# GPT: Gaussian Process Trading & Forecasting User Manual and Mathematical Documentation

Dimitrios Thomakos
with Claude (Anthropic)

August 2025

**Abstract**

This manual provides comprehensive documentation for the Enhanced Gaussian Process Trading System (GPT_v3.py), an algorithmic trading framework that combines Gaussian Process regression with Chebyshev polynomial basis functions for financial time series prediction and trading signal generation. The system implements rigorous mathematical foundations while maintaining practical usability through configurable parameters and multi-frequency support. This document covers the underlying mathematics, implementation details, and comprehensive usage instructions.

# Contents

# 1 Introduction

The Enhanced Gaussian Process Trading System represents a state-of-the-art approach to quantitative trading that leverages the power of Gaussian Process (GP) regression for financial time series prediction. Unlike traditional machine learning approaches, Gaussian Processes provide not only point predictions but also uncertainty estimates, making them particularly suitable for risk-aware trading strategies. The core of the code can be used for forecasting exercises and backtesting as well, something that we document in a different program file (available on request). This document also sets the stage for various updates and upgrades of the code that one can easily implement, such as multivariate extensions, walk-forward-optimization and similar.

## 1.1 Key Features

- **Gaussian Process Regression**: Non-parametric Bayesian approach with uncertainty quantification and hyperparameter optimization.
- **Chebyshev Polynomial Basis**: Orthogonal polynomial features for improved numerical stability, they can accept any dimension of inputs.
- **Feature Construction**: In the current implementation lagged returns are used only, but one can amend the code easily to utilize any number and kind of explanatory variables – note that the data are normalized before entering the basis functions.
- **Multi-frequency Support**: Daily, weekly, and monthly data processing is easily available for backtesting.
- **Look-ahead Bias Protection**: Rigorous temporal alignment of features and targets to avoid look-ahead bias.
- **Hyperparameter Optimization**: Automatic tuning via marginal likelihood maximization.
- **Comprehensive Backtesting**: Full performance attribution and risk analysis, including forecasting performance.
- **Interactive Configuration**: User-friendly parameter selection via the console.

## 1.2 System Architecture

The system is built around several core components:

1. `ChebyshevBasis`: Implements orthogonal polynomial basis functions
2. `GaussianProcessRegressor`: Core GP implementation with kernel methods
3. `GPTradingSystem`: Main trading system orchestrator
4. `TradingStrategy`: Signal generation and position management
5. `TradingMetrics`: Performance evaluation and risk metrics

# 2 Mathematical Foundations

## 2.1 The Bayesian paradigm

The core of this approach is the Bayesian worldview, which treats probability as a degree of belief. Learning is a process of updating this belief as new evidence (data) becomes available, a process governed by Bayes' Theorem:

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})}$$

where $p(\theta|\mathcal{D})$ is the posterior belief about parameters $\theta$ after observing data $\mathcal{D}$, $p(\mathcal{D}|\theta)$ is the likelihood of the data given the parameters, and $p(\theta)$ is the prior belief about the parameters. The term $p(\mathcal{D})$ is the marginal likelihood, serving as a normalization constant.

## 2.2 A simplified view of the "Weight/Function-Space" for linear regression

The 'GPT_v3.py' script's model can be understood as a Bayesian linear regression model with a specific set of basis functions. The model is defined as $y = \boldsymbol{\phi}(\mathbf{x})^\top \mathbf{w} + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma_n^2)$. We place a Gaussian prior on the weights $\mathbf{w}$:

$$p(\mathbf{w}) = \mathcal{N}(0, \Sigma_p)$$

with the computational simplification of $\Sigma_p = \sigma_p^2 \mathbf{I}_p$, and a Gaussian likelihood on the data:

$$p(y|\mathbf{x}, \mathbf{w}) = \mathcal{N}\left(\boldsymbol{\phi}(\mathbf{x})^\top \mathbf{w}, \sigma_n^2\right)$$

Here $p$ denotes the degree of the model – number of elements in the $(p \times 1)$ vector $\boldsymbol{\phi}(\cdot)$ of basis functions – and also note that $\mathbf{x}$ is an $(K \times 1)$ vector of input explanatory variables.

## 2.3 Posterior and predictive distributions

By applying Bayes' theorem, and denoting $\mathcal{D} = (\mathbf{y}, \mathbf{X})$, we find that the posterior distribution over the weights is also a Gaussian:

$$p(\mathbf{w}|\mathcal{D}) = \mathcal{N}\left(\overline{\mathbf{w}}, \mathbf{A}^{-1}\right)$$

with precision matrix $\mathbf{A} = \sigma_n^{-2}\boldsymbol{\Phi}\boldsymbol{\Phi}^\top + \sigma_p^{-2}\mathbf{I}_p$ and mean vector $\overline{\mathbf{w}} = \sigma_n^{-2}\mathbf{A}^{-1}\boldsymbol{\Phi}\mathbf{y}$, where we define the design matrix as $\boldsymbol{\Phi}$ with components the vectors $\boldsymbol{\phi}(\mathbf{x})$. The predictive distribution for a new input $\mathbf{x}_*$ and target value $y_*$ is found by marginalizing over the posterior of the weights:

$$p(y_*|\mathbf{x}_*) = \int p(y_*|\mathbf{x}_*, \mathbf{w}) p(\mathbf{w}|\mathcal{D}) d\mathbf{w}$$

This integral yields a Gaussian predictive distribution with mean value, and associated variance, given by:

$$\mathbb{E}[y_*] = \boldsymbol{\phi}(\mathbf{x}_*)^\top \overline{\mathbf{w}}$$
$$\mathbb{V}[y_*] = \boldsymbol{\phi}(\mathbf{x}_*)^\top \mathbf{A}^{-1} \boldsymbol{\phi}(\mathbf{x}_*)$$

It is these expressions that we shall use in our work. Note that to avoid direct matrix inversion for the matrix $\mathbf{A}$, we use the Cholesky decomposition (makes for more efficient and faster computations).

### 2.4 Feature Construction

The foundation of the model lies in its features, which are constructed within the code in the `_prepare_features_and_target` method. The model's inputs are a combination of past returns and, optionally, any other explanatory variable of one's choice can be added. Let $r_t$ be the return at time $t$. The primary features are a set of lagged returns. For a given time series of returns, the feature vector at time $t$ is:

$$\mathbf{x}_t = [r_{t-1}, r_{t-2}, \ldots, r_{t-n_{\text{lags}}}]^\top$$

The number of lags is defined by the `n_lags` parameter. This process creates a DataFrame 'lagged_returns', where a column 'lag_i' contains the returns from $i$ periods ago. The target variable for the model is the return from the next period, $y_t = r_{t+1}$.

### 2.5 Chebyshev Polynomials for Feature Transformation

The core of the model's complexity is the use of Chebyshev polynomials, as implemented in the `ChebyshevBasis` class. The model does not directly use the raw features. Instead, it transforms them into a higher-dimensional space. The Chebyshev polynomials are defined for the domain $[-1, 1]$. To use them, the `ChebyshevBasis` class first normalizes the input features. For a given input vector $\mathbf{x}$, the normalization function is:

$$\mathbf{x}_{\text{norm}} = \frac{2(\mathbf{x} - \min(\mathbf{x}))}{\max(\mathbf{x}) - \min(\mathbf{x})} - 1$$

This ensures that all input values fall within the correct range for the polynomials. The method then computes the Chebyshev polynomials of the first kind, $T_n(x)$, up to the specified `max_degree`. This is done efficiently using the following recurrence relation:

$$T_0(x) = 1 T_1(x) = x T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \quad \text{for } n \geq 1$$

Each column of the normalized feature matrix is passed to this method, resulting in a design matrix $\Phi$, where the element $\Phi_{ij} = T_j(\mathbf{x}_{\text{norm},i})$ represents a different basis function evaluation for the input data. Specifically, we create the following features $\boldsymbol{\phi}(\mathbf{x}) = [1, \phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \ldots, \phi_d(\mathbf{x})]^\top$ where the components include:

1. **Constant term**: $\phi_0 = 1$
2. **Linear terms**: $\phi_j = x_j$ for $j = 1, \ldots, p$
3. **Higher-order terms**: $\phi_{j,k} = T_k(\tilde{x}_j)$ for degrees $k = 2, \ldots, d$
4. **Interaction terms**: $\phi_{j,\ell} = \tilde{x}_j \cdot \tilde{x}_\ell$ for $j < \ell$

### 2.6 Hyperparameter Optimization

The hyperparameters $\boldsymbol{\theta} = [\sigma_n^2, \sigma_p^2]$ are optimized by maximizing the log marginal likelihood:

$$\log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = -\frac{1}{2}\mathbf{y}^\top \mathbf{A}^{-1}\mathbf{y} - \frac{1}{2}\log|\mathbf{A}| - \frac{n}{2}\log(2\pi)$$

We use L-BFGS-B optimization with log-transformed hyperparameters to ensure their positivity:

$$\tilde{\theta}_1 = \log(\sigma_n^2), \tilde{\theta}_2 = \log(\sigma_p^2)$$

and the optimization problem is solved with given bounds on the transformed variance parameters.

# 3 Implementation Guide

## 3.1 System Requirements

**Prerequisites**

**Required Python Packages:**

- `numpy` $\geq$ 1.20.0
- `pandas` $\geq$ 1.3.0
- `matplotlib` $\geq$ 3.3.0
- `seaborn` $\geq$ 0.11.0
- `scipy` $\geq$ 1.7.0
- `yfinance` $\geq$ 0.1.87

**Python Version:** 3.7+

## 3.2 Basic Usage

### 3.2.1 Simple Execution

The simplest way to use the system is to run the main function:

```
# Run with default parameters
python GPT_v3.py

# Follow the interactive prompts:
# - Enter ticker symbol (e.g., SPY, QQQ, AAPL)
# - Enter start date (YYYY-MM-DD)
# - Enter end date (YYYY-MM-DD)
# - Select frequency (D/W/M)
# - Choose rolling window length
```

### 3.2.2 Programmatic Usage

For integration into larger systems:

```
import numpy as np
import yfinance as yf
from GPT_v3 import GPTradingSystem

# Download data
data = yf.download("SPY", start="2020-01-01", end="
    2024-12-31")
prices = data['Close']

# Initialize system
trading_system = GPTradingSystem(
n_lags=5,               # Number of lagged returns
max_degree=3,           # Maximum polynomial degree
lookback_window=63,     # Rolling window size
min_periods=21,         # Minimum periods for training
transaction_cost=0.001, # Transaction cost (10 bps)
frequency='D'           # Daily frequency
```

```
)

# Run backtest
results = trading_system.backtest(prices,
    optimize_hyperparams=True)

# Display results
trading_system.plot_results(results)
```

### 3.3 Parameter Configuration

| Parameter | Default | Description |
|---|---|---|
| `n_lags` | user input | Number of lagged return features |
| `max_degree` | user input | Maximum Chebyshev polynomial degree |
| `lookback_window` | defined by frequency | Rolling training window size |
| `min_periods` | 10 or defined by frequency | Minimum observations for training |
| `transaction_cost` | user defined | Round-trip transaction cost |
| `frequency` | user defined | Data frequency (D/W/M) |

Table 1: Core System Parameters

The system automatically adjusts parameters based on data frequency:

| Frequency | Lookback Window | Min Periods | Annualization |
|---|---|---|---|
| Daily (D) | 63 periods | 21 periods | 252 |
| Weekly (W) | 26 periods | 8 periods | 52 |
| Monthly (M) | 12 periods | 6 periods | 12 |

Table 2: Frequency-Specific Parameter Defaults

### 3.4 Advanced Configuration

#### 3.4.1 Custom Basis Functions

To implement custom basis functions, inherit from the `BasisFunction` class:

```
from GPT_v3 import BasisFunction

class CustomBasis(BasisFunction):
def __init__(self, n_components=10):
self.n_components = n_components

def compute(self, x, fit=False):
# Implement custom basis computation
return basis_matrix

def get_num_features(self):
return self.n_components

# Use custom basis
custom_basis = CustomBasis(n_components=15)
gp = GaussianProcessRegressor(basis_function=
    custom_basis)
```

### 3.4.2 Custom Trading Strategies

Implement custom signal generation by inheriting from `TradingStrategy`:

```
from GPT_v3 import TradingStrategy

class ConfidenceTradingStrategy(TradingStrategy):
def __init__(self, confidence_threshold=0.5, **kwargs):
super().__init__(**kwargs)
self.confidence_threshold = confidence_threshold

def generate_signal(self, prediction, uncertainty=None)
    :
if uncertainty is None:
return np.sign(prediction)

confidence = 1.0 / (1.0 + uncertainty)

if confidence < self.confidence_threshold:
return 0  # No trade

return np.sign(prediction)
```

# 4 Appendix: Code Examples

## 4.1 Complete Example Script

```
#!/usr/bin/env python3
"""
Complete example of GP Trading System usage
"""

import numpy as np
import pandas as pd
import yfinance as yf
from GPT_v3 import GPTradingSystem

def main():
# Configuration
TICKER = "QQQ"
START_DATE = "2020-01-01"
END_DATE = "2024-08-01"
FREQUENCY = "D"

print(f"Running GP Trading System for {TICKER}")

# Download data
data = yf.download(TICKER, start=START_DATE, end=
    END_DATE)
prices = data['Close']

# Initialize trading system
system = GPTradingSystem(
n_lags=4,
max_degree=3,
lookback_window=63,
min_periods=21,
transaction_cost=0.0005,
frequency=FREQUENCY
)

# Run backtest
results = system.backtest(prices, optimize_hyperparams=
    True)

# Display results
system.plot_results(results)

return results

if __name__ == "__main__":
results = main()
```

## 4.2 Custom Metric Calculation

```
def calculate_custom_metrics(returns):
"""Calculate additional performance metrics"""

# Value at Risk (95%)
var_95 = np.percentile(returns, 5)
```

```
                # Expected Shortfall (95%)
                es_95 = np.mean(returns[returns <= var_95])

                # Skewness
                skewness = pd.Series(returns).skew()

                # Kurtosis
                kurtosis = pd.Series(returns).kurtosis()

                return {
                        'VaR_95': var_95,
                        'ES_95': es_95,
                        'Skewness': skewness,
                        'Kurtosis': kurtosis
                }

                # Example usage
                strategy_returns = results['strategy_returns']
                valid_returns = strategy_returns[~np.isnan(
                    strategy_returns)]
                custom_metrics = calculate_custom_metrics(valid_returns
                    )
                print("Custom Metrics:", custom_metrics)
```

### 4.3 Parameter Sensitivity Analysis

```
                def parameter_sensitivity_analysis(prices):
                """Analyze sensitivity to key parameters"""

                results_grid = {}

                # Parameter ranges
                n_lags_range = [2, 3, 4, 5, 6]
                max_degree_range = [2, 3, 4, 5]
                lookback_range = [40, 63, 100, 150]

                for n_lags in n_lags_range:
                for max_degree in max_degree_range:
                for lookback in lookback_range:
                key = f"lags_{n_lags}_degree_{max_degree}_window_{
                    lookback}"

                try:
                system = GPTradingSystem(
                n_lags=n_lags,
                max_degree=max_degree,
                lookback_window=lookback,
                min_periods=20,
                frequency='D'
                )

                result = system.backtest(prices, optimize_hyperparams=
                    False)
                metrics = result['strategy_metrics']

                results_grid[key] = {
```

```
                'sharpe': metrics.sharpe_ratio,
                'total_return': metrics.total_return,
                'max_drawdown': metrics.max_drawdown
            }

            except Exception as e:
            print(f"Failed for {key}: {e}")

            return results_grid

            # Usage
            sensitivity_results = parameter_sensitivity_analysis(
                prices)
```

# 5 Advanced Topics

## 5.1 Multi-Asset Portfolio Extension

The system can be extended to handle multiple assets simultaneously:

### 5.1.1 Mathematical Framework

For a universe of $N$ assets, we model the return vector $\mathbf{r}_t = [r_{1,t}, r_{2,t}, \ldots, r_{N,t}]^T$ using separate GP models:

$$r_{i,t}|\mathbf{X}_{i,t} \sim \mathcal{GP}(m_i(\mathbf{X}_{i,t}), k_i(\mathbf{X}_{i,t}, \mathbf{X}_{i,t'})) \tag{1}$$

The portfolio return becomes:

$$r_{p,t} = \mathbf{w}_t^T \mathbf{r}_t \tag{2}$$

$$\mathbb{E}[r_{p,t}] = \mathbf{w}_t^T \boldsymbol{\mu}_{GP,t} \tag{3}$$

$$\text{Var}[r_{p,t}] = \mathbf{w}_t^T \boldsymbol{\Sigma}_{GP,t} \mathbf{w}_t \tag{4}$$

where $\boldsymbol{\mu}_{GP,t}$ and $\boldsymbol{\Sigma}_{GP,t}$ are the GP predictions and covariance matrix.

### 5.1.2 Implementation

```
            class MultiAssetGPSystem:
            def __init__(self, tickers, **kwargs):
            self.tickers = tickers
            self.systems = {}

            for ticker in tickers:
            self.systems[ticker] = GPTradingSystem(**kwargs)

            def backtest_portfolio(self, price_data, weights=None):
            if weights is None:
            weights = np.ones(len(self.tickers)) / len(self.tickers
                )

            portfolio_results = {}
            individual_results = {}

            for i, ticker in enumerate(self.tickers):
```

```
                prices = price_data[ticker]
                result = self.systems[ticker].backtest(prices)
                individual_results[ticker] = result

                # Combine results with portfolio weights
                portfolio_returns = self._combine_returns(
                    individual_results, weights)
                return portfolio_returns, individual_results

                def _combine_returns(self, results, weights):
                # Implementation for portfolio aggregation
                pass
```

## 5.2 Regime-Aware Extensions

### 5.2.1 Hidden Markov Models

Incorporate regime switching through HMM:

$$S_t|S_{t-1} \sim \text{Categorical}(\boldsymbol{\pi}_{S_{t-1}}) \tag{5}$$

$$\boldsymbol{\theta}_t|S_t = \boldsymbol{\theta}_{S_t} \tag{6}$$

$$r_t|\mathbf{X}_t, S_t \sim \mathcal{GP}(m_{S_t}(\mathbf{X}_t), k_{S_t}(\mathbf{X}_t, \mathbf{X}_t')) \tag{7}$$

### 5.2.2 Volatility Clustering

Use realized volatility as an additional feature:

$$\text{RV}_t = \sum_{j=1}^{M} r_{t-1+j/M}^2 \tag{8}$$

$$\mathbf{X}_t = [r_{t-p}, \ldots, r_{t-1}, \text{RV}_t, \text{RV}_{t-1}, \ldots]^T \tag{9}$$

## 5.3 Online Learning

### 5.3.1 Recursive Updates

For real-time applications, implement recursive GP updates:

**Algorithm 1** Recursive GP Update

**Require:** New observation $(\mathbf{x}_{n+1}, y_{n+1})$
**Require:** Previous Cholesky factor $\mathbf{L}_n$
1: Compute $\mathbf{k}_* = k(\mathbf{X}_n, \mathbf{x}_{n+1})$
2: Solve $\mathbf{L}_n \boldsymbol{\ell} = \mathbf{k}_*$
3: Compute $c = k(\mathbf{x}_{n+1}, \mathbf{x}_{n+1}) - \boldsymbol{\ell}^T \boldsymbol{\ell}$
4: Update $\mathbf{L}_{n+1} = \begin{bmatrix} \mathbf{L}_n & \mathbf{0} \\ \boldsymbol{\ell}^T & \sqrt{c} \end{bmatrix}$
5: **return** Updated $\mathbf{L}_{n+1}$

```
                class OnlineGPTradingSystem(GPTradingSystem):
                def __init__(self, **kwargs):
                super().__init__(**kwargs)
                self.online_mode = True
                self.buffer_size = 1000
```

```
        def update_model(self, new_x, new_y):
        """Update GP model with new observation"""
        # Add to buffer
        self.X_buffer = np.vstack([self.X_buffer, new_x])
        self.y_buffer = np.append(self.y_buffer, new_y)

        # Maintain buffer size
        if len(self.y_buffer) > self.buffer_size:
        self.X_buffer = self.X_buffer[-self.buffer_size:]
        self.y_buffer = self.y_buffer[-self.buffer_size:]

        # Refit model
        self.gp.fit(self.X_buffer, self.y_buffer)
```

# 6 Validation and Testing

## 6.1 Cross-Validation Framework

### 6.1.1 Walk-Forward Analysis

Implement proper time series cross-validation:

**Algorithm 2** Walk-Forward Cross-Validation

**Require:** Time series data $\{(\mathbf{x}_t, y_t)\}_{t=1}^{T}$
**Require:** Initial training size $n_0$, test size $h$, step size $s$
1: Initialize results list $\mathcal{R} = []$
2: **for** $k = 0, s, 2s, \ldots$ while $n_0 + k + h \leq T$ **do**
3: train_idx $= [1, 2, \ldots, n_0 + k]$
4: test_idx $= [n_0 + k + 1, \ldots, n_0 + k + h]$
5: Fit model on $\{(\mathbf{x}_t, y_t)\}_{t \in \text{train\_idx}}$
6: Predict on $\{(\mathbf{x}_t, y_t)\}_{t \in \text{test\_idx}}$
7: Store results in $\mathcal{R}$
8: **end for**
9: **return** Aggregated results from $\mathcal{R}$

```
        def walk_forward_validation(prices, n_folds=10):
        """Perform walk-forward cross-validation"""

        n_total = len(prices)
        fold_size = n_total // n_folds

        results = []

        for i in range(n_folds - 2):  # Leave room for test set
        train_end = (i + 2) * fold_size
        test_start = train_end
        test_end = test_start + fold_size

        if test_end > n_total:
        break

        train_prices = prices.iloc[:train_end]
        test_prices = prices.iloc[test_start:test_end]
```

```
        # Create system for this fold
        system = GPTradingSystem(
        n_lags=3,
        max_degree=3,
        lookback_window=min(63, len(train_prices)//2),
        min_periods=20
        )

        # Train on training set
        train_result = system.backtest(train_prices)

        # Test on test set (using last model)
        # Implementation would require model persistence

        results.append({
                'fold': i,
                'train_sharpe': train_result['strategy_metrics'
                    ].sharpe_ratio,
                'train_return': train_result['strategy_metrics'
                    ].total_return
        })

        return results
```

## 6.2 Statistical Significance Testing

### 6.2.1 Bootstrap Analysis

Test strategy robustness through bootstrap resampling:

$$\text{Bootstrap Sharpe} = \frac{1}{B}\sum_{b=1}^{B}\text{Sharpe}(\mathbf{r}_b^*) \tag{10}$$

where $\mathbf{r}_b^*$ are bootstrap samples of strategy returns.

```
        def bootstrap_analysis(strategy_returns, n_bootstrap
            =1000):
        """Bootstrap analysis of strategy performance"""

        valid_returns = strategy_returns[~np.isnan(
            strategy_returns)]
        n_obs = len(valid_returns)

        bootstrap_sharpes = []
        bootstrap_returns = []

        for b in range(n_bootstrap):
        # Bootstrap sample
        boot_indices = np.random.choice(n_obs, size=n_obs,
            replace=True)
        boot_returns = valid_returns[boot_indices]

        # Calculate metrics
        if np.std(boot_returns) > 0:
        boot_sharpe = np.mean(boot_returns) / np.std(
            boot_returns) * np.sqrt(252)
```

```
                boot_total_return = np.prod(1 + boot_returns) - 1
                else:
                boot_sharpe = 0
                boot_total_return = 0

                bootstrap_sharpes.append(boot_sharpe)
                bootstrap_returns.append(boot_total_return)

                return {
                        'sharpe_mean': np.mean(bootstrap_sharpes),
                        'sharpe_std': np.std(bootstrap_sharpes),
                        'sharpe_ci_95': np.percentile(bootstrap_sharpes
                           , [2.5, 97.5]),
                        'return_mean': np.mean(bootstrap_returns),
                        'return_ci_95': np.percentile(bootstrap_returns
                           , [2.5, 97.5])
                }
```

## 6.3 Model Diagnostics

### 6.3.1 Residual Analysis

Examine model residuals for patterns:

```
                def analyze_residuals(results):
                """Analyze GP model residuals"""

                predictions = results['predictions']
                targets = results['targets']

                valid_mask = ~np.isnan(predictions)
                residuals = targets[valid_mask] - predictions[
                   valid_mask]

                # Statistical tests
                from scipy import stats

                # Normality test
                _, p_normal = stats.jarque_bera(residuals)

                # Autocorrelation test (Ljung-Box)
                from statsmodels.stats.diagnostic import acorr_ljungbox
                lb_stat, lb_pvalue = acorr_ljungbox(residuals, lags=10,
                    return_df=False)

                # Heteroscedasticity test (Breusch-Pagan)
                from statsmodels.stats.diagnostic import
                   het_breuschpagan
                bp_stat, bp_pvalue, _, _ = het_breuschpagan(residuals,
                predictions[valid_mask].reshape(-1, 1))

                return {
                        'residual_mean': np.mean(residuals),
                        'residual_std': np.std(residuals),
                        'normality_pvalue': p_normal,
                        'ljung_box_pvalue': lb_pvalue[-1],  # Last lag
                           p-value
                        'heteroscedasticity_pvalue': bp_pvalue,
```

```
                'residuals': residuals
            }
```

# 7 Risk Management Extensions

## 7.1 Dynamic Position Sizing

### 7.1.1 Kelly Criterion

Optimal position sizing based on GP predictions:

$$f^* = \frac{p \cdot b - q}{b} \tag{11}$$

where:

- $p = P(\text{prediction} > 0)$ (probability of win)
- $q = 1 - p$ (probability of loss)
- $b =$ average win / average loss ratio

For GP predictions with uncertainty:

$$p = \Phi\left(\frac{\mu_{GP}}{\sigma_{GP}}\right) \tag{12}$$

$$f^* = \frac{\mu_{GP}}{\sigma^2_{\text{returns}}} \tag{13}$$

```
            class KellyPositionSizer:
            def __init__(self, max_position=0.25, min_confidence
                =0.1):
            self.max_position = max_position
            self.min_confidence = min_confidence

            def calculate_position(self, prediction, uncertainty,
                return_vol):
            """Calculate Kelly-optimal position size"""

            if uncertainty <= 0:
            return 0

            # Convert to probability using normal CDF
            from scipy.stats import norm
            prob_positive = norm.cdf(prediction / uncertainty)

            # Kelly fraction (simplified)
            if return_vol > 0:
            kelly_fraction = prediction / (return_vol**2)
            else:
            kelly_fraction = 0

            # Apply confidence and size limits
            confidence = 1.0 / (1.0 + uncertainty)

            if confidence < self.min_confidence:
            return 0
```

```
                position = np.clip(kelly_fraction * confidence,
                -self.max_position, self.max_position)

                return position

                # Integration with GP system
                class RiskManagedGPSystem(GPTradingSystem):
                def __init__(self, position_sizer=None, **kwargs):
                super().__init__(**kwargs)
                self.position_sizer = position_sizer or
                    KellyPositionSizer()

                def generate_signal_with_sizing(self, prediction,
                    uncertainty, return_vol):
                """Generate position-sized signal"""
                return self.position_sizer.calculate_position(
                prediction, uncertainty, return_vol)
```

## 7.2 Risk Budgeting

### 7.2.1 Volatility Targeting

Adjust position sizes to target specific volatility:

$$w_t = \frac{\sigma_{\text{target}}}{\sigma_{\text{forecast},t}} \cdot \text{signal}_t \tag{14}$$

```
                class VolatilityTargeting:
                def __init__(self, target_vol=0.15, lookback=20):
                self.target_vol = target_vol
                self.lookback = lookback

                def adjust_position(self, base_position, return_history
                    ):
                """Adjust position for volatility targeting"""

                if len(return_history) < self.lookback:
                return base_position

                # Estimate current volatility
                recent_returns = return_history[-self.lookback:]
                current_vol = np.std(recent_returns) * np.sqrt(252)

                if current_vol <= 0:
                return 0

                # Volatility scalar
                vol_scalar = self.target_vol / current_vol

                # Apply to base position
                return base_position * vol_scalar
```

# 8 Performance Attribution

## 8.1 Factor Decomposition

Decompose strategy returns into factor exposures:

$$r_{s,t} = \alpha + \sum_{j=1}^{K} \beta_j f_{j,t} + \epsilon_t \tag{15}$$

where $f_{j,t}$ are factor returns (market, size, value, etc.).

```
def factor_attribution(strategy_returns, factor_data):
"""Perform factor attribution analysis"""

import statsmodels.api as sm

# Align dates
aligned_data = pd.concat([strategy_returns, factor_data
    ], axis=1).dropna()

y = aligned_data.iloc[:, 0]  # Strategy returns
X = aligned_data.iloc[:, 1:]  # Factor returns
X = sm.add_constant(X)  # Add intercept

# Regression
model = sm.OLS(y, X).fit()

# Attribution
factor_contributions = {}
for i, factor_name in enumerate(X.columns[1:], 1):  #
    Skip constant
beta = model.params[factor_name]
factor_return = X[factor_name].mean()
contribution = beta * factor_return
factor_contributions[factor_name] = {
        'beta': beta,
        'avg_factor_return': factor_return,
        'contribution': contribution,
        't_stat': model.tvalues[factor_name],
        'p_value': model.pvalues[factor_name]
}

return {
        'alpha': model.params['const'],
        'alpha_tstat': model.tvalues['const'],
        'alpha_pvalue': model.pvalues['const'],
        'r_squared': model.rsquared,
        'factor_contributions': factor_contributions,
        'model': model
}
```