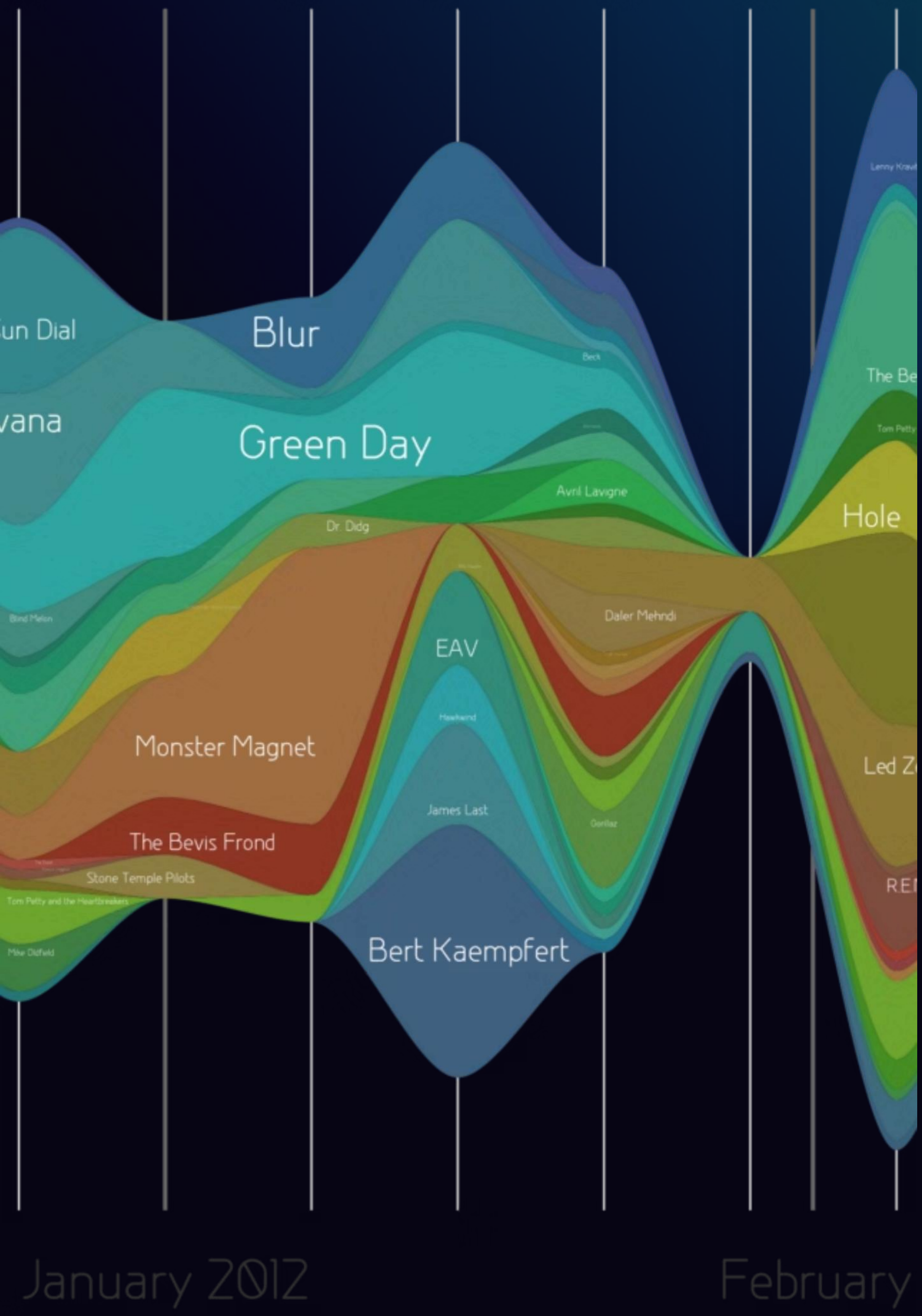


Time Series Analysis

Advanced mathematical techniques for analyzing temporal data in physics and computational sciences

Time Series Analysis for Physics & Computer Science

Week 3 – Master's Course: Data Science and Machine LEarning

Instructor: Hernán Andrés Morales-Navarrete

This comprehensive course module explores the mathematical foundations and practical applications of time series analysis in physics and computer science research environments.

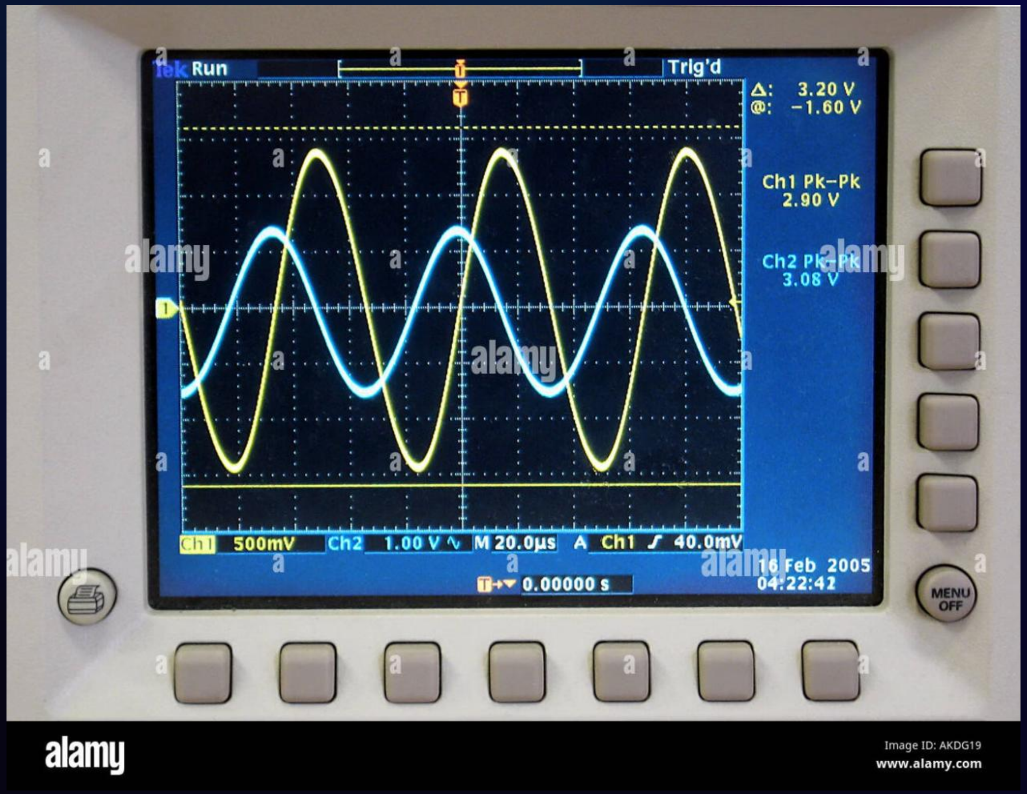
What is a Time Series?

A time series represents a sequence of data points collected and ordered chronologically, where each observation is indexed by time. This fundamental concept forms the backbone of temporal data analysis across scientific disciplines.

Common Physics Applications:

- **Mechanical oscillations:** Pendulum displacement, spring-mass systems
- **Electrical circuits:** LC oscillator voltage and current measurements
- **Environmental monitoring:** Temperature, pressure, and humidity readings
- **Spectroscopy:** Intensity measurements over wavelength or frequency

The primary goal is to **describe underlying patterns, analyze system behavior, and predict future dynamics** based on historical observations.



Why Study Time Series Analysis?

Extract Hidden Patterns

Discover underlying **periodicity, memory effects, and temporal correlations** that aren't immediately visible in raw data. These patterns often reveal fundamental physical processes.

Predictive Modeling

Develop mathematical models to **forecast future system states** based on historical behavior, essential for experimental planning and theoretical validation.

Signal Enhancement

Separate meaningful signals from background noise, improving data quality and enabling more accurate measurements in experimental physics.

Model Validation

Connect experimental data with theoretical models, testing hypotheses and verifying physical laws through quantitative analysis.

Big Data Processing

Handle **large experimental datasets efficiently**, essential for modern physics research involving particle accelerators, astronomical observations, and quantum experiments.

Autocorrelation – Mathematical Foundation

Autocorrelation measures the **linear relationship between a time series and a time-shifted version of itself**. This powerful statistical tool reveals how past values influence current observations.

Mathematical Definition:

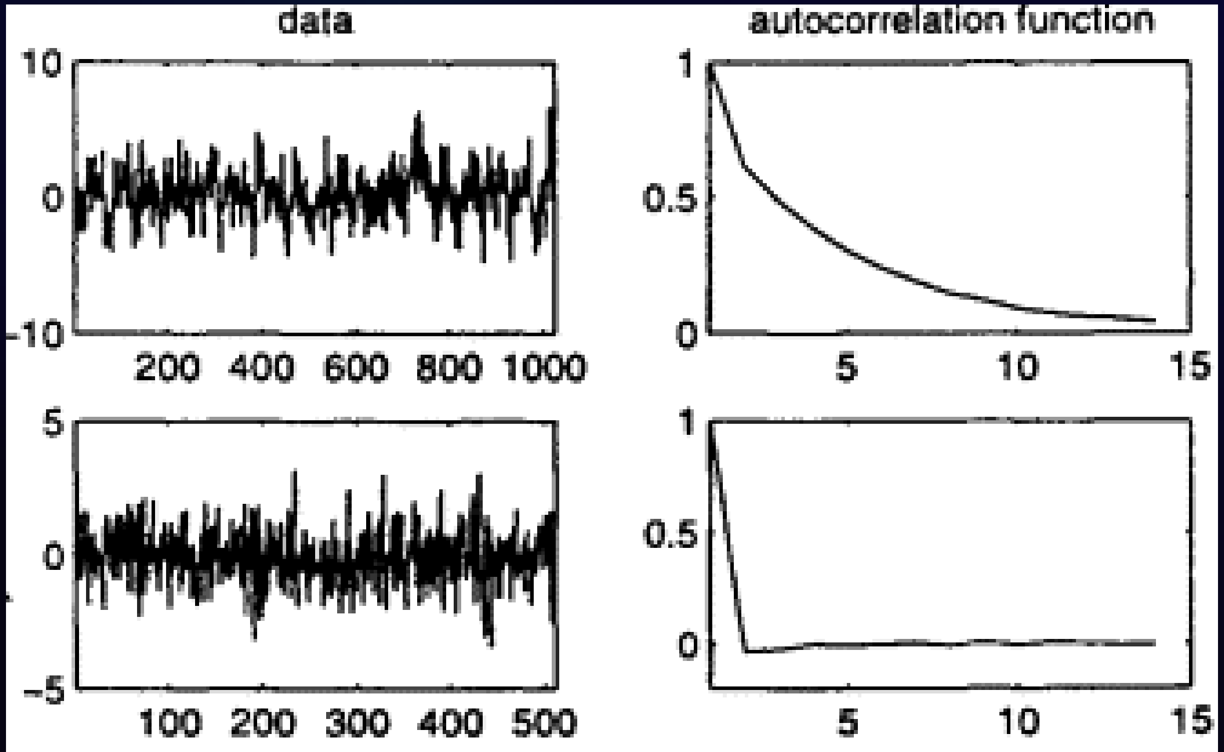
$$R(\tau) = \frac{1}{N-\tau} \sum_{t=1}^{N-\tau} (x_t - \bar{x})(x_{t+\tau} - \bar{x})$$

Where τr epresents the time lag, N is the series length, and $\bar{X}$ is the sample mean.

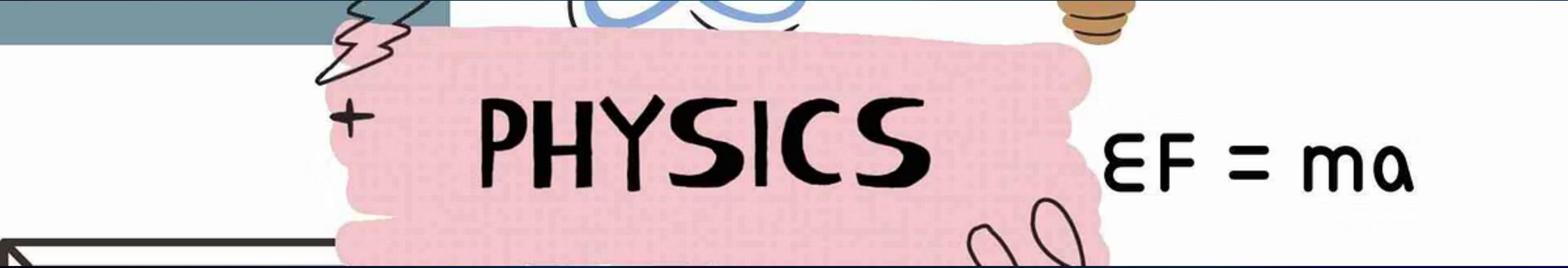
Key Applications:

- Detecting **periodic behaviors** in oscillatory systems
- Measuring **memory effects** and temporal dependencies
- Assessing **noise characteristics** and signal quality

A mathematical representation of the degree of similarity between a given time series and a lagged version of itself over successive time intervals.



In-deep lecture:
https://www.sciencedirect.com/topics/engineering/autocorrelation-function



Autocorrelation in Physics

Velocity Autocorrelation

In molecular dynamics simulations, the **velocity autocorrelation function directly relates to the diffusion coefficient** through the Green-Kubo relation, providing insights into transport properties.

Radar and Sonar Systems

Echo analysis using autocorrelation enables precise distance measurements and object detection by comparing transmitted and received signal patterns.

Oscillatory System Analysis

Identify **characteristic frequencies and damping coefficients** in mechanical oscillators, electrical circuits, and quantum harmonic systems.

Experimental Signal Processing

Extract **repeated patterns from noisy experimental data**, improving signal-to-noise ratio and revealing underlying physical phenomena.

Python Implementation: Autocorrelation

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

# Generate sample signal: damped oscillation
t = np.linspace(0, 10, 1000)
x = np.exp(-0.1*t) * np.cos(2*np.pi*2*t)

# Calculate autocorrelation
x_centered = x - np.mean(x)
acf = np.correlate(x_centered, x_centered, mode='full')
acf = acf[acf.size//2:] / np.max(acf)

# Plot results
plt.subplot(2,1,1)
plt.plot(t, x, label='Original Signal')
plt.legend()

plt.subplot(2,1,2)
plt.plot(t, acf, label='Autocorrelation', color='red')
plt.xlabel('Time Lag')
plt.legend()
plt.show()
```

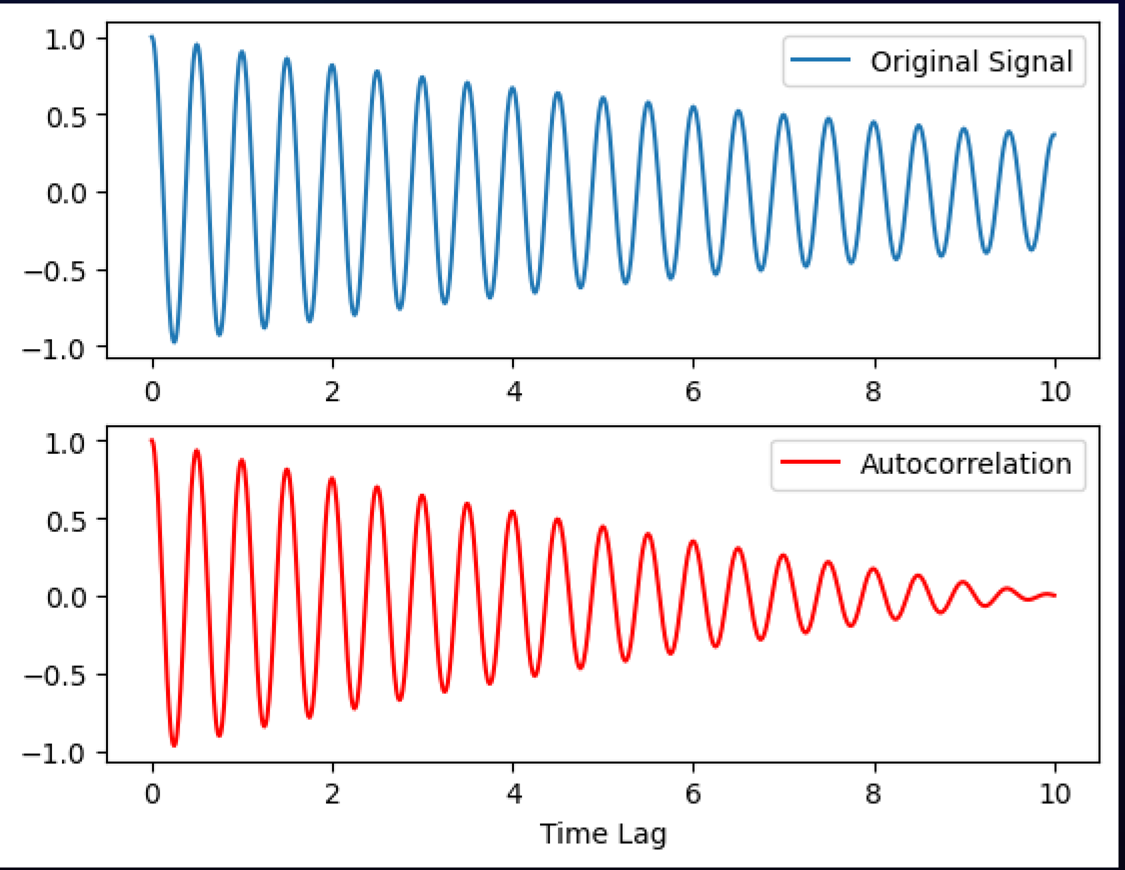
$$x(t) = e^{-0.1t} \cos(2\pi f t)$$

This code demonstrates the calculation of autocorrelation for a **damped harmonic oscillator**, a fundamental system in physics.

Key Features:

- Signal centering removes DC component
- Full correlation provides complete lag information
- Normalization enables comparison between signals

The autocorrelation reveals both the **oscillation frequency and damping characteristics** of the system.



ARIMA Models: Comprehensive Framework

AutoRegressive (AR)

Current values depend on past values with linear relationships. Models systems with memory effects where previous states influence current behavior.

Integrated (I)

Differencing removes trends and achieves stationarity. Essential for handling data with systematic drift or long-term evolution.

Moving Average (MA)

Current values depend on past residual errors. Captures short-term fluctuations and noise characteristics in the system.

General ARIMA(p,d,q) Mathematical Form:

$$\Delta^d x_t = c + \sum_{i=1}^{p} \phi_i \Delta^d x_{t-i} + \varepsilon_t + \sum_{j=1}^{q} \theta_j \varepsilon_{t-j}$$

Where p is the AR order, d is the degree of differencing, q is the MA order, and ϵ_t represents white noise innovations.

ARIMA Applications in Physics

Accelerator Physics

Predict beam intensity fluctuations in particle accelerators, enabling proactive adjustments to maintain stable experimental conditions.

Noise Modeling

Model and filter irregular experimental signals, separating systematic variations from random measurement uncertainties.

Forecasting

Forecast experimental outcomes based on historical patterns, optimizing measurement strategies and resource allocation.

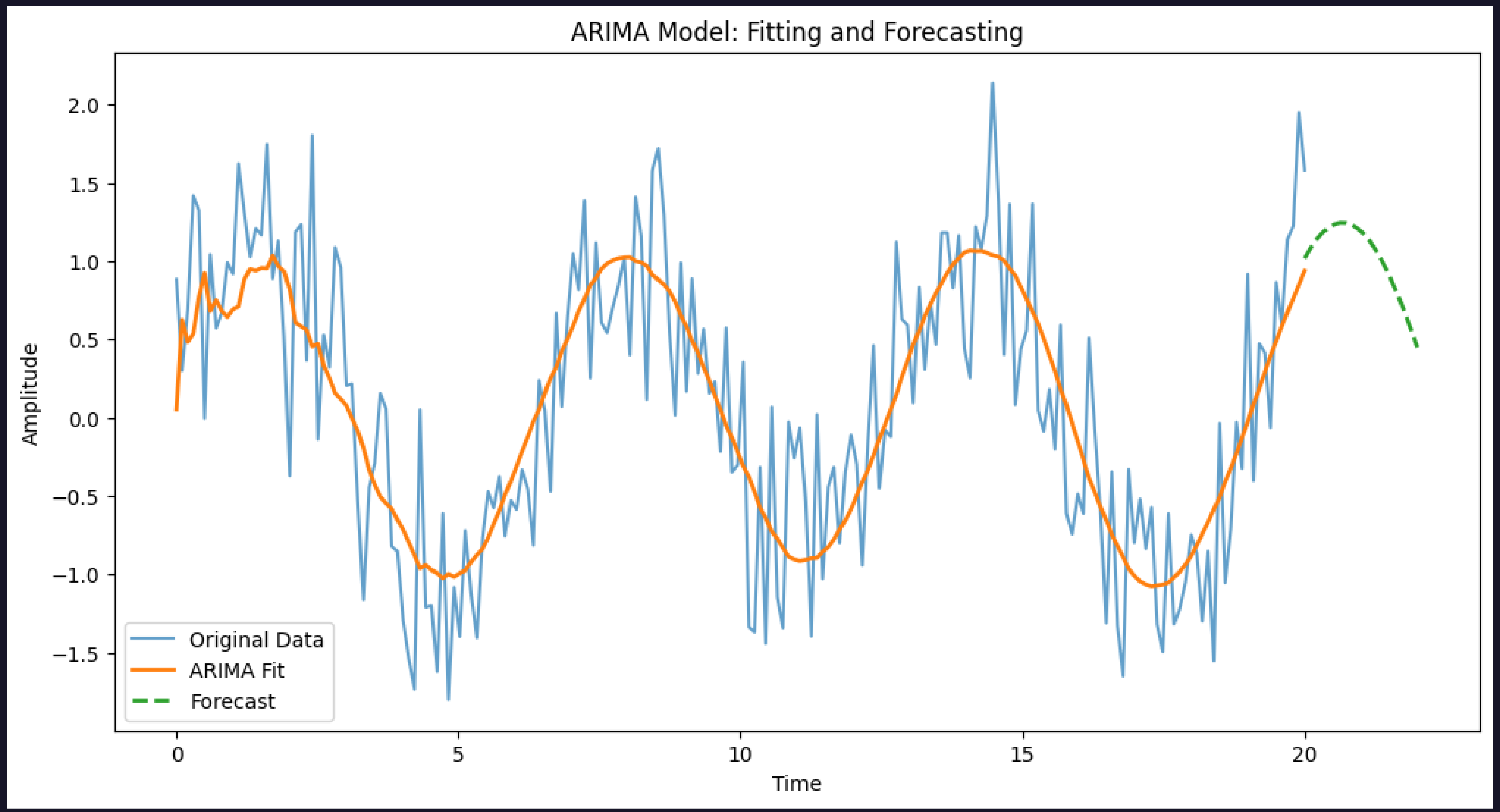
Data Preprocessing

Prepare raw data for advanced analysis by removing trends and modeling temporal dependencies before applying other techniques.

Python Implementation: ARIMA

This implementation demonstrates ARIMA modeling on **noisy sinusoidal data**, typical of experimental measurements with both systematic oscillations and random noise.

```
import statsmodels.api as sm
import numpy as np
import matplotlib.pyplot as plt

# Generate sample noisy sinusoidal data
np.random.seed(0)
t = np.linspace(0, 20, 200)
x = np.sin(t) + 0.5*np.random.randn(200)

# Fit ARIMA model
# Order (2,0,2): AR(2) + MA(2), no differencing needed
model = sm.tsa.ARIMA(x, order=(2, 0, 2))
result = model.fit()

# Generate predictions
predictions = result.fittedvalues
forecast = result.forecast(steps=20)

# Visualization
plt.figure(figsize=(12, 6))
plt.plot(t, x, label='Original Data', alpha=0.7)
plt.plot(t, predictions, label='ARIMA Fit', linewidth=2)
plt.plot(np.linspace(20, 22, 20), forecast,
         label='Forecast', linestyle='--', linewidth=2)
plt.legend()
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('ARIMA Model: Fitting and Forecasting')
plt.show()

# Display model summary
print(result.summary())
```



Fourier Analysis: Frequency Domain Perspective

Fourier analysis decomposes complex time-domain signals into their constituent **sine and cosine frequency components**. This transformation reveals the spectral content of temporal data.

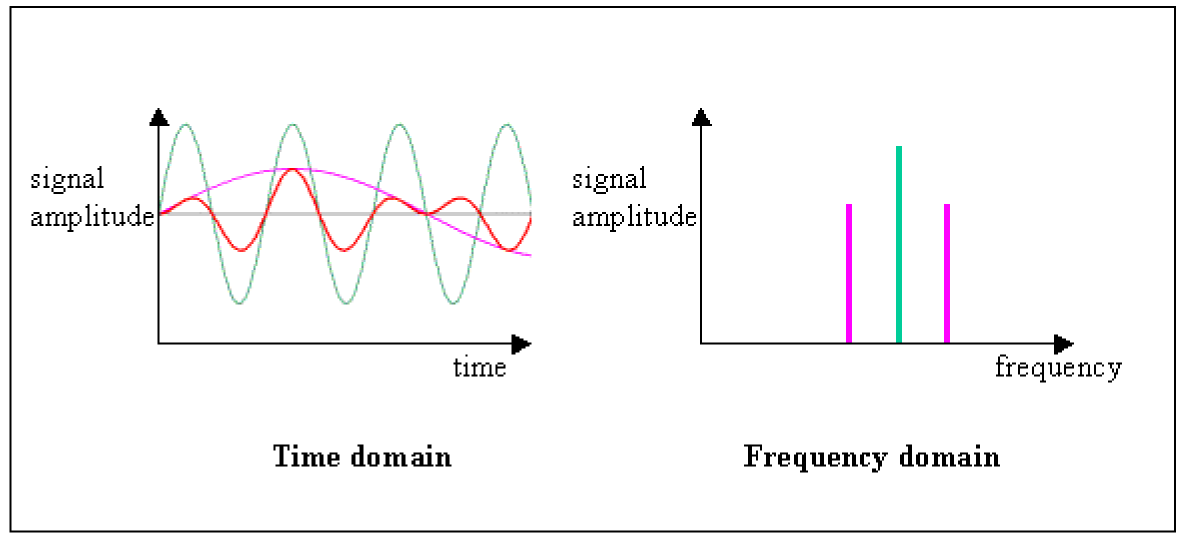
Continuous Fourier Transform:

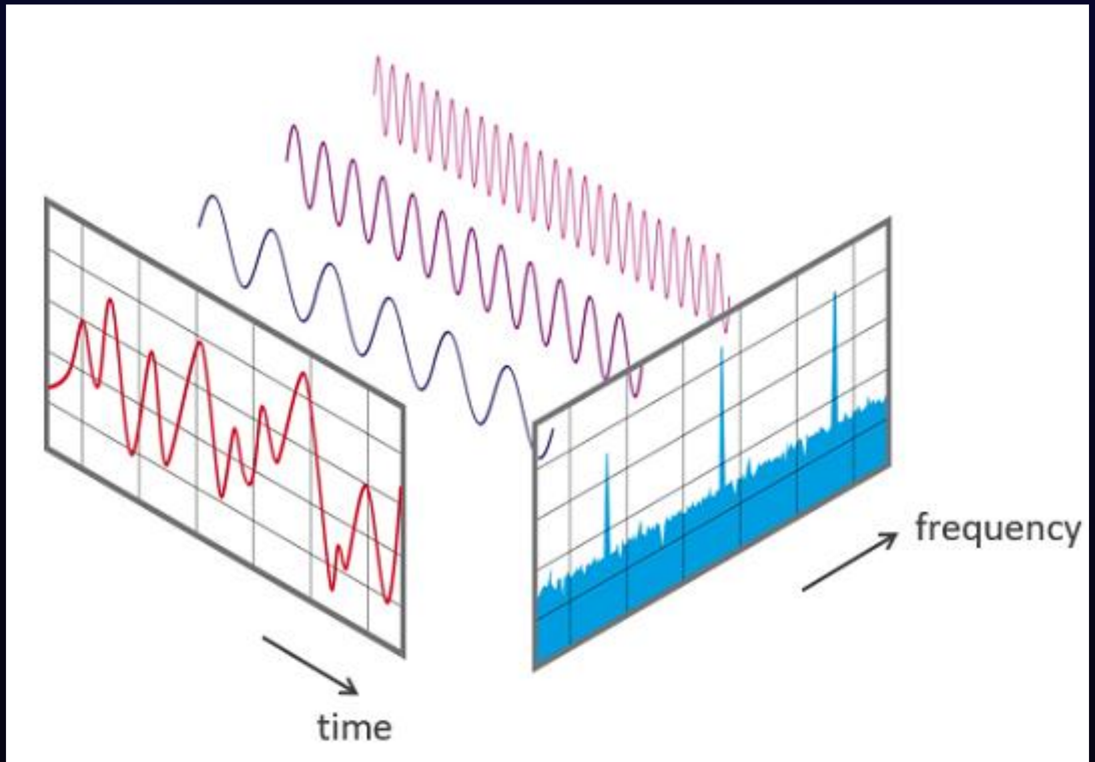
$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-2\pi i f t} dt$$

Key Properties:

- **Linearity:** Superposition principle applies
- **Time-frequency duality:** Narrow pulses have broad spectra
- **Energy conservation:** Parseval's theorem
- **Phase information:** Complete signal reconstruction possible

The Fourier transform is **fundamental to understanding wave phenomena, resonance, and spectroscopy** across all branches of physics.





Fourier Analysis in Physics

Resonance Analysis

Identify **natural frequencies in mechanical and electrical systems**. Critical for understanding structural vibrations, circuit behavior, and quantum energy levels.

Noise Characterization

Analyze **power spectral density of thermal and quantum noise**. Essential for optimizing measurement sensitivity and understanding fundamental limits.

Wave Physics

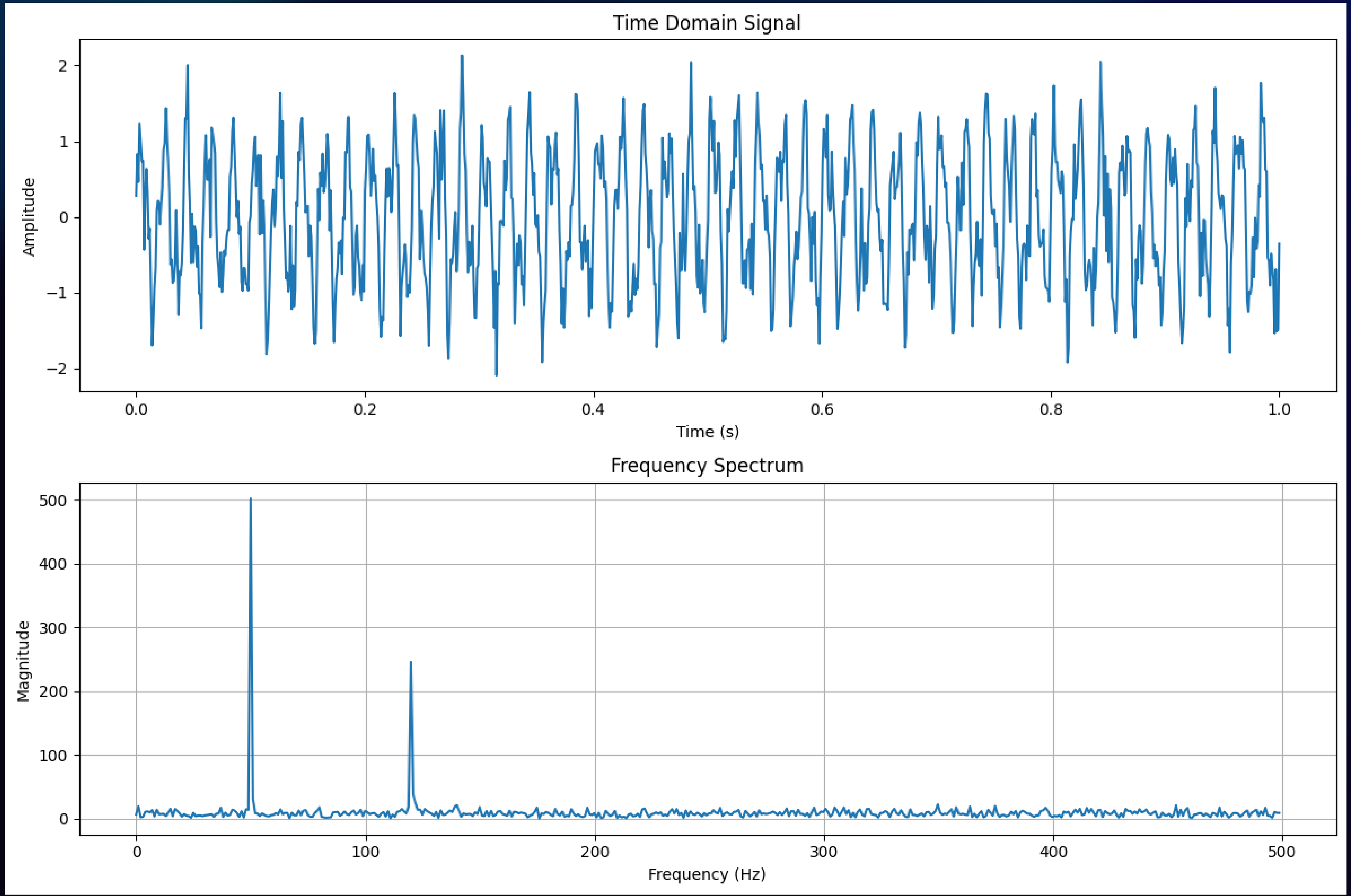
Study **oscillations in optics, acoustics, and plasma physics**. Reveals wave propagation characteristics, dispersion relations, and nonlinear effects.

Crystallography

Determine **crystal structures through X-ray diffraction**. Fourier transforms of diffraction patterns reveal atomic arrangements and lattice parameters.

Python Implementation: Fast Fourier Transform

```
from numpy.fft import fft, fftfreq
import numpy as np
import matplotlib.pyplot as plt

# Generate complex signal: multiple frequencies + noise
t = np.linspace(0, 1, 1000)
signal = (np.sin(2*np.pi*50*t) +
          0.5*np.sin(2*np.pi*120*t) +
          0.3*np.random.randn(len(t)))

# Compute FFT
yf = fft(signal)
xf = fftfreq(len(t), t[1]-t[0])

# Plot frequency spectrum (positive frequencies only)
plt.figure(figsize=(12, 8))

plt.subplot(2,1,1)
plt.plot(t, signal)
plt.title('Time Domain Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')

plt.subplot(2,1,2)
plt.plot(xf[:len(signal)//2],
         np.abs(yf[:len(signal)//2]))
plt.title('Frequency Spectrum')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.grid(True)

plt.tight_layout()
plt.show()

# Find dominant frequencies
dominant_freqs = xf[np.argsort(np.abs(yf))[-5:]]
print(f"Dominant frequencies: {dominant_freqs}")
```



Limitations of Fourier Analysis

No Temporal Localization

Fourier transforms provide **global frequency information but lose all time-specific details**. You know what frequencies are present, but not when they occur.

Stationarity Assumption

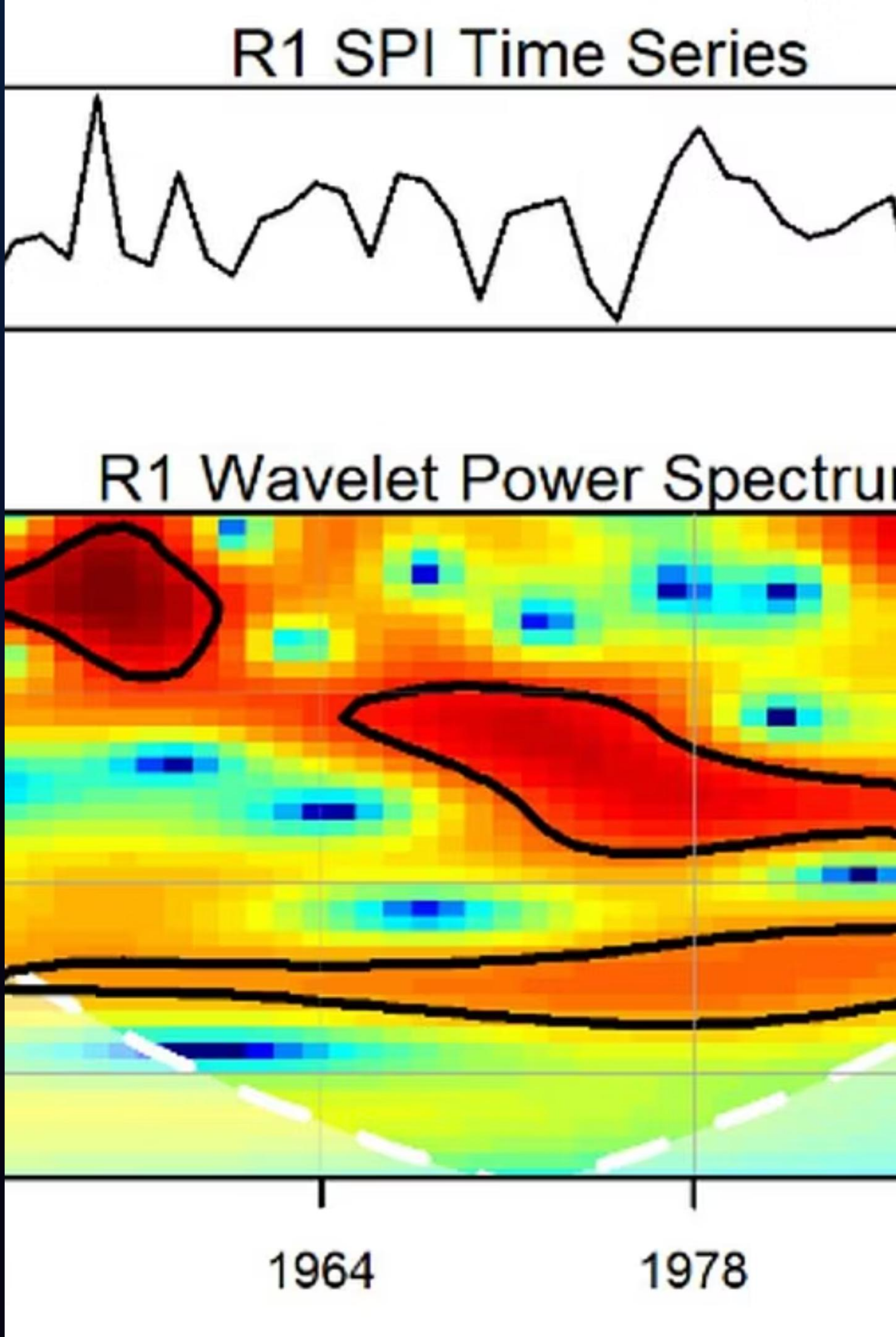
Standard Fourier analysis assumes **signal properties remain constant over time**. This fails for transient events, or time-varying systems.

Resolution Trade-off

The **uncertainty principle limits simultaneous time and frequency resolution**. Better frequency resolution requires longer observation times.

Solution: Wavelet Transform

Wavelets overcome these limitations by providing **simultaneous time and frequency localization**, making them ideal for analyzing non-stationary signals and transient phenomena.



Wavelet Analysis: Time-Frequency Localization

Wavelet analysis uses **localized basis functions** that are scaled and shifted to analyze signals in both time and frequency domains simultaneously.

Continuous Wavelet Transform:

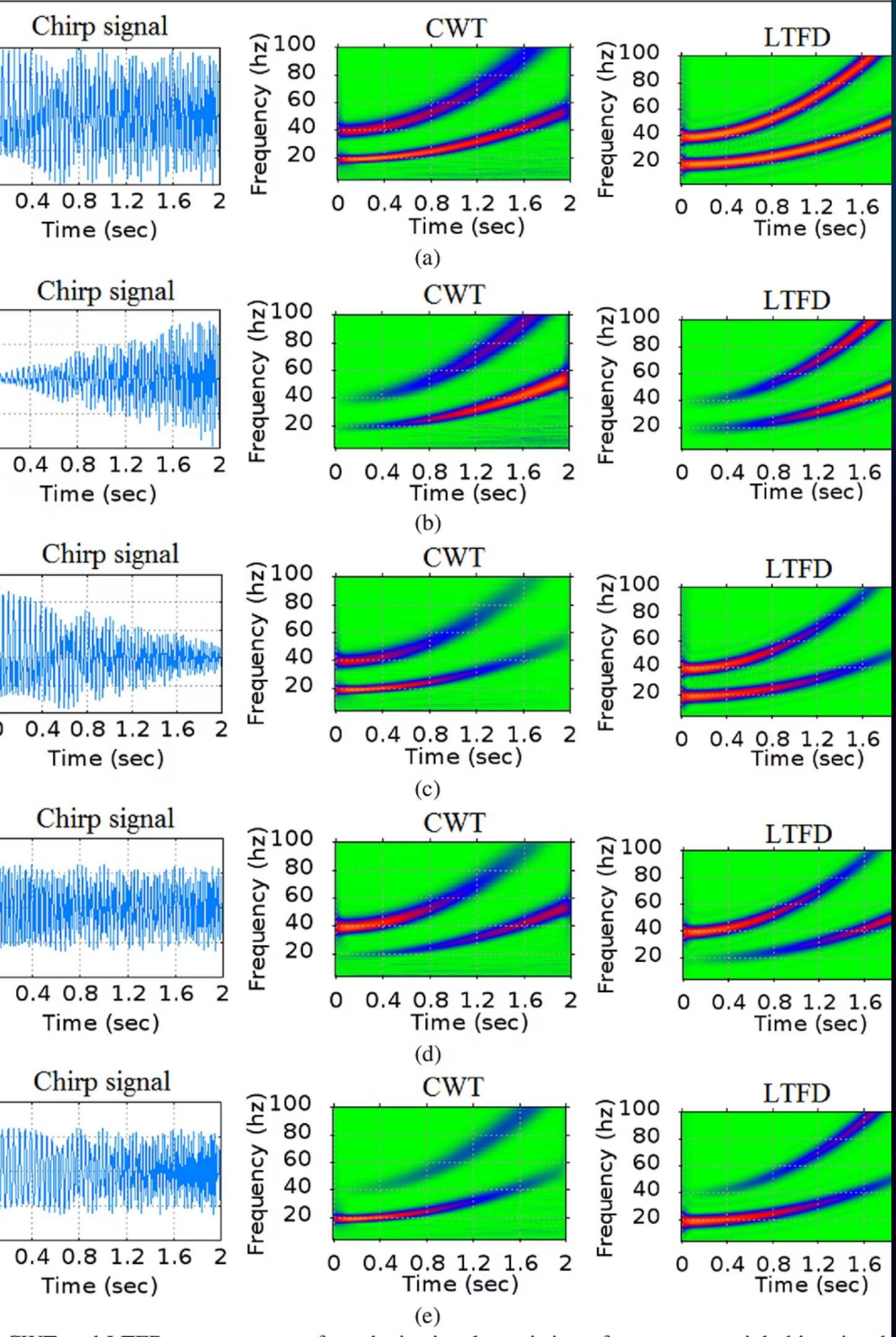
$$W(a,b) = \frac{1}{\sqrt{|a|}} \int x(t)\psi\left(\frac{t-b}{a}\right) dt$$

Where:

- **a** = scale parameter (inverse of frequency)
- **b** = translation parameter (time shift)
- **ψ(t)** = mother wavelet function

Key Advantages:

- Preserves both time and frequency information
- Adaptive resolution: good time resolution at high frequencies
- Ideal for transient and non-stationary signals



Wavelet Applications in Physics

1 **Transient Event Detection**

Analyze short-duration pulses in laser physics, gravitational wave detection, and particle collision events where precise timing is crucial.

2 **Turbulence Analysis**

Detect intermittent structures in fluid turbulence, revealing energy cascade mechanisms and coherent structures in complex flows.

3 **Non-stationary Signals**

Process signals with time-varying characteristics such as earthquake seismograms, neuronal spike trains, and variable star observations.

4 **Multi-scale Phenomena**

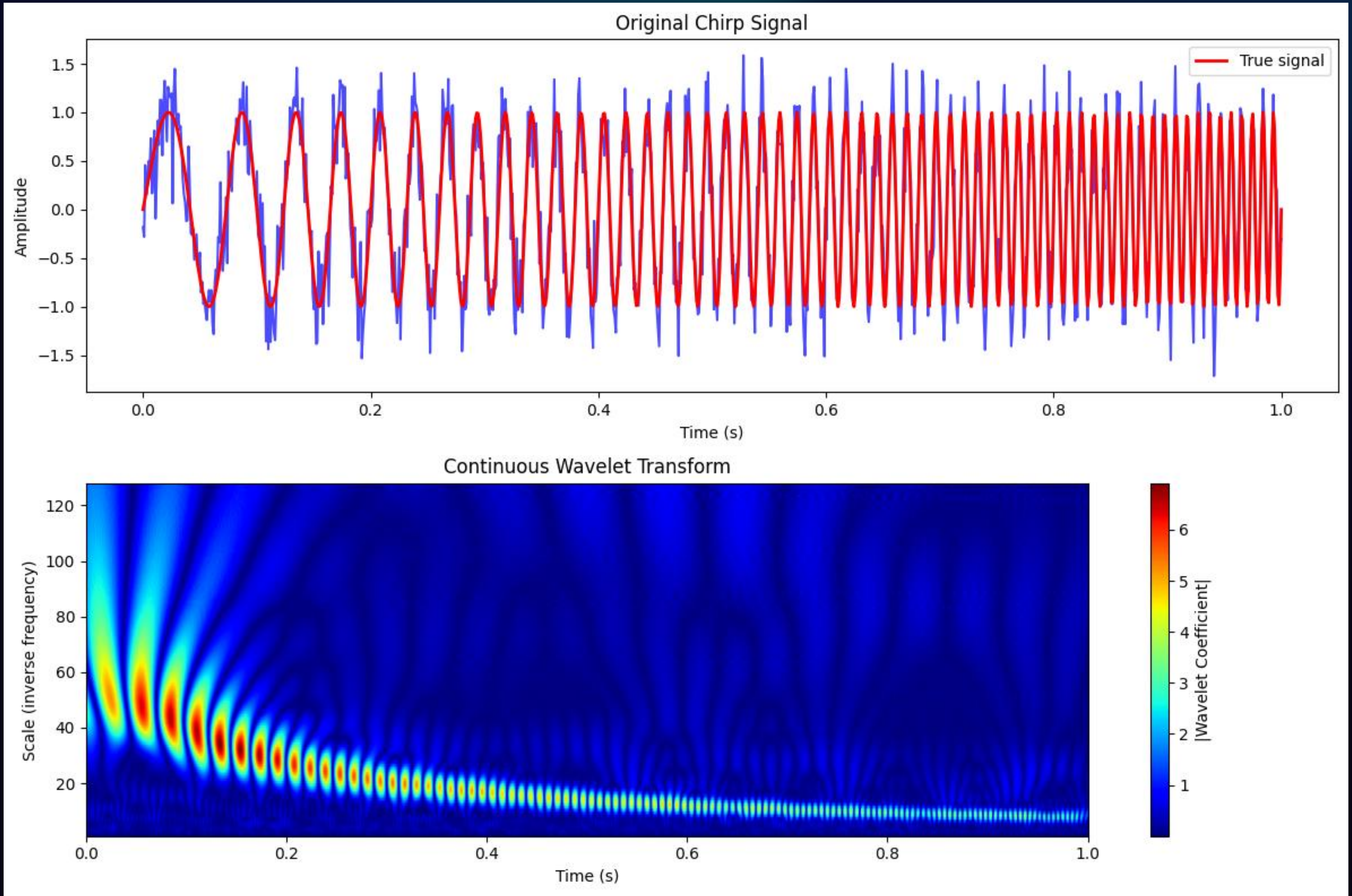
Study systems with multiple characteristic time scales, from quantum decoherence to climate oscillations and biological rhythms.

Python Implementation: Wavelet Transform

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

# Generate chirp signal: frequency increasing with time
t = np.linspace(0, 1, 1000)
chirp = np.sin(2*np.pi*(10*t + 50*t**2))
noise_chirp = chirp + 0.3*np.random.randn(len(t))

# Define scales for wavelet transform
scales = np.arange(1, 128)

# Perform continuous wavelet transform
# 'morl' = Morlet wavelet (good for oscillatory signals)
coeffs, freqs = pywt.cwt(noise_chirp, scales, 'morl',
                         sampling_period=t[1]-t[0])
```

```python
# Create time-frequency plot
plt.figure(figsize=(12, 8))

plt.subplot(2,1,1)
plt.plot(t, noise_chirp, 'b-', alpha=0.7)
plt.plot(t, chirp, 'r-', linewidth=2, label='True signal')
plt.title('Original Chirp Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()

plt.subplot(2,1,2)
plt.imshow(np.abs(coeffs), extent=[t.min(), t.max(), 1, 128],
           cmap='jet', aspect='auto', origin='lower')
plt.colorbar(label='|Wavelet Coefficient|')
plt.ylabel('Scale (inverse frequency)')
plt.xlabel('Time (s)')
plt.title('Continuous Wavelet Transform')
plt.tight_layout()
plt.show()

# Extract ridge (dominant frequency vs time)
ridge_indices = np.argmax(np.abs(coeffs), axis=0)
instantaneous_freq = freqs[ridge_indices]
plt.plot(t, instantaneous_freq, 'r-', linewidth=2)
plt.title('Instantaneous Frequency')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.show()
```

Python Implementation: Wavelet Transform



Best Practices for Time Series Analysis

Data Preprocessing

- **Detrend data:** Remove linear or polynomial trends
- **Normalize signals:** Center and scale for comparison
- **Handle missing values:** Interpolation or gap-filling
- **Remove outliers:** Statistical or physics-based filtering

Sampling Considerations

- **Nyquist criterion:** Sample at >2× highest frequency
- **Aliasing prevention:** Anti-aliasing filters before digitization
- **Resolution trade-offs:** Balance time vs. frequency resolution
- **Record length:** Sufficient data for statistical significance

Windowing Techniques

- **Spectral leakage reduction:** Apply Hanning, Hamming, or Blackman windows
- **Endpoint effects:** Taper signals to reduce artifacts
- **Overlapping segments:** Improve spectral estimates
- **Zero-padding:** Interpolate frequency domain

Method Selection

- **ARIMA:** Prediction and trend modeling
- **FFT:** Steady-state frequency analysis
- **Wavelets:** Transient events and non-stationary signals
- **Autocorrelation:** Memory effects and periodicity detection

Course Summary and Next Steps

Autocorrelation

Find **periodicity and memory effects** in temporal data. Essential for understanding system dynamics and detecting hidden patterns.

Wavelet Transform

Time-frequency localization for non-stationary signals. Ideal for transient events, chirps, and multi-scale phenomena.

ARIMA Models

Predictive modeling framework combining autoregression, differencing, and moving averages for forecasting and trend analysis.

Fourier Analysis

Extract **global frequency content** from signals. Fundamental for spectroscopy, resonance analysis, and wave physics applications.