Module 4: Statistical Arbitrage Frameworks

Course Information

- **Duration:** 170 minutes
- **Level:** Advanced
- **Prerequisites:** Time Series Analysis (Module 3), Statistics, Linear Algebra
- **Learning Path:** Advanced Mathematical Frameworks → Module 4

Learning Objectives

By the end of this module, you will be able to:

1. **Implement Cointegration Analysis** for DeFi pairs trading strategies
2. **Build Mean Reversion Models** for statistical arbitrage in cryptocurrency markets
3. **Design Portfolio Construction** algorithms for multi-asset statistical arbitrage
4. **Apply Kalman Filtering** for dynamic hedge ratio estimation
5. **Develop Risk Management** frameworks for statistical arbitrage
6. **Create Performance Attribution** models for strategy analysis

---

Module Overview

Statistical arbitrage leverages mathematical relationships between assets to generate profit from temporary price discrepancies. This module focuses on advanced statistical techniques for identifying, modeling, and trading these relationships in DeFi and cryptocurrency markets.

Key Topics Covered

- **Cointegration and Error Correction Models**
- **Pairs Trading Strategies**
- **Mean Reversion and Ornstein-Uhlenbeck Processes**
- **Kalman Filters for Dynamic Relationships**
- **Portfolio Optimization for Statistical Arbitrage**
- **Risk Management and Position Sizing**
- **Performance Attribution and Factor Models**

---

1. Foundations of Statistical Arbitrage

1.1 Cointegration Theory and Testing

Cointegration forms the theoretical foundation for statistical arbitrage, identifying long-term equilibrium relationships between asset prices.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from scipy.optimize import minimize
from statsmodels.tsa.stattools import coint, adfuller
from statsmodels.tsa.vector_ar.vecm import coint_johansen
from statsmodels.regression.linear_model import OLS
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.stats.diagnostic import het_breuschpagan
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
import warnings

def setup_matplotlib_for_plotting():
    """Setup matplotlib and seaborn for plotting with proper
configuration."""
    warnings.filterwarnings('default')
    plt.switch_backend("Agg")
    plt.style.use("seaborn-v0_8")
    sns.set_palette("husl")
    plt.rcParams["font.sans-serif"] = ["Noto Sans CJK SC", "WenQuanYi
Zen Hei", "PingFang SC", "Arial Unicode MS", "Hiragino Sans GB"]
    plt.rcParams["axes.unicode_minus"] = False

setup_matplotlib_for_plotting()

class StatisticalArbitrageFramework:
    """
    Comprehensive statistical arbitrage framework for DeFi and
cryptocurrency markets
    """

    def __init__(self, price_data: pd.DataFrame):
        """
        Initialize statistical arbitrage framework

        Args:
            price_data: DataFrame with datetime index and price columns
        """
        self.price_data = price_data.copy()
```

```
        self.price_data.index = pd.to_datetime(self.price_data.index)
        self.log_prices = np.log(self.price_data)
        self.returns = self.price_data.pct_change().dropna()
        self.cointegration_results = {}
        self.models = {}

    def test_cointegration_pairs(self, method: str = 'engle_granger',
                                 significance_level: float = 0.05) ->
dict:
        """
        Test all asset pairs for cointegration relationships
        """
        assets = self.price_data.columns.tolist()
        n_assets = len(assets)
        results = {}

        if method == 'engle_granger':
            # Pairwise Engle-Granger cointegration tests
            for i in range(n_assets):
                for j in range(i + 1, n_assets):
                    asset_1, asset_2 = assets[i], assets[j]

                    # Extract price series
                    y = self.log_prices[asset_1].dropna()
                    x = self.log_prices[asset_2].dropna()

                    # Align series
                    aligned_data = pd.DataFrame({'y': y, 'x':
x}).dropna()
                    if len(aligned_data) < 50:  # Minimum data
requirement
                        continue

                    y_aligned = aligned_data['y']
                    x_aligned = aligned_data['x']

                    # Engle-Granger test
                    score, pvalue, crit_values = coint(y_aligned,
x_aligned)

                    # OLS regression for cointegrating relationship
                    ols_model = OLS(y_aligned,
```

```
sm.add_constant(x_aligned)).fit()
                    residuals = ols_model.resid

                    # Test residuals for stationarity
                    adf_stat, adf_pvalue, _, _, adf_crit, _ =
adfuller(residuals, regression='c')

                    pair_result = {
                        'assets': (asset_1, asset_2),
                        'coint_score': score,
                        'coint_pvalue': pvalue,
                        'is_cointegrated': pvalue < significance_level,
                        'critical_values': crit_values,
                        'beta': ols_model.params[1],
                        'alpha': ols_model.params[0],
                        'residuals': residuals,
                        'adf_stat': adf_stat,
                        'adf_pvalue': adf_pvalue,
                        'half_life':
self._calculate_half_life(residuals)
                    }

                    pair_key = f"{asset_1}_{asset_2}"
                    results[pair_key] = pair_result

        elif method == 'johansen':
            # Johansen cointegration test for multiple assets
            try:
                # Use all assets for multivariate test
                data_matrix = self.log_prices.dropna()
                johansen_result = coint_johansen(data_matrix,
det_order=0, k_ar_diff=1)

                results['johansen'] = {
                    'trace_stats': johansen_result.lr1,
                    'max_eigen_stats': johansen_result.lr2,
                    'critical_values_trace': johansen_result.cvt,
                    'critical_values_max_eigen': johansen_result.cvm,
                    'eigenvectors': johansen_result.evec,
                    'eigenvalues': johansen_result.eig,
                    'n_cointegrating_relationships':
self._count_johansen_relationships(johansen_result)
```

```
                }
            except Exception as e:
                results['johansen'] = {'error': str(e)}

        self.cointegration_results = results
        return results

    def _calculate_half_life(self, residuals: pd.Series) -> float:
        """
        Calculate half-life of mean reversion for residuals
        """
        try:
            # Fit AR(1) model to residuals
            lagged_residuals = residuals.shift(1).dropna()
            current_residuals = residuals[1:]

            # Ensure same length
            min_len = min(len(lagged_residuals),
len(current_residuals))
            lagged_residuals = lagged_residuals.iloc[:min_len]
            current_residuals = current_residuals.iloc[:min_len]

            # OLS regression: y_t = α + φ*y_{t-1} + ε_t
            model =
LinearRegression().fit(lagged_residuals.values.reshape(-1, 1),
current_residuals.values)
            phi = model.coef_[0]

            if phi >= 1 or phi <= 0:
                return np.inf  # No mean reversion

            # Half-life calculation
            half_life = -np.log(2) / np.log(phi)
            return half_life

        except Exception:
            return np.inf

    def _count_johansen_relationships(self, johansen_result) -> int:
        """
        Count number of cointegrating relationships from Johansen test
        """
```

```
        trace_stats = johansen_result.lr1
        critical_values = johansen_result.cvt[:, 1]  # 5% significance
level

        n_relationships = 0
        for i, (stat, crit_val) in enumerate(zip(trace_stats,
critical_values)):
            if stat > crit_val:
                n_relationships = len(trace_stats) - i
                break

        return n_relationships

# Import statsmodels properly
import statsmodels.api as sm

# Generate synthetic DeFi token price data for demonstration
np.random.seed(42)
dates = pd.date_range('2023-01-01', periods=500, freq='D')

# Create cointegrated price series
# Base price process
base_factor = np.cumsum(np.random.normal(0, 0.02, 500))

# ETH price (base asset)
eth_noise = np.cumsum(np.random.normal(0, 0.03, 500))
eth_prices = 2000 * np.exp(base_factor + eth_noise)

# WETH price (nearly identical to ETH with small tracking error)
weth_tracking_error = np.cumsum(np.random.normal(0, 0.005, 500))
weth_prices = eth_prices * np.exp(weth_tracking_error)

# USDC price (stablecoin with small variations)
usdc_noise = np.cumsum(np.random.normal(0, 0.001, 500))
usdc_prices = 1.0 + 0.02 * np.sin(np.arange(500) / 50) + usdc_noise

# DAI price (cointegrated with USDC but with temporary divergences)
dai_spread = np.random.normal(0, 0.01, 500)
dai_prices = usdc_prices + dai_spread

# BTC price (partially correlated with ETH)
btc_factor = 0.7 * base_factor + 0.3 * np.cumsum(np.random.normal(0,
```

```
0.04, 500))
btc_prices = 40000 * np.exp(btc_factor)

# Create DataFrame
defi_prices = pd.DataFrame({
    'ETH': eth_prices,
    'WETH': weth_prices,
    'USDC': usdc_prices,
    'DAI': dai_prices,
    'BTC': btc_prices
}, index=dates)

# Initialize statistical arbitrage framework
stat_arb = StatisticalArbitrageFramework(defi_prices)

print("Statistical Arbitrage Framework Analysis:")
print("=" * 50)

# Test pairwise cointegration
print("\n--- Cointegration Analysis ---")
coint_results =
stat_arb.test_cointegration_pairs(method='engle_granger')

# Display cointegrated pairs
cointegrated_pairs = []
for pair_name, result in coint_results.items():
    if result['is_cointegrated']:
        cointegrated_pairs.append((pair_name, result))
        print(f"\nCointegrated Pair: {result['assets'][0]} -
{result['assets'][1]}")
        print(f"  P-value: {result['coint_pvalue']:.4f}")
        print(f"  Beta (hedge ratio): {result['beta']:.4f}")
        print(f"  Half-life: {result['half_life']:.2f} days")

print(f"\nTotal cointegrated pairs found: {len(cointegrated_pairs)}")

# Johansen test for multivariate cointegration
print("\n--- Johansen Multivariate Cointegration ---")
johansen_results = stat_arb.test_cointegration_pairs(method='johansen')
if 'error' not in johansen_results.get('johansen', {}):
    johansen_data = johansen_results['johansen']
    print(f"Number of cointegrating relationships:
```

```
{johansen_data['n_cointegrating_relationships']}")
    print(f"Eigenvalues: {johansen_data['eigenvalues'][:3]}")  # Show
first 3
```

---

2. Pairs Trading Strategies

2.1 Mean Reversion Models and Trading Signals

```
class PairsTradingStrategy:
    """
    Comprehensive pairs trading strategy implementation
    """

    def __init__(self, asset1_prices: pd.Series, asset2_prices:
pd.Series,
                 lookback_period: int = 60):
        """
        Initialize pairs trading strategy

        Args:
            asset1_prices: Price series for first asset
            asset2_prices: Price series for second asset
            lookback_period: Rolling window for parameter estimation
        """
        self.asset1 = asset1_prices.name or 'Asset1'
        self.asset2 = asset2_prices.name or 'Asset2'
        self.prices_1 = asset1_prices
        self.prices_2 = asset2_prices
        self.lookback_period = lookback_period

        # Align price series
        self.aligned_data = pd.DataFrame({
            self.asset1: self.prices_1,
            self.asset2: self.prices_2
        }).dropna()

        self.log_prices = np.log(self.aligned_data)
        self.spread_series = None
        self.hedge_ratios = None
        self.trading_signals = None

    def calculate_dynamic_hedge_ratio(self, method: str =
'rolling_ols') -> pd.Series:
        """
        Calculate dynamic hedge ratio using various methods
        """
        if method == 'rolling_ols':
            # Rolling OLS regression
            hedge_ratios = []
```

```
            for i in range(self.lookback_period, len(self.log_prices)):
                window_data = self.log_prices.iloc[i-
self.lookback_period:i]
                y = window_data[self.asset1]
                x = window_data[self.asset2]

                # OLS regression
                model = LinearRegression().fit(x.values.reshape(-1, 1),
y.values)
                hedge_ratio = model.coef_[0]
                hedge_ratios.append(hedge_ratio)

            # Create series with proper index
            ratio_index = self.log_prices.index[self.lookback_period:]
            self.hedge_ratios = pd.Series(hedge_ratios,
index=ratio_index)

        elif method == 'kalman_filter':
            # Kalman filter for dynamic hedge ratio
            self.hedge_ratios = self._kalman_filter_hedge_ratio()

        return self.hedge_ratios

    def _kalman_filter_hedge_ratio(self) -> pd.Series:
        """
        Implement Kalman filter for dynamic hedge ratio estimation
        """
        # Simplified Kalman filter implementation
        n = len(self.log_prices)
        hedge_ratios = np.zeros(n)

        # Initialize state
        hedge_ratio = 1.0  # Initial hedge ratio
        error_cov = 1.0    # Initial error covariance

        # Kalman filter parameters
        process_noise = 0.0001  # Process noise variance
        observation_noise = 0.01  # Observation noise variance

        prices_1 = self.log_prices[self.asset1].values
        prices_2 = self.log_prices[self.asset2].values
```

```
        for t in range(1, n):
            # Predict step
            hedge_ratio_pred = hedge_ratio  # Random walk model
            error_cov_pred = error_cov + process_noise

            # Update step
            innovation = prices_1[t] - hedge_ratio_pred * prices_2[t]
            innovation_cov = error_cov_pred * prices_2[t]**2 +
observation_noise

            # Kalman gain
            kalman_gain = error_cov_pred * prices_2[t] / innovation_cov

            # State update
            hedge_ratio = hedge_ratio_pred + kalman_gain * innovation
            error_cov = (1 - kalman_gain * prices_2[t]) *
error_cov_pred

            hedge_ratios[t] = hedge_ratio

        return pd.Series(hedge_ratios[1:],
index=self.log_prices.index[1:])

    def calculate_spread(self, hedge_ratio_method: str = 'rolling_ols')
-> pd.Series:
        """
        Calculate spread series using dynamic hedge ratios
        """
        # Calculate hedge ratios
        hedge_ratios =
self.calculate_dynamic_hedge_ratio(hedge_ratio_method)

        # Calculate spread
        spreads = []
        for date in hedge_ratios.index:
            if date in self.log_prices.index:
                price_1 = self.log_prices.loc[date, self.asset1]
                price_2 = self.log_prices.loc[date, self.asset2]
                hedge_ratio = hedge_ratios.loc[date]

                spread = price_1 - hedge_ratio * price_2
                spreads.append(spread)
```

```
        self.spread_series = pd.Series(spreads,
index=hedge_ratios.index)
        return self.spread_series

    def generate_trading_signals(self, entry_threshold: float = 2.0,
                                 exit_threshold: float = 0.5,
                                 stop_loss: float = 3.0) -> pd.DataFrame:
        """
        Generate trading signals based on spread statistics
        """
        if self.spread_series is None:
            raise ValueError("Spread series must be calculated first")

        # Calculate rolling statistics
        rolling_mean =
self.spread_series.rolling(window=self.lookback_period).mean()
        rolling_std =
self.spread_series.rolling(window=self.lookback_period).std()

        # Z-score of spread
        z_score = (self.spread_series - rolling_mean) / rolling_std

        # Initialize signals
        signals = pd.DataFrame(index=z_score.index)
        signals['spread'] = self.spread_series
        signals['z_score'] = z_score
        signals['position'] = 0
        signals['entry_signal'] = 0
        signals['exit_signal'] = 0

        current_position = 0

        for i, date in enumerate(z_score.index):
            current_z = z_score.loc[date]

            if current_position == 0:  # No position
                if current_z > entry_threshold:
                    # Spread too high - short spread (short asset1,
long asset2)
                    signals.loc[date, 'position'] = -1
                    signals.loc[date, 'entry_signal'] = -1
```

```
                    current_position = -1
                elif current_z < -entry_threshold:
                    # Spread too low - long spread (long asset1, short
asset2)
                    signals.loc[date, 'position'] = 1
                    signals.loc[date, 'entry_signal'] = 1
                    current_position = 1

            elif current_position == 1:  # Long spread position
                if current_z > -exit_threshold and current_z <
exit_threshold:
                    # Exit condition - spread returned to mean
                    signals.loc[date, 'position'] = 0
                    signals.loc[date, 'exit_signal'] = 1
                    current_position = 0
                elif current_z < -stop_loss:
                    # Stop loss
                    signals.loc[date, 'position'] = 0
                    signals.loc[date, 'exit_signal'] = -1
                    current_position = 0
                else:
                    signals.loc[date, 'position'] = 1

            elif current_position == -1:  # Short spread position
                if current_z > -exit_threshold and current_z <
exit_threshold:
                    # Exit condition
                    signals.loc[date, 'position'] = 0
                    signals.loc[date, 'exit_signal'] = 1
                    current_position = 0
                elif current_z > stop_loss:
                    # Stop loss
                    signals.loc[date, 'position'] = 0
                    signals.loc[date, 'exit_signal'] = -1
                    current_position = 0
                else:
                    signals.loc[date, 'position'] = -1

        self.trading_signals = signals
        return signals

    def backtest_strategy(self, transaction_costs: float = 0.001) ->
```

```
dict:
        """
        Backtest the pairs trading strategy
        """
        if self.trading_signals is None:
            raise ValueError("Trading signals must be generated first")

        signals = self.trading_signals.copy()

        # Calculate returns for each asset
        returns_1 = self.aligned_data[self.asset1].pct_change()
        returns_2 = self.aligned_data[self.asset2].pct_change()

        # Align with signals
        common_dates =
signals.index.intersection(returns_1.index).intersection(returns_2.index)
        signals = signals.loc[common_dates]
        returns_1 = returns_1.loc[common_dates]
        returns_2 = returns_2.loc[common_dates]

        # Calculate hedge ratios for returns calculation
        hedge_ratios =
self.hedge_ratios.reindex(common_dates).fillna(method='ffill')

        # Calculate strategy returns
        strategy_returns = []
        transaction_costs_incurred = []

        prev_position = 0
        for date in common_dates:
            position = signals.loc[date, 'position']
            ret_1 = returns_1.loc[date]
            ret_2 = returns_2.loc[date]
            hedge_ratio = hedge_ratios.loc[date] if date in
hedge_ratios.index else 1.0


# Strategy return = position * (return_asset1 - hedge_ratio *
return_asset2)
            strategy_return = position * (ret_1 - hedge_ratio * ret_2)

            # Transaction costs when position changes
```

```
            position_change = abs(position - prev_position)
            transaction_cost = position_change * transaction_costs

            net_return = strategy_return - transaction_cost

            strategy_returns.append(net_return)
            transaction_costs_incurred.append(transaction_cost)
            prev_position = position

        strategy_returns = pd.Series(strategy_returns,
index=common_dates)

        # Performance metrics
        total_return = (1 + strategy_returns).prod() - 1
        annualized_return = (1 + total_return) ** (252 /
len(strategy_returns)) - 1
        volatility = strategy_returns.std() * np.sqrt(252)
        sharpe_ratio = annualized_return / volatility if volatility > 0
else 0

        # Maximum drawdown
        cumulative_returns = (1 + strategy_returns).cumprod()
        running_max = cumulative_returns.expanding().max()
        drawdown = (cumulative_returns - running_max) / running_max
        max_drawdown = drawdown.min()

        # Trade statistics
        trades = signals[(signals['entry_signal'] != 0) |
(signals['exit_signal'] != 0)]
        n_trades = len(trades[trades['entry_signal'] != 0])

        # Win rate calculation
        if n_trades > 0:
            trade_returns = []
            entry_idx = None

            for i, (date, row) in enumerate(trades.iterrows()):
                if row['entry_signal'] != 0:
                    entry_idx = date
                elif row['exit_signal'] != 0 and entry_idx is not None:
                    trade_period = strategy_returns.loc[entry_idx:date]
                    trade_return = trade_period.sum()
```

```
                    trade_returns.append(trade_return)
                    entry_idx = None

            win_rate = sum(1 for ret in trade_returns if ret > 0) /
len(trade_returns) if trade_returns else 0
            avg_win = np.mean([ret for ret in trade_returns if ret >
0]) if trade_returns else 0
            avg_loss = np.mean([ret for ret in trade_returns if ret <
0]) if trade_returns else 0
        else:
            win_rate = 0
            avg_win = 0
            avg_loss = 0

        return {
            'total_return': total_return,
            'annualized_return': annualized_return,
            'volatility': volatility,
            'sharpe_ratio': sharpe_ratio,
            'max_drawdown': max_drawdown,
            'n_trades': n_trades,
            'win_rate': win_rate,
            'avg_win': avg_win,
            'avg_loss': avg_loss,
            'strategy_returns': strategy_returns,
            'cumulative_returns': cumulative_returns,
            'total_transaction_costs': sum(transaction_costs_incurred)
        }

# Example: Pairs trading strategy for ETH-WETH
print("\n--- Pairs Trading Strategy: ETH-WETH ---")

# Initialize pairs trading strategy
pairs_strategy = PairsTradingStrategy(
    defi_prices['ETH'],
    defi_prices['WETH'],
    lookback_period=30
)

# Calculate spread using rolling OLS
spread =
pairs_strategy.calculate_spread(hedge_ratio_method='rolling_ols')
```

```
print(f"Average spread: {spread.mean():.6f}")
print(f"Spread volatility: {spread.std():.6f}")
print(f"Spread half-life: {stat_arb._calculate_half_life(spread):.2f}
days")

# Generate trading signals
signals = pairs_strategy.generate_trading_signals(
    entry_threshold=1.5,
    exit_threshold=0.5,
    stop_loss=2.5
)

# Count trading signals
n_entries = len(signals[signals['entry_signal'] != 0])
n_exits = len(signals[signals['exit_signal'] != 0])
print(f"Entry signals: {n_entries}")
print(f"Exit signals: {n_exits}")

# Backtest strategy
backtest_results =
pairs_strategy.backtest_strategy(transaction_costs=0.0005)

print(f"\n--- Backtest Results ---")
print(f"Total return: {backtest_results['total_return']*100:.2f}%")
print(f"Annualized return: {backtest_results['annualized_return']*100:.
2f}%")
print(f"Volatility: {backtest_results['volatility']*100:.2f}%")
print(f"Sharpe ratio: {backtest_results['sharpe_ratio']:.3f}")
print(f"Maximum drawdown: {backtest_results['max_drawdown']*100:.2f}%")
print(f"Number of trades: {backtest_results['n_trades']}")
print(f"Win rate: {backtest_results['win_rate']*100:.1f}%")
print(f"Total transaction costs:
{backtest_results['total_transaction_costs']*100:.3f}%")
```

---

3. Mean Reversion Models

3.1 Ornstein-Uhlenbeck Process and Parameter Estimation

```
class MeanReversionModeling:
    """
    Mean reversion modeling for statistical arbitrage
    """

    def __init__(self, price_series: pd.Series):
        """
        Initialize mean reversion modeling framework
        """
        self.price_series = price_series
        self.log_prices = np.log(price_series)
        self.model_params = {}

    def fit_ornstein_uhlenbeck(self, method: str = 'mle') -> dict:
        """
        Fit Ornstein-Uhlenbeck process to price series

        The OU process: dX_t = θ(μ - X_t)dt + σdW_t
        Discrete form: X_{t+1} = X_t + θ(μ - X_t)Δt + σ√Δt * ε_t
        """
        if method == 'mle':
            # Maximum likelihood estimation
            result = self._ou_mle_estimation()
        elif method == 'least_squares':
            # Least squares estimation
            result = self._ou_least_squares()
        else:
            raise ValueError("Method must be 'mle' or 'least_squares'")

        self.model_params['ou'] = result
        return result

    def _ou_mle_estimation(self) -> dict:
        """
        Maximum likelihood estimation for OU process
        """
        X = self.log_prices.values
        n = len(X)
        dt = 1.0  # Assuming daily data, dt = 1

        def negative_log_likelihood(params):
            """Negative log-likelihood function"""
```

```
            theta, mu, sigma = params

            if theta <= 0 or sigma <= 0:
                return np.inf

            # Calculate transitions
            X_lag = X[:-1]
            X_current = X[1:]

            # OU process mean and variance
            mean_transition = X_lag + theta * (mu - X_lag) * dt
            var_transition = sigma**2 * dt

            # Log-likelihood
            residuals = X_current - mean_transition
            log_likelihood = -0.5 * n * np.log(2 * np.pi *
var_transition) - \
                             0.5 * np.sum(residuals**2) / var_transition

            return -log_likelihood

        # Initial parameter guess
        initial_guess = [0.1, np.mean(X), np.std(np.diff(X))]

        # Optimization
        from scipy.optimize import minimize
        result = minimize(negative_log_likelihood, initial_guess,
                          bounds=[(0.001, 10), (None, None), (0.001,
10)],
                          method='L-BFGS-B')

        if result.success:
            theta, mu, sigma = result.x
            half_life = np.log(2) / theta

            return {
                'theta': theta,
                'mu': mu,
                'sigma': sigma,
                'half_life': half_life,
                'log_likelihood': -result.fun,
                'aic': 2 * 3 - 2 * (-result.fun),  # 2k - 2ln(L)
```

```
                'success': True
            }
        else:
            return {'success': False, 'error': result.message}

    def _ou_least_squares(self) -> dict:
        """
        Least squares estimation for OU process using discrete
approximation
        """
        X = self.log_prices.values
        dt = 1.0

        # Discrete OU: X_{t+1} - X_t = θμΔt - θX_t*Δt + σ√Δt * ε_t
        # Rearrange: ΔX_t = α + β*X_t + ε_t where α = θμΔt, β = -θΔt

        X_lag = X[:-1]
        delta_X = np.diff(X)

        # OLS regression
        X_matrix = np.column_stack([np.ones(len(X_lag)), X_lag])

        try:
            params = np.linalg.lstsq(X_matrix, delta_X, rcond=None)[0]
            alpha, beta = params

            # Convert back to OU parameters
            theta = -beta / dt
            mu = alpha / (theta * dt) if theta != 0 else 0

            # Estimate sigma from residuals
            fitted_delta_X = X_matrix @ params
            residuals = delta_X - fitted_delta_X
            sigma = np.sqrt(np.var(residuals) / dt)

            # Calculate R-squared
            ss_res = np.sum(residuals**2)
            ss_tot = np.sum((delta_X - np.mean(delta_X))**2)
            r_squared = 1 - (ss_res / ss_tot)

            half_life = np.log(2) / theta if theta > 0 else np.inf
```

```
            return {
                'theta': theta,
                'mu': mu,
                'sigma': sigma,
                'half_life': half_life,
                'r_squared': r_squared,
                'alpha': alpha,
                'beta': beta,
                'residuals': residuals,
                'success': True
            }
        except Exception as e:
            return {'success': False, 'error': str(e)}

    def forecast_ou_process(self, steps: int, current_value: float =
None) -> dict:
        """
        Forecast using fitted OU process
        """
        if 'ou' not in self.model_params:
            raise ValueError("OU model must be fitted first")

        params = self.model_params['ou']
        if not params['success']:
            raise ValueError("OU model fitting was not successful")

        theta = params['theta']
        mu = params['mu']
        sigma = params['sigma']

        if current_value is None:
            current_value = self.log_prices.iloc[-1]

        dt = 1.0  # Daily forecast
        forecasts = []
        forecast_vars = []

        for step in range(1, steps + 1):
            # OU process forecast
            t = step * dt

            # Mean forecast: E[X_t | X_0] = μ + (X_0 - μ)e^{-θt}
```

```
            forecast_mean = mu + (current_value - mu) * np.exp(-theta *
t)

            # Variance forecast: Var[X_t | X_0] = σ²/(2θ) * (1 -
e^{-2θt})
            forecast_var = (sigma**2) / (2 * theta) * (1 - np.exp(-2 *
theta * t))

            forecasts.append(forecast_mean)
            forecast_vars.append(forecast_var)

        return {
            'forecasts': np.array(forecasts),
            'variances': np.array(forecast_vars),
            'std_errors': np.sqrt(forecast_vars),
            'confidence_intervals_95': np.column_stack([
                forecasts - 1.96 * np.sqrt(forecast_vars),
                forecasts + 1.96 * np.sqrt(forecast_vars)
            ])
        }

    def adf_test_stationarity(self) -> dict:
        """
        Augmented Dickey-Fuller test for stationarity
        """
        from statsmodels.tsa.stattools import adfuller

        adf_result = adfuller(self.log_prices, autolag='AIC')

        return {
            'adf_statistic': adf_result[0],
            'p_value': adf_result[1],
            'critical_values': adf_result[4],
            'is_stationary': adf_result[1] < 0.05,
            'used_lags': adf_result[2]
        }

    def variance_ratio_test(self, lags: List[int] = [2, 4, 8, 16]) ->
dict:
        """
        Variance ratio test for random walk hypothesis
        """
```

```
        log_returns = self.log_prices.diff().dropna()
        n = len(log_returns)

        variance_ratios = {}
        test_statistics = {}
        p_values = {}

        # Base variance (1-period)
        var_1 = np.var(log_returns)

        for k in lags:
            if k >= n:
                continue

            # k-period returns
            k_returns = []
            for i in range(k, n):
                k_period_return = log_returns.iloc[i-k+1:i+1].sum()
                k_returns.append(k_period_return)

            if len(k_returns) == 0:
                continue

            var_k = np.var(k_returns)

            # Variance ratio
            vr = var_k / (k * var_1)
            variance_ratios[k] = vr

            # Test statistic (under random walk, VR should be 1)
            # Asymptotic variance of VR under random walk
            theta = 1  # Assuming no autocorrelation
            asymptotic_var = 2 * (2*k - 1) * (k - 1) / (3 * k * n)

            test_stat = (vr - 1) / np.sqrt(asymptotic_var)
            test_statistics[k] = test_stat

            # Two-tailed p-value
            p_value = 2 * (1 - stats.norm.cdf(abs(test_stat)))
            p_values[k] = p_value

        return {
```

```
            'variance_ratios': variance_ratios,
            'test_statistics': test_statistics,
            'p_values': p_values,
            'reject_random_walk': {k: p < 0.05 for k, p in
p_values.items()}
        }

# Example: Mean reversion analysis for USD stablecoin pairs
print("\n--- Mean Reversion Analysis: USDC-DAI Spread ---")

# Calculate USDC-DAI spread
usdc_dai_spread = np.log(defi_prices['USDC']) -
np.log(defi_prices['DAI'])

# Initialize mean reversion model
mr_model = MeanReversionModeling(usdc_dai_spread)

# Test for stationarity
adf_test = mr_model.adf_test_stationarity()
print(f"ADF Test p-value: {adf_test['p_value']:.4f}")
print(f"Is stationary: {adf_test['is_stationary']}")

# Fit Ornstein-Uhlenbeck process
ou_params_mle = mr_model.fit_ornstein_uhlenbeck(method='mle')
if ou_params_mle['success']:
    print(f"\n--- OU Process Parameters (MLE) ---")
    print(f"Mean reversion rate (θ): {ou_params_mle['theta']:.4f}")
    print(f"Long-term mean (μ): {ou_params_mle['mu']:.6f}")
    print(f"Volatility (σ): {ou_params_mle['sigma']:.4f}")
    print(f"Half-life: {ou_params_mle['half_life']:.2f} days")
    print(f"AIC: {ou_params_mle['aic']:.2f}")

# Alternative least squares estimation
ou_params_ls = mr_model.fit_ornstein_uhlenbeck(method='least_squares')
if ou_params_ls['success']:
    print(f"\n--- OU Process Parameters (Least Squares) ---")
    print(f"Mean reversion rate (θ): {ou_params_ls['theta']:.4f}")
    print(f"Long-term mean (μ): {ou_params_ls['mu']:.6f}")
    print(f"Volatility (σ): {ou_params_ls['sigma']:.4f}")
    print(f"Half-life: {ou_params_ls['half_life']:.2f} days")
    print(f"R-squared: {ou_params_ls['r_squared']:.4f}")
```

```
# Forecast spread
forecast_result = mr_model.forecast_ou_process(steps=10)
print(f"\n--- 10-Day Spread Forecast ---")
print(f"Current spread: {usdc_dai_spread.iloc[-1]:.6f}")
print(f"1-day forecast: {forecast_result['forecasts'][0]:.6f}")
print(f"10-day forecast: {forecast_result['forecasts'][-1]:.6f}")
print(f"Forecast uncertainty (1-day): ±{forecast_result['std_errors']
[0]:.6f}")

# Variance ratio test
vr_test = mr_model.variance_ratio_test()
print(f"\n--- Variance Ratio Test ---")
for lag, vr in vr_test['variance_ratios'].items():
    reject = vr_test['reject_random_walk'][lag]
    print(f"Lag {lag}: VR = {vr:.3f}, Reject RW = {reject}")
```

---

4. Portfolio Optimization for Statistical Arbitrage

4.1 Multi-Asset Statistical Arbitrage Portfolio

```
class StatArbPortfolioOptimizer:
    """
    Portfolio optimization framework for statistical arbitrage
strategies
    """

    def __init__(self, returns_data: pd.DataFrame,
cointegration_relationships: dict):
        """
        Initialize portfolio optimizer

        Args:
            returns_data: DataFrame with asset returns
            cointegration_relationships: Dict with cointegration
vectors
        """
        self.returns = returns_data
        self.coint_relationships = cointegration_relationships
        self.portfolio_weights = None
        self.risk_model = None

    def construct_spread_portfolios(self) -> dict:
        """
        Construct spread portfolios from cointegration relationships
        """
        spread_portfolios = {}

        for pair_name, coint_data in self.coint_relationships.items():
            if coint_data.get('is_cointegrated', False):
                assets = coint_data['assets']
                beta = coint_data['beta']

                # Create spread portfolio weights
                # Long-short portfolio: +1 unit of asset1, -beta units
of asset2
                portfolio_weights = pd.Series(0.0,
index=self.returns.columns)
                portfolio_weights[assets[0]] = 1.0
                portfolio_weights[assets[1]] = -beta

                # Normalize to unit portfolio (optional)
                portfolio_weights = portfolio_weights /
```

```
np.sqrt(np.sum(portfolio_weights**2))

                spread_portfolios[pair_name] = {
                    'weights': portfolio_weights,
                    'expected_return':
self._calculate_expected_return(portfolio_weights),
                    'volatility':
self._calculate_portfolio_volatility(portfolio_weights),
                    'half_life': coint_data.get('half_life', np.inf)
                }

        return spread_portfolios

    def _calculate_expected_return(self, weights: pd.Series) -> float:
        """
        Calculate expected return for a portfolio
        """
        mean_returns = self.returns.mean()
        return np.dot(weights, mean_returns)

    def _calculate_portfolio_volatility(self, weights: pd.Series) ->
float:
        """
        Calculate portfolio volatility
        """
        cov_matrix = self.returns.cov()
        return np.sqrt(np.dot(weights, np.dot(cov_matrix, weights)))

    def optimize_portfolio(self, spread_portfolios: dict,
                           objective: str = 'max_sharpe',
                           constraints: dict = None) -> dict:
        """
        Optimize allocation across multiple spread portfolios
        """
        if not spread_portfolios:
            return {'error': 'No spread portfolios provided'}

        n_spreads = len(spread_portfolios)
        spread_names = list(spread_portfolios.keys())

        # Expected returns and covariance of spread portfolios
        spread_returns = []
```

```
        spread_weights_matrix = []

        for name in spread_names:
            spread_data = spread_portfolios[name]
            spread_returns.append(spread_data['expected_return'])
            spread_weights_matrix.append(spread_data['weights'].values)

        spread_returns = np.array(spread_returns)
        spread_weights_matrix = np.array(spread_weights_matrix)

        # Calculate spread covariance matrix
        asset_cov = self.returns.cov().values
        spread_cov = spread_weights_matrix @ asset_cov @
spread_weights_matrix.T

        if objective == 'max_sharpe':
            # Maximize Sharpe ratio
            result = self._maximize_sharpe_ratio(spread_returns,
spread_cov)
        elif objective == 'min_variance':
            # Minimize variance
            result = self._minimize_variance(spread_cov)
        elif objective == 'max_return':
            # Maximize return subject to risk constraint
            target_vol = constraints.get('target_volatility', 0.1)
            result = self._maximize_return_target_vol(spread_returns,
spread_cov, target_vol)
        else:
            raise ValueError("Objective must be 'max_sharpe',
'min_variance', or 'max_return'")

        if result['success']:
            # Map back to individual assets
            spread_allocation = result['weights']
            asset_weights = spread_weights_matrix.T @ spread_allocation

            # Normalize asset weights
            total_abs_weight = np.sum(np.abs(asset_weights))
            if total_abs_weight > 0:
                asset_weights = asset_weights / total_abs_weight

            return {
```

```
                'success': True,
                'spread_allocation': dict(zip(spread_names,
spread_allocation)),
                'asset_weights': pd.Series(asset_weights,
index=self.returns.columns),
                'expected_return': np.dot(spread_allocation,
spread_returns),
                'volatility': np.sqrt(spread_allocation @ spread_cov @
spread_allocation),
                'sharpe_ratio': result.get('sharpe_ratio', 0)
            }
        else:
            return result

    def _maximize_sharpe_ratio(self, expected_returns: np.ndarray,
                               cov_matrix: np.ndarray) -> dict:
        """
        Maximize Sharpe ratio optimization
        """
        n = len(expected_returns)

        def negative_sharpe(weights):
            portfolio_return = np.dot(weights, expected_returns)
            portfolio_vol = np.sqrt(np.dot(weights, np.dot(cov_matrix,
weights)))
            return -portfolio_return / portfolio_vol if portfolio_vol >
0 else -np.inf

        # Constraints: weights sum to 1
        constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1})

        # Bounds: allow long and short positions
        bounds = tuple((-2, 2) for _ in range(n))  # Allow up to 200%
long/short

        # Initial guess
        initial_guess = np.ones(n) / n

        result = minimize(negative_sharpe, initial_guess,
                          method='SLSQP', bounds=bounds,
constraints=constraints)
```

```
        if result.success:
            weights = result.x
            portfolio_return = np.dot(weights, expected_returns)
            portfolio_vol = np.sqrt(np.dot(weights, np.dot(cov_matrix,
weights)))
            sharpe_ratio = portfolio_return / portfolio_vol if
portfolio_vol > 0 else 0

            return {
                'success': True,
                'weights': weights,
                'sharpe_ratio': sharpe_ratio,
                'expected_return': portfolio_return,
                'volatility': portfolio_vol
            }
        else:
            return {'success': False, 'error': result.message}

    def _minimize_variance(self, cov_matrix: np.ndarray) -> dict:
        """
        Minimum variance optimization
        """
        n = cov_matrix.shape[0]

        def portfolio_variance(weights):
            return np.dot(weights, np.dot(cov_matrix, weights))

        constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1})
        bounds = tuple((-2, 2) for _ in range(n))
        initial_guess = np.ones(n) / n

        result = minimize(portfolio_variance, initial_guess,
                          method='SLSQP', bounds=bounds,
constraints=constraints)

        if result.success:
            return {
                'success': True,
                'weights': result.x,
                'volatility': np.sqrt(result.fun)
            }
        else:
```

```
            return {'success': False, 'error': result.message}

    def _maximize_return_target_vol(self, expected_returns: np.ndarray,
                                    cov_matrix: np.ndarray, target_vol:
float) -> dict:
        """
        Maximize return subject to volatility constraint
        """
        n = len(expected_returns)

        def negative_return(weights):
            return -np.dot(weights, expected_returns)

        def vol_constraint(weights):
            vol = np.sqrt(np.dot(weights, np.dot(cov_matrix, weights)))
            return target_vol - vol

        constraints = [
            {'type': 'eq', 'fun': lambda x: np.sum(x) - 1},
            {'type': 'eq', 'fun': vol_constraint}
        ]

        bounds = tuple((-2, 2) for _ in range(n))
        initial_guess = np.ones(n) / n

        result = minimize(negative_return, initial_guess,
                          method='SLSQP', bounds=bounds,
constraints=constraints)

        if result.success:
            weights = result.x
            portfolio_return = np.dot(weights, expected_returns)
            portfolio_vol = np.sqrt(np.dot(weights, np.dot(cov_matrix,
weights)))

            return {
                'success': True,
                'weights': weights,
                'expected_return': portfolio_return,
                'volatility': portfolio_vol
            }
        else:
```

```
            return {'success': False, 'error': result.message}

    def risk_attribution(self, portfolio_weights: pd.Series) -> dict:
        """
        Calculate risk attribution for the portfolio
        """
        cov_matrix = self.returns.cov()

        # Portfolio variance
        portfolio_var = np.dot(portfolio_weights, np.dot(cov_matrix,
portfolio_weights))
        portfolio_vol = np.sqrt(portfolio_var)

        # Marginal risk contribution
        marginal_contrib = np.dot(cov_matrix, portfolio_weights) /
portfolio_vol

        # Component risk contribution
        component_contrib = portfolio_weights * marginal_contrib

        # Percentage risk contribution
        percent_contrib = component_contrib / portfolio_vol

        return {
            'portfolio_volatility': portfolio_vol,
            'marginal_risk_contribution': pd.Series(marginal_contrib,
index=portfolio_weights.index),
            'component_risk_contribution': pd.Series(component_contrib,
index=portfolio_weights.index),
            'percent_risk_contribution': pd.Series(percent_contrib,
index=portfolio_weights.index)
        }

# Example: Multi-asset statistical arbitrage portfolio
print("\n--- Statistical Arbitrage Portfolio Optimization ---")

# Calculate returns from price data
returns_data = defi_prices.pct_change().dropna()

# Initialize portfolio optimizer
portfolio_optimizer = StatArbPortfolioOptimizer(returns_data,
coint_results)
```

```
# Construct spread portfolios
spread_portfolios = portfolio_optimizer.construct_spread_portfolios()

print(f"Number of spread portfolios: {len(spread_portfolios)}")
for name, portfolio in spread_portfolios.items():
    print(f"\n{name}:")
    print(f"  Expected return: {portfolio['expected_return']*252*100:.
2f}% (annualized)")
    print(f"  Volatility: {portfolio['volatility']*np.sqrt(252)*100:.
2f}% (annualized)")
    print(f"  Half-life: {portfolio['half_life']:.2f} days")

    # Show non-zero weights
    non_zero_weights = portfolio['weights'][portfolio['weights'] != 0]
    for asset, weight in non_zero_weights.items():
        print(f"    {asset}: {weight:.4f}")

# Optimize portfolio allocation
if spread_portfolios:
    optimization_result = portfolio_optimizer.optimize_portfolio(
        spread_portfolios,
        objective='max_sharpe'
    )

    if optimization_result['success']:
        print(f"\n--- Optimal Portfolio Allocation ---")
        print(f"Expected return:
{optimization_result['expected_return']*252*100:.2f}% (annualized)")
        print(f"Volatility:
{optimization_result['volatility']*np.sqrt(252)*100:.2f}%
(annualized)")
        print(f"Sharpe ratio:
{optimization_result['sharpe_ratio']*np.sqrt(252):.3f} (annualized)")

        print(f"\nSpread allocations:")
        for spread, allocation in
optimization_result['spread_allocation'].items():
            if abs(allocation) > 0.01:  # Show significant allocations
                print(f"  {spread}: {allocation:.3f}")

        print(f"\nAsset weights:")
```

```
        asset_weights = optimization_result['asset_weights']
        for asset, weight in asset_weights.items():
            if abs(weight) > 0.01:
                print(f"  {asset}: {weight:.3f}")

        # Risk attribution
        risk_attrib =
portfolio_optimizer.risk_attribution(asset_weights)
        print(f"\nRisk attribution (top contributors):")
        risk_contrib_sorted =
risk_attrib['percent_risk_contribution'].abs().sort_values(ascending=False)
        for asset, contrib in risk_contrib_sorted.head(3).items():
            print(f"  {asset}: {contrib*100:.2f}%")
```

---

5. Practical Exercises

Exercise 1: Cross-Chain Arbitrage Analysis

Develop a cointegration-based strategy for arbitrage between Ethereum and Polygon versions of the same tokens.

Exercise 2: DeFi Protocol Arbitrage

Build a mean reversion model for arbitrage opportunities between different DeFi lending protocols (Aave, Compound, etc.).

Exercise 3: Stablecoin Arbitrage Portfolio

Create an optimized portfolio of stablecoin pairs trading strategies with dynamic risk management.

Exercise 4: Yield Farming Arbitrage

Design a statistical arbitrage framework for yield farming opportunities across different DeFi protocols.

---

6. Summary and Next Steps

Key Concepts Covered

1. **Cointegration Analysis**: Finding long-term equilibrium relationships

2. **Pairs Trading**: Implementing mean reversion strategies
3. **Ornstein-Uhlenbeck Models**: Mathematical modeling of mean reversion
4. **Portfolio Optimization**: Multi-asset statistical arbitrage
5. **Risk Management**: Attribution and position sizing
6. **Performance Evaluation**: Backtesting and strategy assessment

Practical Applications in DeFi

- **Cross-Exchange Arbitrage**: Price discrepancies between DEXs
- **Stablecoin Arbitrage**: Mean reversion in stablecoin pairs
- **Yield Arbitrage**: Rate differences between lending protocols
- **Liquidity Pool Arbitrage**: Imbalances in AMM pools

Advanced Techniques for Further Study

- **Multivariate Cointegration**: VECM models for multiple assets
- **Non-Linear Mean Reversion**: Threshold and regime-switching models
- **High-Frequency Statistical Arbitrage**: Sub-second trading strategies
- **Machine Learning Enhancement**: AI-driven relationship discovery

Next Module Preview

Module 5: Portfolio Optimization Theory will cover:
- Modern Portfolio Theory for MEV strategies
- Black-Litterman model applications
- Risk parity and factor-based approaches
- Dynamic portfolio allocation methods

Additional Resources

Statistical Arbitrage References

- "Quantitative Trading" by Ernest Chan
- "Algorithmic Trading: Winning Strategies" by Ernest Chan
- "Statistical Arbitrage" by Andrew Pole

Econometric Methods

- "Time Series Analysis" by James Hamilton
- "Econometric Analysis" by William Greene
- "Analysis of Integrated and Cointegrated Time Series" by Søren Johansen

Python Libraries

- `statsmodels` for econometric analysis
- `scipy` for optimization
- `numpy` and `pandas` for data manipulation
- `matplotlib` for visualization
- `arch` for GARCH modeling

Module Completion Time: 170 minutes
Difficulty Level: Advanced
Prerequisites: Time Series Analysis, Statistics, Linear Algebra

This module provides comprehensive statistical arbitrage capabilities for DeFi and cryptocurrency markets. Master these techniques before proceeding to portfolio optimization theory in Module 5.