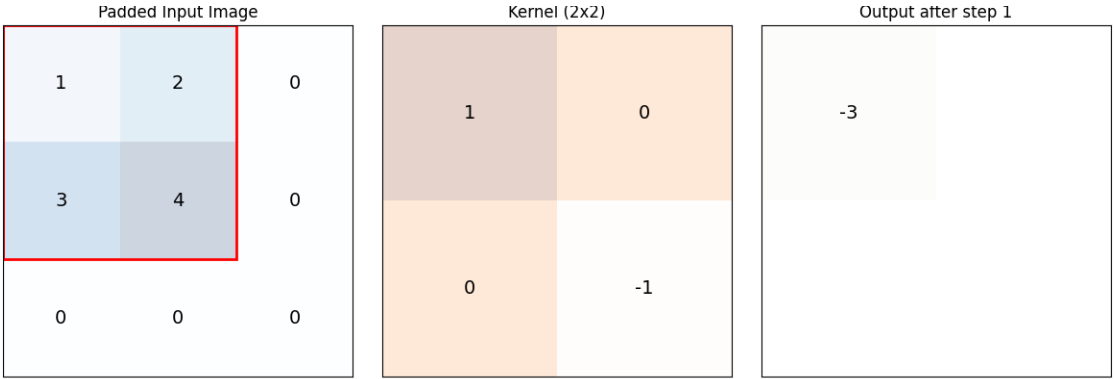
$$K = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad I = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

- Apply **zero-padding** to I:

$$I_{\text{padded}} = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

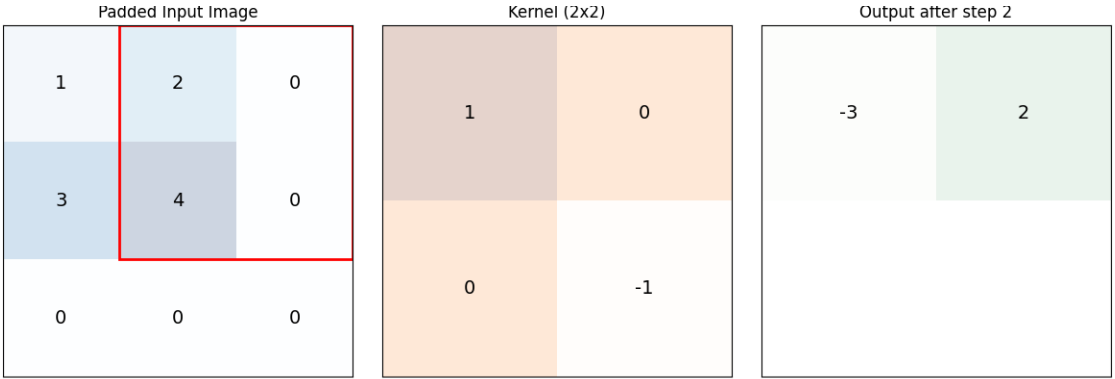
Convolution by Example (2/5)

$$1 \times 1 + 2 \times 0 + 3 \times 0 + 4 \times (-1) = -3$$



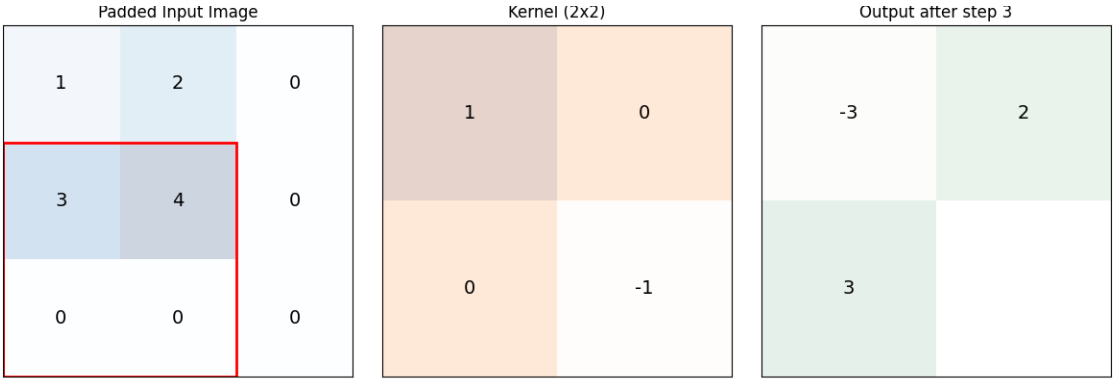
Convolution by Example (3/5)

$$2 \times 1 + 0 \times 0 + 4 \times 0 + 0 \times (-1) = 2$$



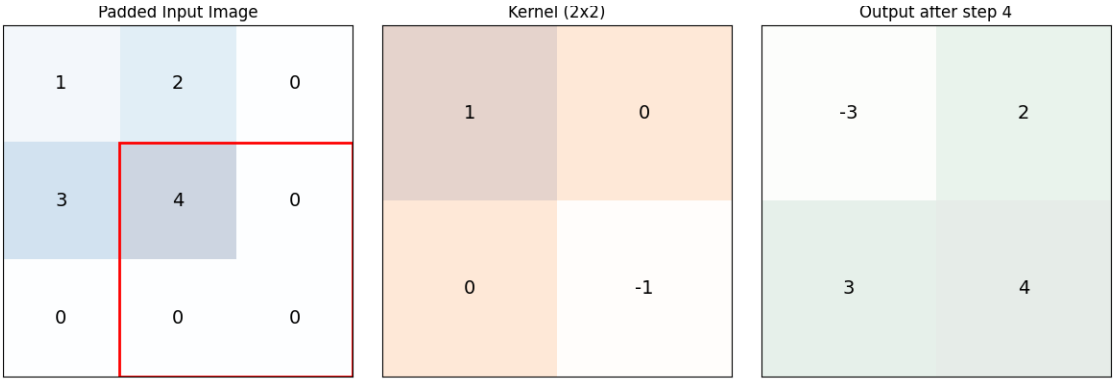
Convolution by Example (4/5)

$$3 \times 1 + 4 \times 0 + 0 \times 0 + 0 \times (-1) = 3$$



Convolution by Example (5/5)

$$4 \times 1 + 0 \times 0 + 0 \times 0 + 0 \times (-1) = 4$$



2D Convolution: Summary

- Slide a small matrix (**kernel**) across the image
- At each position, compute a **weighted sum** of the neighborhood:

$$I'(x,y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} K(i,j) \cdot I(x+i, y+j)$$

  - $I(x,y)$: input image pixel; $I'(x,y)$: output image pixel
  - $K(i,j)$: kernel weight; k: half-size of kernel

Image Smoothing via Average Filter

- Reduces noise by averaging pixel values
- Results in *blurring*, especially at edges and fine details
- Kernel with equal weights:

$$K_{\text{avg}} = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

```
import numpy as np
from scipy.ndimage import convolve

average_filter = np.ones((3, 3)) / 9.0 # 3x3 average filter
filtered_image = convolve(image, average_filter)
# apply the filter
```

Original Grayscale Image

Filtered Image (3x3 Average)

Edge Detection via Sobel & Canny Filters

- **Edge detection:** to identify sudden changes in intensity
- **Sobel filter** computes gradients:
  - Horizontal filter: $K_{\text{sobel\_x}}$; vertical filter: $K_{\text{sobel\_y}}$

$$K_{\text{sobel\_x}} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad K_{\text{sobel\_y}} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

- **Canny filter**: Combines noise reduction, gradient calculation, suppression, and edge tracking

Applying Sobel and Canny in Python

```
from skimage.filters import sobel
from skimage.feature import canny

# Sobel edge detection
edge_sobel = sobel(image)

# Canny edge detection
edge_canny = canny(image, sigma=sigma)
```

- `sobel(image)`: emphasizes gradient changes
- `canny(image, sigma)`: smoothing controlled by sigma (σ):
  - Higher sigma $\rightarrow$ smoother edges, less noise
  - Lower sigma $\rightarrow$ sharper edges, more noise-sensitive

Applying Sobel and Canny in Python

Original Grayscale Image

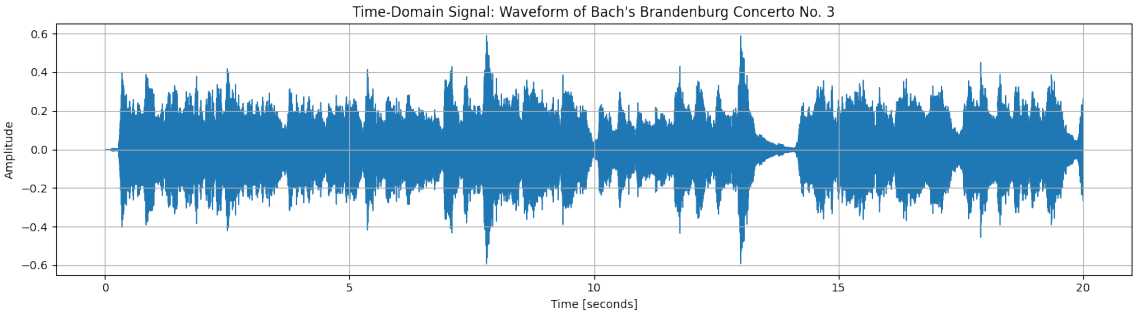
Edge Detection (Sobel)

Edge Detection (Canny, $\sigma = 1$)

What Is Sound?

- Sound is a **mechanical wave** that travels through air or another medium.
- Basic properties of sound:
  - **Amplitude** → loudness
  - **Frequency** → pitch
  - **Period** → time of one cycle
  - **Phase** → shift in wave start
- Human hearing interprets complex waveforms via *timbre* (quality of sound; 音色)

Sound as a Time-Domain Signal



- X-axis: time (seconds)
- Y-axis: amplitude (volume)
- Shows dynamic variations over 20 seconds

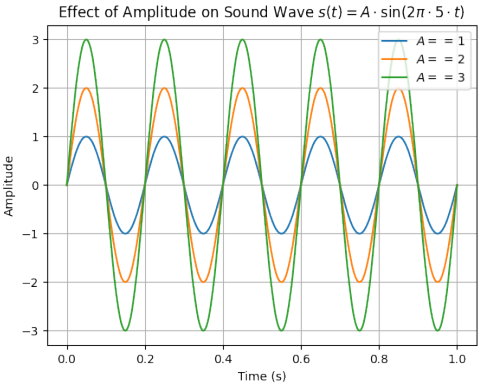
A Mathematical Model of Sound

A simple periodic sound wave is modeled as:

$$s(t) = A \cdot \sin(2\pi f t + \phi)$$

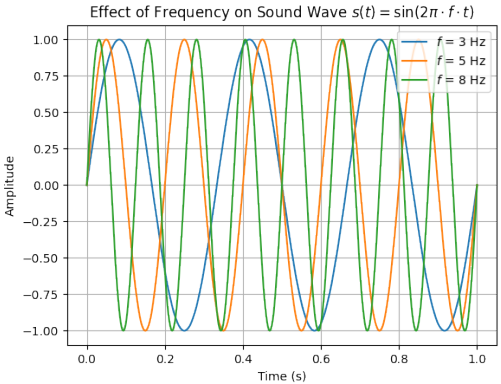
- A: Amplitude (loudness)
- f: Frequency (Hz, pitch)
- t: Time (seconds)
- ϕ: Phase (horizontal shift)

Understanding Amplitude



- Amplitude: height of the wave
  - Larger $A \rightarrow$ louder
  - Smaller $A \rightarrow$ quieter

Understanding Frequency



- Frequency f: how often wave repeats (cycles/second)
  - Measured in Hertz (Hz)
  - $f = 1$ Hz $\rightarrow$ 1 wave per second
  - $f = 2$ Hz $\rightarrow$ 2 waves per second
- Higher $f \rightarrow$ Higher pitch
- Lower $f \rightarrow$ Lower pitch

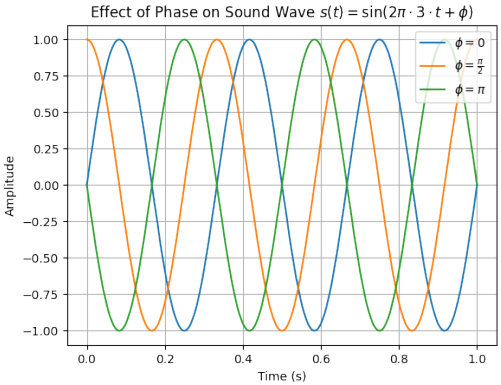
Period and Frequency Relationship

- **Period:** (T) time to complete one cycle
- Related to frequency:

$$T = \frac{1}{f}$$

- Examples:
  - $f = 1$ Hz $\rightarrow T = \frac{1}{1} = 1$ second
  - $f = 2$ Hz $\rightarrow T = \frac{1}{2} = 0.5$ second
  - $f = 4$ Hz $\rightarrow T = \frac{1}{4} = 0.25$ second

Understanding Phase



- Phase ϕ (pronounced /fi /): controls *horizontal shift* of the wave
  - Two waves with same A, f but different ϕ start at different points
- Useful in aligning or comparing signals

Digital Representation of Sound

Analog-to-Digital Conversion (ADC): to analyze sound on computers, we convert analog waves to digital through:

- **Sampling**: measure wave at fixed intervals
- **Quantization**: round values to nearest digital level

Sampling & Quantization in Digital Audio

- **Sampling**: Measure sound amplitude at fixed time intervals
  - Sampling rate = samples per second (Hz)
  - Example: 44,100 samples/sec = 44.1 kHz
  - Higher rate → captures more detail
- **Quantization**: Convert each sample to a digital level
  - Bit depth = number of possible amplitude values
    - 16-bit → $2^{16} = 65{,}536$ levels
    - 24-bit → $2^{24} = 16{,}777{,}216$ levels
  - More bits → better accuracy, larger file size

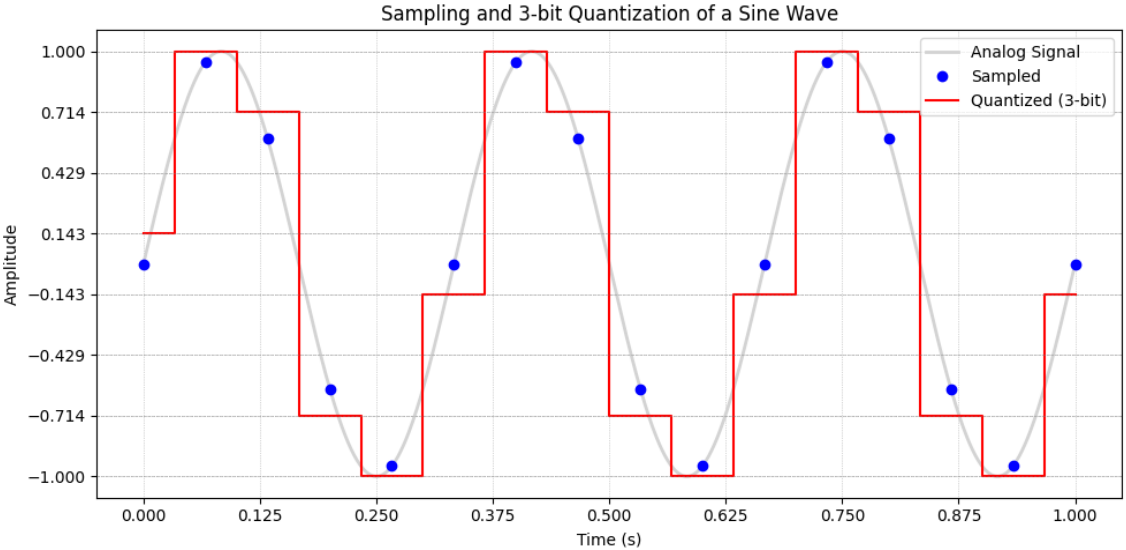


Figure 1: bg contain

Preprocessing Audio for ML Models

- Before deep learning: feature engineering done manually
  - CV: edge detectors, corners
  - NLP: N-grams, term frequencies
  - Audio: phoneme (音素) extraction
- Deep learning replaces manual steps:
  - Learns features automatically from raw data
  - Converts audio to **spectrograms** $\rightarrow$ process with CNNs

What Is a Spectrum?

- Every sound = mix of many frequencies
- **Spectrum** = snapshot of all frequencies present + their amplitudes
- *Fundamental frequency*: lowest component
- *Harmonics*: integer multiples of the fundamental

Spectrum helps reveal the “ingredients” of complex sounds

Time Domain vs. Frequency Domain

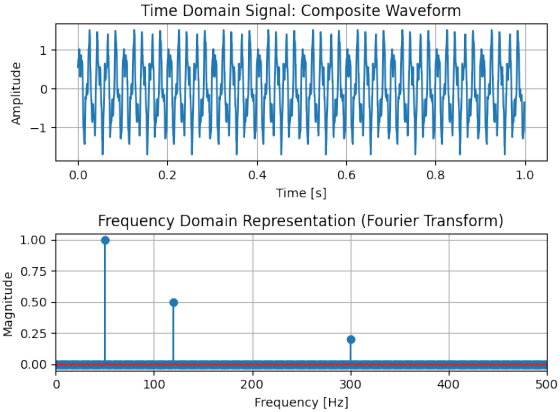
- **Time domain**: amplitude over time (waveform)
- **Frequency domain**: magnitude vs frequency via *Fourier Transform*:

$$X(f) = \int_{-\infty}^{\infty} x(t)\, e^{-j2\pi ft}\, dt$$

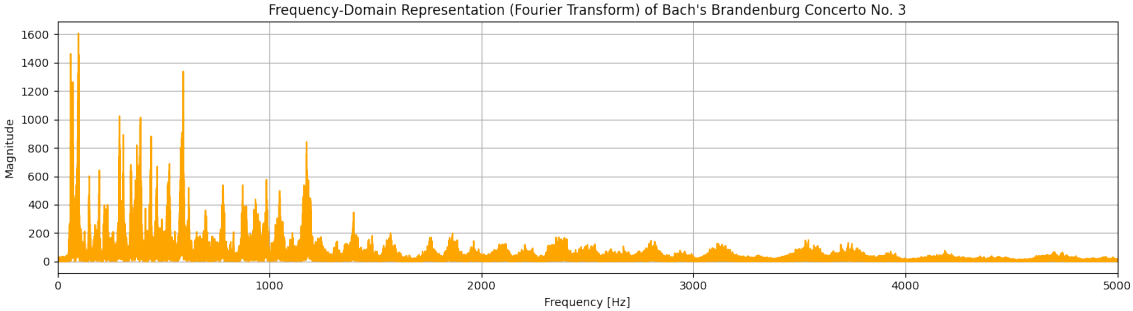
- f: frequency in Hz;
- i: imaginary unit ($i^2 = -1$)
- $X(f)$: complex amplitude for each frequency

Example of a simple composite signal:

$$s(t) = A_1 \cdot \sin(2\pi \cdot 50 \cdot t) + A_2 \cdot \sin(2\pi \cdot 120 \cdot t + \frac{\pi}{4}) + A_3 \cdot \sin(2\pi \cdot 300 \cdot t + \frac{\pi}{2})$$



Bach No. 3 in Frequency Domain

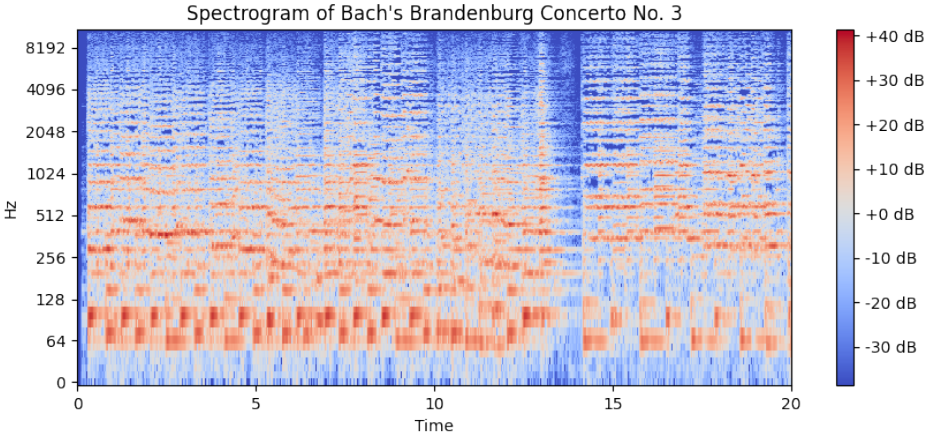


- Peaks = dominant musical notes / harmonics
- Helps analyze timbre, instrument identity, and energy

What Is a Spectrogram?

- A **spectrogram** shows *frequency content over time*:
- X-axis = time
- Y-axis = frequency
- Color = amplitude (in dB)
- Each vertical slice = frequency spectrum at that time
- dB: logarithmic unit measuring amplitude, aligns with human hearing perception

Red = stronger frequency component; Blue = weaker amplitude



- Lower frequencies (0-512 Hz): bass; Higher frequencies: high-pitch instruments
- Visual patterns reflect musical structure & intensity

Process to Create a Spectrogram

1. **STFT (Short-Time Fourier Transform)**
   - Slice signal into windows
   - Apply Fourier Transform to each
2. Convert to decibels:

$$X_{\text{dB}}(t, f) = 20 \cdot \log_{10}(|X(t, f)|)$$

   - $X(t, f)$: complex-valued output of the STFT
   - $|X(t, f)|$: amplitude at time t and frequency f
   - $X_{\text{dB}}(t, f)$: amplitude expressed in decibels (dB scale)
3. Plot: X = time, Y = frequency, Color = amplitude

Decibel Scale (dB)

- Reflects how humans perceive loudness
- Logarithmic scale:
  - 0 dB = silence (threshold of hearing)
  - 10 dB = 10× more intense
  - 20 dB = 100×, 30 dB = 1,000×

Spectrograms usually use dB for intensity visualization

Siginal Processing with Python: Install and Load Audio

```
!pip install librosa
import librosa

# Load example audio (20 sec)
audio_path = librosa.example('brahms')
audio_data, sample_rate = librosa.load(audio_path, duration=20)
```

- `librosa.example()`: loads demo file
- Default sample rate: 22050 Hz

Siginal Processing with Python: Plot Waveform

```
import matplotlib.pyplot as plt

plt.plot(audio_data)
plt.title('Time-Domain Waveform')
plt.xlabel('Samples')
plt.ylabel('Amplitude')
plt.show()
```

Useful to inspect loudness and silence over time

Siginal Processing with Python: Generate Spectrogram

```
import librosa.display

D = librosa.stft(audio_data)                        # Step 1: STFT
D_db = librosa.amplitude_to_db(abs(D))              # Step 2: Convert to dB

librosa.display.specshow(D_db, sr=sample_rate)    # Step 3: Plot
plt.colorbar(format='%+2.0f dB')
plt.title('Spectrogram')
plt.show()
```

Result: time-frequency representation for audio analysis