|  |
| --- |
| Portfolio Optimization Framework User Manual |
| User Manual |

|  |
| --- |
| 2024 | By: Dennis Jung |

****

|  |
| --- |
| Table of Contents |
| 0. Setup and Installation  1. Introduction  2. Getting Started  3. Basic Usage  4. Working with Constraints  5. Robust Portfolio Optimization  6. Backtesting  7. Efficient Frontier Analysis |
| 0. SETUP AND INSTALLATION Due to internal security concerns and data protection first you have to have to change the sent PortOp.txt into a Python. Project Structure to run the Code Create a new directory for your project with the following structure:  your\_project\_folder/  └── PortOpt.py # Copy the provided code here  └── your\_analysis.py # Your main script  Standard Required Dependencies:  Install all required packages using pip:  pip install numpy pandas scipy cvxpy matplotlib tqdm  Create your analysis script (your\_analysis.py)  Importing the Package  Method 1 - Direct Import - *this possibility you will also find in the other examples.*  from PortOpt import (  RobustPortfolioOptimizer,  RobustBacktestOptimizer,  RobustEfficientFrontier,  )  Method 2 - Module Import:  from PortOpt import \*  optimizer = PortfolioOptimizer(...) – with this you can directly initialize all classes. |

1. INTRODUCTION

The portfolio optimization framework provides comprehensive tools for portfolio management:

Key Features:

* Multiple optimization methods (SCIPY, CVXPY)
* Various objective functions
* Robust optimization with uncertainty parameters
* Efficient frontier computation
* Backtesting capabilities
* Comprehensive constraints handling
* Performance analytics and visualization

This framework is designed to provide a comprehensive solution for portfolio optimization with a focus on robustness and practical implementation. The SCIPY and CVXPY optimization methods offer flexibility in solving complex portfolio optimization problems, while the robust optimization features help address uncertainty in parameter estimation.

The framework incorporates several key innovations in portfolio optimization:

* Robust estimation techniques to handle parameter uncertainty
* Multiple objective functions to suit different investment strategies
* Comprehensive constraint handling for real-world portfolio restrictions
* Advanced backtesting capabilities for strategy validation
* Efficient frontier analysis for portfolio selection

2. GETTING STARTED

Input data should be in pandas DataFrame format with datetime index:

Example:

returns\_data = pd.DataFrame({

'AAPL': [0.01, 0.02, -0.01],

'GOOGL': [0.015, -0.01, 0.02]

}, index=pd.date\_range('2020-01-01', periods=3, freq='M'))

The data structure is crucial for proper functioning of the optimizer. Some important considerations:

* Returns should be clean and properly calculated
* The datetime index should be regular (e.g., monthly) without gaps
* Missing values should be handled before passing to the optimizer
* Column names should be consistent across all input data (returns, expected returns, etc.)
* The data should be sufficiently long to allow for robust estimation (recommended minimum of 24 periods)
* - If you insert no expected returns the expected return is calculated automatically as the mean of the historical returns.

Basic Portfolio Optimizer: Initialize the basic optimizer:

optimizer = PortfolioOptimizer(

returns=returns\_data,

expected\_returns=None, # Optional

optimization\_method='SCIPY',

half\_life=36,

risk\_free\_rate=0.0,

transaction\_cost=0.001

)

Key Parameters:

• returns: DataFrame of historical returns

• expected\_returns: Optional DataFrame of expected returns

• optimization\_method: 'SCIPY' or 'CVXPY'

• half\_life: Number of periods for exponential weighting

• risk\_free\_rate: Annual risk-free rate

• transaction\_cost: Transaction cost as decimal

3. BASIC USAGE

The framework supports multiple objective functions, each suited to different investment goals and market conditions:

* MINIMUM\_VARIANCE: Focuses on minimizing portfolio volatility, suitable for risk-averse investors
* MEAN\_VARIANCE: Traditional Markowitz optimization balancing return and risk
* MAXIMUM\_SHARPE: Maximizes the risk-adjusted return using the Sharpe ratio
* GARLAPPI\_ROBUST: Incorporates parameter uncertainty in the optimization
* RISK\_PARITY: Equalizes risk contribution from each asset
* MAXIMUM\_DIVERSIFICATION: Maximizes portfolio diversification benefits
* MINIMUM\_TRACKING\_ERROR: Minimizes deviation from a benchmark
* MINIMUM\_CVAR: Focuses on minimizing tail risk using Conditional Value at Risk

Available Objective Functions:

from PortOpt import ObjectiveFunction

Example:

from PortOpt import OptimizationConstraints

constraints = OptimizationConstraints(

long\_only=True,

target\_return=0.10,

target\_risk=0.15

)

result = optimizer.optimize(

objective=ObjectiveFunction.MEAN\_VARIANCE,

constraints=constraints)

# Access results

optimal\_weights = result['weights']

expected\_return = result['return']

portfolio\_risk = result['risk']

sharpe\_ratio = result['sharpe\_ratio']

4. WORKING WITH CONSTRAINTS

Constraints are essential for creating realistic and implementable portfolios. The framework provides several types of constraints that can be combined to match specific investment requirements:

Group Constraints: Useful for:

* Sector allocation limits
* Geographic exposure controls
* Asset class restrictions
* Risk factor exposure limits

Box Constraints: Appropriate for:

* Individual position limits
* Regulatory requirements
* Liquidity considerations
* Concentration risk management

Turnover Constraints: Important for:

* Transaction cost control
* Portfolio stability
* Trading frequency reduction
* Market impact management

Tracking Error Constraints: Valuable for:

* Benchmark-relative management
* Active risk control
* Investment mandate compliance
* Performance attribution

Types of Constraints:

1. **Group Constraints**
2. **Box Constraints**
3. **Turnover Constraints**
4. **Tracking Error Constraints**
5. **Combined Constraints**

**To 1. Group Constraints**

Example:

from PortOpt import GroupConstraint

tech\_sector = GroupConstraint(

assets=[0, 1, 2], # Asset indices

bounds=(0.1, 0.4) # Min 10%, Max 40%)

finance\_sector = GroupConstraint(

assets=[3, 4, 5],

bounds=(0.15, 0.35))

constraints = OptimizationConstraints(

group\_constraints={

'tech': tech\_sector,

'finance': finance\_sector }

)

**To 2. Box Constraints**

Example:

constraints = OptimizationConstraints(

box\_constraints={

0: (0.05, 0.15), # Asset 0: min 5%, max 15%

1: (0.0, 0.20), # Asset 1: min 0%, max 20%

2: (0.10, 0.30) # Asset 2: min 10%, max 30%

}

)

**To 3. Turnover Constraints**

Example:

constraints = OptimizationConstraints(

max\_turnover=0.20 # Maximum 20% turnover

)

**To 4. Tracking Error Constraints**

Example:

constraints = OptimizationConstraints(

max\_tracking\_error=0.05,

benchmark\_weights=np.array([0.2, 0.3, 0.5])

)

**To 4. Combined Constraints**

Example:

constraints = OptimizationConstraints(

long\_only=True,

group\_constraints={

'tech': tech\_sector,

'finance': finance\_sector },

box\_constraints={

0: (0.05, 0.15),

1: (0.0, 0.20)},

max\_turnover=0.20,

target\_return=0.10,

target\_risk=0.15,

max\_tracking\_error=0.05,

benchmark\_weights=benchmark\_weights

)

5. ROBUST PORTFOLIO OPTIMIZATION

Initializing Robust Optimizer - Example:

from PortOpt import RobustPortfolioOptimizer

robust\_optimizer = RobustPortfolioOptimizer(

returns=returns\_data, expected\_returns=expected\_returns,

epsilon=0.1,

alpha=1.0,

omega\_method='bayes',

optimization\_method='SCIPY',

half\_life=36,

risk\_free\_rate=0.01,

transaction\_cost=0.001

)

Key Parameters:

• epsilon: Uncertainty parameter

• alpha: Risk aversion parameter

• omega\_method: 'asymptotic', 'bayes', or 'factor'

The robust optimization approach helps address several key challenges in portfolio management:

Parameter Uncertainty:

* Epsilon controls the uncertainty set size
* Higher epsilon values lead to more conservative portfolios
* Can be customized per asset or time period

Risk Aversion:

* Alpha parameter controls the trade-off between return and risk
* Higher alpha values result in more conservative allocations
* Can be adjusted based on market conditions

Estimation Error:

* Omega\_method determines how estimation errors are modelled
* 'bayes' method incorporates prior beliefs
* 'asymptotic' method uses classical statistical theory
* 'factor' method considers factor structure in returns

In the next step we can run the Robust Optimization after initializing the RobustPortfolioOptimizer:

robust\_result = robust\_optimizer.optimize(

objective=ObjectiveFunction.GARLAPPI\_ROBUST,

constraints=constraints,

current\_weights=current\_portfolio

)

Here you can add from the section before your individual constraints or current portfolio weights as the starting point.

6. BACKTESTING

Initializing Backtest Optimizer - Example:

from PortOpt import RobustBacktestOptimizer

backtest\_optimizer = RobustBacktestOptimizer(

returns=returns\_data,

expected\_returns=expected\_returns,

epsilon=epsilon\_data,

alpha=alpha\_data,

lookback\_window=36,

rebalance\_frequency=3,

estimation\_method='robust',

transaction\_cost=0.001,

benchmark\_returns=benchmark\_data,

risk\_free\_rate=0.01,

min\_history=24,

out\_of\_sample=True

)

Key Parameters:

• lookback\_window: Estimation window length

• rebalance\_frequency: Periods between rebalancing

• estimation\_method: 'robust' or 'standard'

• out\_of\_sample: True for out-of-sample testing

• benchmark\_returns can be None if there is no Benchmark

The backtesting framework provides a comprehensive way to evaluate portfolio strategies:

Lookback Window Considerations:

* Should be long enough for stable estimation
* But short enough to capture relevant market conditions
* Typically 24-36 months for monthly data

Rebalancing Frequency Trade-offs:

* More frequent rebalancing can better capture opportunities
* But increases transaction costs
* Consider market liquidity and costs
* Typical choices are monthly, quarterly, or semi-annual

Out-of-Sample Testing:

* Helps avoid overfitting
* More realistic performance assessment
* Uses only information available at each point in time
* Critical for strategy validation

In the next step we can run the Backtest after initializing the RobustBacktestOptimizer:

backtest\_results = backtest\_optimizer.run\_backtest(

objective=ObjectiveFunction.GARLAPPI\_ROBUST,

constraints=constraints,

initial\_weights=initial\_portfolio

)

As you can see you can select here the objective function, constraints or initial weights.

Accessing Results:

• portfolio\_returns = backtest\_results['returns']

• portfolio\_weights = backtest\_results['weights']

• metrics\_history = backtest\_results['metrics\_history']

• realized\_costs = backtest\_results['realized\_costs']

• backtest\_metrics = backtest\_results['backtest\_metrics']

7. EFFICIENT FRONTIER ANALYSIS

Initializing Efficient Frontier Calculator - Example:

from PortOpt import RobustEfficientFrontier

frontier\_calculator = RobustEfficientFrontier(

returns=returns\_data, expected\_returns=expected\_returns,

epsilon=epsilon\_data, alpha=alpha\_data,

optimization\_method='SCIPY',

risk\_free\_rate=0.01,

transaction\_cost=0.001,

min\_history=36,

half\_life=36

)

In the next step we can compute the efficient frontier after initializing the RobustEfficientFrontier:

frontier\_results = frontier\_calculator.compute\_efficient\_frontier(

n\_points=15,

constraints=constraints

)

Accessing Results:

• frontier\_returns = frontier\_results['returns']

• frontier\_risks = frontier\_results['risks']

• frontier\_weights = frontier\_results['weights']

• sharpe\_ratios = frontier\_results['sharpe\_ratios']

• tracking\_errors = frontier\_results['tracking\_errors']

The efficient frontier analysis provides crucial insights for portfolio selection:

Using the Results:

* Frontier returns and risks show the range of available risk-return combinations
* Weights across the frontier show how allocations change with risk tolerance
* Sharpe ratios help identify the most efficient portfolios
* Tracking errors are important for benchmark-relative management

Key Applications:

* Strategic asset allocation decisions
* Risk tolerance assessment
* Portfolio rebalancing guidance
* Investment policy formulation
* Performance attribution baseline