www.portfolioeffect.com
High Frequency Portfolio Analytics

User Manual

PortfolioEffect hft Package for Python

Stephanie Toper
stephanie.toper@portfolioeffect.com

Andrey Kostin
andrey.kostin@portfolioeffect.com

Released Under GPL-3 License
by Snowfall Systems, Inc.

October 20, 2016

Contents

1 Package Installation

PortfolioEffect hft package for Python relies on a Java package, which assumes that Java runtime is installed and configured on your system. To install Java runtime and to configure your Python engine to work with it, follow these steps.

1.1 Install Latest JDK/JRE Runtime

1.1.1 Windows or Linux

Download and install latest Java distribution (JDK or JRE) for your platform from Oracle's website

1.1.2 Mac

Download and install latest Java distribution (JDK or JRE) for your platform from app store

1.2 Configure Java Environment

1.2.1 Windows

If you are using Windows, you need to set up JAVA_HOME environment variable. Java could have been installed to a different folder and jdk version could be different, this is an example.

```
JAVA_HOME  C:\Program Files\Java\jdk1.8.0\_101
```

In your System variables path, you should add:

```
Path  C:\Program Files\Java\jdk1.8.0\_101\jre\bin\server (or client);
      C:\Program Files\Java\jdk1.8.0\_101\bin
```

Note: Please check that you downloaded the same version, if not just update the version number in the path (e.g.: Let's say you downloaded 6 version, change jdk1.8.0_101 to jdk1.6.0_101)

1.2.2 Linux

If you are on Linux and you used a tarball file, you will need to manually append the following lines to /etc/environment using your favorite text editor:

```
export JAVA_HOME=/path/to/java/folder
export PATH=$PATH:$JAVA_HOME/bin
```

Apply environment changes:

```
source /etc/environment
```

1.2.3 Mac

In Mac OSX 10.5 or later, set the JAVA_HOME variable

```
$ vim .bash_profile

export JAVA_HOME=$(/usr/libexec/java_home)

$ source .bash_profile

$ echo $JAVA_HOME
/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

1.3 Installation hft Package

1.3.1 With Anaconda

Download and installed Anaconda2 for Python 2.7 at https://www.continuum.io/downloads.

Once installed, open command line and run:

```
conda install -c portfolioeffect hft
```

Anaconda installation is the most straightforward as the binary are already ready for the different platforms.

1.3.2 With Pip

C++ compiler for Python (optional)

On windows you can download: Microsoft Visual C++ Compiler for Python 2.7 For Ubuntu or Mac it should integrated. Linux and Mac Os have by default a compiler.

Pip install

You are now ready to install the PortfolioEffect hft package directly from Pypi

```
pip install hft
```

www.portfolioeffect.com downloads section.

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.

2.1 Locate API Credentials

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

2.2 Set API Credentials in Python

Run the following commands to set your account API credentials for the PortfolioEffect hft Python Package installed.

```
util_setCredentials("API Username", "API Password", "API Key")
```

You are now ready to call PortfolioEffect hft methods.

3 Portfolio Construction

3.1 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.

3.1.1 Create Portfolio

Method Portfolio(). takes a list of index prices in the format [UTC timestamp, price] with UTC timestamp expressed in milliseconds from 1970-01-01 00:00:00 EST.

```
[1409666400000L, 1409668200000L, 1409670000000L, 1409671800000L, 1409673600000L, 1409675400000L, 1409677200000L,
    1409679000000L, 1409680800000L, 1409682600000L, 1409684400000L, 1409686200000L, 1409688000000L]
[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]
```

If index symbol is specified, it is silently ignored.

```
data(spy.data)

# Create portfolio
portfolio=Portfolio(priceDataIx=spy_data)
```

3.1.2 Add Positions

Positions are added using portfolio.add_position() with 'priceData' in the same format as index price.

```
data(goog.data)
data(aapl.data)

# Single position without rebalancing
positionGOOG=portfolio.add_position(
  symbol='GOOG',
  quantity=100,
  priceData=goog_data)

# Single position with rebalancing
positionAAPL=portfolio.add_position(
  symbol='AAPL',
  quantity=[300,150],
  time=["2014-09-01 09:00:00","2014-09-07 14:30:00"],
  priceData=aapl_data)
```

3.2 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.

3.2.1 Create Portfolio

Method Portfolio() creates new asset portfolio or overwrites an existing portfolio object with the same name.

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

```
# Timestamp in "yyyy-MM-dd HH:MM:SS" format
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

# Timestamp in "yyyy-MM-dd" format
portfolio=Portfolio(fromTime="2014-10-01",
                    toTime="2014-10-02")

# Timestamp in "t-N" format
portfolio=Portfolio(fromTime="t-5",
                    toTime="t")
```

3.2.2 Get Symbols List

Once portfolio is created, portfolio_availableSymbols() 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.

```
portfolio.symbols_available()

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

3.2.3 Add Positions

Positions are added by calling position_add() method on a portfolio object with a list of symbols and quantities. For positions that were rebalanced or had non-default holding periods a 'time' argument could be used to specify rebalancing timestamps.

```
# Single position without rebalancing
positionGOOG=portfolio.add_position(
  symbol='GOOG',
  quantity=100)


# Single position with rebalancing
positionAAPL=portfolio.add_position(
  symbol='AAPL',
  quantity=[300,150],
  time=["2014-09-01 09:00:00","2014-09-07 14:30:00"])
```

4 Portfolio Settings

4.1 Portfolio Metrics

These settings regulate how portfolio returns and return moments are computed

4.1.1 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".

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionAAPL=portfolio.add_position(
  symbol='AAPL', quantity=[300,150],
  time=["2014-10-01 09:30:00","2014-10-02 09:30:00"])

positionGOOG=portfolio.add_position(
  symbol='GOOG', quantity=[100,150],
  time=["2014-10-01 09:30:00","2014-10-02 09:30:00"])

# "price" mode
portfolio.settings( portfolioMetricsMode="price")
variance_price=portfolio.variance();

# "portfolio" mode
portfolio.settings( portfolioMetricsMode="portfolio")
variance_portfolio=portfolio.variance();
util_plot(variance_portfolio,variance_price,title="Variance, portfolioMetricsMode",legend=["portfolio","price"])
```

4.1.2 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.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionAAPL=portfolio.add_position('AAPL',
                                    [0,300,0,150],
                                    time=["2014-09-30 09:30:00",
                                          "2014-10-01 09:30:00",
                                          "2014-10-01 15:30:00",
                                          "2014-10-02 11:30:00"])

# enable holdingPeriodsOnly
portfolio.settings( holdingPeriodsOnly='true')
variance_holdingPeriodsOnly_TRUE=portfolio.variance();


# disable holdingPeriodsOnly
portfolio.settings( holdingPeriodsOnly='false')
variance_holdingPeriodsOnly_FALSE=portfolio.variance();

util_plot(variance_holdingPeriodsOnly_TRUE,variance_holdingPeriodsOnly_FALSE,title="Variance,holdingPeriodsOnly",
     legend=["true","false"])
```

4.1.3 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.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionC=portfolio.add_position('C',-500,)
positionGOOG=portfolio.add_position('GOOG',600)

# weights are normalized based on a simple sum (Markowitz)
portfolio.settings( shortSalesMode="markowitz")
variance_markowitz=portfolio.variance();

# weights are normalized based on a sum of absolute values (Lintner)
portfolio.settings( shortSalesMode="lintner")
variance_lintner=portfolio.variance();

util_plot(variance_markowitz,variance_lintner,title="Variance,shortSalesMode",legend=["markowitz","lintner"])
```

4.2 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

4.2.1 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.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-01 16:00:00")

positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)

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

# sample results every 5 minutes
portfolio.settings( resultsSamplingInterval="15m")
variance_15m=portfolio.variance();

util_plot(variance_30s,variance_15m,title="Variance,resultsSamplingInterval",legend=["30s","15m"])
```

4.2.2 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.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)

# sample input prices every 30 seconds
portfolio.settings( inputSamplingInterval="30s")
variance_30s=portfolio.variance();

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

util_plot(variance_30s,variance_5m,title="Variance,inputSamplingInterval",legend=["30s","5m"])
```

4.3 Model Pipeline

4.3.1 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.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)

# 1 hour rolling window
portfolio.settings( windowLength="1h")
variance_1h=portfolio.variance();

# 1 week rolling window
portfolio.settings( windowLength="1d")
variance_1d=portfolio.variance();

util_plot(variance_1h,variance_1d,title="Variance,windowLength",legend=["1h","1d"])
```

4.3.2 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.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)

# 1 hour time scale
portfolio.settings( timeScale="1h")
variance_1h=portfolio.variance();

# 1 week time scale
portfolio.settings( timeScale="1d")
variance_1d=portfolio.variance();

util_plot(variance_1h,variance_1d,title="Variance,timeScale",legend=["1h","1d"])
```

4.3.3 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.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)

# HF noise model is enabled
portfolio.settings( noiseModel='true')
variance_noiseModel_TRUE=portfolio.variance();

# HF noise model is disabled
portfolio.settings( noiseModel='false')
variance_noiseModel_FALSE=portfolio.variance();

util_plot(variance_noiseModel_TRUE,variance_noiseModel_FALSE,title="Variance,noiseModel",legend=["true","false"])
```

4.3.4 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.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)

# Price jumps detection is enabled for returns and moments
portfolio.settings( jumpsModel="all")
variance_all=portfolio.variance();

# Price jumps detection is disabled
portfolio.settings( jumpsModel="none")
variance_none=portfolio.variance();

util_plot(variance_all,variance_none,title="Variance,jumpsModel",legend=["all","none"])
```

4.3.5 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.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)

# Using normal density
portfolio.settings( densityModel="NORMAL")
util_util_plotDensity(dist_density(portfolio,pValueLeft=0.6,pValueRight=1,nPoints=100,addNormalDensity='true'))

# Using Generalized Lambda density
portfolio.settings( densityModel="GLD")
util_util_plotDensity(dist_density(portfolio,pValueLeft=0.6,pValueRight=1,nPoints=100,addNormalDensity='true'))
```

4.3.6 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.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)

# Single Index Model is used
portfolio.settings( factorModel="sim")
variance_sim=portfolio.variance();

# Direct model is used
portfolio.settings( factorModel="direct")
variance_direct=portfolio.variance();

util_plot(variance_sim,variance_direct,title="Variance,factorModel",legend=["sim","direct"])
```

4.3.7 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.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)

# Fractal price model is enabled
portfolio.settings( fractalPriceModel='true')
variance_fractal=portfolio.variance();

# Fractal price model is disabled
portfolio.settings( fractalPriceModel='false')
variance_nonfractal=portfolio.variance();

util_plot(variance_fractal,variance_nonfractal,title="Variance,fractalPriceModel",legend=["enabled","disabled"])
```

4.3.8 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.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00", toTime="2014-10-02 16:00:00")

positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)

# Drift term is enabled
portfolio.settings( driftTerm='true')
ES_driftTerm_TRUE=portfolio.expected_shortfall(0.95);
```

```
# Drift term is disabled
portfolio.settings( driftTerm='false')
ES_driftTerm_FALSE=portfolio.expected_shortfall(0.95);

util_plot(ES_driftTerm_TRUE,ES_driftTerm_FALSE,title="Expected Shortfall,driftTerm",legend=["true","false"])
```

4.4 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.

4.4.1 Cost Per Share

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

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionAAPL=portfolio.add_position('AAPL',
                                    [100,300,500,150],
time=["2014-10-01 09:30:00",
      "2014-10-01 11:30:00",
      "2014-10-01 15:30:00",
      "2014-10-02 11:30:00"])

# Transactional costs per share are 0.5 cent
portfolio.settings( txnCostPerShare=0.005)
return_5=portfolio.log_return()
txn_costs_5=portfolio.txn_costs()

# Transactional costs per share are 0.1 cent
portfolio.settings( txnCostPerShare=0.001)
return_1=portfolio.log_return()
txn_costs_1=portfolio.txn_costs()

util_plot(return_5,return_1,title="Return,txnCostPerShare",legend=["0.005","0.001"])

util_plot(txn_costs_5,txn_costs_1,title="Txn Costs,txnCostPerShare",legend=["0.005","0.001"])
```

4.4.2 Cost Per Transaction

Amount of fixed costs per transaction. Defaults to 0.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")
positionAAPL=portfolio.add_position('AAPL',
                                    [100,300,500,150],
                                    time=["2014-10-01 09:30:00",
                                          "2014-10-01 11:30:00",
                                          "2014-10-01 15:30:00",
                                          "2014-10-02 11:30:00"])

# Fixed costs per transaction are 9 dollars
portfolio.settings( txnCostFixed=19.0)
return_19=portfolio.log_return()
txn_costs_19=portfolio.txn_costs()
```

```
# Fixed costs per transaction are 1 dollar
portfolio.settings( txnCostFixed=1.0)
return_1=portfolio.log_return()
txn_costs_1=portfolio.txn_costs()

util_plot(return_19,return_1,title="Return,txnCostFixed",legend=["19.0","1.0"])

util_plot(txn_costs_19,txn_costs_1,title="Txn Costs,txnCostFixed",legend=["19.0","1.0"])
```

5 Portfolio Optimization

5.1 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.

5.1.1 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 method of parallel tangents (PARTAN).
- 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.

5.1.2 Optimization Goals

Optimization algorithm requires a single maximization/minimization goal to be set using optimization_goal() method that operates on a portfolio (see portfolio construction). Returned optimizer object could be used to add optional optimization costraints and then passed to the optimization_run method to launch portfolio optimization.

```
portfolio=Portfolio(fromTime="2014-10-01 09:30:00", toTime="2014-10-02 16:00:00")
positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)
portfolio.settings(
  portfolioMetricsMode='price',
  resultsSamplingInterval='30m')

# set optimization goal
optimizer=Optimizer(portfolio.log_return(),"max")

# launch optimization and obtain optimal portfolio
optimalPortfolio=optimizer.run()

util_plot(portfolio.log_return(),optimalPortfolio.log_return(),title="Portfolio Return",legend=["Simple Portfolio"
    ,"Optimal Portfolio"])
```

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

“portfolio.variance()”
portfolio returns variance

“portfolio.value_at_risk()”
portfolio Value-at-Risk

“portfolio.expected_shortfall()”
portfolio Conditional Value-at-Risk (Expected Tail Loss)

“portfolio.expected_return()”
portfolio expected return

“portfolio.log_return()”
portfolio return

“portfolio.sharpe_ratio()”
portfolio Sharpe Ratio

“portfolio.mod_sharpe_ratio()”
portfolio modified Sharpe Ratio

“portfolio.starr_ratio()”
portfolio STARR Ratio

“portfolio.weight_transform(”equiweight”)”
no optimization is performed and constraints are not processes. Portfolio positions are returned with equal weights

5.1.3 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.

```
# create portfolio and add positions
portfolio=Portfolio(fromTime="2014-10-01 09:30:00", toTime="2014-10-05 16:00:00")
positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)
positionAAPL=portfolio.add_position('MSFT',200)

portfolio.settings(
  portfolioMetricsMode='price',
  resultsSamplingInterval='30m')

# set optimization goal
optimizer=Optimizer(portfolio.variance(),"min")

# add constraints
optimizer=optimizer.constraint(portfolio.beta(),"<=",1)
```

```
optimizer=optimizer.constraint(positionGOOG.weight(),"=",0.1)

# launch optimization and obtain optimal portfolio
optimalPortfolio=optimizer.run()

util_plot(portfolio.variance(),optimalPortfolio.variance(),title="Variance",legend=["Simple Portfolio","Optimal
    Portfolio"])

util_plot(portfolio.beta(),optimalPortfolio.beta(),title="Beta",legend=["Simple Portfolio","Optimal Portfolio"])

util_plot(portfolio.get_position('GOOG').weight(),optimalPortfolio.get_position('GOOG').weight(),title="Weight
    GOOG",legend=["Simple Portfolio","Optimal Portfolio"])
```

5.1.4 Scalar Constraints

Scalar constraints are the simplest type of optimization boundaries. They require a single constant that is applied over a full time span of portfolio optimization. An example below sets portfolio beta constraint to be less or equal to 0.1.

```
# create portfolio and add positions
portfolio=Portfolio(fromTime="2014-10-01 09:30:00",
                    toTime="2014-10-02 16:00:00")

positionC=portfolio.add_position('C',500)
positionGOOG=portfolio.add_position('GOOG',600)

# rebalancing every minute, static portfolio ignores rebalancing history
portfolio.settings(
  portfolioMetricsMode='price',
  resultsSamplingInterval='30m')

# set optimization goal and define constraints
optimizer=Optimizer(portfolio.sharpe_ratio(),"max")
optimizer=optimizer.constraint(portfolio.beta(), "<=", constraintValue = 0.1)

# run optimization
optimalPortfolio=optimizer.run()

# util_plot results
util_plot(optimalPortfolio.beta(),portfolio.beta(), title="Beta", legend=["Optimal Beta","Original Beta"])
```