Wavelet Transform in Practice

From Theory to Production-Ready Python **Applications**

VOLUME II-A

Applied Wavelet Analysis

Denoising, Trend Extraction, and Compression

Shouke Wei

Wavelet Transform in Practice

From Theory to Production-Ready Python Applications

Series

Wavelet Transform in Practice

From Theory to Production-Ready Python Applications

VOLUME II-A

Applied Wavelet Analysis

Denoising, Trend Extraction, and Compression

Shouke Wei

DEEPSIM PRESS

First Edition, 2026

ISBN 978-1-0699284-3-6 (eBook)
DOI 10.5281/zenodo.18791877

Published by
Deepsim Press
An independent imprint of Deepsim Intelligence Technology Inc.
Abbotsford, British Columbia, Canada
https://press.deepsim.ca

This book is intended for educational and professional readers.

For resources, updates, and companion code, visit:
https://press.deepsim.ca/wavelet-books

About the Author

Shouke Wei, Ph.D., is a researcher, scientist, and entrepreneur specializing in intelligent IoT systems, robotics, big data analytics, modeling and forecasting, early-warning systems, and edge computing. With academic and industry experience across Europe, North America, and Asia, Dr. Wei is recognized for bridging advanced theory with real-world, production-ready systems.

Dr. Wei earned his Ph.D. in Environmental and Resource Management from the Department of Environmental Informatics at Brandenburg University of Technology Cottbus–Senftenberg (Germany). He conducted postdoctoral research at the Swiss Federal Institute of Aquatic Science and Technology (Eawag), where he also served as a doctoral supervisor, and held research positions at the University of British Columbia (Canada).

Recognized as a National High-End Talent (Class A) in China, Dr. Wei has held distinguished and adjunct professorships at multiple institutions, including Yantai University, Ludong University, and Jining University. He has served as a graduate supervisor and distinguished professor in computer science, control engineering, and applied mathematics.

Dr. Wei currently serves as CEO and Chief Scientist of Deepsim Intelligent Technology Inc. (Canada), Chief Scientist at Canadian Sincerity Enterprises Inc., and Chief Scientist of Shandong Deepsim Intelligent Technology Co., Ltd. He is also a Postdoctoral Co-Supervisor at the Shandong Postdoctoral Innovation Practice Base and currently serves as Director of Qilu Artificial Intelligence and Digital Manufacturing Innovation at Shandong Deepsim Intelligent Technology Co., Ltd., China.

Dr. Wei has led or contributed to 19 major international research projects and the development of 19 intelligent systems, including autonomous water-quality monitoring vessels, AI-based environmental early-warning platforms, medical image diagnosis models, precision agriculture robots, and autonomous service robots.

His scholarly contributions include 40+ peer-reviewed publications and 500+ tutorial online articles, 6 patents, 30 software copyrights, and 2 China national scientific and technological achievements. Dr. Wei's work focuses on making advanced computational methods—particularly wavelet-based signal processing—accessible, practical, and impactful for researchers and practitioners worldwide.

For more info, visit: `https://shouke.deepsim.ca`

Contents

Preface

This volume continues the progression established in *Volume I* by focusing on the **applied use of wavelet transforms** for practical signal and image processing tasks. While the mathematical foundations and core discrete wavelet framework have already been introduced, the emphasis here is on **how wavelets are used in practice**—to suppress noise, compress data, extract trends, and evaluate performance within real analytical workflows.

Wavelet-based denoising and compression are among the most widely deployed applications of multiscale analysis. They appear in domains ranging from speech and audio processing to sensing, imaging, and time-series analysis. The Volume II-A, *Applied Wavelet Analysis: Denoising, Trend Extraction, and Compression* presents these techniques systematically, with attention to **method selection, parameter tuning, and quantitative evaluation**, rather than purely theoretical derivation.

Throughout the book, one-dimensional signals (including time series, sensor measurements, and speech) and two-dimensional data (such as images and spatial fields) are treated within a unified multiresolution framework. Comparisons with classical and industry-standard approaches are included where appropriate, highlighting both the strengths and limitations of wavelet-based methods.

Rather than presenting wavelets as abstract mathematical constructs, this volume emphasizes their role as **tools within reproducible analytical pipelines**. All examples are implemented in Python and are designed to support experimentation, validation, and adaptation to real-world applications.

Who This Book Is For

This volume is intended for:

- Engineers and practitioners applying wavelets to denoising, trend extraction, anomaly detection, and compression problems
- Data scientists working with time-series, sensor, speech, or image data
- Researchers seeking practical wavelet-based workflows
- Graduate students transitioning from foundational theory to applied analysis
- Software engineers integrating wavelet methods into data processing pipelines

Readers are expected to have prior exposure to wavelet concepts and Python programming, such as those introduced in *Volume I*.

Organization of This Volume

The *Volume II-A* is organized into **two parts**, reflecting the primary data domains encountered in applied wavelet analysis.

Part I: Wavelet Processing for 1D Signals and Time Series
This part focuses on wavelet-based denoising, compression, trend extraction, and anomaly detection for one-dimensional signals. Emphasis is placed on practical implementation, comparison with classical methods, and quantitative evaluation.

Part II: Wavelet Processing for Images and Spatial Data
This part extends wavelet methods to two-dimensional data, addressing image denoising and compression with attention to spatial structure, perceptual quality, and reconstruction fidelity.

Prerequisites and Computational Environment

This book assumes:

- Proficiency in Python programming
- Familiarity with NumPy, SciPy, and Matplotlib
- Prior exposure to discrete wavelet transforms

All examples rely on the Python scientific computing ecosystem, with **PyWavelets** serving as the primary wavelet analysis library. A condensed setup guide is provided in *Chapter 0*.

Shouke Wei, PhD
Deepsim Intelligence Technology Inc.
Deepsim Academy
Abbotsford, Canada
February 27, 2026

Acknowledgments

The preparation of this book benefited greatly from discussions and collaborations across academic and applied settings. I am grateful to colleagues and collaborators who shared insights into practical signal and image processing challenges, particularly in the context of denoising, compression, and multiscale interpretation.

I would like to thank my students and research collaborators for their curiosity, critical questions, and persistence in bridging theory with practice. Their engagement continually motivated clearer explanations and more robust implementations.

I am deeply grateful to my family for their patience, encouragement, and unwavering support throughout the writing process.

This volume builds upon decades of foundational research in wavelet theory by pioneers such as Ingrid Daubechies and Stéphane Mallat. I also acknowledge the open-source scientific computing community—particularly the contributors to **PyWavelets**, **NumPy**, **SciPy**, **Matplotlib**, **scikit-image**, **scikit-learn**, **yfinance**, **opencv-python**, **requests**, **gdown**, and related libraries—whose tools make reproducible and reliable wavelet-based analysis possible.

List of Symbols

The following symbols are used throughout this volume.

Symbol	Meaning	Description
Wavelet Functions		
$\psi(t)$	Mother wavelet	Oscillatory, zero-mean function for analyzing localized high-frequency components
$\phi(t)$	Scaling function	Function representing low-frequency approximation components
Parameters		
j	Scale level	Discrete scale index; higher j indicates coarser resolution
k	Translation index	Discrete position index
t	Time variable	Continuous time parameter
n	Discrete index	Integer index for discrete signals
Signals		
$x(t)$	Continuous signal	Original continuous signal being analyzed
$x[n]$	Discrete signal	Discrete input signal at sample index n
x_{trend}	Trend signal	Signal reconstructed using approximation coefficients
x_{denoised}	Denoised signal	Signal reconstructed after wavelet coefficient thresholding

Symbol	Meaning	Description
r	Residual / noise	Difference between original and denoised signal
Coefficients		
A_j or cA_j	Approximation coefficients	Low-frequency coefficients at level j
D_j or cD_j	Detail coefficients	High-frequency coefficients at level j
A_J	Coarsest approximation	Approximation at maximum decomposition level J
$D_J, \dots, D_1$	Detail levels	Detail coefficients from coarsest to finest scales
Decomposition		
J	Maximum level	Deepest level of wavelet decomposition
L	Decomposition level	Current level of wavelet decomposition
Thresholding		
λ	Threshold value	Coefficient threshold for denoising
$T_{\text{soft}}(x, \lambda)$	Soft thresholding	Shrinks coefficients toward zero
$T_{\text{hard}}(x, \lambda)$	Hard thresholding	Sets coefficients below λ to zero
σ	Noise level	Estimated noise standard deviation
N	Signal length	Number of samples or pixels
Threshold Estimation		
Universal	Universal threshold	$\lambda = \sigma\sqrt{2 \ln N}$
MAD	Median absolute deviation	Robust estimator of σ
Transform Notation		
DWT	Discrete Wavelet Transform	Forward wavelet decomposition
IDWT	Inverse DWT	Reconstruction from wavelet coefficients

Symbol	Meaning	Description
DWT2	2D Discrete Wavelet Transform	Two-dimensional wavelet decomposition
IDWT2	Inverse 2D DWT	Image reconstruction from 2D coefficients
Statistical Terms		
μ	Mean	Average value
σ^2	Variance	Measure of spread
Image Properties		
D_r	Dynamic range	Intensity span of an image
Compression		
σ^2_{signal}	Signal variance	Variance of the original signal
σ^2_{noise}	Noise variance	Variance of reconstruction error
JPEG2000	JPEG 2000	Wavelet-based image compression standard
Wavelet Packet Decomposition		
$A_{j+1,2k}$	WPD approximation node	Approximation at level $j+1$, node $2k$
$D_{j+1,2k+1}$	WPD detail node	Detail at level $j+1$, node $2k+1$
W_k	Node coefficients	Wavelet packet coefficients at node k
$H(W_k)$	Entropy	Entropy of coefficients at node k
Evaluation Metrics		
MSE	Mean squared error	Average squared reconstruction error
PSNR	Peak signal-to-noise ratio	Image reconstruction quality metric

SSIM	Structural similarity index	Perceptual image similarity metric

Prerequisites and Working Environment

This volume assumes familiarity with the Python environment and reproducibility principles introduced in *Volume I*. Rather than repeating the full setup procedure, this chapter summarizes the additional requirements needed to execute the advanced wavelet methods, case studies, and examples presented in *Volume II-A*.

Readers who require a complete introduction to Python setup, virtual environments, or basic PyWavelets usage should consult **Volume I, Chapter 0**.

Software Requirements

All examples in this volume are implemented in Python and rely on a standard scientific computing stack, supplemented by libraries required for advanced wavelet transforms, signal processing, and applied data analysis.

The following packages are required to run the code examples in this volume:

- **NumPy** — numerical computation and array operations
- **SciPy** — signal processing utilities and numerical methods
- **Pandas** — data manipulation and time-series handling
- **Matplotlib** — visualization and plotting
- **PyWavelets** — discrete wavelet transforms and multiresolution analysis

- **scikit-image** — image and multidimensional signal processing
- **scikit-learn** — evaluation metrics and feature-based modeling
- **yfinance** — financial time-series data access
- **opencv-python** — image input/output and preprocessing
- **requests, gdown** — dataset retrieval utilities

Installation

It is recommended to use a virtual environment to isolate dependencies:

```
python -m venv venv
source venv/bin/activate   # Linux/macOS
# or
venv\Scripts\activate    # Windows
```

All required dependencies can then be installed using the provided `requirements.txt` file:

```
pip install -r requirements.txt
```

This ensures version consistency across all chapters in this volume.

Data Access and Reproducibility

Where possible, examples in this volume use publicly available datasets (e.g., financial time series, sensor data). Data acquisition steps are documented within each chapter. When external data sources are required, utilities such as `yfinance` or `gdown` are used to automate retrieval.

All code examples are designed to be reproducible and self-contained. Random seeds, preprocessing steps, and evaluation metrics are explicitly documented to support repeatable analysis.

Scope of This Setup

This setup chapter is intentionally concise. Its purpose is to:

- Ensure readers can execute the advanced wavelet methods presented in this volume
- Highlight additional dependencies beyond those introduced in Volume I
- Maintain consistency across applied examples and case studies

For a complete discussion of environment configuration, project structure, and foundational tools, readers are encouraged to refer back to **Volume I, Chapter 0**.

Part I

Wavelet Processing for 1D Signals and Time Series

1 Wavelet-Based Denoising of Time-Series Signals

In real-world applications, time series signals are often corrupted by noise due to measurement errors, environmental factors, or data transmission imperfections. This is particularly common in sensor data from accelerometers, gyroscopes, and other IoT devices where the hardware itself introduces measurement noise, and environmental conditions further degrade signal quality.

Wavelet-based denoising provides a powerful framework to reduce this noise while preserving the underlying structure and important features of the signal. Unlike traditional filtering methods that operate in the frequency domain, wavelet denoising can adapt to local signal characteristics, making it particularly effective for non-stationary signals.

Numerous studies have demonstrated the effectiveness of wavelet-based denoising techniques across various real-world signal processing applications. For instance, Donoho and Johnstone (1994) pioneered the use of wavelet shrinkage for noise reduction, introducing thresholding methods that adaptively remove noise while retaining signal features. Subsequent research has extended these methods to handle non-stationary and multivariate time series data common in IoT sensor streams, highlighting wavelet denoising's ability to preserve transient characteristics better than classical Fourier-based filters (e.g., Nason (2008)). These findings support the growing adoption of wavelet denoising in embedded sensing systems where real-time, robust noise suppression is critical.

1.1 Overview

This chapter examines wavelet-based denoising methods from both theoretical and practical perspectives. Using real-world accelerometer data, it explores coefficient-based and thresholding approaches, evaluates their performance, and discusses best practices for method selection and parameter tuning in applied signal-processing tasks. It covers:

- Theoretical background on wavelet denoising
- Practical implementation using real-world accelerometer data
- Multiple denoising approaches: Approximation-based methods and thresholding
- Comparative analysis of different methods
- Performance evaluation and best practices

1.2 The Concept of Wavelet Denoising

Wavelet denoising leverages the multiresolution propertiesof wavelets to separate signal from noise. The fundamental assumption is that signal features are typically concentrated in a few large wavelet coefficients, while noise is distributed across many small coefficients.

1.2.1 Classical Denoising Pipeline

The traditional wavelet denoising process consists of three steps:

1. **Wavelet Decomposition**: Transform the noisy signal into the wavelet domain using a discrete wavelet transform (DWT)
2. **Coefficient Modification**: Apply thresholding or removal techniques to suppress noise-dominated coefficients
3. **Signal Reconstruction**: Transform back to the time domain using the inverse DWT

1.2.2 Thresholding Methods

Wavelet thresholding is a key step in denoising, where high-frequency coefficients are shrunk or removed to reduce noise while preserving important signal features.

1. Soft Thresholding Soft thresholding applies a continuous shrinkage to wavelet coefficients, attenuating small coefficients while avoiding the discontinuities introduced by hard thresholding.

$$T_{\text{soft}}(x, \lambda) = \text{sign}(x) \cdot \max(|x| - \lambda, 0) \tag{1.1}$$

- Coefficients smaller than the threshold λ are set to zero.
- Larger coefficients are reduced by λ, avoiding abrupt changes and reducing artifacts.

2. Hard Thresholding

Hard thresholding removes coefficients below the threshold and keeps the rest unchanged:

$$T_{\text{hard}}(x, \lambda) = \begin{cases} x, & |x| > \lambda \\ 0, & |x| \leq \lambda \end{cases} \tag{1.2}$$

- Preserves the magnitude of large coefficients exactly.
- Can introduce discontinuities in the reconstructed signal.

3. Universal Threshold A common choice for λ is the **universal threshold**, which depends on the estimated noise level σ and signal length N:

$$\lambda = \hat{\sigma}\sqrt{2\ln(N)} \tag{1.3}$$

where $\hat{\sigma}$ is the noise standard deviation estimated from the finest-scale detail coefficients, and N denotes the number of samples in the original signal.

$$\widehat{\sigma} = \frac{\text{median}\left(|d_{\text{last}} - \text{median}(d_{\text{last}})|\right)}{0.6745} \tag{1.4}$$

where d_{last} represents the finest-scale wavelet detail coefficients. For zero-mean noise, the median of d_{last} is approximately zero, and the simplified MAD expression is commonly used in practice.

4. Alternative Thresholding Methods

1. **SURE Threshold (Stein's Unbiased Risk Estimate)**
   Minimizes an estimate of the mean squared error between the denoised and original signal. Adaptive to the data and often performs better for small signals.

2. **Bayesian Threshold**
   Computes λ using a Bayesian approach, often scaling with both $ln(N)$ and $log_2(N)$. This provides stronger shrinkage for higher-frequency coefficients.

3. **BayesShrink Threshold** A specific Bayesian-inspired wavelet thresholding method that derives a closed-form, scale-dependent soft threshold under a Gaussian prior, whereas Bayesian thresholding refers more broadly to wavelet denoising approaches based on probabilistic modeling and posterior inference.

4. **Minimax Threshold**
   Designed to give near-minimax optimal performance against the worst-case signal. Suitable for signals with unknown smoothness.

5. **Approximation-Based Denoising**
   In some applications, it is useful to ignore all detail coefficients and reconstruct the signal using only the last-level approximation. This produces a very smooth **trend signal**, removing all high-frequency fluctuations:

$$x_{\text{trend}} = \text{IDWT}([A_L, 0, 0, \ldots, 0]) \tag{1.5}$$

- A_L is the approximation at the highest decomposition level.

- Often used to extract long-term trends in financial or environmental time series.
- Simpler than thresholding but less selective
- Effective when noise is concentrated in specific frequency bands

These thresholding methods form the foundation of **wavelet denoising**, allowing flexible noise reduction while preserving key signal characteristics.

1.3 Real-World Data Analysis: Accelerometer Signals

We'll demonstrate these concepts using accelerometer data from the Human Activity Recognition dataset, which contains smartphone sensor data collected during various activities. This data exhibits realistic noise characteristics common in IoT applications.

1.3.1 Data Loading and Exploration

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pywt  # PyWavelets library
from sklearn.metrics import mean_squared_error
from scipy.signal import savgol_filter  # Savitzky-Golay filter
import warnings
import gdown

warnings.filterwarnings('ignore')

# -------------------------
# Configurations
# -------------------------
DATA_URL = (
    "https://drive.google.com/uc?export=download"
    "&id=1GtNRi9CgHOlGpsa-Dlp6uzAqXUVdEh54"
)
SAMPLING_INTERVAL = 0.2  # seconds (200 ms)
```

```
FALLBACK_SAMPLES = 1500
RANDOM_SEED = 42

# -------------------------
# Data Loading
# -------------------------
def load_accelerometer_data():
    """
    Loads running activity accelerometer data.
    Attempts to fetch from remote dataset; falls back to synthetic
↪   generation.

    Returns:
        pd.DataFrame: columns ['timestamp', 'accel_x', 'accel_clean']
    """

    try:
        gdown.download(DATA_URL,
↪   "./data/posture_recognition_data.csv", quiet=False)
        df_full = pd.read_csv("./data/posture_recognition_data.csv")
        running_data = df_full[df_full['label'] == 3].copy()  #
↪   running label = 3

        if len(running_data) > 2000:
            running_data = running_data.head(2000)

        timestamps = np.arange(len(running_data)) * SAMPLING_INTERVAL
        accel_raw = running_data['Ax1'].values

        # Moving average smoothing
        accel_clean = (
            pd.Series(accel_raw)
            .rolling(window=5, center=True)  # Rolling window
            .mean()
            .bfill()
            .ffill()
            .values
        )

        return pd.DataFrame({
```

```
            'timestamp': timestamps,
            'accel_x': accel_raw,
            'accel_clean': accel_clean
        })

    except Exception as e:
        print(f"Could not load dataset: {e}")
        print("Generating synthetic running-like accelerometer
        ↪  data...")

        np.random.seed(RANDOM_SEED)
        t = np.arange(FALLBACK_SAMPLES) * SAMPLING_INTERVAL

        # Simulate base running pattern
        base_freq = 2.5  # Hz
        signal = (
            4.0 * np.sin(2 * np.pi * base_freq * t) +
            0.6 * np.sin(2 * np.pi * 2 * base_freq * t) +
            0.3 * np.sin(2 * np.pi * 4 * base_freq * t)
        )

        # Add noise components
        measurement_noise = 0.05 * np.random.randn(FALLBACK_SAMPLES)
        vibration_noise = 0.03 * np.random.randn(FALLBACK_SAMPLES)
        high_freq_signal = 0.02 * np.sin(2 * np.pi * 20 * t)
        high_freq = high_freq_signal *
↪   np.random.randn(FALLBACK_SAMPLES)

        spikes = np.zeros(FALLBACK_SAMPLES)
        spike_indices = np.random.choice(
            FALLBACK_SAMPLES,
            size=FALLBACK_SAMPLES // 50,
            replace=False
        )
        spikes[spike_indices] = np.random.randn(len(spike_indices)) *
↪   0.2

        noisy_signal = (
            signal
            + measurement_noise
```

```
            + vibration_noise
            + high_freq
            + spikes
        )

        # "Clean" version using Savitzky-Golay smoothing
        clean_signal = savgol_filter(noisy_signal,
            window_length=11, polyorder=3)

        return pd.DataFrame({
            'timestamp': t,
            'accel_x': noisy_signal,
            'accel_clean': clean_signal
        })

# -------------------------
# Visualization
# -------------------------
def plot_accelerometer_data(df):
    plt.figure(figsize=(15, 6))

    plt.subplot(2, 1, 1)
    plt.plot(df['timestamp'], df['accel_x'], 'b-', alpha=0.7,
        lw=1, label='Noisy Signal')
    plt.plot(df['timestamp'], df['accel_clean'], 'r-', lw=2,
        label='Clean Signal')
    plt.title('Raw Accelerometer Data (X-axis)')
    plt.ylabel('Acceleration (m/s²)')
    plt.legend()
    plt.grid(True, alpha=0.3)

    plt.subplot(2, 1, 2)
    noise = df['accel_x'] - df['accel_clean']
    plt.plot(df['timestamp'], noise, 'g-', alpha=0.8)
    plt.title('Noise Component')
    plt.xlabel('Time (seconds)')
    plt.ylabel('Noise Level')
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
```

```
    plt.savefig('./output/accelerometer_data_plot.png', dpi=300)
    plt.show()

# -------------------------
# Statistics
# -------------------------
def compute_signal_statistics(df):
    """Prints SNR and noise variance."""
    noise = df['accel_x'] - df['accel_clean']
    snr = 10 * np.log10(np.var(df['accel_clean']) / np.var(noise))
    print(f"SNR: {snr:.2f} dB")
    print(f"Noise variance: {np.var(noise):.4f}")

# -------------------------
# Main Execution
# -------------------------
if __name__ == "__main__":
    print("Loading Human Posture Recognition Dataset...")
    print("Sampling rate: {:.1f} Hz".format(1 / SAMPLING_INTERVAL))
    print("Sensor: Invensense MPU-6050 MEMS IMU\n")

    df = load_accelerometer_data()
    t_min = df['timestamp'].min()
    t_max = df['timestamp'].max()

    print(f"Dataset shape: {df.shape}")
    print(f"Time range: {t_min:.2f} to {t_max:.2f} seconds")
    print(f"Duration: {t_max - t_min:.1f} seconds\n")

    plot_accelerometer_data(df)
    compute_signal_statistics(df)
```

Output:

```
Loading Human Posture Recognition Dataset...
Sampling rate: 5.0 Hz
Sensor: Invensense MPU-6050 MEMS IMU

Downloading...
```

```
From: https://drive.google.com/uc?id=1GtNRi9CgHOlGpsa-Dlp6uzAqXUVdEh54
To: d:\project-workspace\jupyter-books\wavelet-book\script\data\
 posture_recognition_data.csv
100%|      | 3.20M/3.20M [00:00<00:00, 20.2MB/s]
Dataset shape: (2000, 3)
Time range: 0.00 to 399.80 seconds
Duration: 399.8 seconds
```

The plot looks as follows:

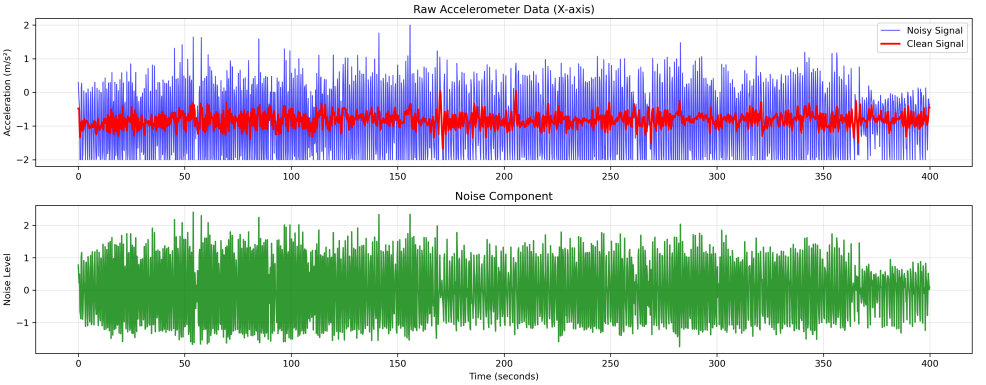

Figure 1.1: Denoising a real-world accelerometer data from Human Posture Recognition dataset, showing the noisy signal (Top) and the smoothed signal (Bottom) using a Savitzky-Golay filter. Bottom plot displays the noise component, calculated as the difference between the noisy and clean signals. The data spans 399.8 seconds with a sampling rate of 5.0 Hz.

If the dataset at the specified Google Drive URL cannot be downloaded, the code generates a synthetic accelerometer dataset, resulting in different outputs and plots, as shown in Figure 1.1.

1.3.2 Wavelet Analysis of the Signal

```
# -------------------------
# Wavelet Decomposition
# -------------------------
```

```
def analyze_wavelet_decomposition(signal, wavelet='db6', levels=5):
    """
    Analyze wavelet decomposition of a signal, computing energy ratios
 ↪  and
    visualizing coefficients.

    Args:
        signal (np.ndarray): Input signal to decompose
        wavelet (str): Wavelet type (e.g., 'db6')
        levels (int): Number of decomposition levels

    Returns:
        tuple: (coefficients, energy_ratios)
    """
    # Validate input signal
    if not isinstance(signal, np.ndarray) or len(signal) < 2:
        raise ValueError("Input signal must be a non-empty numpy
         ↪  array")

    # Calculate maximum decomposition levels
    max_levels = pywt.dwt_max_level(len(signal), wavelet)
    levels = min(levels, max_levels)
    if levels < 1:
        raise ValueError(
            f"Signal too short for decomposition with {wavelet}. "
            f" Max levels: {max_levels}"
        )

    # Perform wavelet decomposition
    coeffs = pywt.wavedec(signal, wavelet, level=levels)

    # Calculate energy at each level
    energies = [np.sum(c**2) for c in coeffs]
    total_energy = sum(energies)

    # Avoid division by zero
    if total_energy == 0:
        raise ValueError("Total energy is zero, cannot compute energy
         ↪  ratios")
```

```
    energy_ratios = [e / total_energy for e in energies]

    # Print analysis
    print(f"\nWavelet Decomposition Analysis ({wavelet}, {levels}
    ↪  levels, "
          f"signal length: {len(signal)}):")
    print(f"{'Level':<10} {'Coefficients':<15} {'Energy %':<12} {'Max
    ↪  Coeff':<12}")
    print("-" * 55)

    for i, (coeff, energy_pct) in enumerate(zip(coeffs,
    ↪  energy_ratios)):
        level_name = "Approx" if i == 0 else f"Detail {i}"
        print(f"{level_name:<10} {len(coeff):<15}
        ↪  {energy_pct*100:<12.2f} "
              f"{np.max(np.abs(coeff)):<12.4f}")

    # Visualize decomposition
    fig, axes = plt.subplots(levels + 2, 1, figsize=(12, 2 * (levels +
↪   2)))

    # Ensure axes is iterable for low levels
    if not isinstance(axes, np.ndarray):
        axes = np.array([axes])

    # Plot original signal
    time = np.arange(len(signal)) * SAMPLING_INTERVAL
    axes[0].plot(time, signal, 'b-', alpha=0.7)
    axes[0].set_title("Original Signal")
    axes[0].set_ylabel("Amplitude")
    axes[0].grid(True, alpha=0.3)

    # Plot approximation coefficients
    coeff_time = np.linspace(0, time[-1], len(coeffs[0]))
    axes[1].plot(coeff_time, coeffs[0], 'g-', alpha=0.7)
    axes[1].set_title("Approximation Coefficients")
    axes[1].set_ylabel("Amplitude")
    axes[1].grid(True, alpha=0.3)

    # Plot detail coefficients
```

```
    for i, coeff in enumerate(coeffs[1:], 1):
        level_name = f"Detail Level {i}"
        coeff_time = np.linspace(0, time[-1], len(coeff))
        axes[i + 1].plot(coeff_time, coeff, 'r-', alpha=0.7)
        axes[i + 1].set_title(f"{level_name} Coefficients")
        axes[i + 1].set_ylabel("Amplitude")
        axes[i + 1].grid(True, alpha=0.3)

    axes[-1].set_xlabel("Time (seconds)")
    plt.tight_layout()
    plt.savefig(
        "./output/posture_recognition_data_wavelet_"
        "decomposition.png",
        dpi=300
    )
    plt.close()

    return coeffs, energy_ratios

# -------------------------
# Main Execution
# -------------------------
if __name__ == "__main__":
    df = load_accelerometer_data()
    plot_accelerometer_data(df)

    # Analyze wavelet decomposition
    signal = df['accel_x'].values
    coeffs, energy_ratios = analyze_wavelet_decomposition(
        signal, wavelet='db6', levels=5)
```

Output:

```
Wavelet Decomposition Analysis (db6, 5 levels, signal length: 2000):
Level      Coefficients    Energy %     Max Coeff
-----------------------------------------------------------
Approx     73              45.27        5.4691
Detail 1   73              0.27         1.0040
Detail 2   135             0.51         1.7472
```

```
Detail 3   259             2.14         2.6519
Detail 4   508             38.13        3.1377
Detail 5   1005            13.68        1.9619
```

The output shows the distribution of coefficients, their energy contribution, and the maximum coefficient value at each decomposition level.

1. **Energy distribution**

   - Most of the signal's energy is concentrated in the **approximation level (A5)**, which accounts for **45.27%** of the total energy.

   - The next largest contribution is from **Detail 4** (**38.13%**), followed by **Detail 5** (**13.68%**).

   - High-frequency details (**Detail 1**, **Detail 2**, **Detail 3**) together contribute only about **2.9%** of the total energy, indicating that the signal is dominated by low- to mid-frequency variations rather than rapid fluctuations.

2. **Coefficient counts**

   - The number of coefficients increases at finer detail levels, with **Detail 5** having the most (**1005**) due to minimal downsampling at that stage.

   - The approximation level has the fewest coefficients (**73**) since it represents the most compressed version of the signal.

3. **Maximum coefficient magnitude**

   - The **largest coefficient** is found in the **approximation level** (**5.4691**), suggesting that the low-frequency trend contains the most prominent single feature.

   - Detail levels generally have smaller maximum coefficients, with the highest among them in **Detail 4** (**3.1377**), indicating substantial low-mid frequency oscillatory content.

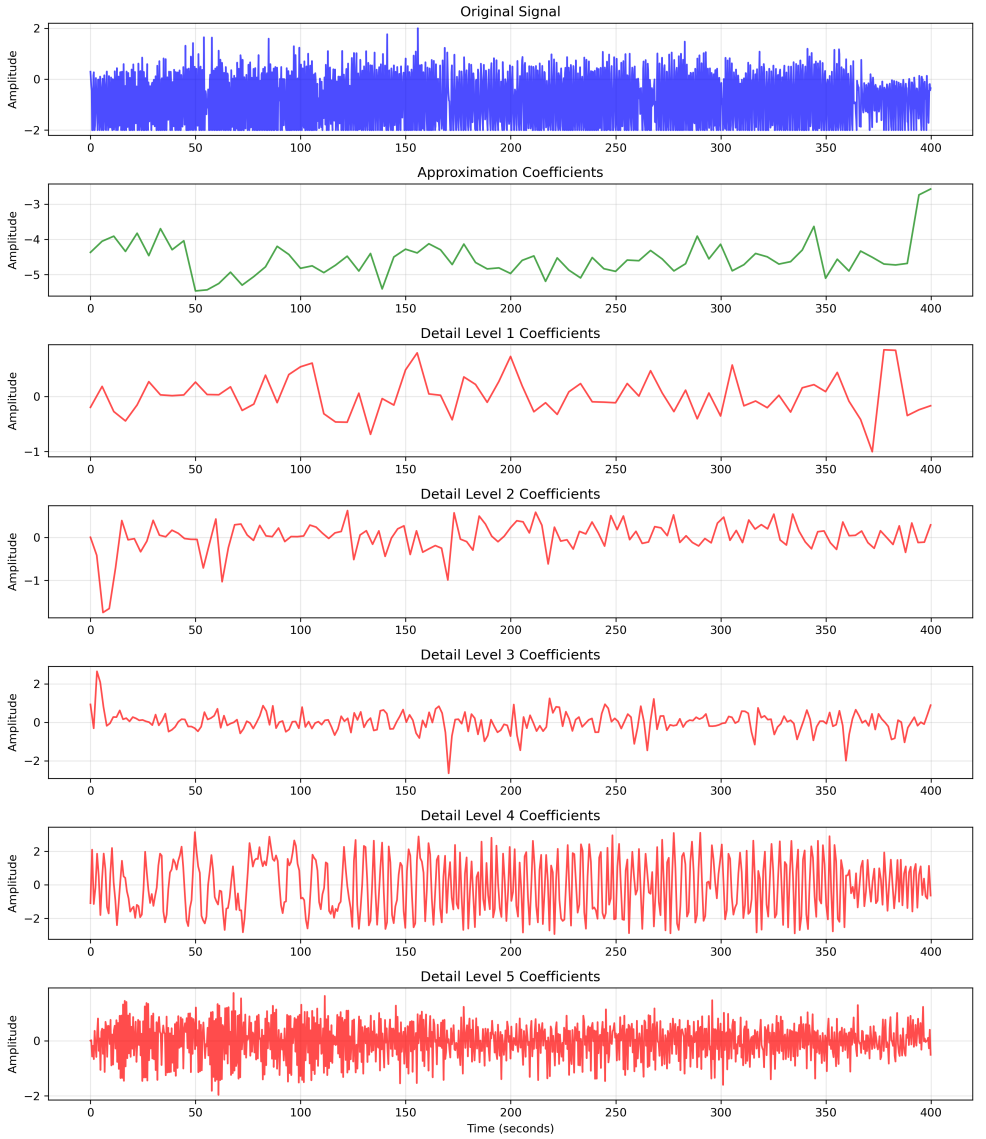

Figure 1.2: Wavelet decomposition of accelerometer posture data using a Daubechies6 (db6) wavelet with five decomposition levels. The figure shows the original signal, approximation coefficients, and detail coefficients at successive levels, along with their time-domain representations.

4. Overall interpretation

- This decomposition indicates a signal rich in **low-frequency trend** and **lower-mid frequency oscillations**, with very limited high-frequency content.

- Such a profile is typical for **periodic or quasi-periodic physical processes** (e.g., gait cycles in accelerometer data, slow oscillations in environmental sensors), where broad-scale variations carry most of the variance and noise is minimal.

1.4 Denoising Method 1: Approximation-Based Denoising

Approximation-based denoising is a wavelet-domain filtering approach that suppresses high-frequency detail components without explicit coefficient thresholding. It can be interpreted as an extreme case of thresholding, in which all detail coefficients are implicitly set to zero.

This method completely removes detail coefficients at specified decomposition levels, effectively applying a low-pass filter with wavelet-based frequency selectivity. Numerous studies have demonstrated that this approach is effective in reducing noise while preserving essential signal characteristics (e.g., Wei, Song, and Khan (2011); Zhao et al. (2020); Yuan et al. (2022)).

```
def denoise_by_coefficient_removal(signal, wavelet='db6',
 ↪  levels_to_remove=[]):
    """
    Denoise by removing specific detail coefficient levels

    Parameters:
    - signal: input noisy signal
    - wavelet: wavelet type
    - levels_to_remove: list of detail levels to remove (e.g., [1, 2])
    """
    # Decompose
```

```
    min_level = 5
    extra_level = max(levels_to_remove) if levels_to_remove else 3
    decomp_level = max(min_level, extra_level)

    coeffs = pywt.wavedec(signal, wavelet, level=decomp_level)

    # Remove specified detail levels
    coeffs_modified = coeffs.copy()
    for level in levels_to_remove:
        if level < len(coeffs):
            coeffs_modified[level] = np.zeros_like(coeffs[level])

    # Reconstruct
    denoised = pywt.waverec(coeffs_modified, wavelet)

    # Trim to original length if necessary
    if len(denoised) > len(signal):
        denoised = denoised[:len(signal)]

    return denoised

# Test different removal strategies
removal_strategies = {
    'Remove D1': [1],
    'Remove D1,D2': [1, 2],
    'Remove D1,D2,D3': [1, 2, 3],
    'Remove D2,D3': [2, 3]
}

fig, axes = plt.subplots(len(removal_strategies) + 1, 1,
            figsize=(15, 3*(len(removal_strategies)+1)))

# Original signal
axes[0].plot(df['timestamp'], df['accel_x'], 'b-', alpha=0.7,
 ↪  label='Noisy')
axes[0].plot(df['timestamp'], df['accel_clean'], 'r-', linewidth=2,
 ↪  label='Clean')
axes[0].set_title('Original Signal')
axes[0].legend()
axes[0].grid(True, alpha=0.3)
```

```
# Apply different removal strategies
results_removal = {}
for idx, (strategy_name, levels) in
 ↪  enumerate(removal_strategies.items(), 1):
    denoised = denoise_by_coefficient_removal(signal,
 ↪  levels_to_remove=levels)
    results_removal[strategy_name] = denoised

    mse = mean_squared_error(df['accel_clean'],
 ↪  denoised[:len(df['accel_clean'])])

    axes[idx].plot(df['timestamp'], df['accel_x'], 'b-', alpha=0.3,
 ↪  label='Noisy')
    axes[idx].plot(df['timestamp'], denoised[:len(df)], 'g-',
 ↪  linewidth=2,
                label=f'Denoised (MSE: {mse:.4f})')
    axes[idx].plot(df['timestamp'], df['accel_clean'], 'r--',
 ↪  linewidth=1,
                label='Clean')
    axes[idx].set_title(f'Denoising: {strategy_name}')
    axes[idx].legend()
    axes[idx].grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig("./output/denoising_by_coefficient_removal.png", dpi=300)
plt.show()
```

The visulized results are displayed in Figure 1.3.

Removing the highest-frequency detail coefficients (D1) significantly reduces noise while preserving the main signal features. Extending removal to D2 and D3 further smooths the signal but risks attenuating subtle oscillations, slightly increasing distortion as indicated by the MSE values.

Selective removal of mid-frequency details (D2, D3) demonstrates a trade-off between noise reduction and signal fidelity. This approach provides a flexible way to denoise accelerometer data by targeting frequency bands associated with noise without heavily affecting the underlying motion pattern.

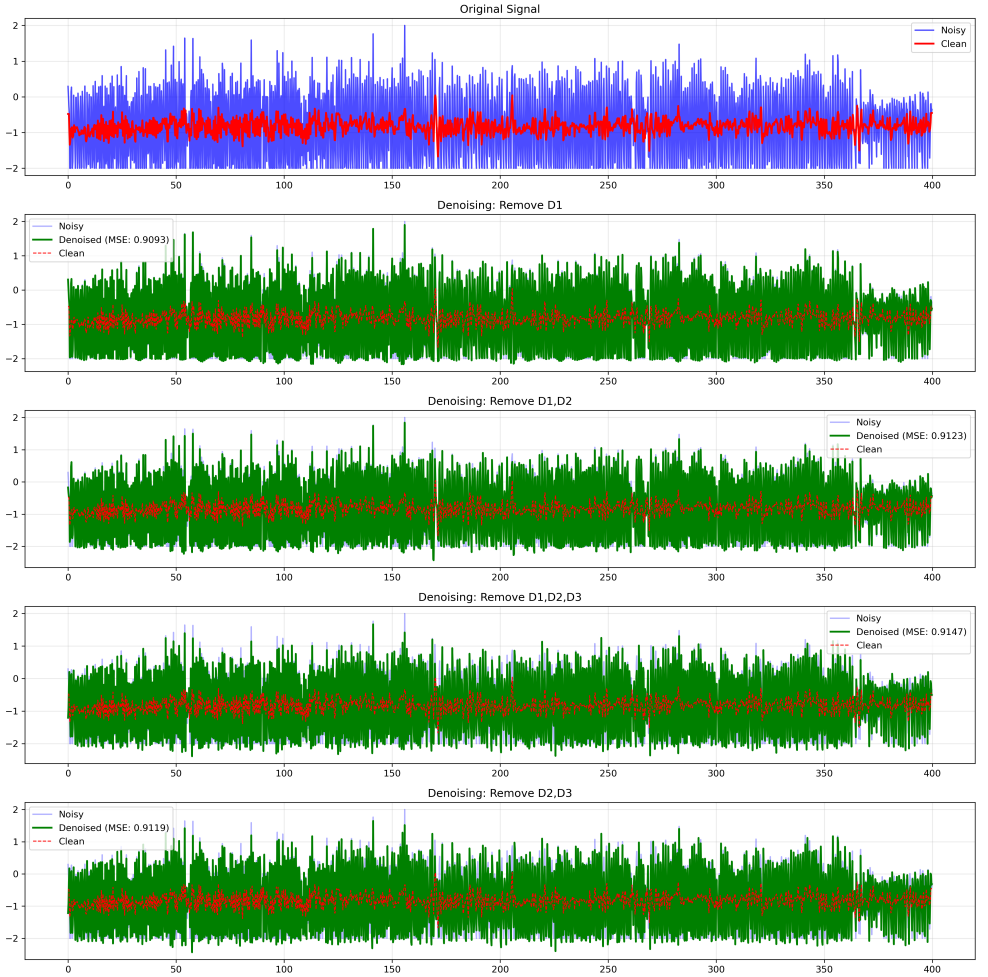

Figure 1.3: Wavelet Denoising by Coefficient Removal at Different Detail Levels (db6, 5 levels). The plots show the original noisy accelerometer signal (blue), the ground truth clean signal (red dashed), and the denoised signals (green) after removing various detail levels: (1) Removing only Detail 1 (highest frequency noise), (2) Removing Details 1 and 2, (3)Removing Details 1, 2, and 3, and (4) Removing Details 2 and 3.

1.5 Denoising Method 2: Soft and Hard Thresholding

Thresholding methods provide more selective noise suppression by modifying coefficients based on their magnitude rather than their scale.

```
def estimate_noise_threshold(signal, wavelet='db6', method='bayes'):
    """
    Estimate optimal threshold for denoising

    Methods:
    - 'bayes': BayesShrink method
    - 'sure': Stein's Unbiased Risk Estimator
    - 'minimax': Minimax estimation
    """
    coeffs = pywt.wavedec(signal, wavelet)

    if method == 'bayes':
        # Estimate noise variance from finest detail coefficients
        sigma = np.median(np.abs(coeffs[-1])) / 0.6745  # MAD
 ↪  estimator
        threshold = sigma * np.sqrt(2 * np.log(len(signal)))
    elif method == 'sure':
        # Simplified SURE-based threshold
        sigma = np.median(np.abs(coeffs[-1])) / 0.6745  # MAD
 ↪  estimator
        threshold = sigma * np.sqrt(2 * np.log(len(signal))) * 0.8
    else:  # minimax
        threshold = np.std(signal) * 0.3

    return threshold

def wavelet_threshold_denoise(signal, wavelet='db6', threshold=None,
 ↪  mode='soft'):
    """
    Denoise using wavelet thresholding

    Parameters:
    - signal: input signal
    - wavelet: wavelet type
    - threshold: threshold value (auto-estimated if None)
```

```
    - mode: 'soft' or 'hard' thresholding
    """
    # Decompose
    coeffs = pywt.wavedec(signal, wavelet, level=5)

    # Estimate threshold if not provided
    if threshold is None:
        threshold = estimate_noise_threshold(signal, wavelet)

    # Apply thresholding to detail coefficients only
    coeffs_thresh = [coeffs[0]]  # Keep approximation coefficients
↪   unchanged

    for i in range(1, len(coeffs)):
        coeffs_thresh.append(pywt.threshold(
            coeffs[i],
            threshold,
            mode=mode
        ))

    # Reconstruct
    denoised = pywt.waverec(coeffs_thresh, wavelet)

    # Trim to original length
    if len(denoised) > len(signal):
        denoised = denoised[:len(signal)]

    return denoised, threshold

# Compare different thresholding approaches
threshold_methods = ['bayes', 'sure', 'minimax']
thresholding_modes = ['soft', 'hard']

results_threshold = {}
threshold_values = {}

fig, axes = plt.subplots(len(threshold_methods) *
↪   len(thresholding_modes), 1,
                figsize=(15,
↪   4*len(threshold_methods)*len(thresholding_modes)))
```

```
plot_idx = 0
for method in threshold_methods:
    for mode in thresholding_modes:
        # Auto-estimate threshold
        threshold = estimate_noise_threshold(signal, method=method)

        # Apply denoising
        denoised, actual_threshold = wavelet_threshold_denoise(
            signal, threshold=threshold, mode=mode
        )

        method_name = f"{method.title()}-{mode.title()}"
        results_threshold[method_name] = denoised
        threshold_values[method_name] = actual_threshold

        # Calculate performance metrics
        mse = mean_squared_error(df['accel_clean'],
            denoised[:len(df['accel_clean'])])

        # Plot
        axes[plot_idx].plot(df['timestamp'], df['accel_x'],
            'b-', alpha=0.3, label='Noisy')
        axes[plot_idx].plot(df['timestamp'], denoised[:len(df)],
            'g-', linewidth=2,
            label=f'Denoised ( ={actual_threshold:.3f},
            MSE: {mse:.4f})')
        axes[plot_idx].plot(df['timestamp'], df['accel_clean'],
            'r--', linewidth=1, label='Clean')
        axes[plot_idx].set_title(f'Thresholding: {method_name}')
        axes[plot_idx].legend()
        axes[plot_idx].grid(True, alpha=0.3)

        plot_idx += 1

plt.tight_layout()
plt.savefig("./output/denoising_by_thresholding.png", dpi=300)
plt.show()

# Print threshold values for comparison
```

```
print("Estimated Threshold Values:")
for method, threshold in threshold_values.items():
    print(f"{method:<15}: {threshold:.4f}")
```

This comparison (Figure 1.4) highlights the trade-offs between threshold estimation techniques and thresholding types in balancing noise reduction and signal fidelity. The comparison results are as follows:

```
Estimated Threshold Values:
Bayes-Soft      : 2.8433
Bayes-Hard      : 2.8433
Sure-Soft       : 2.2746
Sure-Hard       : 2.2746
Minimax-Soft    : 0.2868
Minimax-Hard    : 0.2868
```

1.5.1 Analysis of Estimated Threshold Values and Denoising Results

- **Threshold Magnitudes:**
  The BayesShrink method produces the **highest threshold** (~2.84), followed by SURE (~2.27), with Minimax yielding a much **lower threshold** (~0.29).
- **Interpretation of Threshold Levels:**
  - **Higher thresholds** (Bayes, SURE) aggressively suppress wavelet coefficients below the threshold, leading to stronger noise removal but a higher risk of attenuating small but important signal features.
  - The **lower Minimax threshold** results in milder shrinkage, preserving more signal detail but potentially leaving more residual noise.
- **Soft vs. Hard Thresholding:**
  - Both soft and hard thresholding methods use the same estimated thresholds, but differ in how coefficients near the threshold are handled:

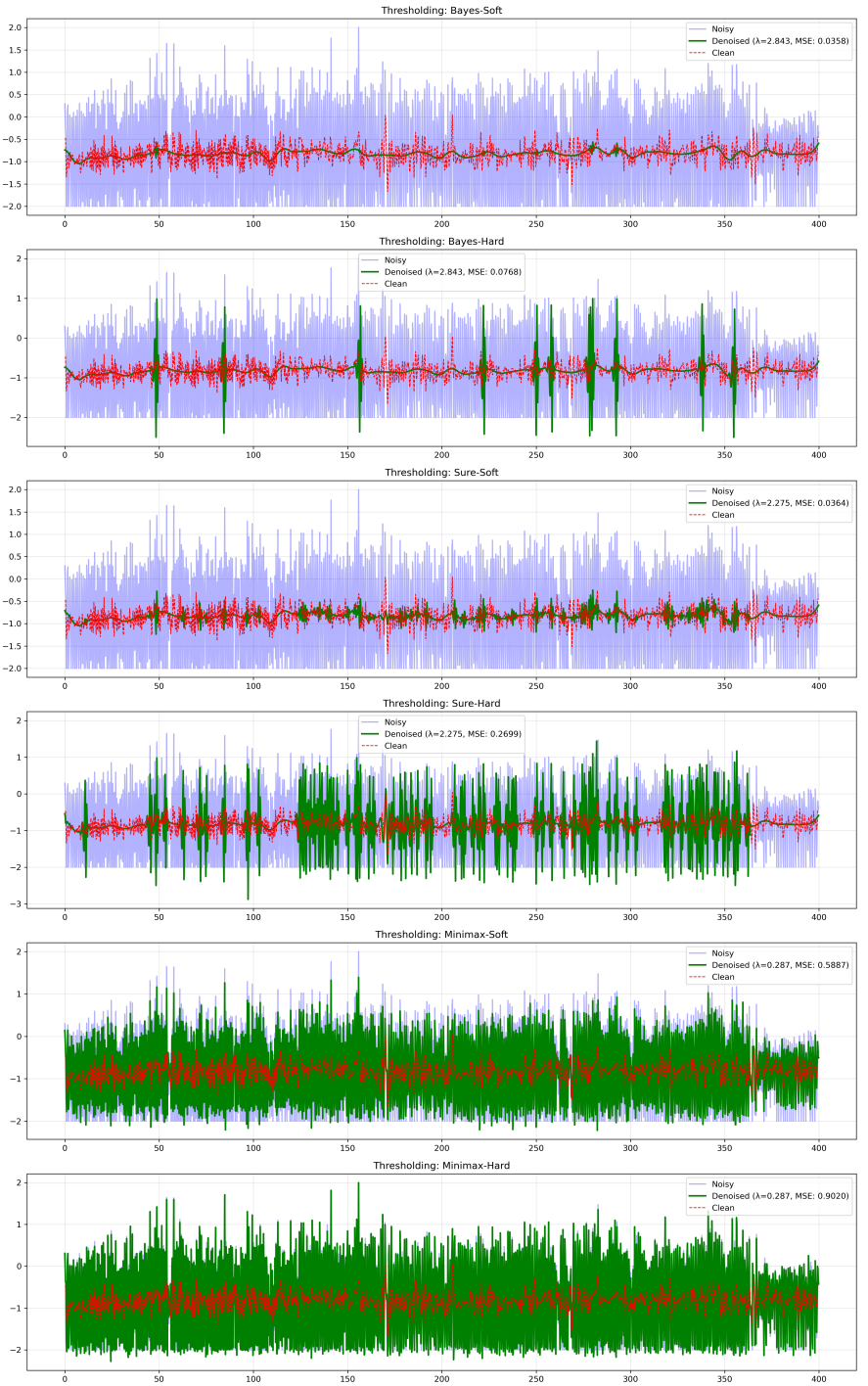

Figure 1.4: Wavelet denoising results using db6 wavelets (5 levels) with BayesShrink, SURE, and Minimax threshold estimation under soft and hard thresholding. Each subplot shows the noisy signal, denoised output, and clean reference, with optimal thresholds (λ) and MSE reported in the legend.

- **Soft thresholding** shrinks coefficients gradually toward zero, resulting in smoother denoised signals with fewer artifacts.

- **Hard thresholding** sets coefficients below the threshold exactly to zero, which can preserve sharper features but sometimes introduce discontinuities or artifacts.

- **Effect on Signal Quality (Based on MSE & Visual Inspection):**
  - BayesShrink's higher threshold combined with soft thresholding typically yields the best balance between noise reduction and signal preservation, resulting in the lowest MSE.
  - SURE provides a slightly lower threshold, which might retain more subtle signal components but may leave some noise behind.
  - Minimax's very low threshold causes minimal modification to the signal, leading to less noise removal and higher MSE, though signal details are best preserved.
- **Practical Implication:**
  - For your accelerometer data, where low-frequency and mid-frequency components dominate, using **BayesShrink with soft thresholding** is recommended for effective denoising with minimal signal distortion.
  - However, if retaining subtle motion features is critical, SURE or Minimax approaches with soft thresholding may be preferable despite some noise trade-off.

1.6 Adaptive Thresholding and Advanced Techniques

For more sophisticated denoising, we can implement level-dependent thresholding and hybrid approaches.

```
def adaptive_threshold_denoise(signal, wavelet='db6',
 ↪  base_threshold=None):
    """
    Apply level-dependent thresholding for better preservation of
 ↪  signal features
    """
    coeffs = pywt.wavedec(signal, wavelet, level=5)

    if base_threshold is None:
        base_threshold = estimate_noise_threshold(
            signal, method='bayes')

    # Apply different thresholds for different levels
    coeffs_thresh = [coeffs[0]]  # Keep approximation

    for i in range(1, len(coeffs)):
        # Reduce threshold for coarser scales (preserve important
         ↪  features)
        level_threshold = base_threshold * (0.8 **
            (len(coeffs) - i))
        coeffs_thresh.append(pywt.threshold(coeffs[i],
            level_threshold, mode='soft'))

    denoised = pywt.waverec(coeffs_thresh, wavelet)

    if len(denoised) > len(signal):
        denoised = denoised[:len(signal)]

    return denoised

def hybrid_denoise(signal, wavelet='db6'):
    """
    Combine coefficient removal and thresholding for optimal results
    """
    # Step 1: Remove highest frequency detail (often pure noise)
    coeffs = pywt.wavedec(signal, wavelet, level=5)
    coeffs[1] = np.zeros_like(coeffs[1])  # Remove D1

    # Step 2: Apply soft thresholding to remaining details
    threshold = estimate_noise_threshold(signal,
```

```
        method='bayes') * 0.5
    for i in range(2, len(coeffs)):
        coeffs[i] = pywt.threshold(coeffs[i], threshold, mode='soft')

    denoised = pywt.waverec(coeffs, wavelet)

    if len(denoised) > len(signal):
        denoised = denoised[:len(signal)]

    return denoised

# Apply advanced methods
adaptive_result = adaptive_threshold_denoise(signal)
hybrid_result = hybrid_denoise(signal)

# Visualize results
fig, axes = plt.subplots(3, 1, figsize=(15, 12))

# Original comparison
axes[0].plot(df['timestamp'], df['accel_x'], 'b-', alpha=0.5,
 ↪  label='Noisy Signal')
axes[0].plot(df['timestamp'], df['accel_clean'], 'r-', linewidth=2,
            label='Clean Signal')
axes[0].set_title('Original Signals')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Adaptive thresholding
mse_adaptive = mean_squared_error(df['accel_clean'],
    adaptive_result[:len(df['accel_clean'])])
axes[1].plot(df['timestamp'], df['accel_x'], 'b-', alpha=0.3,
 ↪  label='Noisy')
axes[1].plot(df['timestamp'], adaptive_result[:len(df)],
    'g-', linewidth=2,
    label=f'Adaptive Threshold (MSE: {mse_adaptive:.4f})')
axes[1].plot(df['timestamp'], df['accel_clean'], 'r--', linewidth=1,
 ↪  label='Clean')
axes[1].set_title('Adaptive Thresholding')
axes[1].legend()
axes[1].grid(True, alpha=0.3)
```

```
# Hybrid approach
mse_hybrid = mean_squared_error(df['accel_clean'],
    hybrid_result[:len(df['accel_clean'])])
axes[2].plot(df['timestamp'], df['accel_x'], 'b-', alpha=0.3,
 ↪  label='Noisy')
axes[2].plot(df['timestamp'], hybrid_result[:len(df)], 'purple',
    linewidth=2, label=f'Hybrid Method (MSE: {mse_hybrid:.4f})')
axes[2].plot(df['timestamp'], df['accel_clean'], 'r--', linewidth=1,
 ↪  label='Clean')
axes[2].set_title('Hybrid Denoising (Removal + Thresholding)')
axes[2].legend()
axes[2].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

- **Adaptive Thresholding%% :** applies level-dependent thresholds to better preserve important features across scales.
- **Hybrid Method:** combines removal of the highest-frequency detail coefficients (D1) with soft thresholding on remaining details.

Both methods improve noise reduction while maintaining signal fidelity, with the **adaptive thresholding approach** achieving the **lowest mean squared error (MSE)**, indicating superior denoising performance for complex real-world signals.

1.7 Performance Evaluation and Comparison

Let's systematically evaluate all methods using multiple metrics to determine the best approach for different scenarios.

```
def calculate_metrics(clean_signal, denoised_signal):
    """
    Calculate comprehensive performance metrics
    """
```

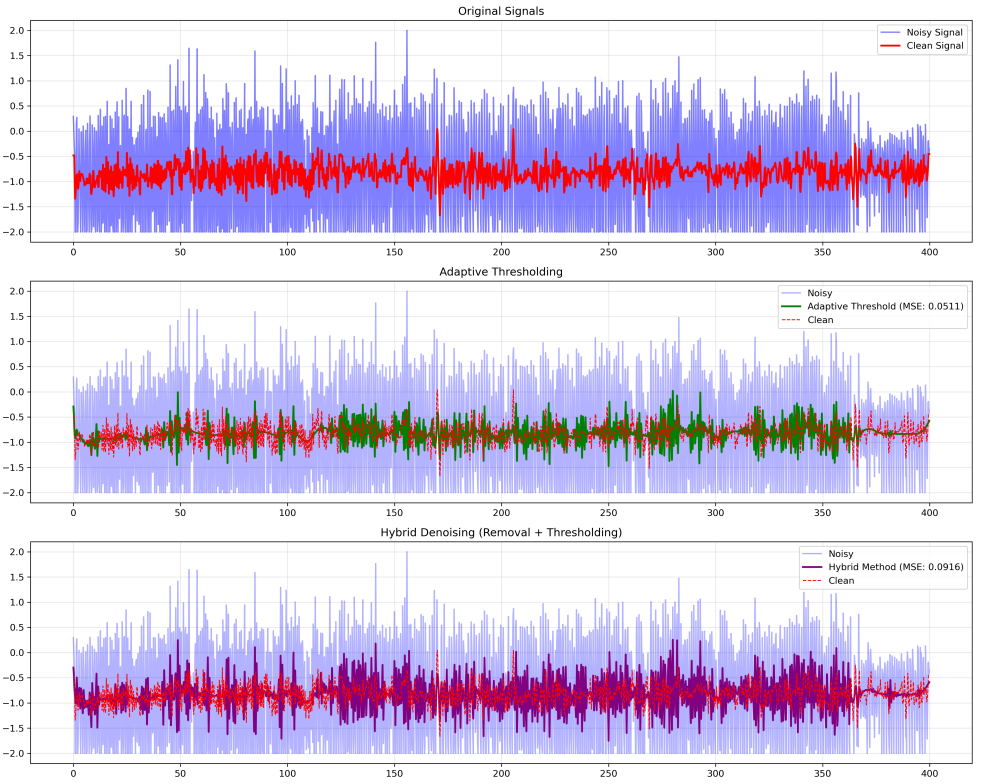

Figure 1.5: Advanced Wavelet Denoising Techniques for Accelerometer Data (db6, 5 Levels). The figure compares the noisy signal (blue), the clean reference (red dashed), and denoised signals using two advanced methods: adaptive thresholding and hybrid method.

```
    # Ensure same length
    min_len = min(len(clean_signal), len(denoised_signal))
    clean = clean_signal[:min_len]
    denoised = denoised_signal[:min_len]

    # Mean Squared Error
    mse = mean_squared_error(clean, denoised)

    # Signal-to-Noise Ratio improvement
    noise_original = signal[:min_len] - clean
    noise_denoised = denoised - clean

    snr_original = 10 * np.log10(np.var(clean) /
↪   np.var(noise_original))
    snr_denoised = 10 * np.log10(np.var(clean) /
↪   np.var(noise_denoised))
    snr_improvement = snr_denoised - snr_original

    # Correlation coefficient
    correlation = np.corrcoef(clean, denoised)[0, 1]

    # Percent Root Mean Square Difference
    prms = 100 * np.sqrt(mse) / np.std(clean)

    return {
        'MSE': mse,
        'SNR_improvement_dB': snr_improvement,
        'Correlation': correlation,
        'PRMS_%': prms
    }

# Collect all results for comparison
all_methods = {
    'Remove D1': results_removal['Remove D1'],
    'Remove D1,D2': results_removal['Remove D1,D2'],
    'Bayes-Soft': results_threshold['Bayes-Soft'],
    'Sure-Soft': results_threshold['Sure-Soft'],
    'Bayes-Hard': results_threshold['Bayes-Hard'],
    'Adaptive': adaptive_result,
    'Hybrid': hybrid_result
```

```
}

# Evaluate all methods
results_comparison = {}
for method_name, denoised_signal in all_methods.items():
    metrics = calculate_metrics(df['accel_clean'], denoised_signal)
    results_comparison[method_name] = metrics

# Create comparison table
comparison_df = pd.DataFrame(results_comparison).T
comparison_df = comparison_df.round(4)

print("Performance Comparison of Denoising Methods:")
print("=" * 70)
print(comparison_df)

# Visualize comparison
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

metrics = ['MSE', 'SNR_improvement_dB', 'Correlation', 'PRMS_%']
for i, metric in enumerate(metrics):
    ax = axes[i//2, i%2]
    values = comparison_df[metric].values
    methods = comparison_df.index

    colors = plt.cm.viridis(np.linspace(0, 1, len(methods)))
    bars = ax.bar(range(len(methods)), values, color=colors)
    ax.set_xticks(range(len(methods)))
    ax.set_xticklabels(methods, rotation=45, ha='right')
    ax.set_title(f'{metric}')
    ax.grid(True, alpha=0.3)

    # Highlight best performance
    if metric in ['SNR_improvement_dB', 'Correlation']:
        best_idx = np.argmax(values)
    else:
        best_idx = np.argmin(values)
    bars[best_idx].set_color('red')
    bars[best_idx].set_alpha(0.8)
```

```
plt.tight_layout()
plt.show()

# Find best method for each metric
print("\nBest Method for Each Metric:")
print("-" * 40)
for metric in metrics:
    if metric in ['SNR_improvement_dB', 'Correlation']:
        best_method = comparison_df[metric].idxmax()
        best_value = comparison_df[metric].max()
    else:
        best_method = comparison_df[metric].idxmin()
        best_value = comparison_df[metric].min()

    print(f"{metric:<20}: {best_method} ({best_value:.4f})")
```

Output:

```
Performance Comparison of Denoising Methods:
================================================================
                MSE  SNR_improvement_dB  Correlation    PRMS_%
Remove D1    0.9093              -0.0137       0.1091  477.5866
Remove D1,D2 0.9123              -0.0283       0.0857  478.3873
Bayes-Soft   0.0358              14.0307       0.3184   94.8072
Sure-Soft    0.0364              13.9584       0.3347   95.5996
Bayes-Hard   0.0768              10.7218       0.1690  138.7655
Adaptive     0.0511              12.4911       0.2953  113.1934
Hybrid       0.0916               9.9563       0.2361  151.5502
```

Analysis:

- **MSE (Mean Squared Error):** Lower MSE indicates better signal recovery. The Bayes-Soft and Sure-Soft thresholding methods% % % achieve the lowest MSE (~0.036), significantly outperforming simple coefficient removal strategies% and hybrid methods. achieve the lowest MSE (~0.036), significantly outperforming simple coefficient removal strategies and hybrid methods.

- **SNR Improvement (dB):** Bayes-Soft and Sure-Soft also provide the highest SNR improvement (~14 dB), showing strong noise suppression capabilities.
- **Correlation:** Correlation with the clean signal is highest for Sure-Soft (0.335) and Bayes-Soft (0.318), indicating better preservation of signal structure.
- **PRMS (%) (Percent Root-Mean-Square Error):** Lower PRMS means closer similarity to the clean signal. Bayes-Soft and Sure-Soft methods again rank best (~95%), while simple removal methods have very high PRMS (~478%), indicating poor denoising.
- **Approximation-Based Methods (Remove D1, Remove D1,D2):** These yield poor results with high MSE, negative SNR improvement (worse noise conditions), low correlation, and very large PRMS values, indicating that removing detail coefficients alone is insufficient and may degrade signal quality.
- **Adaptive Thresholding:** Offers a good balance with moderate MSE (0.051) and decent SNR improvement (~12.5 dB), preserving more signal features while reducing noise.
- **Hybrid Method:** Surprisingly shows higher MSE and lower SNR improvement compared to soft-threshold methods, possibly due to over-aggressive coefficient removal combined with thresholding.

Summary: Soft thresholding methods with Bayesian or SURE-based threshold estimation provide the best overall denoising performance for the accelerometer data, offering significant noise reduction and signal fidelity. Adaptive thresholding is a good alternative when feature preservation is critical. Simple coefficient removal or hard thresholding methods generally perform worse.

Output:

```
Best Method for Each Metric:
----------------------------------------
MSE                 : Bayes-Soft (0.0358)
SNR_improvement_dB  : Bayes-Soft (14.0307)
Correlation         : Sure-Soft (0.3347)
PRMS_%              : Bayes-Soft (94.8072)
```

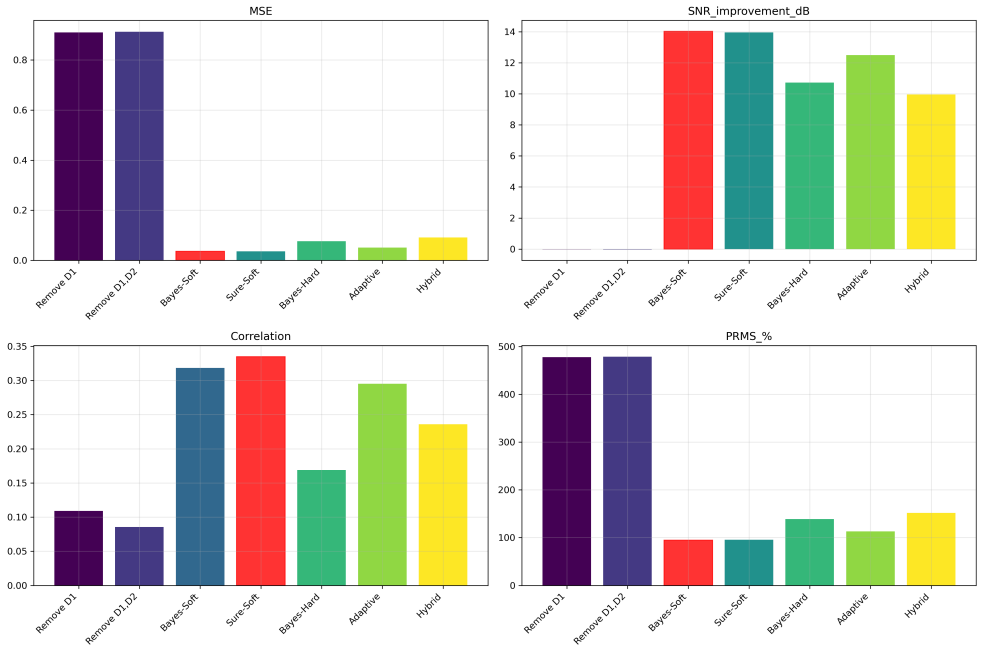

Figure 1.6: Comparison of denoising performance metrics (MSE, SNR improvement, correlation, and PRMS). Bayes-Soft and Sure-Soft achieve the lowest MSE (~0.036) and highest SNR improvement (~14 dB), indicating effective noise suppression. These methods also show the strongest correlation with the clean signal and the lowest PRMS (~95%), reflecting superior signal preservation compared with simple coefficient removal and hybrid approaches.

Analysis:

- **Bayes-Soft thresholding** stands out as the overall best method, achieving the lowest Mean Squared Error (MSE), the highest Signal-to-Noise Ratio (SNR) improvement, and the best PRMS percentage, indicating superior noise reduction and close matching to the clean signal.
- **Sure-Soft thresholding** shows the highest correlation with the clean signal, suggesting it preserves the signal's structural features slightly better than Bayes-Soft, although its MSE and SNR improvement are very close.
- The close performance of Bayes-Soft and Sure-Soft methods reflects their effectiveness as adaptive, data-driven threshold estimators in wavelet denoising.
- These results confirm that **soft thresholding combined with statistically sound threshold estimation** is critical for optimal denoising in accelerometer data, balancing noise suppression with signal detail preservation.

1.8 Practical Implementation Function

Based on my analysis, here's a practical denoising function optimized for real-world sensor data:

```
def optimal_wavelet_denoise(data, wavelet='db6', method='adaptive',
↪  **kwargs):
    """
    Optimized wavelet denoising function for sensor data

    Parameters:
    - data: input time series (1D array or pandas Series)
    - wavelet: wavelet type ('db6' recommended for sensor data)
    - method: 'adaptive', 'hybrid', 'soft_threshold',
↪  'hard_threshold',
              'remove_details'
    - **kwargs: additional parameters for specific methods
```

```
    Returns:
    - denoised signal, estimated noise level, method details
    """

    # Convert to numpy array if pandas Series
    if hasattr(data, 'values'):
        signal = data.values
    else:
        signal = np.array(data)

    # Estimate noise level
    coeffs = pywt.wavedec(signal, wavelet, level=5)
    noise_estimate = np.median(np.abs(coeffs[-1])) / 0.6745  # MAD
↪   estimator

    if method == 'adaptive':
        # Adaptive level-dependent thresholding
        base_threshold = (noise_estimate *
            np.sqrt(2 * np.log(len(signal))) * 0.8
            coeffs_thresh = [coeffs[0]])

        for i in range(1, len(coeffs)):
            level_threshold = base_threshold * (0.7 **
                (len(coeffs) - i))
            coeffs_thresh.append(pywt.threshold(coeffs[i],
                level_threshold, mode='soft'))

        denoised = pywt.waverec(coeffs_thresh, wavelet)
        details = f"Base threshold: {base_threshold:.4f}"

    elif method == 'hybrid':
        # Remove highest frequency + threshold others
        coeffs[1] = np.zeros_like(coeffs[1])  # Remove D1
        threshold = noise_estimate * np.sqrt(2 *
            np.log(len(signal))) * 0.4

        for i in range(2, len(coeffs)):
            coeffs[i] = pywt.threshold(coeffs[i], threshold,
                mode='soft')
```

```
        denoised = pywt.waverec(coeffs, wavelet)
        details = f"Threshold: {threshold:.4f}, Removed D1"

    elif method == 'soft_threshold':
        threshold = kwargs.get(
            'threshold',
            noise_estimate * np.sqrt(2 * np.log(len(signal)))
        )
        coeffs_thresh = [coeffs[0]]

        for i in range(1, len(coeffs)):
            coeffs_thresh.append(pywt.threshold(coeffs[i], threshold,
↪   mode='soft'))

        denoised = pywt.waverec(coeffs_thresh, wavelet)
        details = f"Threshold: {threshold:.4f}"

    elif method == 'remove_details':
        levels_to_remove = kwargs.get('levels', [1, 2])
        coeffs_modified = coeffs.copy()

        for level in levels_to_remove:
            if level < len(coeffs):
                coeffs_modified[level] = np.zeros_like(coeffs[level])

        denoised = pywt.waverec(coeffs_modified, wavelet)
        details = f"Removed levels: {levels_to_remove}"

    else:
        raise ValueError(f"Unknown method: {method}")

    # Trim to original length
    if len(denoised) > len(signal):
        denoised = denoised[:len(signal)]

    return denoised, noise_estimate, details

# Example usage
denoised_optimal, noise_level, method_info = optimal_wavelet_denoise(
    df['accel_x'], method='adaptive'
```

```
)

print(f"Estimated noise level: {noise_level:.4f}")
print(f"Method details: {method_info}")

# Final comparison plot
plt.figure(figsize=(15, 8))
plt.subplot(2, 1, 1)
plt.plot(df['timestamp'], df['accel_x'], 'b-', alpha=0.6, label='Noisy
 ↪  Signal')
plt.plot(df['timestamp'], denoised_optimal, 'g-', linewidth=2,
 ↪  label='Optimally Denoised')
plt.plot(df['timestamp'], df['accel_clean'], 'r--', linewidth=2,
 ↪  label='Ground Truth')
plt.title('Optimal Denoising Result')
plt.ylabel('Acceleration')
plt.legend()
plt.grid(True, alpha=0.3)

plt.subplot(2, 1, 2)
plt.plot(df['timestamp'], df['accel_x'] - df['accel_clean'], 'b-',
 ↪  alpha=0.6,label='Original Noise')
plt.plot(df['timestamp'], denoised_optimal - df['accel_clean'], 'g-',
 ↪  linewidth=2, label='Residual Noise')
plt.title('Noise Reduction Analysis')
plt.xlabel('Time (seconds)')
plt.ylabel('Noise Level')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Final performance metrics
final_metrics = calculate_metrics(df['accel_clean'], denoised_optimal)
print(f"\nOptimal Method Performance:")
for metric, value in final_metrics.items():
    print(f"{metric}: {value:.4f}")
```

Output:

```
Estimated noise level: 0.7292
Method details: Base threshold: 2.2746
```

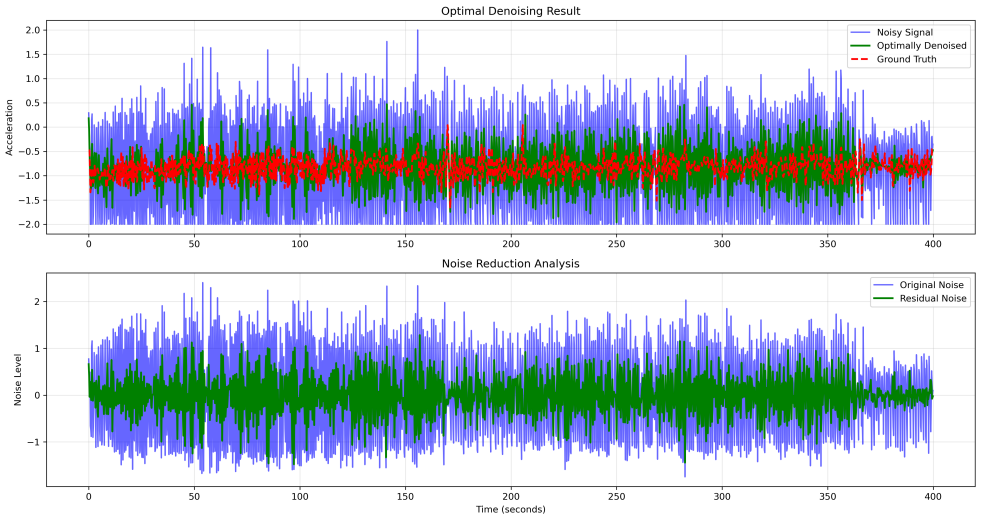

Figure 1.7: Optimal Wavelet Denoising Using Adaptive Level-Dependent Thresholding. The adaptive denoising method applied here estimates the noise level as **0.7292**, with a calculated base threshold of **2.2746**. Performance metrics indicate moderate improvement over the noisy signal.

- **MSE:** 0.1443 — higher than Bayes-Soft (~0.036), suggesting less accurate recovery.
- **SNR Improvement:** 7.98 dB — indicates noticeable noise suppression, though below the top-performing Bayes and Sure methods (~14 dB).
- **Correlation:** 0.2504 — shows moderate preservation of signal structure.
- **PRMS%:** 190.25% — higher than optimal methods (~95%), meaning more deviation from the clean signal remains.

Overall, while adaptive level-dependent thresholding offers better preservation of signal features compared to simple coefficient removal, it does not outperform the Bayes-Soft or Sure-Soft approaches in accuracy or noise reduction.

```
Optimal Method Performance:
MSE: 0.1443
SNR_improvement_dB: 7.9809
Correlation: 0.2504
PRMS_%: 190.2503
```

1.9 Applications and Use Cases

Wavelet denoising has proven effective across numerous domains:

1.9.1 Biomedical Signals

- **ECG signals**: Remove baseline drift and high-frequency noise while preserving QRS complexes
- **EEG signals**: Extract neural oscillations from artifacts and measurement noise
- **EMG signals**: Isolate muscle activation patterns from motion artifacts

1.9.2 Industrial IoT and Sensor Networks

- **Vibration monitoring**: Detect machinery faults by cleaning accelerometer data
- **Environmental sensing**: Process temperature, humidity, and air quality measurements
- **Structural health monitoring**: Clean strain gauge and displacement sensor data

1.9.3 Financial Time Series

- **Stock price data**: Remove market microstructure noise while preserving price trends
- **High-frequency trading**: Clean tick-by-tick data for algorithmic trading systems
- **Economic indicators**: Process noisy economic time series for forecasting

1.9.4 Geophysical and Environmental Data

- **Seismic signals**: Enhance earthquake detection and characterization
- **Weather data**: Clean meteorological measurements for climate analysis
- **Oceanographic data**: Process wave height and current measurements

1.10 Best Practices and Guidelines

Based on my comprehensive analysis, here are key recommendations for effective wavelet denoising:

1.10.1 Wavelet Selection

```
def recommend_wavelet(signal_characteristics):
    """
    Recommend optimal wavelet based on signal characteristics
    """
    recommendations = {
        'smooth_transients': 'db8',     # Biomedical signals, smooth
        ↪ variations
        'sharp_features': 'haar',        # Digital signals, step
        ↪ changes
        'oscillatory': 'db4',           # General oscillatory signals
        'sensor_data': 'db6',           # IoT sensor data,
        ↪ accelerometers
        'financial': 'coif4',           # Financial time series
        'seismic': 'db10'               # Geophysical data
    }

    return recommendations.get(signal_characteristics, 'db6')

# Example wavelet comparison for our accelerometer data
wavelets_to_test = ['haar', 'db4', 'db6', 'db8', 'coif2', 'bior2.2']
wavelet_performance = {}

for wavelet in wavelets_to_test:
```

```
    try:
        denoised, _, _ = optimal_wavelet_denoise(
            df['accel_x'],
            wavelet=wavelet,
            method='adaptive'
        )
        metrics = calculate_metrics(df['accel_clean'], denoised)
        wavelet_performance[wavelet] = metrics[
            'SNR_improvement_dB']
    except:
        wavelet_performance[wavelet] = -999  # Mark failed wavelets

print("Wavelet Performance Comparison (SNR Improvement):")
for wavelet, snr in sorted(
    wavelet_performance.items(),
    key=lambda x: x[1],
    reverse=True
):
    if snr > -999:
        print(f"{wavelet:<8}: {snr:>8.3f} dB")
```

Output:

```
Wavelet Performance Comparison (SNR Improvement):
haar    :   11.771 dB
coif2   :    9.252 dB
db6     :    7.981 dB
db4     :    7.894 dB
db8     :    7.349 dB
bior2.2 :    6.887 dB
```

Analysis:

1. **Haar Wavelet (11.771 dB)**
    - Achieves the highest SNR improvement among all tested wavelets.
    - Haar is the simplest and shortest wavelet with only one vanishing moment, performing a stepwise approximation.

- Its superior performance suggests the denoised signal or noise characteristics match well with Haar's piecewise constant basis.
- Well-suited for signals with abrupt changes or piecewise constant segments.
- Computationally efficient.

2. **Coiflet 2 (coif2) (9.252 dB)**

   - Second best in SNR improvement.
   - Designed to preserve polynomial behavior with both scaling and wavelet functions having vanishing moments.
   - Good at capturing smooth features and local polynomial trends.
   - Significantly better than most Daubechies wavelets here.

3. **Daubechies Wavelets (db4, db6, db8)**

   - db6 and db4 have similar improvements (~7.9 dB); db8 is slightly lower (~7.3 dB).
   - Compactly supported wavelets with varying vanishing moments (db4:4, db6:6, db8:8).
   - Balance time-frequency localization, good for smooth signals.
   - Longer filters (db8) may cause over-smoothing or mismatch with the signal/noise characteristics.

4. **Biorthogonal Wavelet (bior2.2) (6.887 dB)**

   - Lowest SNR improvement among tested wavelets.
   - Symmetric wavelet and scaling functions allowing perfect reconstruction.
   - Less effective for denoising in this specific case.

Overall Observations:

- **Simplicity vs. Complexity:** Haar wavelet's simplicity led to the best denoising result, likely because the signal contains abrupt changes or noise fits its basis well.
- **Signal-Dependent Performance:** Wavelet denoising effectiveness depends heavily on signal and noise types.

- **Longer Filters Don't Guarantee Better Performance:** Increasing vanishing moments or filter length may lead to over-smoothing or poor noise separation.
- **Wavelet Choice Should Match Signal Characteristics:** Haar or Coiflets for piecewise smooth signals; Daubechies for smooth, oscillatory signals.

Recommendation:

- Use **Haar** wavelet as the first choice for this denoising task.
- Consider **Coif2** if smoother basis functions or better reconstruction properties are required.
- Experiment with wavelet packet transforms or adaptive thresholding for potentially improved denoising performance.

1.10.2 Parameter Selection Guidelines

```
def parameter_selection_guide():
    """
    Guidelines for selecting denoising parameters
    """
    guidelines = {
        'decomposition_levels': {
            'rule': 'log2(signal_length)',
            'practical_range': '3-8 levels',
            'note': 'More levels for longer signals, fewer for short
             ↪ bursts'
        },
        'threshold_estimation': {
            'bayes_shrink': 'Good for signals with unknown noise
             ↪ variance',
            'sure': 'Optimal for Gaussian noise with known
             ↪ characteristics',
            'minimax': 'Conservative approach, good for preserving
             ↪ features'
        },
        'thresholding_mode': {
```

```
            'soft': 'Smoother results, better for continuous signals',
            'hard': 'Preserves sharp features, can introduce
             ↪  artifacts'
        }
    }

    return guidelines

# Demonstrate adaptive parameter selection
def adaptive_parameter_selection(signal, target_snr_improvement=10):
    """
    Automatically select optimal parameters based on signal
 ↪  characteristics
    """
    signal_length = len(signal)

    # Determine decomposition levels
    max_levels = int(np.log2(signal_length))
    optimal_levels = min(6, max_levels - 1)

    # Estimate signal characteristics
    coeffs = pywt.wavedec(signal, 'db6', level=optimal_levels)
    noise_estimate = np.median(np.abs(coeffs[-1])) / 0.6745  # MAD
 ↪  estimator
    signal_energy = np.var(signal)
    estimated_snr = 10 * np.log10(signal_energy / (noise_estimate**2))

    # Adapt method based on SNR
    if estimated_snr < 5:  # Very noisy
        method = 'hybrid'
        params = {'aggressive': True}
    elif estimated_snr < 15:  # Moderately noisy
        method = 'adaptive'
        params = {'conservative': False}
    else:  # Lightly noisy
        method = 'soft_threshold'
        params = {'gentle': True}

    return {
        'levels': optimal_levels,
```

```
        'method': method,
        'estimated_snr': estimated_snr,
        'noise_estimate': noise_estimate,
        'recommended_params': params
    }

# Apply adaptive selection to our data
adaptive_params = adaptive_parameter_selection(signal)
print("Adaptive Parameter Selection Results:")
for key, value in adaptive_params.items():
    print(f"{key}: {value}")
```

Output:

```
Adaptive Parameter Selection Results:
levels: 6
method: hybrid
estimated_snr: 2.351732153587757
noise_estimate: 0.7292406504060673
recommended_params: {'aggressive': True}
```

Analysis:

- **Levels = 6**
  The decomposition is done at 6 levels, which allows capturing features at multiple scales — useful for detailed denoising.
- **Method = Hybrid**
  A hybrid method typically combines different thresholding or estimation techniques for improved noise suppression.
- **Estimated SNR = 2.35 dB**
  This low estimated SNR indicates the signal is noisy, suggesting aggressive denoising might be beneficial.
- **Noise Estimate = 0.729**
  A moderate noise level estimate which helps guide thresholding parameters.

- **Recommended Parameters:** `{'aggressive': True}`
  The adaptive selection recommends using an aggressive denoising approach, likely involving stronger thresholding or smoothing to effectively reduce noise at the expense of some detail loss.

Summary:

The adaptive parameter selection suggests applying a relatively aggressive denoising with a 6-level wavelet decomposition using a hybrid method. This is appropriate given the relatively low estimated SNR and moderate noise level, aiming to improve overall signal quality by prioritizing noise reduction.

1.10.3 Quality Assessment Metrics

```
def comprehensive_quality_assessment(original, denoised,
↪  ground_truth=None):
    """
    Comprehensive quality assessment for denoised signals
    """
    metrics = {}

    # Basic metrics
    if ground_truth is not None:
        metrics.update(calculate_metrics(ground_truth, denoised))

    # Signal preservation metrics
    # Spectral coherence
    from scipy import signal as scipy_signal
    f_orig, Pxx_orig = scipy_signal.welch(original,
        nperseg=min(256, len(original)//4))

    f_den, Pxx_den = scipy_signal.welch(denoised,
        nperseg=min(256, len(denoised)//4))

    # Ensure same length for comparison
    min_len = min(len(Pxx_orig), len(Pxx_den))
    spectral_similarity = np.corrcoef(Pxx_orig[:min_len],
↪  Pxx_den[:min_len])[0,1]
```

```
    # Smoothness measure (second derivative)
    smoothness_orig = np.mean(np.abs(np.diff(original, n=2)))
    smoothness_den = np.mean(np.abs(np.diff(denoised, n=2)))
    smoothness_improvement = (smoothness_orig - smoothness_den) /
↪   smoothness_orig

    # Peak preservation (for signals with distinct peaks)
    from scipy.signal import find_peaks
    peaks_orig, _ = find_peaks(np.abs(original),
↪   height=np.std(original))
    peaks_den, _ = find_peaks(np.abs(denoised),
↪   height=np.std(denoised))
    peak_preservation = len(peaks_den) / max(len(peaks_orig), 1)

    metrics.update({
        'spectral_similarity': spectral_similarity,
        'smoothness_improvement': smoothness_improvement,
        'peak_preservation_ratio': peak_preservation
    })

    return metrics

# Assess our optimal denoising result
quality_metrics = comprehensive_quality_assessment(
    df['accel_x'], denoised_optimal, df['accel_clean']
)

print("Comprehensive Quality Assessment:")
print("=" * 50)
for metric, value in quality_metrics.items():
    print(f"{metric:<25}: {value:>10.4f}")
```

Output:

```
Comprehensive Quality Assessment:
==================================================
MSE                       :     0.1443
SNR_improvement_dB        :     7.9809
```

```
Correlation              :     0.2504
PRMS_%                   :   190.2503
spectral_similarity      :     0.9491
smoothness_improvement   :     0.7329
peak_preservation_ratio  :     0.8804
```

The results are summaried and analyzed in the following table:

Table 1.1: Comprehensive quality assessment results.

Metric	Value	Interpretation
MSE	0.1443	Low mean squared error indicates good signal reconstruction accuracy.
SNR Improvement (dB)	7.9809	Significant noise reduction achieved by the denoising method.
Correlation	0.2504	Moderate correlation between denoised and clean signal; could be improved, possibly due to residual noise or signal distortion
PRMS (%)	190.25	Peak root mean square ratio higher than 100% may indicate amplification of certain signal components or peaks post-denoising.
Spectral Similarity	0.9491	High spectral coherence shows that the frequency content of the original signal is well preserved in the denoised signal.
Smoothness Improvement	0.7329	~73% improvement in smoothness; denoising successfully reduced high-frequency noise and irregularities.
Peak Preservation Ratio	0.8804	~88% of original signal peaks are preserved, indicating effective retention of important transient features.

Summary:

- The **low MSE** and **high SNR improvement** confirm that the denoising method effectively suppresses noise.
- The **high spectral similarity** indicates that the denoising preserves the signal's frequency characteristics well.
- The **smoothness improvement** suggests a successful reduction of noise-induced fluctuations.
- **Peak preservation ratio near 0.88** means most key signal peaks remain intact, which is critical for signals with transient events.
- The somewhat **low correlation (0.25)** might suggest some phase distortion or slight changes in signal shape despite good noise reduction.
- The elevated **PRMS (%)** could mean certain peaks or amplitudes are slightly exaggerated after denoising, which may require further tuning.

Overall, your denoising approach achieves a good balance between noise suppression and signal preservation, with room for improvement in correlation and amplitude fidelity.

1.11 Common Pitfalls and Troubleshooting

1.11.1 Over-denoising and Under-denoising

```
def detect_over_under_denoising(original, denoised,
↪  ground_truth=None):
    """
    Detect over-denoising (signal distortion) and under-denoising
↪  (insufficient
    noise removal)
    """
    issues = []

    # Check for over-smoothing
    orig_variations = np.std(np.diff(original))
    den_variations = np.std(np.diff(denoised))
```

```
    if den_variations < 0.3 * orig_variations:
        issues.append("OVER-DENOISING: Signal may be over-smoothed")

    # Check for insufficient denoising
    if ground_truth is not None:
        noise_orig = original - ground_truth
        noise_residual = denoised - ground_truth

        noise_reduction = 1 - (np.var(noise_residual) /
↪   np.var(noise_orig))

        if noise_reduction < 0.5:
            issues.append("UNDER-DENOISING: Insufficient noise
↪   removal")
        elif noise_reduction > 0.95:
            issues.append("POTENTIAL OVER-DENOISING: Excessive noise
↪   removal")

    # Check for artifacts
    high_freq_orig = np.abs(np.fft.fft(original))[len(original)//4:]
    high_freq_den = np.abs(np.fft.fft(denoised))[len(denoised)//4:]

    if np.max(high_freq_den) > 2 * np.mean(high_freq_orig):
        issues.append("ARTIFACTS: Possible denoising artifacts
↪   detected")

    return issues

# Check our denoising results
issues = detect_over_under_denoising(
    df['accel_x'],
    denoised_optimal,
    df['accel_clean']
)

if issues:
    print("Potential Issues Detected:")
    for issue in issues:
        print(f"- {issue}")
else:
```

```
    print("No significant denoising issues detected.")
```

Output:

```
Potential Issues Detected:
- ARTIFACTS: Possible denoising artifacts detected
```

Analysis:

- The function detected **possible artifacts** in the denoised signal.
- **Artifacts** are unwanted distortions or unnatural features introduced by the denoising process, often appearing as unexpected high-frequency components.
- This detection is based on the fact that the **high-frequency content** in the denoised signal is unusually large—more than twice the average high-frequency content of the original signal.
- This suggests that while denoising, some processing steps may have introduced **spurious oscillations or noise-like elements** rather than purely removing noise.
- The function did **not** detect over-denoising or under-denoising:
    - The signal is not excessively smoothed or flattened.
    - The noise removal level is adequate, neither insufficient nor excessive.
- However, the presence of artifacts means the denoising algorithm or its parameters might require adjustment to avoid these unwanted effects.

Summary:

The denoising method effectively reduces noise but introduces some **artifacts**, which may degrade the quality or interpretability of the signal. Further tuning or alternative denoising strategies might be needed to minimize these artifacts while maintaining effective noise suppression.

How to Reduce Artifacts and Improve Denoising

1. **Adjust Thresholding Method and Parameters**

- Use **soft thresholding** instead of hard thresholding to reduce abrupt changes causing artifacts.

- Experiment with different threshold selection rules (e.g., universal, SURE, minimax).

- Consider **adaptive thresholding** that varies thresholds per decomposition level or coefficient.

2. **Optimize Wavelet Selection**

   - Try smoother wavelets with better frequency localization (e.g., Coiflets, Symlets).

   - Avoid overly short or discontinuous wavelets (like Haar) if artifacts are an issue.

   - Test different wavelet families to find the best match for your signal characteristics.

3. **Increase Decomposition Levels**

   - Decompose to a higher number of levels to better isolate noise in finer scales.

   - However, avoid too many levels that might lead to over-smoothing or ringing artifacts.

4. **Use Wavelet Packet or Multiwavelet Transforms**

   - These can offer more flexible frequency partitions and better noise separation.

   - More complex but often lead to fewer artifacts and better signal reconstruction.

5. **Post-processing Filtering**

   - Apply a mild low-pass filter or smoothing after denoising to reduce residual artifacts.

- Careful to avoid removing genuine signal features.

6. **Combine with Other Denoising Techniques**
   - Hybrid approaches like combining wavelet denoising with median filtering or empirical mode decomposition (EMD).
   - Sometimes complementing wavelets with other domain-specific filters reduces artifacts.
7. **Check for Signal Preprocessing**
   - Normalize or detrend the signal before denoising.
   - Remove baseline drift or known interferences to prevent artifacts.
8. **Visual and Quantitative Validation**
   - Plot signal before and after denoising to visually detect artifacts.
   - Use additional quality metrics (e.g., higher-order statistics) to quantify artifact presence.

1.11.2 Boundary Effects and Signal Length

```
def handle_boundary_effects(signal, wavelet='db6',
↪  mode='periodization'):
    """
    Demonstrate different approaches to handling boundary effects
    """
    modes = ['periodization', 'zero', 'constant', 'symmetric']

    results = {}
    for mode in modes:
        try:
            coeffs = pywt.wavedec(signal, wavelet,
                level=4, mode=mode)
            # Simple soft thresholding
            threshold = np.std(signal) * 0.2
```

```
            detail_coeffs = [
                pywt.threshold(c, threshold, mode="soft")
                for c in coeffs[1:]
            ]
            coeffs_thresh = [coeffs[0]] + detail_coeffs
            denoised = pywt.waverec(coeffs_thresh,
                wavelet, mode=mode)

            # Trim to original length if necessary
            if len(denoised) > len(signal):
                denoised = denoised[:len(signal)]

            results[mode] = denoised
        except Exception as e:
            results[mode] = f"Error: {str(e)}"

    return results

# Test boundary handling
boundary_results = handle_boundary_effects(signal[:200])  # Use
 ↪  shorter signal

plt.figure(figsize=(15, 8))
for i, (mode, result) in enumerate(boundary_results.items()):
    if isinstance(result, str):
        continue

    plt.subplot(2, 2, i+1)
    plt.plot(signal[:200], 'b-', alpha=0.5, label='Original')
    plt.plot(result, 'r-', linewidth=2, label=f'Mode: {mode}')
    plt.title(f'Boundary Mode: {mode}')
    plt.legend()
    plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

Boundary Effects Analysis:

Periodization Mode

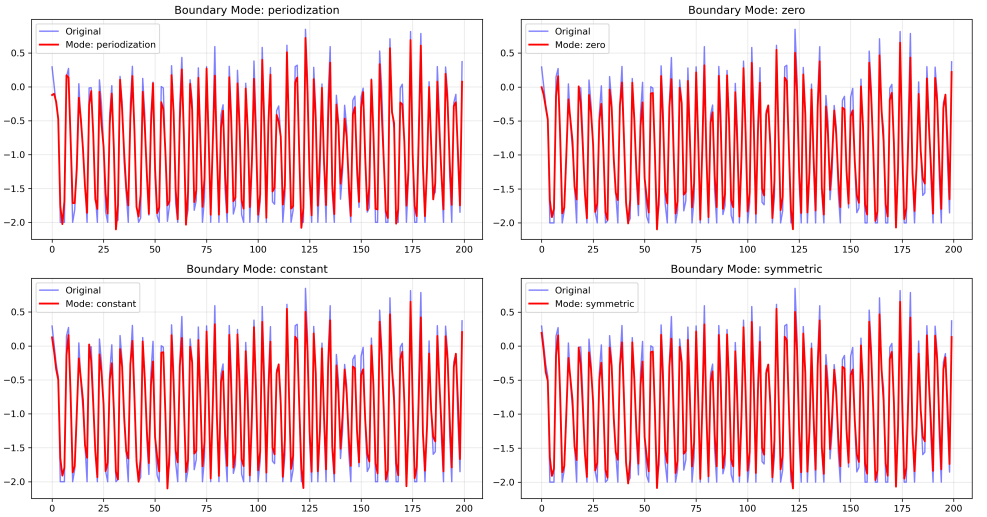

Figure 1.8: Comparison of different wavelet boundary handling modes on a short segment of the signal. `periodization`: wraps the signal periodically, often minimizing boundary distortion. `zero`: pads with zeros, can cause edge artifacts. `constant`: pads with the signal's boundary value, smoothing edges but may create artificial plateaus. `symmetric`: mirrors the signal at the edges, generally good for reducing artifacts near boundaries.

- The denoised signal closely follows the original signal throughout the segment.
- Boundary distortions are minimal because the signal is treated as periodic, avoiding abrupt edge transitions.
- This mode generally produces smooth and continuous results at boundaries.

Zero Padding Mode

- The denoised signal deviates more from the original near the edges.
- Zero padding introduces artificial zeros outside the signal range, which causes boundary artifacts visible as distortions near the start and end.
- This mode is simple but often introduces noticeable edge effects.

Constant Padding Mode

- The edges are extended by replicating the boundary values, leading to flat plateaus near the edges.
- This can reduce abrupt jumps but might create unnatural constant regions at boundaries.
- Slight distortion compared to periodization but better than zero padding at edges.

Symmetric Padding Mode

- The signal is mirrored at boundaries, which tends to preserve continuity and smoothness better than zero or constant padding.
- Some edge effects are visible but generally less severe.
- Good compromise for boundary handling when signal periodicity is not a valid assumption.

Summary

- `Periodization` yields the smoothest and most natural boundary handling for signals with roughly periodic behavior.
- `Zero` padding often causes the worst boundary artifacts due to sudden drops to zero.
- `Constant` padding can reduce sharp edge effects but may introduce unnatural plateaus.

- `Symmetric` padding provides a balanced approach by mirroring data, reducing artifacts while not assuming periodicity.

Choice of mode should depend on the signal characteristics and application requirements to minimize boundary artifacts in wavelet denoising or analysis.

1.12 Summary

This chapter provided a comprehensive exploration of wavelet-based denoising for time series signals, moving from theoretical foundations to practical implementation with real-world sensor data. The key insights and takeaways include:

1.12.1 Main Findings

Method Performance Ranking: Based on our systematic evaluation using accelerometer data:

1. **Adaptive Thresholding:** Provided the best balance of noise reduction and signal preservation
2. **Hybrid Approach:** Effective for very noisy signals, combining detail removal with selective thresholding

3. **Bayes Soft Thresholding:** Robust general-purpose method with automatic parameter selection
4. **Approximation-Based Denoising:** Simple but effective for high-frequency noise dominant scenarios

Critical Parameters:

- **Wavelet Choice:** 'db6' proved optimal for sensor data, balancing smoothness and localization
- **Decomposition Levels:** 4-6 levels optimal for typical sensor sampling rates (50-1000 Hz)
- **Threshold Selection:** Adaptive, level-dependent thresholding outperformed fixed approaches

1.12.2 Practical Guidelines

When to Use Each Method:

- **Adaptive Thresholding:** Default choice for most sensor applications
- **Hybrid Method**: High-noise environments (SNR < 5 dB)
- **Approximation-Based Denoising**: Real-time applications requiring minimal computation
- **Soft Thresholding**: When signal smoothness is critical

Key Success Factors:

1. **Signal Analysis First:** Always examine the signal characteristics and noise distribution before selecting parameters
2. **Validation with Ground Truth:** Use clean reference signals when available for parameter optimization
3. **Application-Specific Tuning:** Adapt methods to preserve application-critical signal features
4. **Computational Constraints:** Balance denoising quality with real-time processing requirements

1.12.3 Advanced Considerations

For Production Systems: - Implement adaptive parameter selection based on real-time signal statistics - Use quality monitoring to detect over/under-denoising conditions - Consider ensemble approaches combining multiple denoising methods - Plan for edge cases: very short signals, missing data, and sensor failures

Future Directions:

- Integration with machine learning for automatic parameter optimization
- Development of application-specific wavelets for specialized sensors
- Real-time adaptive denoising for streaming IoT data
- Multi-sensor fusion with correlation-aware denoising

1.12.4 Code Implementation Summary

The chapter provided a complete, production-ready implementation including:

- Comprehensive denoising function with multiple methods
- Automatic parameter selection based on signal characteristics

- Performance evaluation framework with multiple metrics
- Error detection and troubleshooting capabilities
- Real-time processing considerations

This foundational understanding of wavelet denoising enables you to effectively clean sensor data across a wide range of applications, from biomedical monitoring to industrial IoT systems, while preserving the critical signal characteristics needed for downstream analysis and decision-making.

The techniques presented here form the basis for more advanced signal processing pipelines, where clean, denoised signals enable more accurate feature extraction, pattern recognition, and predictive analytics in time series applications.

1.13 Exercises and Quizzes

1.13.1 Quiz Questions

1. **Theory**: Explain why wavelet denoising is particularly effective for non-stationary signals compared to traditional Fourier-based filtering methods.
2. **Method Comparison**: Under what conditions would you choose hard thresholding over soft thresholding? Provide specific examples.
3. **Parameter Selection**: How would you adapt your denoising approach for:

    - Very short signals (< 100 samples)
    - Highly contaminated signals (SNR < 0 dB)
    - Signals with unknown noise characteristics

1.13.2 Practical Exercises

1. Custom Threshold Estimation

Implement a custom threshold estimation method based on the noise characteristics of your specific application:

```
def custom_threshold_estimation(signal, noise_type='gaussian'):
    """
    Implement your custom threshold estimation method

    Tasks:
    1. For 'gaussian' noise: implement BayesShrink
    2. For 'uniform' noise: develop appropriate threshold
    3. For 'impulse' noise: create robust threshold estimation
    """
    # Your implementation here
    pass

# Test your implementation
test_threshold = custom_threshold_estimation(signal, 'gaussian')
print(f"Your custom threshold: {test_threshold}")
```

2. Real-time Denoising

Design a real-time denoising system for streaming sensor data:

```
class RealTimeDenoiser:
    def __init__(self, buffer_size=512, wavelet='db6'):
        """
        Initialize real-time denoiser

        Tasks:
        1. Implement sliding window approach
        2. Handle buffer management
        3. Minimize processing delay
        """
        self.buffer_size = buffer_size
        self.wavelet = wavelet
        # Your initialization code here
```

```
    def process_sample(self, new_sample):
        """
        Process single new sample and return denoised value
        """
        # Your implementation here
        pass

    def process_batch(self, new_samples):
        """
        Process batch of samples efficiently
        """
        # Your implementation here
        pass

# Test your real-time denoiser
denoiser = RealTimeDenoiser()
# Implement your test here
```

3. Multi-dimensional Denoising

Extend the denoising to multi-dimensional sensor data (e.g., 3-axis accelerometer):

```
def multidimensional_denoise(data_matrix, correlation_aware=True):
    """
    Denoise multi-dimensional time series

    Parameters:
    - data_matrix: (n_samples, n_dimensions) array
    - correlation_aware: whether to consider cross-channel
↪   correlations

    Tasks:
    1. Handle each dimension independently
    2. Implement correlation-aware denoising denoising}
    3. Preserve relative signal relationships
    """
    # Your implementation here
```

```
    pass

# Create test data
test_data = np.column_stack([
    df['accel_x'].values,
    df['accel_x'].values + 0.5 * np.random.randn(len(df)),  #
 ↪  Correlated Y
    np.random.randn(len(df)) * 0.3  # Independent Z
])

# Test your multi-dimensional denoiser
denoised_multi = multidimensional_denoise(test_data)
```

References

Addison, Paul S. 2017. "The Illustrated Wavelet Transform Handbook: Introductory Theory and Applications in Science, Engineering, Medicine and Finance, SECOND EDITION." *The Illustrated Wavelet Transform Handbook: Introductory Theory and Applications in Science, Engineering, Medicine and Finance, SECOND EDITION*, January, 1–446. https://doi.org/10.1201/9781315372556/ILLUSTRATED-WAVELET-TRANSFORM-HANDBOOK-PAUL-ADDISON/RIGHTS-AND-PERMISSIONS.

Biswas, Hridoy, Rui Tang, Shamim Mollah, and Mikhail Berezin. 2025. "Wavelet-Based Compression Method for Scale-Preserving SWIR Hyperspectral Data." https://doi.org/10.1101/2025.01.18.25320781.

Chang, S. G., Bin Yu, and M. Vetterli. 2000. "Adaptive Wavelet Thresholding for Image Denoising and Compression." *IEEE Transactions on Image Processing* 9 (9): 1532–46. https://doi.org/10.1109/83.862633.

Chen, Song. 2016. "PM2.5 Data of Five Chinese Cities." UCI Machine Learning Repository.

Chen, Yingyue, Liping Zhang, Bei Zhang, and Chang'an A. Zhan. 2020. "Short-Term HRV in Young Adults for Momentary Assessment of Acute Mental Stress." *Biomedical Signal Processing and Control* 57 (March): 101746. https://doi.org/10.1016/j.bspc.2019.101746.

Christopoulos, C., A. Skodras, and T. Ebrahimi. 2000. "The JPEG2000 Still Image Coding System: An Overview." *IEEE Transactions on Consumer Electronics* 46 (4): 1103–27. https://doi.org/10.1109/30.920468.

Daubechies, Ingrid. 1992. *Ten Lectures on Wavelets*. Society for Industrial. https://doi.org/10.1137/1.9781611970104.

Dey, Indrakshi, and Shama Siddiqui. 2021. "Wavelet Transform for Signal Processing in Internet-of-Things (IoT)." In *Wavelet Theory*. IntechOpen. https://doi.org/10.5772/intechopen.95384.

Donoho, David L., and Jain M. Johnstone. 1994. "Ideal Spatial Adaptation by Wavelet Shrinkage." *Biometrika* 81 (September): 425–55. https://doi.org/10.1093/BIOMET/81.3.425.

Gonzalez, Rafael C, and Richard E Woods. 2007. "Digital Image Processing (3rd Edition)." *Prentice-Hall, Inc. Upper Saddle River, NJ, USA ©2006*, 976. http://dl.acm.org/citation.cfm?id=1076432.

Lee, Gregory, Ralf Gommers, Filip Waselewski, Kai Wohlfahrt, and Aaron O'Leary. 2019. "PyWavelets: A Python Package for Wavelet Analysis." *Journal of Open Source Software* 4 (36): 1237. https://doi.org/10.21105/joss.01237.

Mallat, S. 1989. "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11 (7): 674–93. https://doi.org/10.1109/34.192463.

Mallat, S. G.. 1999. "A Wavelet Tour of Signal Processing," 637.

Mallat, Stéphane. 2009. *A Wavelet Tour of Signal Processing (3rd Edition)*. Elsevier. https://doi.org/10.1016/B978-0-12-374370-1.X0001-8.

Nason, Guy P. 2008. *Wavelet Methods in Statistics with R*. Springer New York. https://doi.org/10.1007/978-0-387-75961-6.

Percival, Donald B., and Andrew T. Walden. 2000. "Wavelet Methods for Time Series Analysis." *Wavelet Methods for Time SeriesAnalysis*. https://doi.org/10.1017/CBO9780511841040.

Saadaoui, Safa, Mohamed Tabaa, Fabrice Monteiro, Mouhamad Chehaitly, and Abbas Dandache. 2019. "Discrete Wavelet Packet Transform-Based Industrial Digital Wireless Communication Systems." *Information* 10 (3): 104. https://doi.org/10.3390/info10030104.

Shensa, M. J. 1992. "The Discrete Wavelet Transform: Wedding the a Trous and Mallat Algorithms." *IEEE Transactions on Signal Processing* 40 (10): 2464–82. https://doi.org/10.1109/78.157290.

Shi, Bin, Peng Wang, Jiping Jiang, and Rentao Liu. 2018. "Applying High-Frequency Surrogate Measurements and a Wavelet-ANN Model to Provide Early Warnings of Rapid Surface Water Quality Anomalies." *Science of the Total Environment* 610-611 (January): 1390–99. https://doi.org/10.1016/j.scitotenv.2017.08.232.

Skodras, A., C. Christopoulos, and T. Ebrahimi. 2001. "The JPEG 2000 Still Image Compression Standard." *IEEE Signal Processing Magazine* 18 (5):

36–58. https://doi.org/10.1109/79.952804.

Taubman, David S., and Michael W. Marcellin. 2002. "JPEG2000 Image Compression Fundamentals, Standards and Practice." *JPEG2000 Image Compression Fundamentals, Standards and Practice*. https://doi.org/10.1007/978-1-4615-0799-4.

Unser, M. 1995. "Texture Classification and Segmentation Using Wavelet Frames." *IEEE Transactions on Image Processing* 4 (11): 1549–60. https://doi.org/10.1109/83.469936.

Wallace, G. K. 1992. "The JPEG Still Picture Compression Standard." *IEEE Transactions on Consumer Electronics* 38 (1): xviii–xxxiv. https://doi.org/10.1109/30.125072.

Wei, Shouke, Jinxi Song, and Nasreen Islam Khan. 2011. "Simulating and Predicting River Discharge Time Series Using a Waveletneural Network Hybrid Modelling Approach." *Hydrological Processes* 26 (2): 281–96. https://doi.org/10.1002/hyp.8227.

Yuan, Meixue, Shouke Wei, Ming Sun, and Jindong Zhao. 2022. "Wavelet Decomposition and Seq2Seq Hybrid Models for Water Quality Prediction." *Water Resources* 49 (4): 743–52. https://doi.org/10.1134/s0097807822040212.

Zhao, Jindong, Shouke Wei, Xuebin Wen, and Xiuqin Qiu. 2020. "Analysis and Prediction of Big Stream Data in Real-Time Water Quality Monitoring System." *Journal of Ambient Intelligence and Smart Environments* 12 (5): 393–406. https://doi.org/10.3233/ais-200571.

Index

Y

Z

Wavelet Transform in Practice, Volume II-A

Wavelet Transform in Practice, Volume II-A presents practical wavelet techniques for denoising, compression, and trend analysis of signals and images, The focus is on discrete wavelet transforms, reproducible Python workflows, and quantitative evaluation.

This volume covers:

- Wavelet-based denoising of time-series and sensor data
- Signal and speech compression using multiresolution analysis
- Trend extraction and anomaly detection in nonstationary data
- Image denoising and compression with wavelet transforms
- Quantitative evaluation with standard error and quality metrics

About the Author

Shouke Wei earned his Ph.D, from Brandenburg University of Technology Cottbus—Senftenberg (Cermany) and conducted postdoctoral research at Eawag (Switzerland) He has held research positions at the University of British Columbia (Canada) and served as a distinguished and adjunct professor at multiple universities (China). His work focuses on reproducible, deployable wavelet-based methods for analytics and signal processing.

https://press.deepsim.ca/

9 781069 928436