www.portfolioeffect.com
High Frequency Portfolio Analytics

User Manual

PortfolioEffectHFT Package for Java

Oleg Nechaev
oleg.nechaev@portfolioeffect.com

2016

Contents

1 Package Installation

Building from Source

To build the jar from source You need to have Maveninstalled. After that in the folder with pom.xml run:

```
mvn package
```

2 API Credentials

All portfolio computations are performed on PortfolioEffect cloud servers. To obtain a free non-professional account, you need to follow a quick sign-up process on our website: www.portfolioeffect.com/registration.

Please use a valid sign-up address - it will be used to email your account activation link.

Locate API Credentials

Log in to you account and locate your API credentials on the main page

Set API Credentials in Java

Run the following commands to set your account API credentials for the PortfolioEffectHFT Java Library.

```
import com.portfolioeffect.quant.client.api.Util;
Util.util_setCredentials("API Username", "API Password", "API Key");
```

Credentials are not stored between sessions. You would need to repeat this procedure every time you start a new JVM session or if you changed your account password.

You are now ready to call PortfolioEffectHFT methods.

3 Portfolio Construction

User Data

Users may supply their own historical datasets for index and position entries. This external data could be one a OHLC bar column element (e.g. 1-second close prices) or a vector of actual transaction prices that contains non-equidistant data points. You might want to prepend at least N =(4 x windowLength) data points to the beginning of the interval of interest which would be used for initial calibration of portfolio metrics. Times expressed in milliseconds from 1970-01-01 00:00:00 EST.

```
long[] indexTime = new[]{1409666400000L, 1409668200000L, 1409670000000L, 1409671800000L, 1409673600000L,
    1409675400000L, 1409677200000L, 1409679000000L, 1409680800000L, 1409682600000L, 1409684400000L, 1409686200000
    L, 1409688000000L};
double[] indexPrice = new[]{102.99, 103.60, 103.65, 103.58, 103.30, 103.52, 103.34, 103.27, 103.45, 103.32,
    103.20, 103.25, 103.30};
```

Server Data

At PortfolioEffect we are capturing and storing 1-second intraday bar history for a all NASDAQ traded equites. This server-side dataset spans from January 2013 to the latest trading time minus five minutes. It could be used to construct asset portfolios and compute intraday portfolio metrics.

Create Portfolio

```
import com.portfolioeffect.quant.client.api.Portfolio;
...
long[] indexTime;
double[] indexPrice;
...
//Creates new empty Portfolio using user data for index
Portfolio portfolio = new Portfolio(indexPrice, indexTime);
```

When using server-side data, it only requires a time interval that would be treated as a default position holding period unless positions are added with rebalancing. Index symbol could be specified as well with a default value of “SPY” - SPDR S&P 500 ETF Trust.

Interval boundaries are passed in the following format:

- “yyyy-MM-dd HH:MM:SS” (e.g. “2014-10-01 09:30:00”)
- “yyyy-MM-dd” (e.g. “2014-10-01”)
- “t-N” (e.g. “t-5” is latest trading time minus 5 days)
- UTC timestamp in milliseconds (mills from “1970-01-01 00:00:00”) in EST time zone

```
import com.portfolioeffect.quant.client.api.Portfolio;
...
//Creates new empty Portfolio using server data for index
//Timestamp in "yyyy-MM-dd HH:MM:SS" format
Portfolio portfolio= new Portfolio("2014-10-01 09:30:00", "2014-10-02 16:00:00");
...
//Creates new empty Portfolio using server data for index
// Timestamp in "yyyy-MM-dd" format
Portfolio portfolio= new Portfolio("2014-10-01", "2014-10-02");
...
//Creates new empty Portfolio using server data for index
//Timestamp in "t-N" format
Portfolio portfolio= new Portfolio("t-5", "t");
...
//Timestamp in "t-N" format
//Creates new empty Portfolio using server data for index
//Explicitly specify the symbol for the index
Portfolio portfolio= new Portfolio("t-5", "t","SPY");
```

Add Positions

Positions are added by calling position_add() method on a portfolio object with prices and times in the same format as index data. For positions that were rebalanced or had non-default holding periods a 'time' argument could be used to specify rebalancing timestamps.

```
...
import com.portfolioeffect.quant.client.api.Position;
...
long[] priceTime;
double[] price;
...
//Single position without rebalancing with 100 assets for user data
Position positionGOOG = portfolio.add_position("GOOG", 100, priceTime, time);
...
//Single position without rebalancing with 100 assets for server data
Position positionGOOG = portfolio.add_position("GOOG", 100);
...
...
long[] priceTime;
double[] price;
int[] quantity;
long[] quantityTime;
...
//Single position with rebalancing for user data
Position positionGOOG =  portfolio.add_position("GOOG",  quantity,  quantityTime,  price, priceTime);
...
//Single position with rebalancing for server data
Position positionGOOG =  portfolio.add_position("GOOG",  quantity,  quantityTime);
...
...
int[] quantity= new int[]{300,150}
String[] quantityTimeString= new String[]{"2014-09-01 09:40:00","2014-09-07 14:30:00"}
//Single position with rebalancing for server data
Position positionGOOG =  portfolio.add_position("GOOG",  quantity,  quantityTimeString);
```

Get Symbols List

Once portfolio is created, symbols_available() method could be called to receive the list of all available symbols for position creation. Each symbol is accompanied by a full company/instrument description and listing exchange name.

```
//list.get(0) Contains array of  symbols name (id)
//list.get(1) Contains array of exchanges
```

```
//list.get(2) Contains array of descriptions
List<String[]> list = portfolio.symbols_available()


 id      description                                                 exchange
"BBC"   "BioShares Biotechnology Clinical Trials Fund"             "NASDAQ"
"SCS"   "Steelcase Inc. Common Stock"                              "NYSE"
"BBD"   "Banco Bradesco Sa American Depositary Shares"             "NYSE"
"BBG"   "Bill Barrett Corporation Common Stock"                    "NYSE"
"STPP"  "Barclays PLC - iPath US Treasury Steepener ETN"           "NASDAQ"
"BBF"   "BlackRock Municipal Income Investment Trust"              "NYSE"
"BBH"   "Market Vectors Biotech ETF"                               "NYSEARCA"
"SCON"  "Superconductor Technologies Inc. - Common Stock"          "NASDAQ"
"SCX"   "L.S. Starrett Company (The) Common Stock"                 "NYSE"
"BBK"   "Blackrock Municipal Bond Trust"                           "NYSE"
```

4 Metrics

Class Metric

The result of the calculation a portfolio or positional metric stored in the object of Metric class. Methods of this class allows to receive data set and check for errors that may have been during the calculation.

```
import com.portfolioeffect.quant.client.api.Portfolio;
import com.portfolioeffect.quant.client.api.Position;
import com.portfolioeffect.quant.client.api.Metric;
import com.portfolioeffect.quant.client.model.ComputeErrorException;
...

//Timestamp in "t-N" format
//Creates new empty Portfolio using server data for index
//Explicitly specify the symbol for the index
Portfolio portfolio= new Portfolio("t-5", "t","SPY");

//Add  position without rebalancing with 100 assets for server data
Position positionGOOG = portfolio.add_position("GOOG", 100);
//Add  position without rebalancing with 100 assets for server data
Position positionAAPL = portfolio.add_position("AAPL", 100);

//Create  object of Metric class  containing variance of portfolio
Metric variancePortfolio = portfolio.variance();

//Create  object of Metric class  containing variance of GOOG position
Metric varianceGOOG = positionGOOG.variance();

//Check for errors in the calculation of variancePortfolio
if( variancePortfolio.hasError() ){
    //get error message
    String errorMsg = variance.getError()
    ...
}


...
//Another method for check for errors
try {
    double[] t = variance.getValue();
} catch (ComputeErrorException e) {
    //get error message
    String errorMsg = variance.getError();
    ...
}

...

//Get array of  portfolio variance values.
double[] variancePortfolioValue = variancePortfolio.getValue();

//Get the timestamps corresponding to the values in variancePortfolioValue
//Times expressed in milliseconds from 1970-01-01 00:00:00 EST
long[] variancePortfolioTime = variancePortfolio.getTime();

//Get last value of portfolio variance array
//This is equivalent to  variancePortfolioValue[ variancePortfolioValue.length -1]
double variancePortfolioLastValue = variancePortfolio.getLastValue();
```

```
//Get last value of variance timestamp array
//This is equivalent to  variancePortfolioTime[ variancePortfolioTime.length -1]
long variancePortfolioLastTime = variancePortfolio.getLastTime();



...
//Another method for check for errors
double[] variancePortfolioValue;
long[] variancePortfolioTime;
try {
    variancePortfolioValue = variancePortfolio.getValue();
    variancePortfolioTime = variancePortfolio.getTime();
} catch (ComputeErrorException e) {
    //get error message
    String errorMsg = variance.getError();
    ...
}
```

List of the available Portfolio and Position methods for calculating metrics

"alpha_exante()"
Creates (ex-ante) according to the Single Index Model.

"alpha_jensens()"
Creates Jensen's alpha (excess return) according to the Single Index Model.

"beta()"
Creates beta (market sensitivity) according to the Single Index Model.

"calmar_ratio()"
Creates Calmar ratio (cumulative return to maximum drawdown) of a portfolio

"cumulant(int order)"
Creates N-th cumulant of return distribution.

"down_capture_ratio()"
Creates down capture ratio.

"down_number_ratio()"
Creates down number rati.

"down_percentage_ratio()"
Creates down percentage ratio of returns.

"downside_variance (double thresholdReturn) "
Creates downside variance of returns.

"expected_downside_return (double thresholdReturn) "
Creates cumulative expected return below a certain threshold.

"expected_return()"
Creates cumulative expected return.

"expected_shortfall (double confidenceInterval) "
Creates conditional Value-at-Risk (Expected Tail Loss) at a given confidence interval.

"expected_upside_return (double thresholdReturn) "
Creates cumulative expected return above a certain threshold.

"fractal_dimension()"
Creates portfolio fractal dimension. Portfolio fractal dimension is a weighted sum of fractal dimensions of its position returns.

"gain_loss_variance_ratio()"
Creates gain to loss variance ratio of returns.

"gain_variance()"
Creates gain variance of returns.

" hurst_exponent()"
Creates Hurst exponent of returns. Portfolio Hurst exponent is a weighted sum of the Hurst exponents of its position returns.

"information_ratio()"
Creates information ratio.

"kurtosis()"
Creates kurtosis of returns.

"log_return()"
Creates log_return from the beginning of the holding period.

"loss_variance()"
Creates loss variance of returns.

"max_drawdown()"
Creates Creates maximum drawdown of preturns.

"mod_sharpe_ratio (double confidenceInterval) "
Creates modified Sharpe ratio at a given confidence interval.

"moment (int order) "
Creates N-th order central moment of return distribution.

"omega_ratio (double thresholdReturn) "
Creates Omega Ratio.

"profit()"
Creates profit.

"rachev_ratio (double confidenceIntervalA, double confidenceIntervalB) "
Creates Rachev ratio at given confidence intervals.

"sharpe_ratio()"
Creates Sharpe Ratio.

"skewness()"
Creates skewness of returns.

"sortino_ratio (double thresholdReturn) "
Creates Sortino ratio.

"starr_ratio (double confidenceInterval) "
Creates Stable Tail Adjusted Return Ratio (STARR) at given confidence intervals.

"treynor_ratio()"
Creates Treynor Ratio.

"txn_costs()"
Creates monetary value of accumulated transactional costs.

"up_capture_ratio()"
Creates up capture ratio.

"up_number_ratio()"
Creates up number ratio.

"up_percentage_ratio()"
Creates up percentage ratio.

"upside_downside_variance_ratio (double thresholdReturn) "
Creates upside to downside variance ratio.

"upside_variance (double thresholdReturn) "
Creates upside variance of returns.

"value_at_risk (double confidenceInterval) "
Creates portfolio Value-at-Risk at a given confidence interval.

"value()"
Creates monetary value from the beginning of the holding period.

"variance()"
Creates variance of returns.

List of available Position class specific methods for calculating metrics

" quantity()"
Creates total number of shares associated with the given position

"price()"
Creates position price

"weight()"
Creates ratio of a monetary position value to the monetary value of the whole portfolio. Expressed in decimal points of portfolio value.

"return_autocovariance(int lag)"
Creates autocovariance of position returns for a certain time lag in sec.

"correlation(Position positionB)"
Creates correlation between current position and positionB.

"covariance(Position positionB)"
Creates covariance between current position and positionB.

Tips and tricks for metric calculation

It should be noted that request to the server does not take place in time of creation Metric object, but in the moment of query calculation result(call such method as getValue(), hasError(), ...).

To save computing time, try to create metrics in groups. In this case it will generate batch request to the server. During batch request, some server calculations will not be repeated compare to single requests. So it will be faster. For example, metrics belonging to the same portfolio or positions metrics of a single portfolio could be calculated in 1 batch.

```
...
Portfolio portfolio= new Portfolio("t-5", "t","SPY");
Position positionGOOG = portfolio.add_position("GOOG", 100);
Position positionAAPL = portfolio.add_position("AAPL", 100);
```

```
//use this style
Metric variancePortfolio = portfolio.variance();
Metric varianceGOOG = positionGOOG.variance();
Metric varianceAAPL = positionAAPL.variance();
double[] variancePortfolioValue = variancePortfolio.getValue();//one request to server at this moment, will
    calculate variancePortfolio, varianceGOOG and varianceAAPL
double[] varianceGOOGValue = variancePortfolio.getValue();
long[] variancePortfolioTime = variancePortfolio.getTime();


//instead this
Metric variancePortfolio = portfolio.variance();
double[] variancePortfolioValue = variancePortfolio.getValue();//request to server
long[] variancePortfolioTime = variancePortfolio.getTime();
Metric varianceGOOG = positionGOOG.variance();
double[] varianceGOOGValue = variancePortfolio.getValue();//request to server
Metric varianceAAPL = positionAAPL.variance();
```

5 Portfolio Settings

Assignment portfolio settings is done using method settings(String key, String value).

Portfolio Metrics

These settings regulate how portfolio returns and return moments are computed

Portfolio Metrics Mode

One of the two modes for collecting portfolio metrics that could be used:

- "portfolio"- portfolio metrics are computed using previous history of position rebalancing. Portfolio risk and performance metrics account for the periods with no market exposure (i.e. when no positions are held) depending on the holding periods accounting settings (see holding periods mode below).
- "price" - at any given point of time, both position and portfolio metrics are computed for a buy-and-hold strategy. This mode is a common for classic portfolio theory and is often used in academic literature for portfolio optimization or when computing price statistics.

By default, mode is set to "portfolio".

```
// set "price" mode
portfolio.settings("portfolioMetricsMode", "price");

//set "portfolio" mode
portfolio.settings("portfolioMetricsMode", "portfolio");
```

Holding Periods Only

This setting should only be used when portfolio metrics mode is set to "portfolio". When holdingPeriodsOnly is set to "false", trading strategy risk and performance metrics will be annualized to include time intervals when strategy had no market exposure at certain points (i.e. when position quantity were zero). When set to "true", trading strategy metrics are annualized only based on actual holding intervals.

```
// enable holdingPeriodsOnly
portfolio.settings("holdingPeriodsOnly","true");

// disable holdingPeriodsOnly
portfolio.settings("holdingPeriodsOnly","false");
```

Short Sales Mode

This setting is used to specify how position weights are computed. Available modes are:

- "lintner" - the sum of absolute weights is equal to 1 (Lintner assumption)
- "markowitz" - the sum of weights must equal to 1 (Markowitz assumption)

Defaults to "lintner", which implies that the sum of absolute weights is used to normalize investment weights.

```
// weights are normalized based on a simple sum (Markowitz)
portfolio.settings("shortSalesMode","markowitz");


// weights are normalized based on a sum of absolute values (Lintner)
portfolio.settings("shortSalesMode","lintner");
```

Data Sampling

These settings regulate how results of portfolio computations are returned. Depending on your usage scenario, some of them might bring significantly improvement to speed of your portfolio computations

Results Sampling Interval

Interval to be used for sampling computed results before returning them to the caller. Available interval values are:

- "Xs" - seconds
- "Xm" - minutes
- "Xh" - hours
- "Xd" - trading days (6.5 hours in a trading day)
- "Xw" - weeks (5 trading days in 1 week)
- "Xmo" - month (21 trading day in 1 month)
- "Xy" - years (256 trading days in 1 year)
- "none" - no sampling.
- "last" - only the very last data point is returned

Large sampling interval would produce smaller vector of results and would require less time spent on data transfer. Default value of "1s" indicates that data is returned for every second during trading hours.

```


// sample results every 30 seconds
portfolio.settings("resultsSamplingInterval","30s");
variance_30s=portfolio.variance();

// sample results every 5 minutes
portfolio.settings("resultsSamplingInterval","30s");
```

Input Sampling Interval

Interval to be used as a minimum step for sampling input prices. Available interval values are:

- “Xs” - seconds
- “Xm” - minutes
- “Xh” - hours
- “Xd” - trading days (6.5 hours in a trading day)
- “Xw” - weeks (5 trading days in 1 week)
- “Xmo” - month (21 trading day in 1 month)
- “Xy” - years (256 trading days in 1 year)
- “none” - no sampling

Default value is “none”, which indicates that no sampling is applied.

```
// sample input prices every 30 seconds
portfolio.settings("inputSamplingInterval","30s");


//sample input prices every 5 min
portfolio.settings("inputSamplingInterval","5m");
variance_5m=portfolio.variance();
```

Model Pipeline

Window Length

Specifies rolling window length that should be used for computing portfolio and position metrics. When portfolio mode is set to “portfolio”, it is also the length of rebalancing history window to be used. Available interval values are:

- “Xs” - seconds
- “Xm” - minutes
- “Xh” - hours
- “Xd” - trading days (6.5 calendar hours in a trading day)
- “Xw” - weeks (5 trading days in 1 week)
- “Xmo” - month (21 trading day in 1 month)
- “Xy” - years (256 trading days in 1 year)
- “all” - all observations are used

Default value is “1d” - one trading day.

```
// 1 hour rolling window
portfolio.settings("windowLength","1h");


// 1 week rolling window
portfolio.settings("windowLength","1d");
```

Time Scale

Interval to be used for scaling return distribution statistics and producing metrics forecasts at different horizons. Available interval values are:

- “Xs” - seconds
- “Xm” - minutes
- “Xh” - hours
- “Xd” - trading days (6.5 hours in a trading day)
- “Xw” - weeks (5 trading days in 1 week)
- “Xmo” - month (21 trading day in 1 month)
- “Xy” - years (256 trading days in 1 year)
- “all” - actual interval specified during portfolio creation.

Default value is ”1d” - one trading day.

```
// 1 hour time scale
portfolio.settings("timeScale","1h");

// 1 day time scale
portfolio.settings("timeScale","1d");
```

Microstructure Noise Model

Enables market microstructure noise model of distribution returns.
Defaults to ’true’, which means that microstructure effects are modeled and resulting HF noise is removed from metric calculations. When ’false’, HF microstructure noise is not separated from asset returns, which at high trading frequencies could yield noise-contaminated results.

```
// HF noise model is enabled
portfolio.settings("noiseModel","true");

// HF noise model is disabled
portfolio.settings("noiseModel","false");
```

Jumps/Outliers Model

Used to select jump filtering mode when computing return statistics. Available modes are:

- "none" - price jumps are not filtered anywhere
- "moments" - price jumps are filtered only when computing return moments (i.e. for expected return, variance, skewness, kurtosis and derived metrics)
- "all" - price jumps are filtered from computed returns, prices and all return metrics.

```
// Price jumps detection is enabled for returns and moments
portfolio.settings("jumpsModel","all");


// Price jumps detection is disabled
portfolio.settings("jumpsModel","none");
```

Density Model

Used to select density approximation model of return distribution. Available models are:

- "GLD" - Generalized Lambda Distribution
- "CORNER_FISHER" - Corner-Fisher approximation
- "NORMAL" - Gaussian distribution

Defaults to "GLD", which would fit a very broad range of distribution shapes.

```
// Using normal density
portfolio.settings("densityModel","NORMAL");

// Using Generalized Lambda density
portfolio.settings("densityModel","GLD");
```

Factor Model

Factor model to be used when computing portfolio metrics. Available models are:

- "sim" - portfolio metrics are computed using the Single Index Model
- "direct" - portfolio metrics are computed using portfolio value itself (experimental)

Defaults to "sim", which implies that the Single Index Model is used to compute portfolio metrics.

```
// Single Index Model is used
portfolio.settings("factorModel","sim");

// Direct model is used
portfolio.settings("factorModel","direct");
```

Fractal Price Model

Used to enable mono-fractal price assumptions (fGBM) when time scaling return moments. Defaults to 'true', which implies that computed Hurst exponent is used to scale return moments. When 'false', price is assumed to follow regular GBM with Hurst exponent = 0.5.

```
// Fractal price model is enabled
portfolio.settings("fractalPriceModel","true");
variance_fractal=portfolio.variance();

// Fractal price model is disabled
portfolio.settings("fractalPriceModel","false");
```

Drift Term

Used to enable drift term (expected return) when computing probability density approximation and related metrics (e.g. CVaR, Omega Ratio, etc.). Defaults to 'false', which implies that distribution is centered around expected return.

```
// Drift term is enabled
portfolio.settings("driftTerm","true");


// Drift term is disabled
portfolio.settings("driftTerm", "false");
```

Transactional Costs

These settings provide a framework for adding variable and fixed transactional costs into return, expected return and profit calculations. All metrics based on expected return like Sharpe Ratio, VaR (with drift term enabled) would reflect transactional costs in their computations.

Cost Per Share

Amount of transaction costs per share. Default value is 0.

```
// Transactional costs per share are 0.5 cent
portfolio.settings("txnCostPerShare","0.005");

// Transactional costs per share are 0.1 cent
portfolio.settings("txnCostPerShare","0.001");
```

Cost Per Transaction

Amount of fixed costs per transaction. Defaults to 0.

```
                                    "2014-10-02 11:30:00"])
```

```
// Fixed costs per transaction are 9 dollars
portfolio.settings("txnCostFixed","9.0");

// Fixed costs per transaction are 1 dollar
portfolio.settings("txnCostFixed","1.0");
```

6 Portfolio Optimization

Optimization Goals & Constraints

A classic problem of constructing a portfolio that meets certain maximization/minimization goals and constraints is addressed in our version of a multi-start portfolio optimization algorithm. At every time step optimization algorithm tries to find position weights that best meet optimization goals and constraints.

Key Features

- A multi-start approach is used to compare local optima with each other and select a global optimum. Local optima are computed using a modified Hooke-Jeeves method.
- When optimization algorithm is supplied with mutually exclusive constraints, it would try to produce result that is equally close (in absolute terms) to all constraint boundaries. For instance, constraints "x $>$ 6" and "x $<$ 4" are mutually exclusive, so the optimization algorithm would choose "x = 5", which is a value that has the smallest distance to both constraints.
- Portfolio metrics change over time, but optimization uses only the latest value in the time series. Therefore, the faster metric series would change, the more likely current optimal weights would deviate from the optimal weights at the next time step.
- Optimization results depend on provided portfolio settings. For example, short windowLength would produce "spot" versions of portfolio metrics and computed optimal weights would change faster to reflect shortened metric horizon.

Optimization Goals

Optimization algorithm requires a single maximization/minimization goal to be set using constructor of Optimizer class; created Optimizer object could be used to add optional optimization costraints and then call optimization_run() method to launch portfolio optimization.

```
import com.portfolioeffect.quant.client.api.Metric;
import com.portfolioeffect.quant.client.api.Optimizer;
import com.portfolioeffect.quant.client.api.Portfolio;
import com.portfolioeffect.quant.client.api.Position;
...
Portfolio portfolio=new Portfolio("2014-10-01 09:30:00", "2014-10-02 16:00:00");
Position positionC=portfolio.add_position("C",500);
Position positionGOOG=portfolio.add_position("GOOG",600);
portfolio.settings("portfolioMetricsMode","price").settings("resultsSamplingInterval","30m");

//created optimizer object, set optimization goal as log_return and optimization direction as maximization
Optimizer optimizer= new Optimizer(portfolio.log_return(),"max");

// launch optimization and obtain appropriate Metric object
Metric optimalPortfolioMeric=optimizer.run();
// get optimal portfolio
Portfolio optimalPortfolio = optimalPortfolioMetric.getPortfolio();
// get a metic of optimal portfolio
```

```
Metric optimalReturn = optimalPortfolio.log_return();
```

Adding Constraints

Optimization constraints cover both metric-based and weight-based constraints. Metric-based constraints limit portfolio-level metrics to a certain range of values. For example, zero beta constraint would produce market-neutral optimal portfolio. Weight-based constraints operate on optimal position weights or sum of weights to give control over position concentration risks or short-sales assumptions.

Constraint methods could be chained to produce complex optimization rules:

Since position quantities are integer numbers and weights are decimals, a discretization error is introduced while converting optimal position weights to corresponding quantities. By default, optimal portfolio starts with a value of the initial portfolio. Portfolio value could be fixed to a constant level at every optimization step (see corresponding constraint below). Higher portfolio value could be used to keep difference between computed optimal weights and effective weights based on position quantities small. Lower portfolio value or higher asset price would normally increase discretization error.

```
import com.portfolioeffect.quant.client.api.Metric;
import com.portfolioeffect.quant.client.api.Optimizer;
import com.portfolioeffect.quant.client.api.Portfolio;
import com.portfolioeffect.quant.client.api.Position;
...
Portfolio portfolio=new Portfolio("2014-10-01 09:30:00", "2014-10-02 16:00:00");
Position positionC=portfolio.add_position("C",500);
Position positionGOOG=portfolio.add_position("GOOG",600);
Position positionMSFT=portfolio.add_position('MSFT',200);
portfolio.settings("portfolioMetricsMode","price").settings("resultsSamplingInterval","30m");



Optimizer optimizer= new Optimizer(portfolio.variance(),"min");

// add constraints
optimizer.constraint(portfolio.beta(),"<=",1);
optimizer.constraint(positionGOOG.weight(),"=",0.1);

// launch optimization and obtain optimal portfolio
Portfolio optimalPortfolio=optimizer.run().getPortfolio();
```

Use portfolio metric "value" to specify the monetary value in US dollars of the optimized portfolio. This value will be divided between the positions according to the goal and constraints of optimization.

```
import com.portfolioeffect.quant.client.api.Metric;
import com.portfolioeffect.quant.client.api.Optimizer;
import com.portfolioeffect.quant.client.api.Portfolio;
import com.portfolioeffect.quant.client.api.Position;
...
Portfolio portfolio=new Portfolio("2014-10-01 09:30:00", "2014-10-02 16:00:00");
Position positionC=portfolio.add_position("C",500);
Position positionGOOG=portfolio.add_position("GOOG",600);
Position positionMSFT=portfolio.add_position('MSFT',200);
portfolio.settings("portfolioMetricsMode","price").settings("resultsSamplingInterval","30m");



Optimizer optimizer= new Optimizer(portfolio.variance(),"min");

//Set the monetary value in US dollars of the optimized portfolio.
optimizer.constraint(portfolio.value(),"=",1e6);


// launch optimization and obtain optimal portfolio
```

```
Portfolio optimalPortfolio=optimizer.run().getPortfolio();
```

The following portfolio metrics could currently be used as optimization goals or constraints:

"alpha_exante()"
Creates (ex-ante) according to the Single Index Model.

"beta()"

"alpha_jensens()"
Creates Jensen's alpha (excess return) according to the Single Index Model.

Creates beta (market sensitivity) according to the Single Index Model.

"cumulant(int order)"
Creates N-th cumulant of return distribution.

"downside_variance (double thresholdReturn) "
Creates downside variance of returns.

"expected_return()"
Creates cumulative expected return.

"expected_upside_return (double thresholdReturn) "
Creates cumulative expected return above a certain threshold.

"expected_downside_return (double thresholdReturn) "
Creates cumulative expected return below a certain threshold.

"expected_shortfall (double confidenceInterval) "
Creates conditional Value-at-Risk (Expected Tail Loss) at a given confidence interval.

"gain_loss_variance_ratio()"
Creates gain to loss variance ratio of returns.

"gain_variance()"
Creates gain variance of returns.

"information_ratio()"
Creates information ratio.

"kurtosis()"
Creates kurtosis of returns.

"log_return()"
Creates log_return from the beginning of the holding period.

"loss_variance()"
Creates loss variance of returns.

"mod_sharpe_ratio (double confidenceInterval) "
Creates modified Sharpe ratio at a given confidence interval.

"moment (int order) "
Creates N-th order central moment of return distribution.

"omega_ratio (double thresholdReturn) "
Creates Omega Ratio.

"rachev_ratio (double confidenceIntervalA, double confidenceIntervalB) "
Creates Rachev ratio at given confidence intervals.

"sharpe_ratio()"
Creates Sharpe Ratio.

"skewness()"
Creates skewness of returns.

"starr_ratio (double confidenceInterval) "
Creates Stable Tail Adjusted Return Ratio (STARR) at given confidence intervals.

"sortino_ratio (double thresholdReturn) "
Creates Sortino ratio.

"treynor_ratio()"
Creates Treynor Ratio.

"upside_downside_variance_ratio (double thresholdReturn) "
Creates upside to downside variance ratio.

"upside_variance (double thresholdReturn) "
Creates upside variance of returns.

"value_at_risk (double confidenceInterval) "
Creates portfolio Value-at-Risk at a given confidence interval.

"variance()"
Creates variance of returns.

The following portfolio metrics could currently be used as optimization goals:

"constraints_only()"
The goal of optimization is only to satisfy the all constraints.

"equiweight()
No optimization is performed and constraints are not processes. Portfolio positions are returned with equal weights

The following position metric could currently be used as optimization constraint:

"weight()"
Creates ratio of a monetary position value to the monetary value of the whole portfolio. Expressed in decimal points of portfolio value.