Finance Applications - Advanced Handout

Machine Learning for Smarter Innovation

1 Finance Applications - Advanced Handout

Target Audience: Quantitative analysts and ML engineers **Duration**: 90 minutes reading **Level**: Advanced (mathematical foundations + production systems)

1.1 Mathematical Foundations

1.1.1 Stochastic Calculus Background

Financial modeling relies on stochastic differential equations. The fundamental process is geometric Brownian motion:

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

where W_t is a Wiener process with $dW_t \sim N(0, dt)$. By Ito's lemma, for a function $f(S, t)$:

$$df = \left(\frac{\partial f}{\partial t} + \mu S \frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} \right) dt + \sigma S \frac{\partial f}{\partial S} dW_t$$

The solution to geometric Brownian motion:

$$S_T = S_0 \exp\left[(\mu - \frac{\sigma^2}{2})T + \sigma W_T \right]$$

Under the risk-neutral measure $\mathbb{Q}$, the drift becomes r (risk-free rate):

$$S_T = S_0 \exp\left[(r - \frac{\sigma^2}{2})T + \sigma W_T^{\mathbb{Q}} \right]$$

1.1.2 Measure Theory and Pricing

The Fundamental Theorem of Asset Pricing states: a market is arbitrage-free if and only if there exists an equivalent martingale measure $\mathbb{Q}$. Under $\mathbb{Q}$:

$$\mathbb{E}^{\mathbb{Q}}\left[\frac{S_T}{B_T} \middle| \mathcal{F}_t \right] = \frac{S_t}{B_t}$$

where $B_t = e^{rt}$ is the money market account. Derivative prices are:

$$V_t = B_t \mathbb{E}^{\mathbb{Q}}\left[\frac{V_T}{B_T} \middle| \mathcal{F}_t \right] = e^{-r(T-t)} \mathbb{E}^{\mathbb{Q}}[V_T | \mathcal{F}_t]$$

1.2 Portfolio Theory

1.2.1 Mean-Variance Optimization

The Markowitz problem minimizes portfolio variance for target return:

$$\min_w \frac{1}{2} w^T \Sigma w$$

subject to: $\mu^T w \geq r_{target}$, $\mathbf{1}^T w = 1$, $w \geq 0$

The efficient frontier is the set of portfolios with minimum variance for each return level. The tangency portfolio maximizes the Sharpe ratio:

$$w^* =_w \frac{w^T \mu - r_f}{\sqrt{w^T \Sigma w}}$$

The analytical solution (without constraints):

$$w^* = \frac{\Sigma^{-1}(\mu - r_f \mathbf{1})}{\mathbf{1}^T \Sigma^{-1}(\mu - r_f \mathbf{1})}$$

1.2.2 Estimation Error and Shrinkage

Sample covariance estimation suffers from high variance when $p \approx n$. The Ledoit-Wolf shrinkage estimator combines sample and structured estimates:

$$\hat{\Sigma}_{shrunk} = \alpha F + (1 - \alpha) S$$

where S is the sample covariance, F is a structured target (e.g., diagonal or single-factor), and α is the optimal shrinkage intensity:

$$\alpha^* = \frac{\sum_{i,j} \text{Var}(s_{ij})}{||S - F||_F^2}$$

1.2.3 Black-Litterman Model

Combines market equilibrium with investor views:

$$\mu_{BL} = [(\tau\Sigma)^{-1} + P^T \Omega^{-1} P]^{-1} [(\tau\Sigma)^{-1} \Pi + P^T \Omega^{-1} Q]$$

Where: - $\Pi = \delta \Sigma w_{mkt}$ is the equilibrium excess return vector - δ is the risk aversion coefficient (typically 2.5) - P is the view pick matrix (K views x N assets) - Q is the view return vector (K x 1) - Ω is the view uncertainty matrix - τ is a scalar (typically 0.05)

1.3 Implementation: Portfolio Optimization Engine

```
"""
Production portfolio optimization with multiple methods.
"""
```

```python
import numpy as np
from typing import Dict, Optional, Tuple, List
from dataclasses import dataclass
from scipy.optimize import minimize
from scipy.linalg import cholesky
import pandas as pd


@dataclass
class PortfolioResult:
    """Result of portfolio optimization."""
    weights: np.ndarray
    expected_return: float
    volatility: float
    sharpe_ratio: float
    asset_names: List[str]


class PortfolioOptimizer:
    """
    Multi-method portfolio optimizer with production features.
    """

    def __init__(self, returns: pd.DataFrame, risk_free_rate: float = 0.02):
        self.returns = returns
        self.rf = risk_free_rate
        self.mu = returns.mean().values * 252  # Annualized
        self.Sigma = returns.cov().values * 252  # Annualized
        self.n_assets = len(self.mu)
        self.asset_names = returns.columns.tolist()

    def mean_variance(
        self,
        target_return: Optional[float] = None,
        target_volatility: Optional[float] = None
    ) -> PortfolioResult:
        """
        Mean-variance optimization with optional targets.
        """
        def portfolio_volatility(w):
            return np.sqrt(w @ self.Sigma @ w)

        def portfolio_return(w):
            return w @ self.mu

        # Constraints
        constraints = [
            {"type": "eq", "fun": lambda w: np.sum(w) - 1}  # Weights sum to 1
        ]

        if target_return is not None:
            constraints.append({
                "type": "eq",
                "fun": lambda w: portfolio_return(w) - target_return
            })

        if target_volatility is not None:
            constraints.append({
                "type": "eq",
                "fun": lambda w: portfolio_volatility(w) - target_volatility
            })

        bounds = [(0, 1) for _ in range(self.n_assets)]
```

```
        w0 = np.ones(self.n_assets) / self.n_assets

        result = minimize(
            portfolio_volatility,
            w0,
            method="SLSQP",
            bounds=bounds,
            constraints=constraints
        )

        return self._build_result(result.x)

    def max_sharpe(self) -> PortfolioResult:
        """Maximum Sharpe ratio portfolio."""
        def neg_sharpe(w):
            ret = w @ self.mu
            vol = np.sqrt(w @ self.Sigma @ w)
            return -(ret - self.rf) / vol

        constraints = [{"type": "eq", "fun": lambda w: np.sum(w) - 1}]
        bounds = [(0, 1) for _ in range(self.n_assets)]
        w0 = np.ones(self.n_assets) / self.n_assets

        result = minimize(
            neg_sharpe,
            w0,
            method="SLSQP",
            bounds=bounds,
            constraints=constraints
        )

        return self._build_result(result.x)

    def risk_parity(self, risk_budget: Optional[np.ndarray] = None) ->
    PortfolioResult:
        """
        Risk parity: equal risk contribution from each asset.
        """
        if risk_budget is None:
            risk_budget = np.ones(self.n_assets) / self.n_assets

        def objective(w):
            port_vol = np.sqrt(w @ self.Sigma @ w)
            marginal = self.Sigma @ w
            risk_contrib = w * marginal / port_vol
            target_contrib = risk_budget * port_vol
            return np.sum((risk_contrib - target_contrib) ** 2)

        constraints = [{"type": "eq", "fun": lambda w: np.sum(w) - 1}]
        bounds = [(0.01, 1) for _ in range(self.n_assets)]
        w0 = np.ones(self.n_assets) / self.n_assets

        result = minimize(
            objective,
            w0,
            method="SLSQP",
            bounds=bounds,
            constraints=constraints
        )

        return self._build_result(result.x)

    def minimum_variance(self) -> PortfolioResult:
```

```
        """Global minimum variance portfolio."""
        return self.mean_variance(target_return=None, target_volatility=None)

    def black_litterman(
        self,
        market_caps: np.ndarray,
        views_P: np.ndarray,
        views_Q: np.ndarray,
        tau: float = 0.05,
        delta: float = 2.5
    ) -> PortfolioResult:
        """
        Black-Litterman allocation with investor views.
        """
        # Market equilibrium weights
        w_mkt = market_caps / market_caps.sum()

        # Equilibrium returns
        Pi = delta * self.Sigma @ w_mkt

        # View uncertainty
        Omega = np.diag(np.diag(views_P @ (tau * self.Sigma) @ views_P.T))

        # Black-Litterman combined returns
        inv_tau_sigma = np.linalg.inv(tau * self.Sigma)
        inv_omega = np.linalg.inv(Omega)

        M = np.linalg.inv(inv_tau_sigma + views_P.T @ inv_omega @ views_P)
        mu_bl = M @ (inv_tau_sigma @ Pi + views_P.T @ inv_omega @ views_Q)

        # Posterior covariance
        Sigma_bl = self.Sigma + M

        # Store original and use BL parameters
        mu_orig = self.mu.copy()
        self.mu = mu_bl

        result = self.max_sharpe()

        # Restore original
        self.mu = mu_orig

        return result

    def _build_result(self, weights: np.ndarray) -> PortfolioResult:
        """Build result object from weights."""
        weights = np.clip(weights, 0, 1)
        weights = weights / weights.sum()

        ret = weights @ self.mu
        vol = np.sqrt(weights @ self.Sigma @ weights)
        sharpe = (ret - self.rf) / vol

        return PortfolioResult(
            weights=weights,
            expected_return=ret,
            volatility=vol,
            sharpe_ratio=sharpe,
            asset_names=self.asset_names
        )

    def efficient_frontier(self, n_points: int = 50) -> pd.DataFrame:
        """Generate efficient frontier."""
```

```
        min_ret = self.minimum_variance().expected_return
        max_ret = np.max(self.mu)

        target_returns = np.linspace(min_ret, max_ret * 0.95, n_points)
        results = []

        for target in target_returns:
            try:
                result = self.mean_variance(target_return=target)
                results.append({
                    "return": result.expected_return,
                    "volatility": result.volatility,
                    "sharpe": result.sharpe_ratio
                })
            except:
                continue

        return pd.DataFrame(results)
```

1.4 Risk Measures

1.4.1 Value at Risk (VaR)

VaR at confidence level α is the α-quantile of the loss distribution:

$$\text{VaR}_\alpha = -F_L^{-1}(1-\alpha)$$

where F_L is the CDF of the loss distribution $L = -R$.

Parametric VaR (normal assumption):

$$\text{VaR}_\alpha = -(\mu + \sigma\Phi^{-1}(1-\alpha)) \approx \sigma z_\alpha - \mu$$

Historical VaR: empirical quantile of historical returns.

Cornish-Fisher VaR (skewness/kurtosis adjustment):

$$z_{CF} = z_\alpha + \frac{1}{6}(z_\alpha^2 - 1)S + \frac{1}{24}(z_\alpha^3 - 3z_\alpha)(K-3) - \frac{1}{36}(2z_\alpha^3 - 5z_\alpha)S^2$$

where S is skewness and K is kurtosis.

1.4.2 Expected Shortfall (CVaR)

Expected Shortfall is the expected loss given that the loss exceeds VaR:

$$\text{ES}_\alpha = -\mathbb{E}[R | R \leq -\text{VaR}_\alpha]$$

Under normality:

$$\text{ES}_\alpha = \mu - \sigma\frac{\phi(\Phi^{-1}(1-\alpha))}{1-\alpha}$$

ES is a coherent risk measure (subadditive, positive homogeneous, translation invariant, monotonic).

1.4.3 CVaR Optimization

Minimize CVaR subject to return constraint:

$$\min_{w,\gamma} \gamma + \frac{1}{(1-\alpha)n} \sum_{i=1}^{n} \max(0, -r_i^T w - \gamma)$$

This is equivalent to a linear program and can be solved efficiently.

1.5 Implementation: Risk Analytics Engine

```
"""
Comprehensive risk analytics with multiple VaR methods.
"""

import numpy as np
from typing import Dict, List, Tuple, Optional
from dataclasses import dataclass
from scipy import stats


@dataclass
class RiskMetrics:
    """Complete risk metrics for a portfolio."""
    var_parametric: float
    var_historical: float
    var_cornish_fisher: float
    var_monte_carlo: float
    expected_shortfall: float
    max_drawdown: float
    volatility: float
    downside_deviation: float
    sortino_ratio: float


class RiskAnalytics:
    """
    Production risk analytics with multiple methodologies.
    """

    def __init__(self, confidence: float = 0.95, horizon: int = 1):
        self.confidence = confidence
        self.alpha = 1 - confidence
        self.horizon = horizon

    def compute_all(
        self,
        returns: np.ndarray,
        n_simulations: int = 10000
    ) -> RiskMetrics:
        """Compute comprehensive risk metrics."""
        scaled_returns = returns * np.sqrt(self.horizon)

        return RiskMetrics(
            var_parametric=self.var_parametric(scaled_returns),
            var_historical=self.var_historical(scaled_returns),
            var_cornish_fisher=self.var_cornish_fisher(scaled_returns),
            var_monte_carlo=self.var_monte_carlo(
                np.mean(returns), np.std(returns), n_simulations
```

```
            ),
            expected_shortfall=self.expected_shortfall(scaled_returns),
            max_drawdown=self.max_drawdown(returns),
            volatility=np.std(returns) * np.sqrt(252),
            downside_deviation=self.downside_deviation(returns),
            sortino_ratio=self.sortino_ratio(returns)
        )

    def var_parametric(self, returns: np.ndarray) -> float:
        """Parametric VaR assuming normality."""
        mu = np.mean(returns)
        sigma = np.std(returns)
        z = stats.norm.ppf(self.alpha)
        return -(mu + sigma * z)

    def var_historical(self, returns: np.ndarray) -> float:
        """Historical simulation VaR."""
        return -np.percentile(returns, self.alpha * 100)

    def var_cornish_fisher(self, returns: np.ndarray) -> float:
        """Cornish-Fisher adjusted VaR."""
        mu = np.mean(returns)
        sigma = np.std(returns)
        skew = stats.skew(returns)
        kurt = stats.kurtosis(returns)

        z = stats.norm.ppf(self.alpha)

        # Cornish-Fisher expansion
        z_cf = (z + (z**2 - 1) * skew / 6 +
                (z**3 - 3*z) * (kurt - 3) / 24 -
                (2*z**3 - 5*z) * skew**2 / 36)

        return -(mu + sigma * z_cf)

    def var_monte_carlo(
        self,
        mu: float,
        sigma: float,
        n_simulations: int
    ) -> float:
        """Monte Carlo VaR."""
        simulated = np.random.normal(
            mu * self.horizon,
            sigma * np.sqrt(self.horizon),
            n_simulations
        )
        return -np.percentile(simulated, self.alpha * 100)

    def expected_shortfall(self, returns: np.ndarray) -> float:
        """Expected Shortfall (CVaR)."""
        var = self.var_historical(returns)
        tail_returns = returns[returns <= -var]

        if len(tail_returns) == 0:
            return var

        return -np.mean(tail_returns)

    def max_drawdown(self, returns: np.ndarray) -> float:
        """Maximum drawdown from returns series."""
        cumulative = np.cumprod(1 + returns)
        running_max = np.maximum.accumulate(cumulative)
```

```
        drawdown = (cumulative - running_max) / running_max
        return -np.min(drawdown)

    def downside_deviation(
        self,
        returns: np.ndarray,
        mar: float = 0.0
    ) -> float:
        """Downside deviation below minimum acceptable return."""
        downside_returns = returns[returns < mar]
        if len(downside_returns) == 0:
            return 0.0
        return np.sqrt(np.mean((downside_returns - mar) ** 2)) * np.sqrt(252)

    def sortino_ratio(
        self,
        returns: np.ndarray,
        mar: float = 0.0,
        risk_free: float = 0.02
    ) -> float:
        """Sortino ratio using downside deviation."""
        excess_return = np.mean(returns) * 252 - risk_free
        downside = self.downside_deviation(returns, mar)

        if downside == 0:
            return np.inf if excess_return > 0 else 0.0

        return excess_return / downside


class CVaROptimizer:
    """
    CVaR portfolio optimization using linear programming.
    """

    def __init__(self, alpha: float = 0.95):
        self.alpha = alpha

    def optimize(
        self,
        returns: np.ndarray,
        target_return: Optional[float] = None
    ) -> np.ndarray:
        """
        Minimize CVaR subject to return constraint.

        Uses LP formulation:
        min gamma + 1/((1-alpha)*n) * sum(z_i)
        s.t. z_i >= -r_i'w - gamma
             z_i >= 0
             1'w = 1
             w >= 0
             mu'w >= target (optional)
        """
        from scipy.optimize import linprog

        n_scenarios, n_assets = returns.shape

        # Decision variables: [w, gamma, z]
        n_vars = n_assets + 1 + n_scenarios

        # Objective: min gamma + 1/((1-alpha)*n) * sum(z)
        c = np.zeros(n_vars)
```

```
        c[n_assets] = 1  # gamma coefficient
        c[n_assets + 1:] = 1 / ((1 - self.alpha) * n_scenarios)  # z
    coefficients

        # Inequality constraints: z_i >= -r_i'w - gamma
        # Rewrite: -z_i - gamma + r_i'w <= 0
        A_ub = np.zeros((n_scenarios, n_vars))
        A_ub[:, :n_assets] = -returns
        A_ub[:, n_assets] = -1
        A_ub[:, n_assets + 1:] = -np.eye(n_scenarios)
        b_ub = np.zeros(n_scenarios)

        # Equality constraints: sum(w) = 1
        A_eq = np.zeros((1, n_vars))
        A_eq[0, :n_assets] = 1
        b_eq = [1]

        # Optional return constraint
        if target_return is not None:
            mu = returns.mean(axis=0)
            A_eq_ret = np.zeros((1, n_vars))
            A_eq_ret[0, :n_assets] = mu
            A_eq = np.vstack([A_eq, A_eq_ret])
            b_eq.append(target_return)

        # Bounds
        bounds = (
            [(0, 1) for _ in range(n_assets)] +  # w >= 0
            [(None, None)] +  # gamma unbounded
            [(0, None) for _ in range(n_scenarios)]  # z >= 0
        )

        result = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq,
                         bounds=bounds, method='highs')

        return result.x[:n_assets]
```

1.6 Options Pricing

1.6.1 Black-Scholes Framework

The Black-Scholes PDE for derivative pricing:

$$\frac{\partial V}{\partial t} + rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} = rV$$

With boundary condition $V(S,T) = \max(S-K, 0)$ for calls. Solution:

$$C = S_0 N(d_1) - Ke^{-rT}N(d_2)$$

where:

$$d_1 = \frac{\ln(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}, \quad d_2 = d_1 - \sigma\sqrt{T}$$

1.6.2 Greeks

() () () ()
* * * *
0.1701 0.1952 0.2987 Greek Symbol Definition Call Formula 0.3415

() () () ()
* * * *
0.1701 0.1952 0.2013 0.3415 $\frac{\partial V}{\partial S}$ $N(d_1)$

() () () ()
* * * *
0.1701 0.1952 0.2013 0.3415 $\frac{\partial^2 V}{\partial S^2}$ $\frac{N'(d_1)}{S\sigma\sqrt{T}}$

() () () ()
* * * *
0.1701 0.1952 0.2013 0.3415 $\frac{\partial V}{\partial \sigma}$ $SN'(d_1)\sqrt{T}$

() () () ()
* * * *
0.1701 0.1952 0.2013 0.3415 $\frac{\partial V}{\partial t}$ $-\frac{SN'(d_1)\sigma}{2\sqrt{T}} - rKe^{-rT}N(d_2)$

() () () ()
* * * *
0.1701 0.1952 0.2013 0.3415 $\frac{\partial V}{\partial r}$ $KTe^{-rT}N(d_2)$

1.6.3 Volatility Smile and Surface

Implied volatility varies with strike and maturity, forming a surface. The smile can be parameterized using:

SABR Model:

$$dF = \alpha F^{\beta} dW_1, \quad d\alpha = \nu\alpha dW_2, \quad dW_1 dW_2 = \rho dt$$

SVI (Stochastic Volatility Inspired):

$$w(k) = a + b(\rho(k - m) + \sqrt{(k - m)^2 + \sigma^2})$$

where $k = \ln(K/F)$ is log-moneyness.

1.7 Implementation: Derivatives Pricing Engine

```
"""
Options pricing and Greeks calculation engine.
"""

import numpy as np
from scipy.stats import norm
from scipy.optimize import brentq
from typing import Dict, Optional
from dataclasses import dataclass


@dataclass
class OptionPrices:
    """Option pricing results."""
    call_price: float
    put_price: float
    delta: float
```

```
    gamma: float
    vega: float
    theta: float
    rho: float


class BlackScholes:
    """
    Black-Scholes option pricing with Greeks.
    """

    def __init__(self, S: float, K: float, T: float, r: float, sigma: float):
        self.S = S
        self.K = K
        self.T = T
        self.r = r
        self.sigma = sigma

        self._compute_d1_d2()

    def _compute_d1_d2(self):
        """Compute d1 and d2 parameters."""
        self.d1 = (np.log(self.S / self.K) +
                   (self.r + 0.5 * self.sigma**2) * self.T) / \
                  (self.sigma * np.sqrt(self.T))
        self.d2 = self.d1 - self.sigma * np.sqrt(self.T)

    def call_price(self) -> float:
        """European call option price."""
        return (self.S * norm.cdf(self.d1) -
                self.K * np.exp(-self.r * self.T) * norm.cdf(self.d2))

    def put_price(self) -> float:
        """European put option price."""
        return (self.K * np.exp(-self.r * self.T) * norm.cdf(-self.d2) -
                self.S * norm.cdf(-self.d1))

    def delta(self, option_type: str = "call") -> float:
        """Delta: dV/dS."""
        if option_type == "call":
            return norm.cdf(self.d1)
        else:
            return norm.cdf(self.d1) - 1

    def gamma(self) -> float:
        """Gamma: d2V/dS2."""
        return norm.pdf(self.d1) / (self.S * self.sigma * np.sqrt(self.T))

    def vega(self) -> float:
        """Vega: dV/dsigma (per 1% move)."""
        return self.S * norm.pdf(self.d1) * np.sqrt(self.T) / 100

    def theta(self, option_type: str = "call") -> float:
        """Theta: dV/dt (per day)."""
        term1 = -self.S * norm.pdf(self.d1) * self.sigma / (2 * np.sqrt(self.T
    ))

        if option_type == "call":
            term2 = -self.r * self.K * np.exp(-self.r * self.T) * norm.cdf(
    self.d2)
        else:
            term2 = self.r * self.K * np.exp(-self.r * self.T) * norm.cdf(-
    self.d2)
```

```
        return (term1 + term2) / 365

    def rho(self, option_type: str = "call") -> float:
        """Rho: dV/dr (per 1% move)."""
        if option_type == "call":
            return self.K * self.T * np.exp(-self.r * self.T) * norm.cdf(self.
    d2) / 100
        else:
            return -self.K * self.T * np.exp(-self.r * self.T) * norm.cdf(-
    self.d2) / 100

    def all_greeks(self, option_type: str = "call") -> Dict[str, float]:
        """Return all Greeks."""
        return {
            "delta": self.delta(option_type),
            "gamma": self.gamma(),
            "vega": self.vega(),
            "theta": self.theta(option_type),
            "rho": self.rho(option_type)
        }


class ImpliedVolatility:
    """
    Implied volatility calculation and surface fitting.
    """

    @staticmethod
    def from_price(
        market_price: float,
        S: float,
        K: float,
        T: float,
        r: float,
        option_type: str = "call"
    ) -> float:
        """
        Calculate implied volatility using Brent's method.
        """
        def objective(sigma):
            bs = BlackScholes(S, K, T, r, sigma)
            if option_type == "call":
                return bs.call_price() - market_price
            else:
                return bs.put_price() - market_price

        try:
            iv = brentq(objective, 0.001, 5.0)
            return iv
        except ValueError:
            return np.nan

    @staticmethod
    def surface(
        prices: np.ndarray,
        S: float,
        strikes: np.ndarray,
        maturities: np.ndarray,
        r: float,
        option_type: str = "call"
    ) -> np.ndarray:
        """
```

```
        Calculate implied volatility surface.
        """
        n_strikes = len(strikes)
        n_maturities = len(maturities)
        surface = np.zeros((n_strikes, n_maturities))

        for i, K in enumerate(strikes):
            for j, T in enumerate(maturities):
                surface[i, j] = ImpliedVolatility.from_price(
                    prices[i, j], S, K, T, r, option_type
                )

        return surface


class MonteCarloPricer:
    """
    Monte Carlo option pricing for exotic derivatives.
    """

    def __init__(self, n_paths: int = 100000, n_steps: int = 252):
        self.n_paths = n_paths
        self.n_steps = n_steps

    def european_option(
        self,
        S0: float,
        K: float,
        T: float,
        r: float,
        sigma: float,
        option_type: str = "call"
    ) -> Dict[str, float]:
        """Price European option with standard error."""
        dt = T / self.n_steps
        sqrt_dt = np.sqrt(dt)

        # Simulate paths
        paths = np.zeros((self.n_paths, self.n_steps + 1))
        paths[:, 0] = S0

        for t in range(1, self.n_steps + 1):
            z = np.random.standard_normal(self.n_paths)
            paths[:, t] = paths[:, t-1] * np.exp(
                (r - 0.5 * sigma**2) * dt + sigma * sqrt_dt * z
            )

        # Payoffs
        S_T = paths[:, -1]
        if option_type == "call":
            payoffs = np.maximum(S_T - K, 0)
        else:
            payoffs = np.maximum(K - S_T, 0)

        # Discount and average
        discount = np.exp(-r * T)
        price = discount * np.mean(payoffs)
        se = discount * np.std(payoffs) / np.sqrt(self.n_paths)

        return {"price": price, "std_error": se, "confidence_interval": (price
    - 1.96*se, price + 1.96*se)}

    def asian_option(
```

```
        self,
        S0: float,
        K: float,
        T: float,
        r: float,
        sigma: float,
        option_type: str = "call",
        average: str = "arithmetic"
    ) -> Dict[str, float]:
        """Price Asian option with average price."""
        dt = T / self.n_steps
        sqrt_dt = np.sqrt(dt)

        paths = np.zeros((self.n_paths, self.n_steps + 1))
        paths[:, 0] = S0

        for t in range(1, self.n_steps + 1):
            z = np.random.standard_normal(self.n_paths)
            paths[:, t] = paths[:, t-1] * np.exp(
                (r - 0.5 * sigma**2) * dt + sigma * sqrt_dt * z
            )

        # Average price
        if average == "arithmetic":
            S_avg = np.mean(paths[:, 1:], axis=1)
        else:
            S_avg = np.exp(np.mean(np.log(paths[:, 1:]), axis=1))

        if option_type == "call":
            payoffs = np.maximum(S_avg - K, 0)
        else:
            payoffs = np.maximum(K - S_avg, 0)

        discount = np.exp(-r * T)
        price = discount * np.mean(payoffs)
        se = discount * np.std(payoffs) / np.sqrt(self.n_paths)

        return {"price": price, "std_error": se}
```

1.8 Time Series Models

1.8.1 GARCH(p,q)

The GARCH model captures volatility clustering:

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

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

Stationarity condition: $\sum_{i=1}^{p} \alpha_i + \sum_{j=1}^{q} \beta_j < 1$

Unconditional variance: $\mathbb{E}[\sigma_t^2] = \frac{\omega}{1 - \sum \alpha_i - \sum \beta_j}$

1.8.2 Factor Models

Single Factor (CAPM):

$$R_i - R_f = \alpha_i + \beta_i(R_M - R_f) + \epsilon_i$$

Fama-French Three Factor:

$$R_i - R_f = \alpha_i + \beta_i^{MKT}(R_M - R_f) + \beta_i^{SMB} \cdot SMB + \beta_i^{HML} \cdot HML + \epsilon_i$$

Principal Component Analysis extracts statistical factors:

$$\Sigma = V \Lambda V^T$$

where V contains eigenvectors (factor loadings) and Λ contains eigenvalues.

1.9 Common Parameters

()	()	()	()
*	*	*	*
0.1	7	5	Typical
			0.375 Range
()	()	()	()
*	*	*	*
0.1	7	5	High = more conservative
()	()	()	()
*	*	*	*
0.1	7	5	Prior = uncertainty
()	()	()	()
*	*	*	*
0.1	7	5	95% typical
()	()	()	()
*	*	*	*
0.1	7	5	10000 accuracy vs speed

() *	() *	() *	() *
0.1	7	7	Typical 0.3750 Range
() *	() *	() *	() *
0.1	7	7	Per- pha + beta 0.99 Is- tence of volatil- ity
() *	() *	() *	() *
0.1	7	7	An- tioity 0.5 nu- al- ized

1.10 Practice Problems

1. **Mean-Variance Optimization**: Given 5 assets with historical returns, construct the efficient frontier and identify the maximum Sharpe ratio portfolio.
2. **Risk Budgeting**: Implement a risk parity portfolio where each asset contributes equally to total portfolio risk. Compare to equal-weight allocation.
3. **VaR Backtesting**: Implement a VaR model and backtest using Kupiec's POF test. Calculate the violation ratio and assess model performance.
4. **Implied Volatility**: Given options prices across strikes and maturities, construct the volatility surface and fit an SVI parameterization.
5. **Factor Model**: Decompose portfolio risk into factor and idiosyncratic components. Calculate factor exposures and contribution to total variance.

1.11 References

1. Markowitz, H. (1952). "Portfolio Selection." Journal of Finance, 7(1), 77-91.
2. Black, F., & Litterman, R. (1992). "Global Portfolio Optimization." Financial Analysts Journal, 48(5), 28-43.
3. McNeil, A. J., Frey, R., & Embrechts, P. (2015). "Quantitative Risk Management." Princeton University Press.
4. Hull, J. C. (2022). "Options, Futures, and Other Derivatives." 11th Edition. Pearson.
5. Bollerslev, T. (1986). "Generalized Autoregressive Conditional Heteroskedasticity." Journal of Econometrics, 31(3), 307-327.
6. Ledoit, O., & Wolf, M. (2004). "A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices." Journal of Multivariate Analysis, 88(2), 365-411.

Quantitative finance combines mathematical rigor with practical implementation. Models that work in theory must survive the reality of markets, regulation, and operational constraints. Always validate assumptions and monitor performance.