Time Series Analysis & Forecasting for Finance

Complete Theory, Mathematics, and Python Implementation

Chapter 1: Introduction to Time Series in Finance

1.1 What is a Time Series?

A **time series** is a sequence of data points indexed in time order. In finance, time series represent the evolution of financial variables such as stock prices, returns, volatility, trading volumes, interest rates, and economic indicators over time.

Formal Definition:
A time series $\{Y_t\}$ is a stochastic process where Y_t represents the value of a random variable at time t, where $t \in \mathcal{T}$ and $\mathcal{T}$ is an index set (typically integers or continuous time).

Types of Financial Time Series:

1. **Price Series:** Stock prices, bond prices, commodity prices
2. **Return Series:** Daily, weekly, monthly returns on assets
3. **Volatility Series:** Realized volatility, implied volatility (VIX)
4. **Volume Series:** Trading volume, order flow
5. **Fundamental Series:** Earnings, revenue, macroeconomic indicators

1.2 Characteristics of Financial Time Series

Financial time series exhibit unique properties that distinguish them from other types of time series data:

1. Non-Stationarity
Most financial price series are non-stationary, meaning their statistical properties (mean, variance, autocorrelation) change over time. This violates the assumptions of many classical statistical models.

Mathematical Definition of Stationarity:
A time series $\{Y_t\}$ is **strictly stationary** if the joint distribution of $(Y_{t_1}, Y_{t_2}, \ldots, Y_{t_n})$ is identical to $(Y_{t_1+h}, Y_{t_2+h}, \ldots, Y_{t_n+h})$ for all h and all n.

A weaker form is **weak (covariance) stationarity**, requiring:

- $E[Y_t] = \mu$ (constant mean)
- $\text{Var}(Y_t) = \sigma^2$ (constant variance)
- $\text{Cov}(Y_t, Y_{t+h}) = \gamma_h$ (autocorrelation depends only on lag h, not on t)

2. Volatility Clustering
Financial returns exhibit periods of high volatility followed by periods of low volatility. This autocorrelation in squared returns was famously observed by Mandelbrot (1963) and formalized by Engle (1982) with ARCH models.

3. Heavy Tails (Leptokurtosis)
Financial return distributions have fatter tails than the normal distribution, indicating a higher probability of extreme events (crashes, rallies). Kurtosis typically exceeds 3 (the normal distribution value).

4. Leverage Effect
Negative returns tend to increase volatility more than positive returns of the same magnitude. This asymmetric response is captured by GJR-GARCH and related models.

5. Mean Reversion
Some financial series (interest rates, volatility, commodity prices) exhibit mean reversion—the tendency to return to a long-term average.

1.3 Returns vs Prices

In quantitative finance, we almost always work with **returns** rather than **prices** for several reasons:

Simple (Arithmetic) Return:

$$R_t = \frac{P_t - P_{t-1}}{P_{t-1}} = \frac{P_t}{P_{t-1}} - 1$$

Log Return (Continuously Compounded Return):

$$r_t = \ln\left(\frac{P_t}{P_{t-1}}\right) = \ln(P_t) - \ln(P_{t-1})$$

Why Use Returns?

1. **Scale-free:** Returns are dimensionless, allowing comparison across assets
2. **Stationarity:** Returns are closer to stationary than prices
3. **Time-aggregation:** Log returns are additive: $r_{t,t+n} = \sum_{i=1}^{n} r_{t+i}$
4. **Statistical properties:** Returns better satisfy distributional assumptions

Relationship Between Simple and Log Returns:
For small returns, $r_t \approx R_t$. The exact relationship is:

$$R_t = e^{r_t} - 1$$

1.4 Adjusted Close Prices

When analyzing stock data, we must account for **corporate actions** that affect prices but don't represent true economic changes:

Stock Splits: A company increases shares outstanding (e.g., 2-for-1 split doubles shares, halves price)
Dividends: Cash payments to shareholders reduce stock price by the dividend amount
Rights Issues: Offering new shares to existing shareholders at a discount

Adjusted Close Formula:

$$\text{Adj Close}_t = \text{Close}_t \times \prod_{i=t+1}^{T} \text{Adjustment Factor}_i$$

where the adjustment factor for a dividend is:

$$\text{Factor} = \frac{\text{Close} - \text{Dividend}}{\text{Close}}$$

Implementation in Python:

```
import yfinance as yf
import pandas as pd
import numpy as np

# Download stock data (adjusted close already computed)
ticker = yf.Ticker("AAPL")
data = ticker.history(start="2020-01-01", end="2024-01-01")

# Calculate simple returns
data['Simple_Return'] = data['Close'].pct_change()

# Calculate log returns
data['Log_Return'] = np.log(data['Close'] / data['Close'].shift(1))

# Verify adjustment for splits/dividends
data['Manual_Adj'] = data['Close'] / data['Close'].iloc[-1]
```

Chapter 2: Exploratory Data Analysis of Financial Time Series

2.1 Statistical Properties of Returns

Empirical Distribution Analysis

When analyzing return distributions, we examine several key statistics:

1. Mean Return:

$$\bar{r} = \frac{1}{T}\sum_{t=1}^{T} r_t$$

2. Volatility (Standard Deviation):

$$\sigma = \sqrt{\frac{1}{T-1}\sum_{t=1}^{T}(r_t - \bar{r})^2}$$

3. Skewness (Third Moment):

$$\text{Skew} = \frac{1}{T}\sum_{t=1}^{T}\left(\frac{r_t - \bar{r}}{\sigma}\right)^3$$

Negative skewness indicates a left-tailed distribution (more extreme negative returns).

4. Kurtosis (Fourth Moment):

$$\text{Kurt} = \frac{1}{T}\sum_{t=1}^{T}\left(\frac{r_t - \bar{r}}{\sigma}\right)^4$$

Excess kurtosis = Kurt - 3. Financial returns typically have excess kurtosis > 0 (leptokurtic).

Python Implementation:

```
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns

def analyze_returns(returns):
    """Comprehensive statistical analysis of return series"""

    # Remove NaN values
    ret = returns.dropna()

    # Basic statistics
    results = {
        'Mean': ret.mean(),
        'Std Dev': ret.std(),
        'Skewness': stats.skew(ret),
        'Kurtosis': stats.kurtosis(ret),
        'Min': ret.min(),
        'Max': ret.max(),
        '5th Percentile': ret.quantile(0.05),
        '95th Percentile': ret.quantile(0.95)
    }

    # Normality tests
    results['Jarque-Bera p-value'] = stats.jarque_bera(ret)[1]
    results['Shapiro-Wilk p-value'] = stats.shapiro(ret)[1]

    return pd.Series(results)

# Example usage
returns = data['Log_Return'].dropna()
stats_summary = analyze_returns(returns)
print(stats_summary)
```

2.2 QQ-Plots (Quantile-Quantile Plots)

A **QQ-plot** compares the quantiles of empirical data against theoretical quantiles from a reference distribution (typically normal).

Mathematical Basis:

For a sample $\{x_1, x_2, \ldots, x_n\}$ sorted in ascending order:

- Empirical quantile at probability p: $Q_{empirical}(p) = x_{[np]}$
- Theoretical quantile from $N(0,1)$: $Q_{theoretical}(p) = \Phi^{-1}(p)$

If data is normally distributed, points should lie on the 45-degree line $y = x$.

Deviations from Normality:

- **S-shaped curve:** Heavy tails (excess kurtosis)
- **Curved upward/downward:** Positive/negative skewness
- **Points far from line at extremes:** Outliers

Python Implementation:

```
import scipy.stats as stats
import matplotlib.pyplot as plt

def qq_plot_analysis(returns, title="QQ Plot"):
    """Generate QQ-plot comparing returns to normal distribution"""

    fig, axes = plt.subplots(1, 2, figsize=(14, 5))

    # QQ-plot against normal distribution
    stats.probplot(returns, dist="norm", plot=axes[0])
    axes[0].set_title(f'{title} - Normal Distribution')
    axes[0].grid(True, alpha=0.3)

    # QQ-plot against t-distribution (better for financial data)
    # Estimate degrees of freedom
    params = stats.t.fit(returns)
    df = params[0]
    stats.probplot(returns, dist=stats.t, sparams=(df,), plot=axes[1])
    axes[1].set_title(f'{title} - t-Distribution (df={df:.2f})')
    axes[1].grid(True, alpha=0.3)

    plt.tight_layout()
    return fig

# Usage
qq_plot_analysis(returns, title="S&amp;amp;P 500 Log Returns")
```

2.3 The t-Distribution in Finance

The **Student's t-distribution** provides a better fit for financial returns than the normal distribution due to its heavier tails.

Probability Density Function:

$$f(x;\nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\,\Gamma\left(\frac{\nu}{2}\right)}\left(1+\frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

where ν is the **degrees of freedom** parameter.

Properties:

- As $\nu \to \infty$, the t-distribution converges to $N(0,1)$
- Lower ν means heavier tails (more extreme events)
- Empirically, financial returns often fit $\nu \approx 4$ to $\nu \approx 8$

Fitting t-Distribution to Data:

```
from scipy.stats import t

def fit_t_distribution(returns):
    """Fit t-distribution and compare to normal"""

    # Fit t-distribution
    params = t.fit(returns)
    df, loc, scale = params

    # Generate theoretical distributions
    x = np.linspace(returns.min(), returns.max(), 100)

    # t-distribution PDF
    t_pdf = t.pdf(x, df, loc, scale)

    # Normal distribution PDF
    mu, sigma = returns.mean(), returns.std()
    norm_pdf = stats.norm.pdf(x, mu, sigma)

    # Plot comparison
    plt.figure(figsize=(12, 6))
    plt.hist(returns, bins=50, density=True, alpha=0.6,
             label='Empirical Data', color='skyblue')
    plt.plot(x, t_pdf, 'r-', linewidth=2,
             label=f't-dist (df={df:.2f})')
    plt.plot(x, norm_pdf, 'g--', linewidth=2,
             label='Normal dist')
    plt.xlabel('Returns')
    plt.ylabel('Density')
    plt.title('Distribution Fitting: t-distribution vs Normal')
    plt.legend()
    plt.grid(True, alpha=0.3)
```

```
    # Goodness of fit (Kolmogorov-Smirnov test)
    ks_t = stats.kstest(returns, lambda x: t.cdf(x, df, loc, scale))
    ks_norm = stats.kstest(returns, lambda x: stats.norm.cdf(x, mu, sigma))

    print(f"t-distribution KS p-value: {ks_t.pvalue:.4f}")
    print(f"Normal distribution KS p-value: {ks_norm.pvalue:.4f}")

    return df, loc, scale
```

2.4 Autocorrelation and Serial Dependence

Autocorrelation Function (ACF):

The autocorrelation at lag k measures the linear relationship between Y_t and Y_{t-k}:

$$\rho_k = \frac{\text{Cov}(Y_t, Y_{t-k})}{\text{Var}(Y_t)} = \frac{\gamma_k}{\gamma_0}$$

where $\gamma_k = E[(Y_t - \mu)(Y_{t-k} - \mu)]$ is the autocovariance.

Key Facts About Financial Returns:

- Price autocorrelations decay slowly (non-stationary)
- Return autocorrelations are typically weak and near zero
- **Squared returns** show strong autocorrelation (volatility clustering)

Python Implementation:

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import acf, pacf

def autocorrelation_analysis(series, lags=40):
    """Analyze autocorrelation structure"""

    fig, axes = plt.subplots(2, 2, figsize=(14, 10))

    # ACF of returns
    plot_acf(series, lags=lags, ax=axes[0,0], alpha=0.05)
    axes[0,0].set_title('ACF of Returns')

    # PACF of returns
    plot_pacf(series, lags=lags, ax=axes[0,1], alpha=0.05)
    axes[0,1].set_title('PACF of Returns')

    # ACF of squared returns (volatility proxy)
    plot_acf(series**2, lags=lags, ax=axes[1,0], alpha=0.05)
    axes[1,0].set_title('ACF of Squared Returns (Volatility)')

    # ACF of absolute returns
    plot_acf(np.abs(series), lags=lags, ax=axes[1,1], alpha=0.05)
    axes[1,1].set_title('ACF of Absolute Returns')

    plt.tight_layout()
```

```
    # Ljung-Box test for autocorrelation
    from statsmodels.stats.diagnostic import acorr_ljungbox
    lb_test = acorr_ljungbox(series, lags=[10, 20, 30], return_df=True)
    print("\nLjung-Box Test for Returns:")
    print(lb_test)

    lb_test_sq = acorr_ljungbox(series**2, lags=[10, 20, 30], return_df=True)
    print("\nLjung-Box Test for Squared Returns:")
    print(lb_test_sq)
```

Chapter 3: Stationarity and Unit Root Tests

3.1 Why Stationarity Matters

Most time series models (ARMA, regression) assume **stationarity**. Non-stationary series can lead to:

- **Spurious regression:** High R^2 with unrelated variables
- **Invalid inference:** Incorrect confidence intervals and p-values
- **Poor forecasting:** Models extrapolate unsustainable trends

3.2 Testing for Stationarity

1. Augmented Dickey-Fuller (ADF) Test

Tests the null hypothesis H_0: series has a unit root (non-stationary)

The test regression is:

$$\Delta Y_t = \alpha + \beta t + \gamma Y_{t-1} + \sum_{i=1}^{p} \delta_i \Delta Y_{t-i} + \epsilon_t$$

We test $H_0 : \gamma = 0$ (unit root) vs (stationary).

2. KPSS Test

Tests the null hypothesis H_0: series is stationary (opposite of ADF)

Decomposition: $Y_t = \xi t + r_t + \epsilon_t$ where r_t is a random walk.

Test statistic based on cumulative sum of residuals.

Python Implementation:

```
from statsmodels.tsa.stattools import adfuller, kpss

def stationarity_tests(series, name='Series'):
    """Perform ADF and KPSS tests"""

    # Augmented Dickey-Fuller test
```

```
    adf_result = adfuller(series, autolag='AIC')

    print(f"\n{'='*60}")
    print(f"Stationarity Tests for {name}")
    print(f"{'='*60}")

    print("\n1. Augmented Dickey-Fuller Test:")
    print(f"   Test Statistic: {adf_result[0]:.4f}")
    print(f"   p-value: {adf_result[1]:.4f}")
    print(f"   Critical Values:")
    for key, value in adf_result[4].items():
        print(f"      {key}: {value:.4f}")

    if adf_result[1] &lt; 0.05:
        print("   ✓ Reject H0: Series is STATIONARY")
    else:
        print("   ✗ Fail to reject H0: Series is NON-STATIONARY")

    # KPSS test
    kpss_result = kpss(series, regression='c', nlags='auto')

    print("\n2. KPSS Test:")
    print(f"   Test Statistic: {kpss_result[0]:.4f}")
    print(f"   p-value: {kpss_result[1]:.4f}")
    print(f"   Critical Values:")
    for key, value in kpss_result[3].items():
        print(f"      {key}: {value:.4f}")

    if kpss_result[1] &gt; 0.05:
        print("   ✓ Fail to reject H0: Series is STATIONARY")
    else:
        print("   ✗ Reject H0: Series is NON-STATIONARY")

    return adf_result, kpss_result

# Test prices vs returns
adf_price, kpss_price = stationarity_tests(data['Close'], 'Price')
adf_ret, kpss_ret = stationarity_tests(returns, 'Returns')
```

3.3 Transformations to Achieve Stationarity

1. Differencing:

$$\Delta Y_t = Y_t - Y_{t-1}$$

Second-order differencing: $\Delta^2 Y_t = \Delta Y_t - \Delta Y_{t-1}$

2. Log Transformation:

$$\tilde{Y}_t = \ln(Y_t)$$

Stabilizes variance for series with exponential growth.

3. Detrending:
Remove deterministic trend via regression:

$$Y_t = \alpha + \beta t + \epsilon_t$$

Use residuals $\hat{\epsilon}_t$ as detrended series.

4. Seasonal Differencing:
For seasonal data with period s:

$$\Delta_s Y_t = Y_t - Y_{t-s}$$

Chapter 4: ARIMA Models

4.1 Autoregressive (AR) Models

An **AR(p)** model expresses the current value as a linear combination of p past values plus noise:

$$Y_t = c + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_p Y_{t-p} + \epsilon_t$$

where $\epsilon_t \sim WN(0, \sigma^2)$ (white noise).

Stationarity Condition:
The characteristic equation $1 - \phi_1 z - \phi_2 z^2 - \cdots - \phi_p z^p = 0$ must have all roots outside the unit circle .

Example: AR(1) Model

$$Y_t = c + \phi Y_{t-1} + \epsilon_t$$

Stationary if .

Mean: $\mu = \frac{c}{1-\phi}$

Variance: $\sigma_Y^2 = \frac{\sigma^2}{1-\phi^2}$

ACF decays exponentially: $\rho_k = \phi^k$

4.2 Moving Average (MA) Models

An **MA(q)** model expresses the current value as a linear combination of current and q past error terms:

$$Y_t = \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_q \epsilon_{t-q}$$

Key Properties:

- Always stationary (no restrictions on parameters)
- ACF cuts off after lag q (all $\rho_k = 0$ for)
- PACF decays exponentially

Invertibility Condition:
The characteristic equation $1 + \theta_1 z + \theta_2 z^2 + \cdots + \theta_q z^q = 0$ must have roots outside unit circle for unique representation.

4.3 ARIMA(p, d, q) Models

ARIMA = **A**uto**R**egressive **I**ntegrated **M**oving **A**verage

General form:

$$\phi(L)(1-L)^d Y_t = \theta(L)\epsilon_t$$

where:

- p: order of AR component
- d: degree of differencing (integration order)
- q: order of MA component
- L: lag operator ($LY_t = Y_{t-1}$)

Polynomial Notation:

$$\phi(L) = 1 - \phi_1 L - \phi_2 L^2 - \cdots - \phi_p L^p$$

$$\theta(L) = 1 + \theta_1 L + \theta_2 L^2 + \cdots + \theta_q L^q$$

Special Cases:

- ARIMA(1,0,0) = AR(1)
- ARIMA(0,0,1) = MA(1)
- ARIMA(0,1,0) = Random walk
- ARIMA(0,1,1) = Exponentially weighted moving average

4.4 Model Identification via ACF/PACF

Identification Strategy:

Model	ACF Pattern	PACF Pattern
AR(p)	Decays exponentially/sinusoidally	Cuts off after lag p
MA(q)	Cuts off after lag q	Decays exponentially/sinusoidally
ARMA(p,q)	Decays	Decays

Step-by-Step Process:

1. Test stationarity (ADF, KPSS)
2. If non-stationary, difference until stationary
3. Plot ACF/PACF of stationary series
4. Identify preliminary p and q values
5. Estimate model and check residuals
6. Compare models using AIC/BIC

4.5 Python Implementation: ARIMA Modeling

```
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
import itertools
import warnings
warnings.filterwarnings('ignore')

def fit_arima_model(series, order=(1,1,1)):
    """Fit ARIMA model and display diagnostics"""

    # Fit model
    model = ARIMA(series, order=order)
    fitted = model.fit()

    # Print summary
    print(fitted.summary())

    # Diagnostic plots
    fitted.plot_diagnostics(figsize=(14, 10))
    plt.tight_layout()

    # Residual analysis
    residuals = fitted.resid

    # Test for autocorrelation in residuals
    lb_test = acorr_ljungbox(residuals, lags=20, return_df=True)
    print("\nLjung-Box Test on Residuals:")
    print(lb_test.head(10))

    return fitted

# Example: Fit ARIMA(1,1,1)
model_111 = fit_arima_model(data['Close'], order=(1,1,1))
```

Auto-ARIMA (Automated Model Selection):

```
from pmdarima import auto_arima

def auto_arima_selection(series, seasonal=False, m=1):
    """Automatically select best ARIMA model using AIC/BIC"""

    model = auto_arima(
        series,
        start_p=0, start_q=0,
        max_p=5, max_q=5,
        d=None,  # Auto-detect differencing order
        seasonal=seasonal,
        m=m,  # Seasonal period
        stepwise=True,
        suppress_warnings=True,
        information_criterion='aic',
        trace=True
    )
```

```
    print("\n" + "="*60)
    print("Best Model Selected:")
    print(model.summary())

    return model

# Usage
best_model = auto_arima_selection(returns.dropna())
```

4.6 Forecasting with ARIMA

One-Step-Ahead Forecast:

$$\hat{Y}_{T+1|T} = E[Y_{T+1}|Y_T, Y_{T-1}, \ldots]$$

Multi-Step-Ahead Forecast:

$$\hat{Y}_{T+h|T} = E[Y_{T+h}|Y_T, Y_{T-1}, \ldots]$$

Forecast Uncertainty:
Prediction intervals widen as forecast horizon h increases.

Python Implementation:

```
def arima_forecast(model, steps=30, alpha=0.05):
    """Generate forecasts with confidence intervals"""

    # Get forecast
    forecast_result = model.get_forecast(steps=steps)
    forecast_mean = forecast_result.predicted_mean
    forecast_ci = forecast_result.conf_int(alpha=alpha)

    # Plot
    fig, ax = plt.subplots(figsize=(14, 6))

    # Historical data (last 200 points)
    historical = model.data.endog[-200:]
    ax.plot(historical.index, historical.values,
            label='Historical', color='blue')

    # Forecast
    forecast_index = pd.date_range(
        start=historical.index[-1],
        periods=steps+1,
        freq='D'
    )[1:]

    ax.plot(forecast_index, forecast_mean,
            label='Forecast', color='red', linewidth=2)

    # Confidence interval
    ax.fill_between(forecast_index,
```

```
                        forecast_ci.iloc[:, 0],
                        forecast_ci.iloc[:, 1],
                        color='pink', alpha=0.3,
                        label=f'{int((1-alpha)*100)}% Confidence Interval')

    ax.set_xlabel('Date')
    ax.set_ylabel('Value')
    ax.set_title('ARIMA Forecast')
    ax.legend()
    ax.grid(True, alpha=0.3)

    return forecast_mean, forecast_ci

# Generate 30-day forecast
forecast, conf_int = arima_forecast(model_111, steps=30)
```

Chapter 5: GARCH Models for Volatility

5.1 Motivation: Volatility Clustering

Stylized Fact: Financial returns exhibit **volatility clustering**—large changes tend to be followed by large changes, and small changes by small changes.

This violates the constant variance assumption of ARMA models. We need **conditional heteroskedasticity** models where variance depends on past information.

5.2 ARCH Model (Engle, 1982)

ARCH(q) = **A**uto**R**egressive **C**onditional **H**eteroskedasticity

Model specification:

$$r_t = \mu + \epsilon_t, \quad \epsilon_t = \sigma_t z_t, \quad z_t \sim N(0,1)$$

$$\sigma_t^2 = \omega + \alpha_1 \epsilon_{t-1}^2 + \alpha_2 \epsilon_{t-2}^2 + \cdots + \alpha_q \epsilon_{t-q}^2$$

Interpretation: Today's volatility depends on squared past shocks.

Constraints:

-
- $\alpha_i \geq 0$ for all i
- for stationarity

5.3 GARCH Model (Bollerslev, 1986)

GARCH(p,q) = **G**eneralized ARCH

Adds lagged conditional variances to the volatility equation:

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2 + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^2$$

GARCH(1,1) is most commonly used:

$$\sigma_t^2 = \omega + \alpha \epsilon_{t-1}^2 + \beta \sigma_{t-1}^2$$

Properties:

- **Persistence:** $\alpha + \beta$ close to 1 implies high persistence
- **Unconditional variance:** $\sigma^2 = \frac{\omega}{1-\alpha-\beta}$
- Parsimonious: GARCH(1,1) often outperforms ARCH(q) with large q

5.4 GJR-GARCH (Leverage Effect)

Problem: Standard GARCH treats positive and negative shocks symmetrically, but empirically, negative returns increase volatility more.

GJR-GARCH(1,1):

$$\sigma_t^2 = \omega + (\alpha + \gamma I_{t-1}) \epsilon_{t-1}^2 + \beta \sigma_{t-1}^2$$

where $I_{t-1} = 1$ if , else 0.

If , negative shocks have impact $\alpha + \gamma$ vs α for positive shocks.

5.5 Python Implementation: GARCH Modeling

```
from arch import arch_model
from arch.univariate import GARCH, GJR

def fit_garch_model(returns, p=1, q=1, model_type='GARCH'):
    """Fit GARCH or GJR-GARCH model"""

    # Remove mean
    returns_clean = returns.dropna() * 100  # Scale to percentage

    if model_type == 'GARCH':
        model = arch_model(returns_clean, vol='GARCH', p=p, q=q)
    elif model_type == 'GJR':
        model = arch_model(returns_clean, vol='Garch', p=p, o=1, q=q)
    else:
        raise ValueError("model_type must be 'GARCH' or 'GJR'")

    # Fit model
    fitted = model.fit(disp='off')
```

```
    print(fitted.summary())

    # Plot conditional volatility
    fig, axes = plt.subplots(2, 1, figsize=(14, 10))

    # Returns
    axes[0].plot(returns_clean.index, returns_clean,
                 label='Returns', color='blue', alpha=0.6)
    axes[0].set_ylabel('Returns (%)')
    axes[0].set_title(f'{model_type}({p},{q}) Model - Returns')
    axes[0].legend()
    axes[0].grid(True, alpha=0.3)

    # Conditional volatility
    cond_vol = fitted.conditional_volatility
    axes[1].plot(cond_vol.index, cond_vol,
                 label='Conditional Volatility', color='red', linewidth=1.5)
    axes[1].set_ylabel('Volatility (%)')
    axes[1].set_xlabel('Date')
    axes[1].set_title(f'{model_type}({p},{q}) - Conditional Volatility')
    axes[1].legend()
    axes[1].grid(True, alpha=0.3)

    plt.tight_layout()

    return fitted

# Fit GARCH(1,1)
garch_model = fit_garch_model(returns, p=1, q=1, model_type='GARCH')

# Fit GJR-GARCH(1,1)
gjr_model = fit_garch_model(returns, p=1, q=1, model_type='GJR')
```

Volatility Forecasting:

```
def garch_volatility_forecast(model, horizon=30):
    """Forecast volatility using fitted GARCH model"""

    # Generate forecast
    forecast = model.forecast(horizon=horizon)
    forecast_variance = forecast.variance.values[-1, :]
    forecast_volatility = np.sqrt(forecast_variance)

    # Plot
    fig, ax = plt.subplots(figsize=(14, 6))

    # Historical volatility
    hist_vol = model.conditional_volatility
    ax.plot(hist_vol.index[-100:], hist_vol.values[-100:],
            label='Historical Volatility', color='blue')

    # Forecast
    last_date = hist_vol.index[-1]
    forecast_dates = pd.date_range(start=last_date, periods=horizon+1)[1:]
```

```
    ax.plot(forecast_dates, forecast_volatility,
            label='Forecast', color='red', linewidth=2, linestyle='--')

    ax.set_xlabel('Date')
    ax.set_ylabel('Volatility (%)')
    ax.set_title('GARCH Volatility Forecast')
    ax.legend()
    ax.grid(True, alpha=0.3)

    return forecast_volatility

# Forecast next 30 days
vol_forecast = garch_volatility_forecast(garch_model, horizon=30)
```

Chapter 6: Model Selection and Diagnostics

6.1 Information Criteria

Akaike Information Criterion (AIC):

$$\text{AIC} = -2\ln(\mathcal{L}) + 2k$$

Bayesian Information Criterion (BIC):

$$\text{BIC} = -2\ln(\mathcal{L}) + k\ln(n)$$

where:

- $\mathcal{L}$ = maximized likelihood
- k = number of parameters
- n = sample size

Interpretation:

- Lower values indicate better fit (penalized for complexity)
- BIC penalizes complexity more heavily than AIC
- Choose model with minimum AIC/BIC

6.2 Residual Diagnostics

Requirements for Good Model:

1. Residuals should be uncorrelated (white noise)
2. Residuals should be normally distributed
3. No remaining ARCH effects in residuals

Diagnostic Tests:

```
def model_diagnostics(fitted_model, lags=20):
    """Comprehensive residual diagnostics"""

    residuals = fitted_model.resid

    fig, axes = plt.subplots(2, 2, figsize=(14, 10))

    # 1. Residual plot
    axes[0,0].plot(residuals)
    axes[0,0].set_title('Residuals Over Time')
    axes[0,0].set_xlabel('Time')
    axes[0,0].set_ylabel('Residual')
    axes[0,0].grid(True, alpha=0.3)

    # 2. ACF of residuals
    plot_acf(residuals, lags=lags, ax=axes[0,1], alpha=0.05)
    axes[0,1].set_title('ACF of Residuals')

    # 3. QQ-plot
    stats.probplot(residuals, dist="norm", plot=axes[1,0])
    axes[1,0].set_title('QQ-Plot of Residuals')
    axes[1,0].grid(True, alpha=0.3)

    # 4. Histogram
    axes[1,1].hist(residuals, bins=50, density=True, alpha=0.6, color='skyblue')

    # Overlay normal distribution
    mu, sigma = residuals.mean(), residuals.std()
    x = np.linspace(residuals.min(), residuals.max(), 100)
    axes[1,1].plot(x, stats.norm.pdf(x, mu, sigma), 'r-', linewidth=2)
    axes[1,1].set_title('Histogram of Residuals')
    axes[1,1].set_xlabel('Residual')
    axes[1,1].set_ylabel('Density')
    axes[1,1].grid(True, alpha=0.3)

    plt.tight_layout()

    # Statistical tests
    print("\n" + "="*60)
    print("Residual Diagnostic Tests")
    print("="*60)

    # Ljung-Box test
    lb_test = acorr_ljungbox(residuals, lags=[10, 20], return_df=True)
    print("\nLjung-Box Test (Autocorrelation):")
    print(lb_test)

    # Jarque-Bera test
    jb_stat, jb_pval = stats.jarque_bera(residuals)
    print(f"\nJarque-Bera Test (Normality):")
    print(f"  Statistic: {jb_stat:.4f}")
    print(f"  p-value: {jb_pval:.4f}")

    # ARCH-LM test for remaining ARCH effects
    from statsmodels.stats.diagnostic import het_arch
    arch_test = het_arch(residuals, nlags=10)
```

```
    print(f"\nARCH-LM Test (Heteroskedasticity):")
    print(f"  LM Statistic: {arch_test[0]:.4f}")
    print(f"  p-value: {arch_test[1]:.4f}")

    return None

# Run diagnostics
model_diagnostics(model_111)
```

Chapter 7: Advanced Topics and Extensions

7.1 SARIMAX Models

SARIMAX = **S**easonal **ARIMA** with e**X**ogenous regressors

For data with seasonal patterns (e.g., monthly economic data):

$$\text{SARIMAX}(p, d, q) \times (P, D, Q)_s$$

- (p, d, q): non-seasonal components
- $(P, D, Q)_s$: seasonal components with period s

Python Example:

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

# Fit SARIMAX(1,1,1)x(1,1,1,12) for monthly data
model = SARIMAX(
    monthly_data,
    order=(1, 1, 1),
    seasonal_order=(1, 1, 1, 12),
    enforce_stationarity=False,
    enforce_invertibility=False
)

fitted = model.fit(disp=False)
print(fitted.summary())
```

7.2 Exponential Smoothing Methods

Simple Exponential Smoothing (SES):

$$\hat{Y}_{t+1|t} = \alpha Y_t + (1 - \alpha)\hat{Y}_{t|t-1}$$

Holt's Linear Trend:

$$\hat{Y}_{t+h|t} = \ell_t + hb_t$$

where level ℓ_t and trend b_t are updated recursively.

Holt-Winters (Seasonal):
Adds seasonal component s_t.

Python Implementation:

```
from statsmodels.tsa.holtwinters import ExponentialSmoothing

def fit_holtwinters(series, seasonal_periods=12, trend='add', seasonal='add'):
    """Fit Holt-Winters exponential smoothing"""

    model = ExponentialSmoothing(
        series,
        seasonal_periods=seasonal_periods,
        trend=trend,
        seasonal=seasonal,
    )

    fitted = model.fit()

    # Forecast
    forecast = fitted.forecast(steps=24)

    # Plot
    plt.figure(figsize=(14, 6))
    plt.plot(series.index, series, label='Observed')
    plt.plot(fitted.fittedvalues.index, fitted.fittedvalues,
             label='Fitted', linestyle='--')
    plt.plot(forecast.index, forecast, label='Forecast',
             color='red', linewidth=2)
    plt.xlabel('Time')
    plt.ylabel('Value')
    plt.title('Holt-Winters Exponential Smoothing')
    plt.legend()
    plt.grid(True, alpha=0.3)

    return fitted
```

Chapter 8: Practical Trading Applications

8.1 Volatility-Based Position Sizing

Use GARCH forecasts to adjust position sizes:

$$\text{Position Size} = \frac{\text{Risk Budget}}{\sigma_{forecast}}$$

8.2 Pairs Trading with Cointegration

Test for cointegration using Engle-Granger or Johansen tests, then model the spread with ARIMA.

8.3 VaR Calculation

Value at Risk (VaR): Maximum expected loss at confidence level α over horizon h.

Using GARCH forecast:

$$\text{VaR}_{\alpha,h} = -(\mu_h + \sigma_h \cdot z_\alpha)$$

where z_α is the α-quantile of the return distribution.

Conclusion

This textbook has covered the complete theory and implementation of time series analysis for finance, from basic concepts to advanced GARCH models. Key takeaways:

1. Financial data requires careful preprocessing (returns, stationarity)
2. ARIMA models capture linear temporal dependencies
3. GARCH models capture time-varying volatility
4. Model selection requires balancing fit and complexity (AIC/BIC)
5. Diagnostics are critical to validate model assumptions
6. Practical applications include forecasting, risk management, and trading

Further Reading:

- Tsay, R. S. (2010). *Analysis of Financial Time Series*
- Hamilton, J. D. (1994). *Time Series Analysis*
- Brooks, C. (2014). *Introductory Econometrics for Finance*

Python Libraries:

- statsmodels: ARIMA, SARIMAX, diagnostics
- arch: GARCH family models
- pmdarima: Auto-ARIMA
- pandas: Data manipulation
- matplotlib/seaborn: Visualization