A Primer on Wavelet Analysis: From Fourier's Limit to Practical Code

A Guided Introduction

November 17, 2025

Abstract

This document provides a comprehensive, yet accessible, introduction to the theory and application of Wavelet Analysis. We build intuition by starting with the limitations of the Fourier Transform, mathematically define the Continuous and Discrete Wavelet Transforms (CWT and DWT), and provide practical Python code examples for implementation.

Contents

1 Preamble: The Inner Product as a "Similarity" Engine

Before we dive into *why* we need wavelets, we must first understand the central mathematical tool that all ofthese transforms (including Fourier) are built on: the **inner product**.

Your intuition that the inner product measures "equi-directionality" is exactly correct. This idea can be extended from simple 2D vectors to infinitely-dimensional "vectors" we call functions.

1.1 Intuition from Vectors

In 2D, the inner product (or dot product) of $\vec{v}$ and $\vec{w}$ is:

$$\vec{v} \cdot \vec{w} = |\vec{v}||\vec{w}|\cos(\theta)$$

The key term is $\cos(\theta)$.

- **High (Match):** If $\vec{v}$ and $\vec{w}$ point in the same direction, $\theta = 0$ and $\cos(\theta) = 1$. The inner product is at its positive maximum.
- **Zero (Unrelated):** If they are perpendicular (orthogonal), $\theta = 90^\circ$ and $\cos(\theta) = 0$. The inner product is zero.
- **Low (Unmatch):** If they point in opposite directions, $\theta = 180^\circ$ and $\cos(\theta) = -1$. The inner product is at its negative maximum.

1.2 Extending to Functions

Now, imagine a function $f(t)$ is just a vector with an infinite number of "dimensions," where each point t is a new dimension. How do we compute the inner product?

The dot product for discrete vectors is $\sum v_i w_i$. The continuous, infinite-dimensional equivalent of a sum ($\sum$) is an integral ($\int$).

Therefore, the inner product of two real functions $f(t)$ and $g(t)$ is:

$$\langle f, g \rangle = \int_{-\infty}^{\infty} f(t)g(t)dt$$

This integral is just a single number. This number tells us *exactly* the same thing as the vector dot product: how "similar" are the two functions?

- **High (Match):** If $f(t)$ and $g(t)$ are "in sync" (where f is positive, g is positive; where f is negative, g is negative), their product $f(t)g(t)$ will be positive everywhere. The integral of all this positive area will be a **large positive number**.
- **Zero (Unrelated):** If $f(t)$ and $g(t)$ are "out of sync" (they are orthogonal), the product $f(t)g(t)$ will have exactly as much positive area as it has negative area. The integral of these cancelling areas will be **zero**.
- **Low (Unmatch):** If $f(t)$ and $g(t)$ are "perfectly opposite" (where f is positive, g is negative), their product $f(t)g(t)$ will be negative everywhere. The integral of all this negative area will be a **large negative number**.

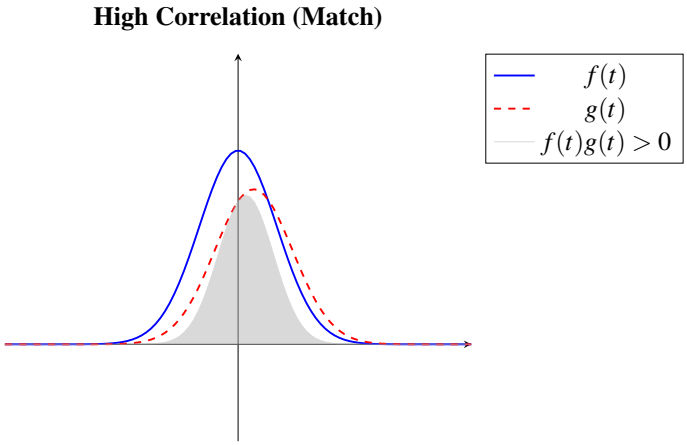

Figure 1: When $f(t)$ and $g(t)$ are "in sync," their product $f(t)g(t)$ is always positive, leading to a large positive integral.

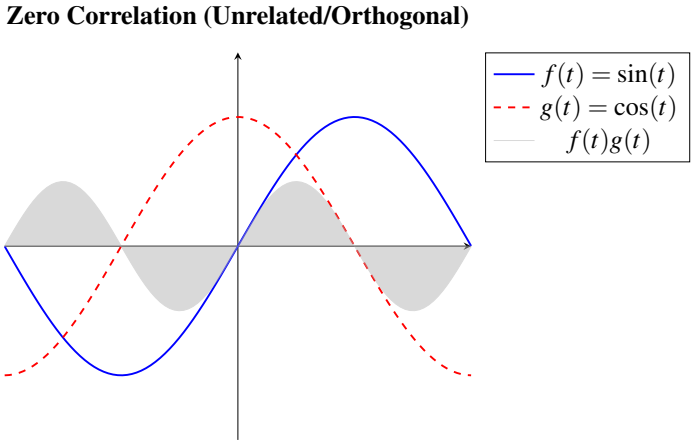

Figure 2: For orthogonal functions like $\sin(t)$ and $\cos(t)$ (over $[-\pi, \pi]$), the positive and negative areas of their product cancel out, integrating to zero.

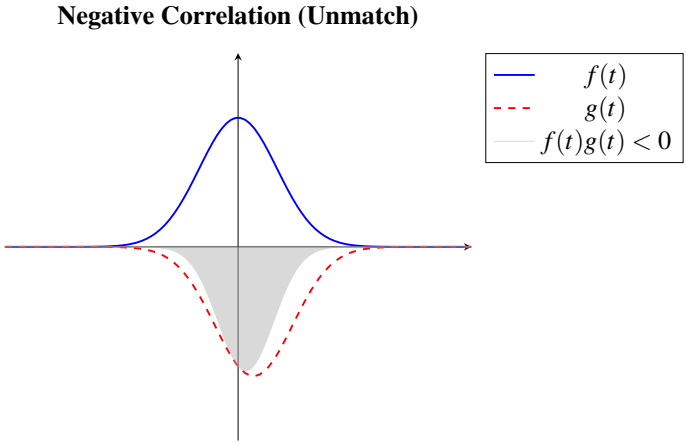

Figure 3: When $f(t)$ and $g(t)$ are "out of sync," their product $f(t)g(t)$ is always negative, leading to a large negative integral.

1.3 From Inner Product to Convolution

What, then, is a convolution or a transform?

$$\text{CWT: } W_f(u,s) = \int_{-\infty}^{\infty} f(t)\psi^*\left(\frac{t-u}{s}\right)dt$$

$$\text{Fourier: } \hat{f}(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt$$

Look closely. These are not just inner products. They are **families of inner products**.

A "transform" like the CWT is not a single calculation; it is an engine for performing a **sliding inner product**. We aren't just asking, "How similar is our signal $f(t)$ to the wavelet $\psi(t)$?" We are asking a much more powerful question:

"How similar is our signal $f(t)$ to the wavelet $\psi(t)$...

- ...when the wavelet is at position u_1 and scale s_1?" (Calculate inner product)
- ...when the wavelet is at position u_2 and scale s_1?" (Calculate inner product)
- ...when the wavelet is at position u_3 and scale s_2?" (Calculate inner product)
- ...for all possible u and s?"

The resulting "scaleogram" (or "spectrogram" for Fourier) is a giant map where every single pixel's brightness is the result of one inner product, telling you the "similarity score" at that specific time (u) and scale (s).

This is the bridge. **An inner product is a similarity score. A transform is a map of similarity scores.**

2 The Problem with Fourier: The "Why"

2.1 The Fourier Transform: A "What," Not a "When"

The **Fourier Transform** is a cornerstone of signal processing. It is built on a profound idea: any complex signal $f(t)$ can be perfectly reconstructed as a sum of simple sine and cosine waves (or complex exponentials).

The transform $\hat{f}(\omega)$ answers the question: "How much of frequency ω is in my signal?"

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt$$

This is, as we just discussed, an inner product. We are asking: "How similar is our signal $f(t)$ to a 'pure' infinite wave $e^{-i\omega t}$?"

The Problem: The basis functions (sines and cosines) are *infinite in time*. They have perfect "frequency localization" (we know exactly what frequency they are) but zero "time localization" (they exist everywhere, so we don't know *when* they are).

If a signal contains a high-frequency chirp at $t = 5$ seconds and a low-frequency rumble at $t = 10$ seconds, the Fourier Transform will simply tell you: "This signal contains both high and low frequencies." It cannot tell you *when* they occurred.

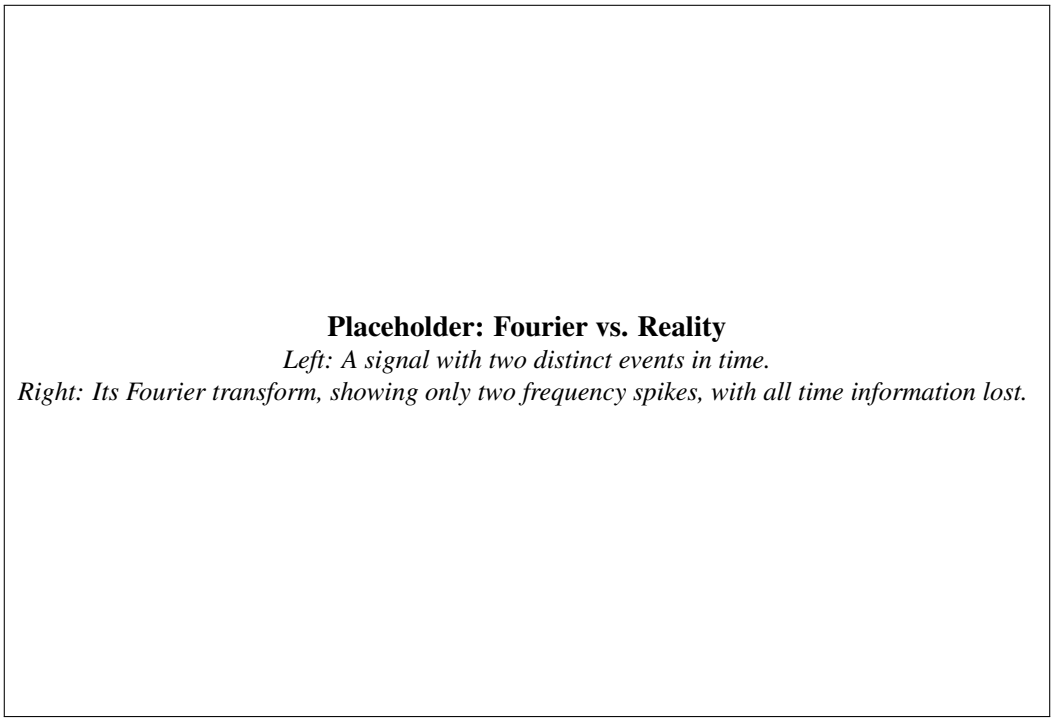

Figure 4: The Fourier Transform identifies *what* frequencies are present, but not *when* or *where* they occur in the signal.

2.2 The Gabor Transform: A First Attempt

The first logical step to fix this is the **Short-Time Fourier Transform (STFT)**, or Gabor Transform. The idea is simple:

1. Choose a "window" function $g(t)$ (like a Gaussian) that is localized in time.
2. Slide this window along your signal $f(t)$.

3. Perform a Fourier Transform only on the "windowed" part of the signal.

$$\text{STFT}_f(u,\omega) = \int_{-\infty}^{\infty} f(t)g(t-u)e^{-i\omega t}dt$$

This works! It gives us a **spectrogram**, which plots frequency content over time.

The New Problem: This introduces the **Heisenberg-Gabor Uncertainty Principle**. The size of our window $g(t)$ is fixed.

- A **narrow window** gives good time localization (we know *when* it happened) but poor frequency localization (the smearing from the window blurs all the nearby frequencies).
- A **wide window** gives good frequency localization (we can resolve close frequencies) but poor time localization (we only know it happened "sometime in this wide window").

This fixed window size is the critical flaw. We can't analyze high-frequency (short-lived) and low-frequency (long-lived) events with the same "ruler."

3 The Wavelet Solution: A "What" *and* a "When"

Wavelet analysis solves the fixed-window problem by using a "window" (the wavelet) that can *stretch* and *shrink*.

3.1 The Mother Wavelet and her "Children"

We start with a single "Mother Wavelet" function, $\psi(t)$. This function must be localized in both time and frequency and must have a zero mean (it must be a "wave").

$$\int_{-\infty}^{\infty} \psi(t)dt = 0$$

A classic example is the Morlet wavelet, which is just a sine wave inside a Gaussian window.

From this one "mother," we generate all our basis functions (the "children") by two operations:

1. **Translation (u):** Sliding the wavelet in time, just like in STFT.
2. **Scaling (s):** Stretching or shrinking the wavelet.

This gives us the wavelet family $\psi_{u,s}(t)$:

$$\psi_{u,s}(t) = \frac{1}{\sqrt{s}}\psi\left(\frac{t-u}{s}\right)$$

- u: The position (time).
- s: The scale (inverse of frequency).
- $\frac{1}{\sqrt{s}}$: An energy normalization factor, ensuring all wavelets have the same "strength."

This is the key insight:

- **Small s (High Frequency):** The wavelet is *compressed* (shrunk). This gives a very narrow, high-frequency "ruler"—perfect for pinpointing the *exact time* of a high-frequency spike.
- **Large s (Low Frequency):** The wavelet is *stretched*. This gives a very wide, low-frequency "ruler"—perfect for analyzing the *frequency* of a long, slow oscillation.

This is called **Multi-Resolution Analysis**. Unlike the STFT's fixed window, our analysis "ruler" automatically adapts its time and frequency precision to the feature it's looking for.

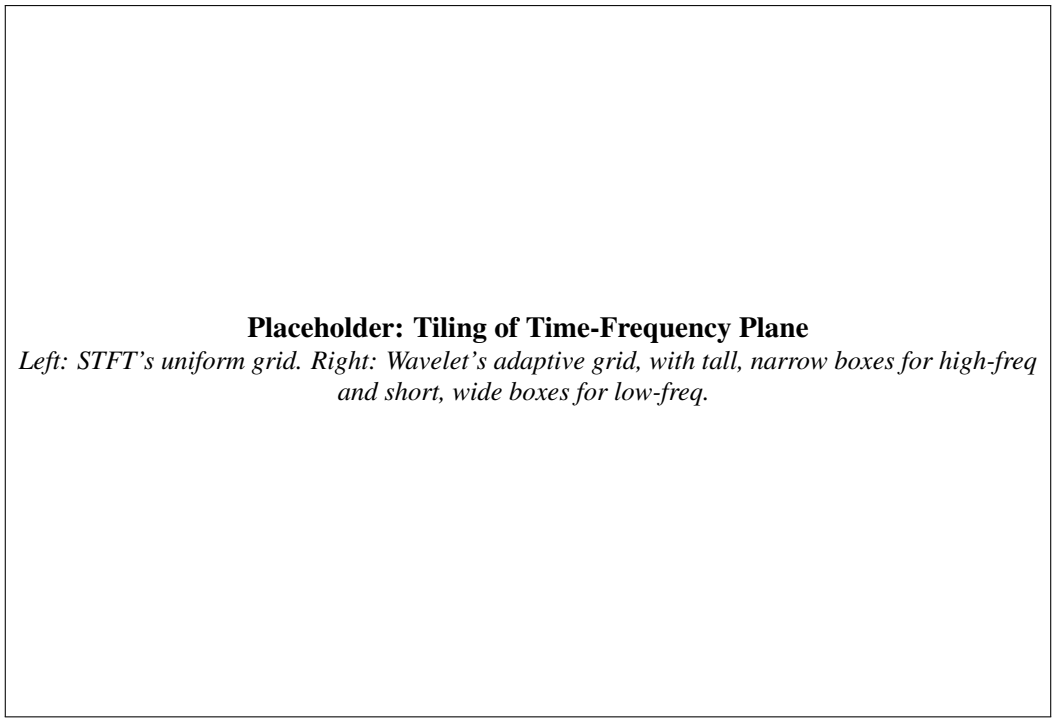

Figure 5: How STFT (left) and Wavelets (right) "see" the time-frequency plane. Wavelets provide high time resolution for high frequencies and high frequency resolution for low frequencies.

3.2 The Continuous Wavelet Transform (CWT)

The CWT is the full, mathematically-rich version of the wavelet transform. It is defined as the inner product of our signal $f(t)$ with the wavelet family $\psi_{u,s}(t)$ for *all possible* scales s and translations u.

$$W_f(u,s) = \langle f, \psi_{u,s} \rangle = \int_{-\infty}^{\infty} f(t) \frac{1}{\sqrt{s}} \psi^* \left(\frac{t-u}{s} \right) dt$$

The result, $W_f(u,s)$, is a 2D map of "similarity scores." This map is called a **scaleogram**. It shows the "energy" (the squared coefficient) of the signal at every time u and scale s.

3.3 CWT Code Example: The Chirp Signal

A "chirp" signal is the perfect test for wavelets, as its frequency content is explicitly time-dependent. We will use the ‘PyWavelets‘ library in Python.

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

# 1. Create the signal
fs = 1000 # Sample rate
t = np.linspace(0, 2, 2 * fs)
# A signal that starts at 5 Hz and sweeps up to 50 Hz
chirp_signal = np.sin(2 * np.pi * (5 * t + 11.25 * t**2))
# Add a high-frequency event at t=1.2s
t_event = np.abs(t - 1.2) < 0.05
chirp_signal[t_event] = chirp_signal[t_event] + np.sin(2 * np.pi * 100 * t[t_event])

# 2. Perform the CWT
# We choose 100 scales, from low to high freq
scales = np.arange(1, 100)
# Use the 'Morlet' wavelet
```

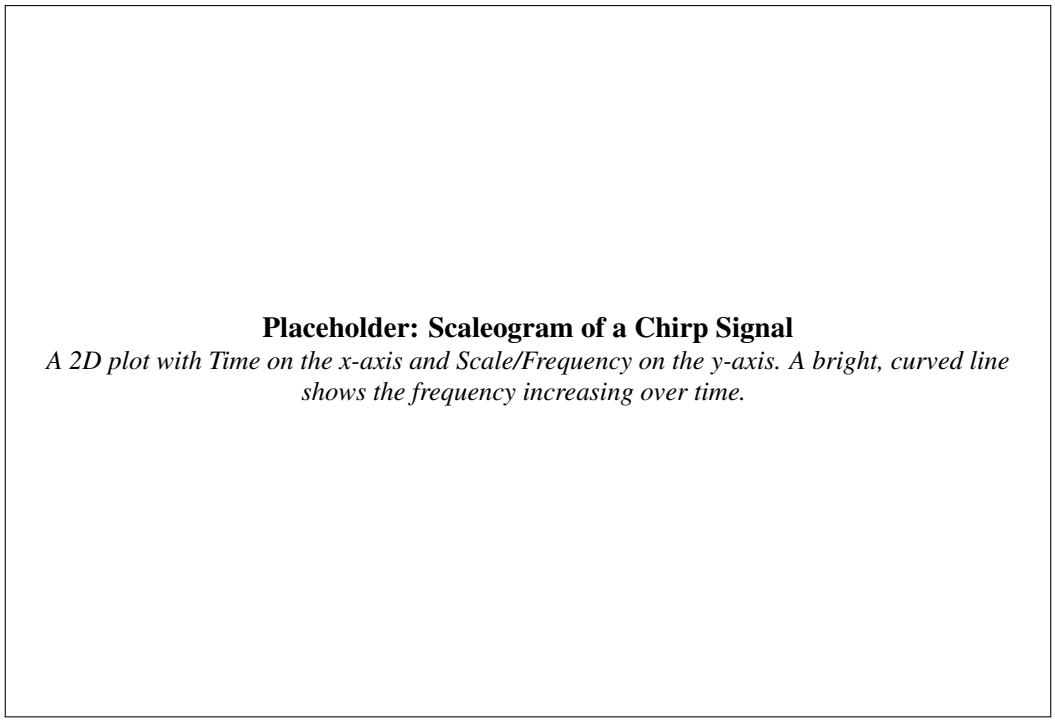

Figure 6: A scaleogram of a "chirp" signal (where frequency increases). Unlike Fourier, the CWT clearly shows the frequency changing as a function of time.

```
coef, freqs = pywt.cwt(chirp_signal, scales, 'morl', 1/fs)

# 3. Plot the results
plt.figure(figsize=(12, 8))

# Plot 1: The signal
plt.subplot(2, 1, 1)
plt.plot(t, chirp_signal)
plt.title("Original Signal (Chirp with high-freq event)")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")

# Plot 2: The Scaleogram
plt.subplot(2, 1, 2)
# We plot the log of the frequencies for better visualization
plt.pcolormesh(t, np.log2(freqs), np.abs(coef)**2, cmap='viridis')
plt.title("CWT Scaleogram (Viridis cmap)")
plt.xlabel("Time (s)")
plt.ylabel("Log(Frequency)")
plt.tight_layout()
plt.show()
```

Listing 1: Python code for CWT of a chirp signal

Interpreting the Output: The resulting plot (see Figure 6) would clearly show:

1. A bright, curved line starting at a low frequency and rising over time (the chirp).
2. A distinct, isolated "blob" of high-frequency energy at $t = 1.2$s (the event).

This is a perfect example of time-frequency localization, something the Fourier Transform (Figure 7) could never do.

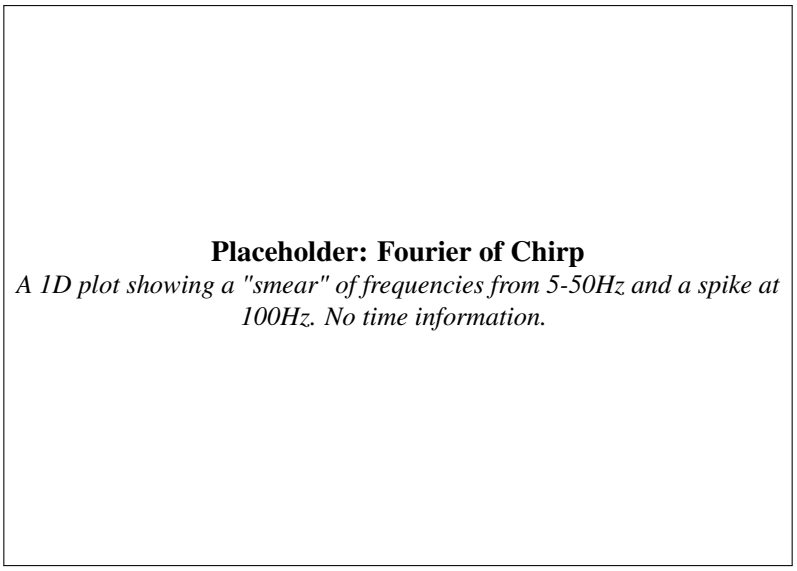

Figure 7: The Fourier transform of the signal from Listing 1. All time information is lost.

4 The Discrete Wavelet Transform (DWT)

The CWT is beautiful, but it is also highly redundant and computationally slow. For every time step, we calculate an inner product with hundreds of scales.

The **Discrete Wavelet Transform (DWT)** provides a fast, efficient, and non-redundant way to compute wavelet coefficients. It is the workhorse behind JPEG2000 image compression, denoising, and data analysis.

4.1 The Filter Bank Intuition

The DWT is best understood as a **filter bank**. At each "level" of decomposition, the DWT splits the signal into two components:

1. **A "Low-Pass" Filter (g):** This filter smooths the signal, capturing the slow, low-frequency "Approximation" components.
2. **A "High-Pass" Filter (h):** This filter sharpens the signal, capturing the fast, high-frequency "Detail" components.

These two filters are mathematically related and are called a **Quadrature Mirror Filter (QMF)** pair.

4.2 Multi-Level Decomposition

Here is the process for a Level-2 DWT:

1. **Level 1:**
   - Pass the **Original Signal** through the low-pass filter g and high-pass filter h.
   - We now have two signals. We "downsample" them (throw away every other sample), because we have twice as much data as we started with.
   - This yields: **Level 1 Approximation (cA1)** and **Level 1 Detail (cD1)**.
2. **Level 2:**
   - Take the **cA1** (the "Approximation" from Level 1) and repeat the process.
   - Pass **cA1** through the low-pass g and high-pass h filters, then downsample.

- This yields: **Level 2 Approximation (cA2)** and **Level 2 Detail (cD2)**.

The final DWT representation of the signal is the set of coefficients: {cA2, cD2, cD1}. We have "decomposed" the signal into different frequency bands at different time resolutions.

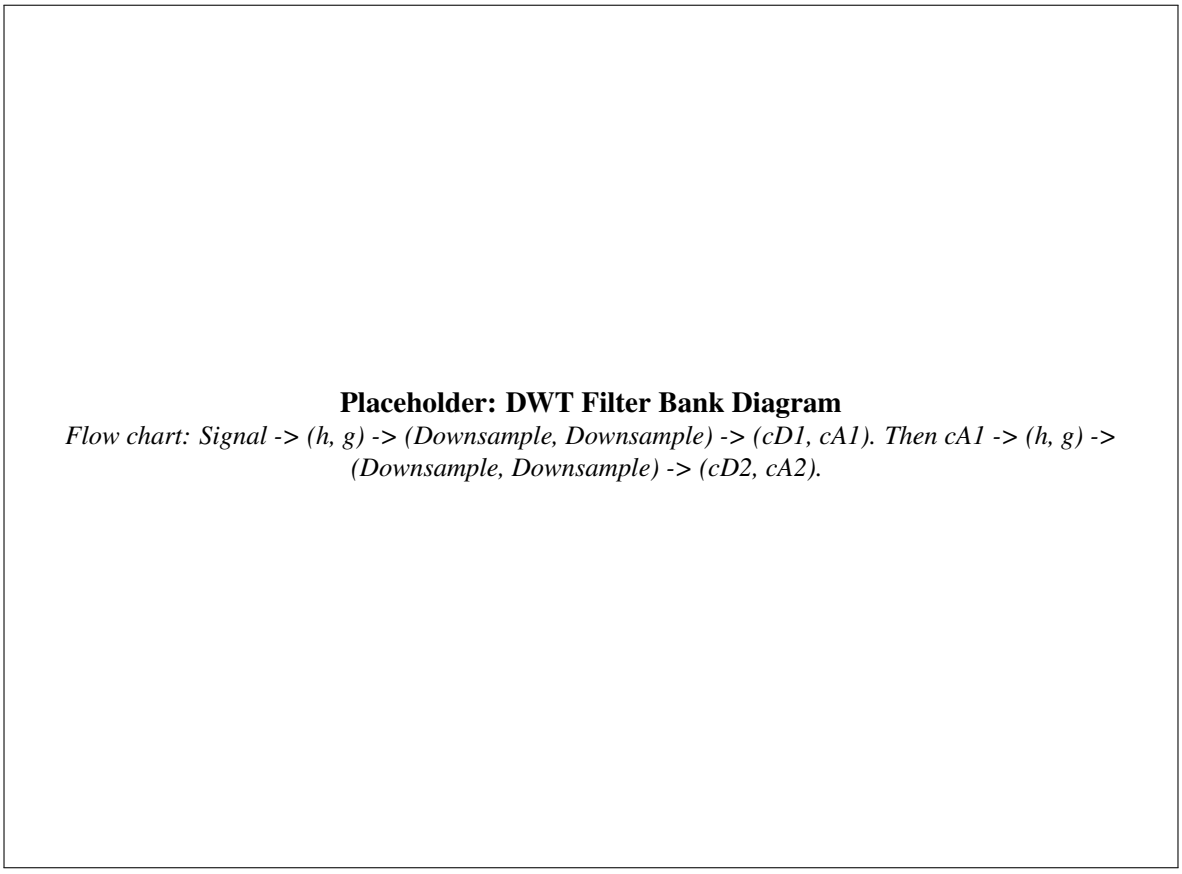

Figure 8: A 2-level DWT decomposition. The process is iterated on the "Approximation" (cA) coefficients.

4.3 DWT Code Example: Denoising

The most famous application of the DWT is denoising. The logic is:

1. Perform a DWT of a noisy signal.
2. *Intuition:* Real signal energy will be concentrated in a few large coefficients, while noise will be spread across many small coefficients.
3. We apply a "threshold"—setting all coefficients below a certain value to zero.
4. We perform an **Inverse DWT (IDWT)** to reconstruct the signal from the "cleaned" coefficients.

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

# 1. Create a noisy signal
t = np.linspace(0, 1, 1000)
clean_signal = np.sin(2 * np.pi * 5 * t) + np.sin(2 * np.pi * 12 * t)
noise = 0.5 * np.random.randn(len(t))
noisy_signal = clean_signal + noise

```

```python
# 2. Decompose the signal
wavelet = 'db4' # Daubechies 4 wavelet
level = 4
coeffs = pywt.wavedec(noisy_signal, wavelet, level=level)

# 3. Threshold the Detail coefficients
# We don't threshold the first coeff (cA4), only cD4, cD3, cD2, cD1
threshold = 0.4
for i in range(1, len(coeffs)):
    coeffs[i] = pywt.threshold(coeffs[i], threshold, mode='soft')

# 4. Reconstruct the signal
denoised_signal = pywt.waverec(coeffs, wavelet)

# 5. Plot the results
plt.figure(figsize=(10, 8))
plt.plot(t, noisy_signal, color='gray', alpha=0.5, label='Noisy')
plt.plot(t, clean_signal, color='black', linestyle='--', label='Original')
plt.plot(t, denoised_signal, color='red', label='Denoised')
plt.legend()
plt.title("DWT Denoising")
plt.show()
```

Listing 2: Python code for DWT denoising

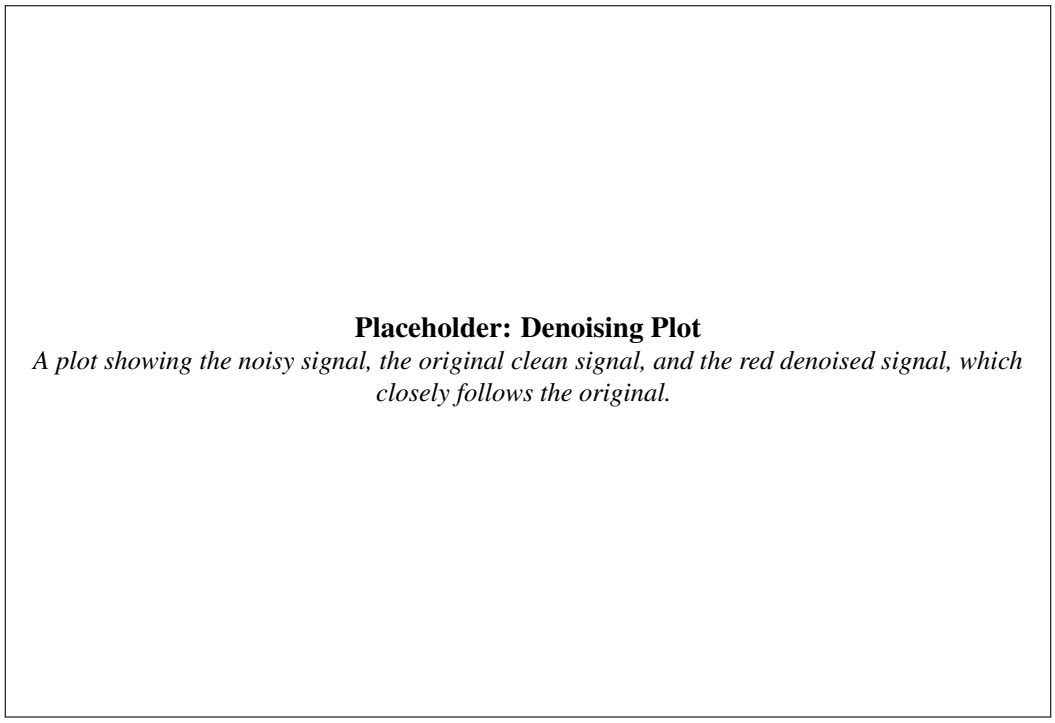

Figure 9: The output of Listing 2, showing the power of DWT thresholding to recover a clean signal from noise.

5 Conclusion: Which to Use?

- **Continuous Wavelet Transform (CWT):** Use for analysis, feature extraction, and high-resolution visualization. It is computationally expensive but provides a rich, interpretable map (the scaleogram).
- **Discrete Wavelet Transform (DWT):** Use for applications requiring speed and efficiency, such as compression (JPEG2000), denoising, and real-time data processing. It is a fast, non-redundant, and perfectly invertible transform.

Wavelet analysis is a fundamental tool that bridges the gap between pure time-domain and pure frequency-domain analysis, allowing us to finally see *what* is happening *when*.