Welcome to PARIS

* Introduction of the Performance And Risk Integrated Investment System

Developer: Yuting Li (01266903)

Imperial College London

Table of Contents

**Design Concept****2**

Current Issue2

Project Target2

**System Explanation3**

Modules Overview4

Graphic Demonstration4

Data Base4

Signal Generator 5

Backtestor 5

Risk Performance Advisor 8

Live Trading Assistant8

**Area of Improvement 10**

**Appendix 11**

# **Design Concept**

## Current Issue:

It is not unusual when we find investors making investment decisions largely depend on subjective judgements, without awareness of what factors they are loading on and what risk levels they are exposed to. This is also an explanation for why back testing results are often unrepeatable. In designing and back testing trading strategies, users would inevitably tune strategy parameters to make performance look great. Sometimes it’s intentional as sell side analysts trying to sell their strategies to the clients, but more often it’s because investors/researchers lack the awareness of what actually made their performance look so great. As such they have no assurance of whether the spectacular performance will continue when the trading strategy is put to live use. Not to mention the continuous diligence required to constantly monitor a strategy’s strength and weakness after it is in live trading. When it becomes too troublesome or technically impossible for investors to monitor their trading decisions rationally, the risk of investors going for gut feelings increase.

Here in the ARR Investment Partners we aim to examine our investment decision process with full honesty, to fairly assess the market timing and stock picking skills of each managers and each subsequent trading strategies. As such we need a system that helps us automate risk and performance monitoring from strategy designing to subsequent automatic trading.

## Project Target

To tackle this exact problem, I designed a ‘*Performance And Risk Integrated (Investment) System’,* or in abbreviation ‘PARIS’. PARIS is a system aiming to help users easily:

1. Perform Risk and Performance analysis on investors’ historical trades to assess their stock picking and market timing skills
2. Implement and back test new trading strategies/ideas
3. Perform risk and performance analysis on these tested trading strategies to know how and why they perform
4. Automate live trading of these outstanding trading strategies/ideas
5. Constant live monitoring of the portfolio holding and show live risk/performance metrics
6. Continuously update and build a complete price/volume database.

# **System Explanation**

As shown in the Chart above, PARIS is made of seven closely inter-connected parts. This chapter starts with an overview of them, then followed by a user demonstration. I’d recommend reading the detailed description section alongside the ‘PARIS – Complete Demonstration’ Jupyter Notebook file. Each python module of the PARIS has a separate documentation provided in the appendix, detailing code structure and all functionalities. User of this manual is encouraged to check out these detailed documentations when having any confusion. All python dependencies are described in detail in each documentation, please ensure those packages are installed before start running PARIS.

In case user lacks experience of the finance market: PARIS is an in-house investment management system not a broker! The current version of PARIS is built on the broker MetaTrader4 (MT4).

I have described in ‘Software\_Configuration.docx’ file how to set up the MT4 software and provided a paper trading account for demonstration, please check it out before start using PARIS.

Not all python dependencies can be pip-installed, one special library our Live Trading Assistant heavily relying on is DarwinEx Lab ZMQ Connector. Please check out Software\_Configuration file for details. Web link referenced here: <https://github.com/darwinex/dwx-zeromq-connector>

## Overview of modules:

1. A Database that contains different data sources, including market information (Price/ Volume), User information (Historical trades) and strategy information (Signal Pool).
2. Signal Generator that calls on the price information from database, alongside other external factors, to make trading decisions and record the signal into the Signal Pool.
3. Backtestor calls price from Price database and either read from Signal pool or calls Signal Generator to form a back test of trading strategies. Then it passes the back-test result onto risk advisor and performance advisor to assess these metrics.
4. Risk Advisor can work out 13 different metrics. It by default runs analysis on users’ historical trades. It is connected to Backtestor to assess the risks of back-tested strategy. It is also connected to Live Trading Assistant to assess the risks of streaming portfolio holding every few seconds. It can be used to assess risks of hypothetical positions too.
5. Performance Advisor calculates several metrics to help investor understand where the returns come from. It is a derived class from Risk Advisor, so it has all the flexibility previously described of Risk Advisor.
6. Live Trading Assistant handles communication with the MT4 platform including requesting and receiving current position holdings, subscribe and collect live market data, trade executions. Live Trading App connects with Risk Advisor and Performance Advisor to give a live-updating demonstration of the portfolio risk metrics and return attributions. It also connects with the database to add latest price data into the database.

## Detailed Demonstration of module:

### Database

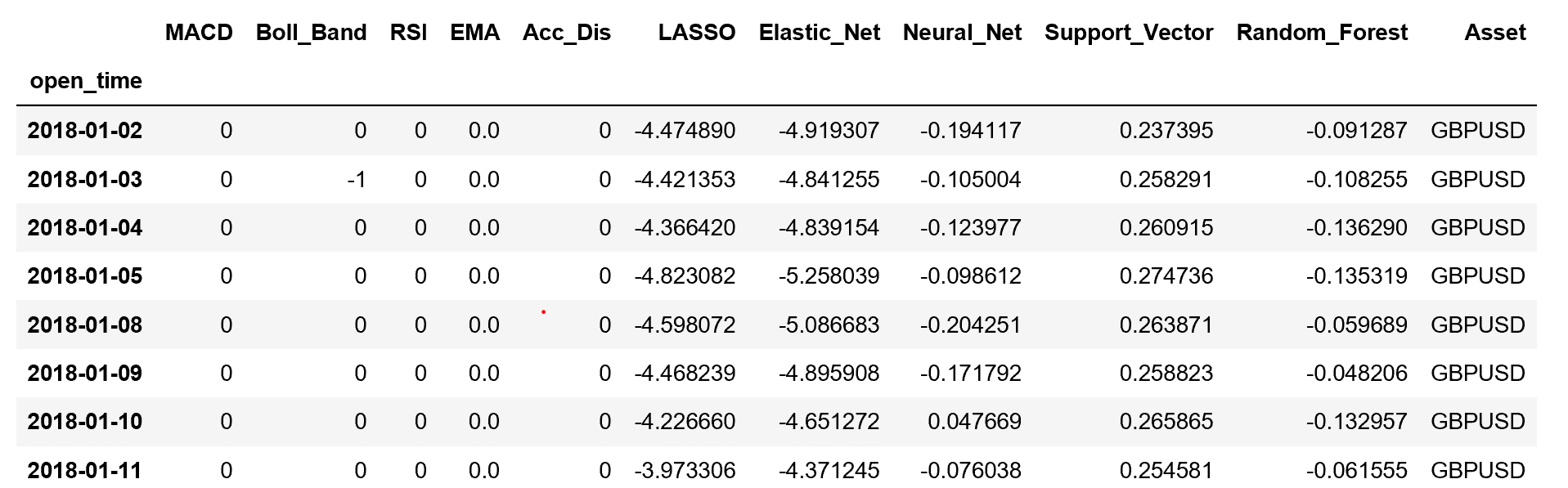
This part focus on description of data in database. In terms of how to build this database from scratch please check out appendix 1.01.

In our database we have in total 126 different assets, those are all assets tradable on the OANDA-MT4 account. Following is a list of their MT4 tickers. They mainly consists of fx, index futures, commodities and bond futures. In our database we stored 8 different frequencies of all these market information: These frequencies are : 1min, 5min, 15min, 30min, 1hour, 4hours, daily and weekly. Naturally the lower frequency the longer our record is. I conducted a data integrity summarisation that summarises the start, end, existing record and missing record of each asset at each frequency. Integrity report is then stored on a csv file for user to check out too. Overall Hour\_4 and Daily data appears to be relatively clean. (Although it looks as if the daily data has a lot of missing values, it’s actually because BITCOIN sometimes have weekend values making other asset appear to be missing value while they’re not trading on weekends. On the Jupyter Notebook demonstration, user can also check out the missing days of each asset at each frequency. MT4 appears to have long period of data missing as a chunk while other time periods data is clean. This is beneficial for the back testing as we don’t want data to be unclean throughout. Currently we can still perform back testing using the period before and after the ‘missing period’.

### Signal\_Generator

Signal\_Generator generates all the trading signals. Currently I programmed 5 Technical Signals (MACD, RSI, BB, EMA and Accumulated Distribution) and 5 Machine Learning Signals (LASSO, Elastic Net, Neural Network, Support Vector and Random Forest).

In the Demonstration I showed how it worked on generating all signal on GBPUSD on daily trading frequency. On Jupyter user can see the following result for 2018:



Focusing especially on the machine learning signals, we generated them using a rolling window with window size of 200 days (default) and refitting frequency of 100 days. The model is trained using lagged return and volume of all assets as well as 5 day accumulative return and volume as model input. One filter is however the asset cannot have more than 10 missing values during the training period otherwise it will be dropped from feature pool. The model is trained toward predicting 3 day forward looking accumulative returns in terms of percentage. Several models above require a standardisation. In order to prevent information leakage problem, I used the mean and standard deviation of features in their training period to normalise the features in the prediction period too. i.e. .

### Backtestor

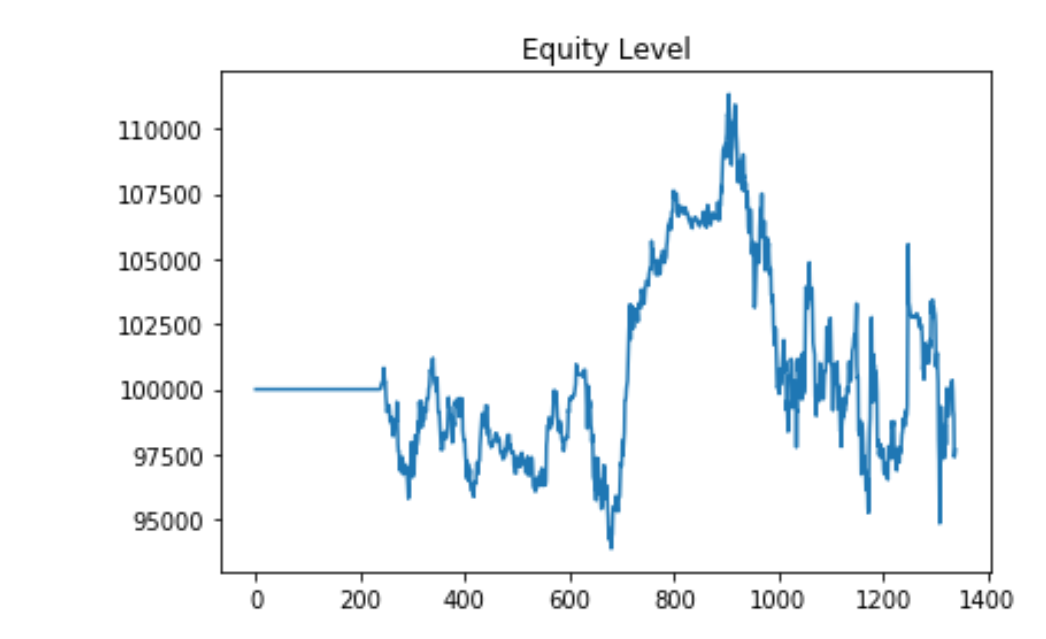
How backtestor work is explained extensively in its documentation file. In simple terms, it calls the price and signal information from database, iterate through days to calculate position based on the weighting method specified, run through exposure check to ensure we are bounded by our criteria and have sufficient margin, then calculate returns and equity. on daily bases.

Here I will focus on show some case of how to use it.

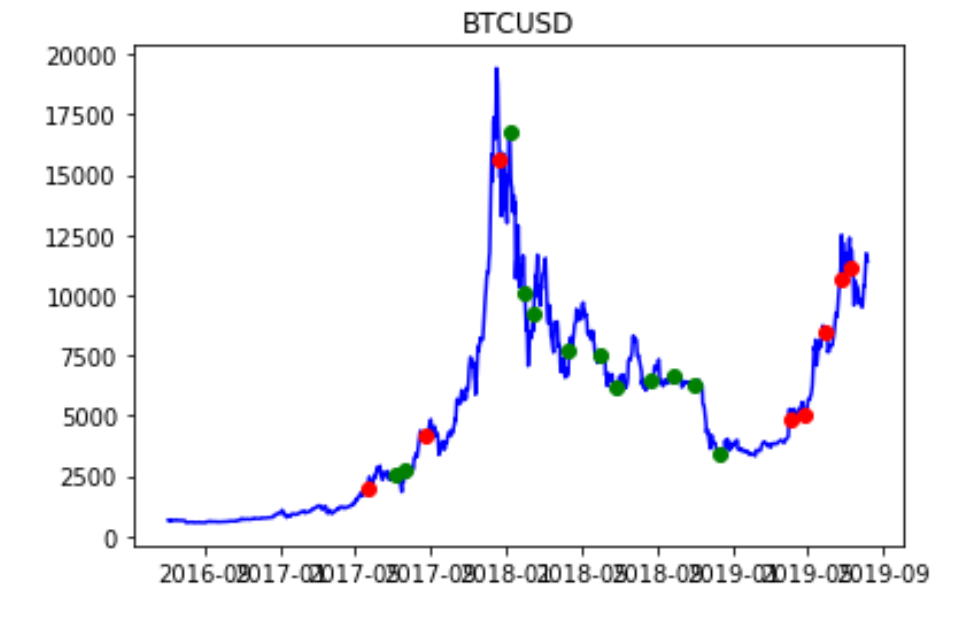
For example we are interested using technical signal ‘MACD’ and ‘Accumulated Distribution’ in combination with machine learning signal ‘LASSO’ to trade on four different assets: [Brent crude oil, Bitcoin, CAD/JPY, Wheat future], and we want to make trading decision on a daily basis throughout the entire history we have. Then what I need to do is specify these information like I did in the Jupyter Notebook.

Like described in the previous sections, our asset pool to select contains 126 different assets at 8 different frequencies and I’ve programmed 10 signals user can try out.

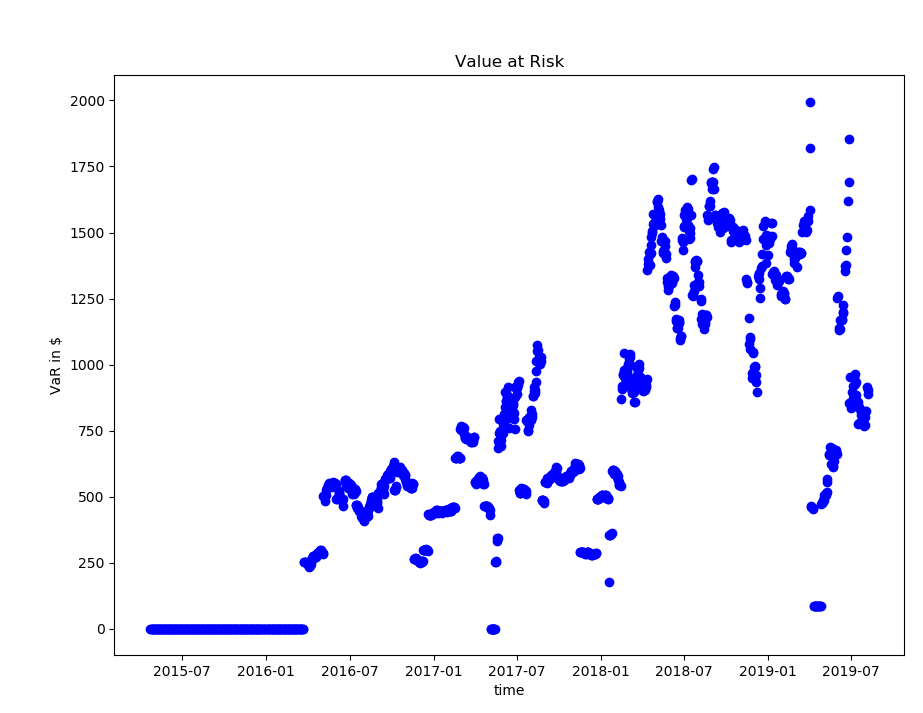
Let’s see some results:



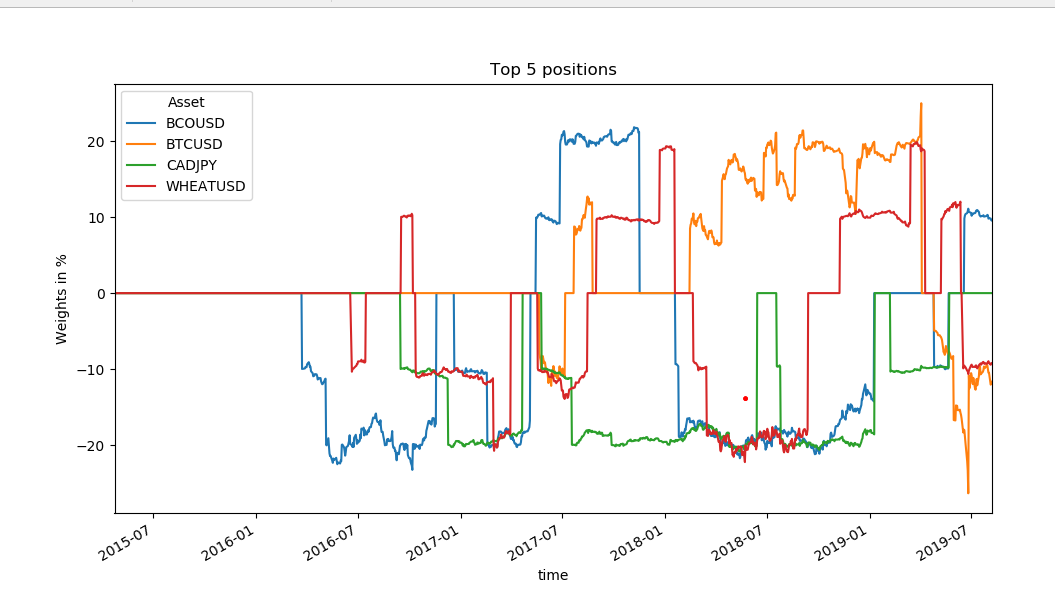
Performance seems to be very volatile, and calling the Plot\_Trades function we see clearly we’re making a lot of time mistakes in the Bitcoin.



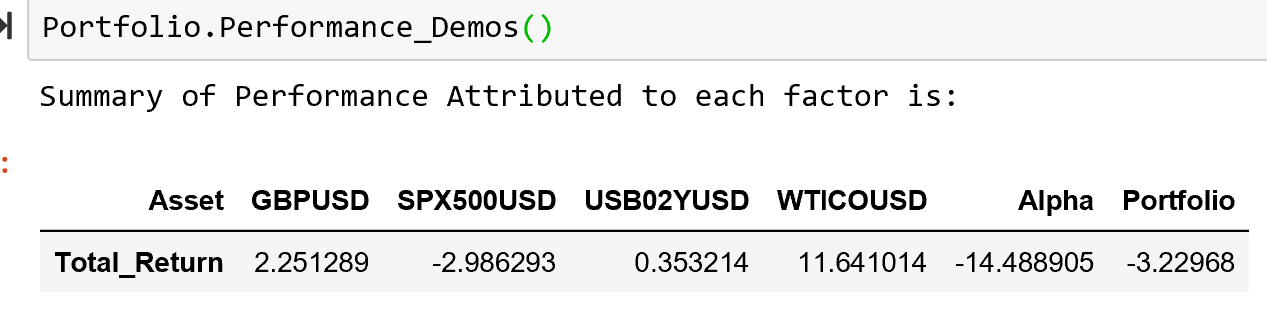
Calling the Risk\_Demos we see many different risk metrics. To pick a few, we see the increased risk clearly reflected from the VaR plot



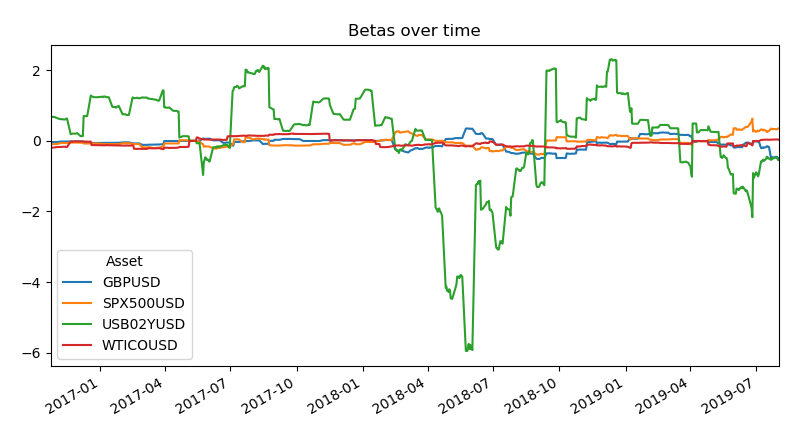
By reading the Top positions chart we see the reason for such increased risk was because we started building positions in the Bitcoin.



Then call for the Performance\_Demos we see many performance metrics. For example, from this table which attributes our return to several common ‘Market Factors’ we can see our return mainly come from our good timing in the Oil sector and loading on this beta when the premium is good. However, we have a horrible stock picking skill. No wonder we were trading Bitcoins.



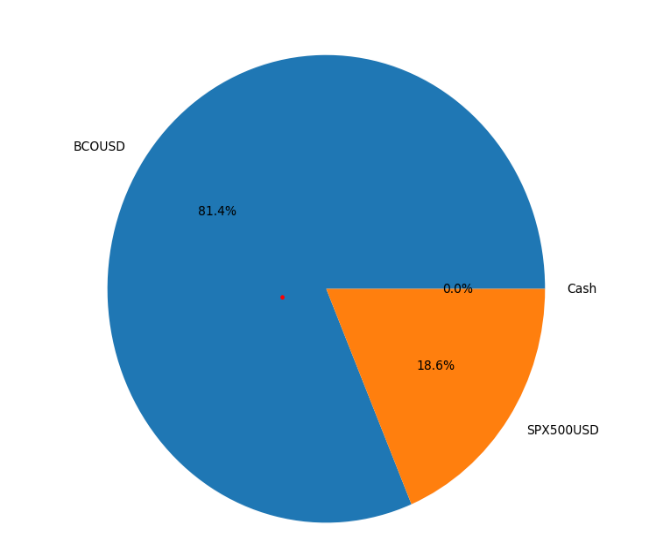
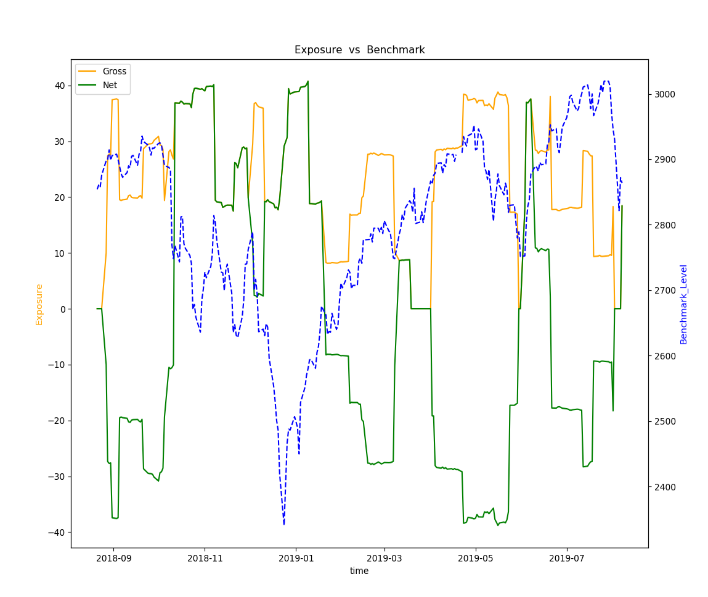
It can also be seen we are often heavily loaded on the bond factor, but it varys a lot over time and it hasn’t contributed to too much return, maybe we should endeavour to improve our market timing skill in the bond factor.



Then after understanding the strategy strength and weakness the user can go back to edit asset pool, edit trading frequency, signal to use and risk threshold in order to eventually develop a profitable trading strategy.

### Risk and Performance Advisor

We’ve had a glance how these two advisors are connected to the Backtestor. However we often use them on their own too. When we provide no argument at all to the Advisors, by default they will go read our historical trades and conduct analysis on our historical trades.

For example, this is the volatility contribution of each asset in our portfolio on the last day of our trading record. Chart next to it is how our portfolio liquidity evolved over time. It’s just as easy to

use these advisors on hypothetical holdings, I’ve shown in the Jupyter Notebook how to do that.

### Live Trading Assistant.

Again, technical mechanism of Live Trading Assistant is described in its documentation. In brief Live Trading Assistant is in charge of all communication with MT4, responsible for monitoring holding position, subscribing and recording live-streaming prices, execute trades from our strategy etc.

Here I focus on the demonstration of how it functions.

First of all, you need the MT4 fully configured and logged in (see documents on how to configure).

When initiating the LTA object user specifies how frequent they wants all information to be updated, as well as the time to wait before kill any action. Upon the successful connection user should see greeting ‘Good morning sir, PARIS is online’

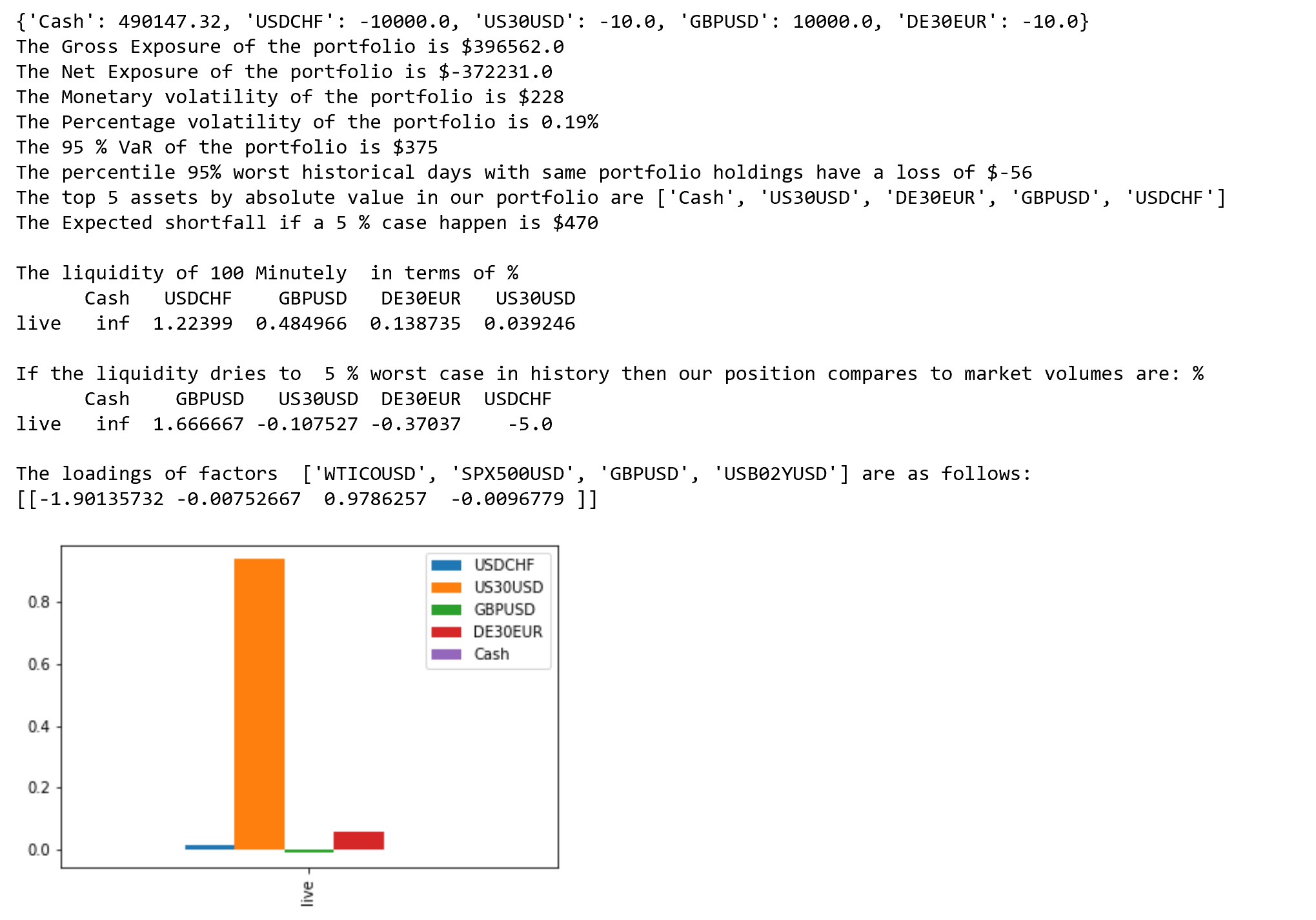
Then User just hit the Run function to have the LTA running. What LTA will return is a real-life risk and performance monitoring that looks like this:

Starting from a summary of position holding right now;

Followed by all the risk metrics

Followed by a plot of current holdings’ contribution to volatility. If all contributions are positive it will be plotted as pie charts, but often assets are negative correlated adding position of some asset will actually decrease portfolio volatility. In these case a bar plot will be shown (pie chart doesn’t plot negative values).

Then followed by a summary of current portfolio’s exposure to different risk factors we concern.



What’s executed in the background but didn’t return anything are two processes:

1. Execution of the trading strategy we picked. Currently for demonstration it’s running a random trading strategy that: - close any trades that’s in the money for over $10 and; -open a random new position with lot 0.1
2. Record the live data stream every second into the price buffer then add the buffer to database when it fills up.

All above process including monitor, trade, record are every 5 seconds in the demonstration case.

# **Area of Improve**

PARIS is designed and programmed with a high focus on scalability and flexibility. I believe the first big improvement that can be done is to extend the APIs of the live trading assistant to make it compatible with more platforms. Platforms like Interactive Broker has a far bigger tradable pool than MT4, extending LTA to allow for multiple platforms can make PARIS more applicable to more funds.

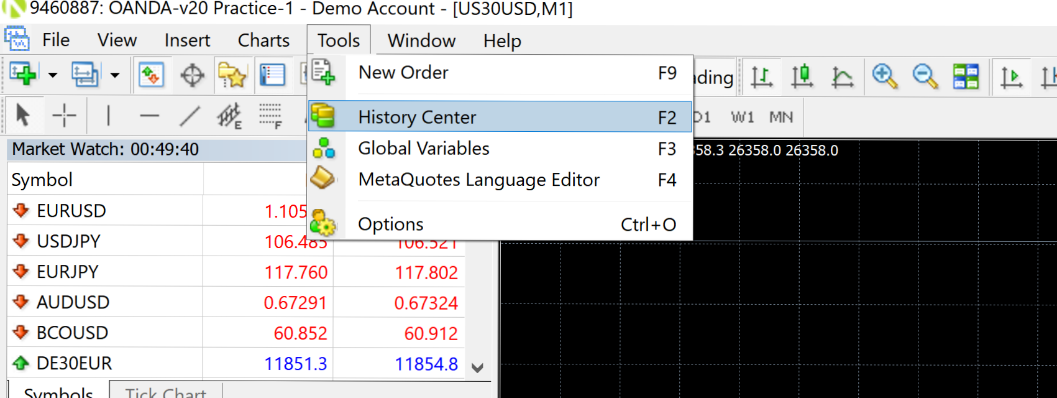
Secondly I think the module dependencies can be reduced by creating an input wrapper class that turns different types of inputs into a standard form that all modules can understand. Currently this is kind of ‘hard-coded’ which makes some constructors lengthy.

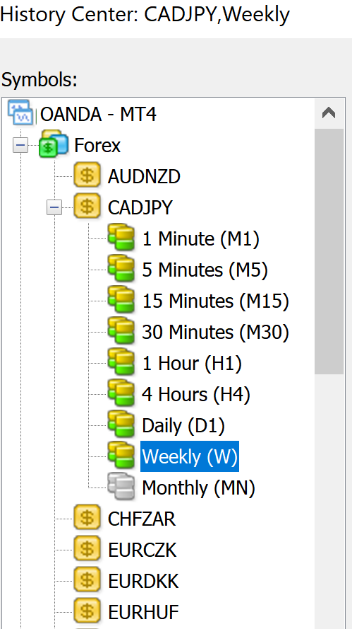
Third I believe the efficiency of storing data in the database can be improved by adopting more advanced indexing methods. That way the size of Database can be reduced further.

# **Appendix**

* 1. **How to build the database:**

I assume you have read the Software\_Configuration docx file already, if not please follow through these steps first to ensure you have a functional MT4 software and account.

Step1 : Open the Trading account and navigate to the historical center shown.

Double click to ensure every price data you want to download shows yellow and green color, this means they are downloaded. Then wait for about 1 hour and navigate to your local meta trader data folder: It’s address should look somewhat like follows:

C:\Users\\*Your\_name\AppData\Roaming\MetaQuotes\Terminal\

\*Your\_Account\history

Step 2: When you see these hst files are downloaded, open the Collect Data jupyter notebook I provided in the folder. It uses another function in the DWX\_ZMQ library to turn hst files into pandas dataframe.

Step 3: Download a sqlite gui, link here: <https://sqlitebrowser.org/>

Step 4: Using the DB\_Operator class created in the Database\_Interactions.py file, we can then easily create a database from the database in step2. If you have any difficulty in achieving this, please read the documentation I provided in the DB\_Operator\_Documentation.docx file

* 1. **DB\_Operator documentation:**

***“Database\_Interaction.py*” – Documentation**

# Description:

This file contains a DB\_Operator class which, as name suggests, take care of communication between a SQL database and python programs. Purpose of having this operator is to free users from having to program two different languages simultaneously to focus on python end, also standardised functions reduce the risk of contaminating the database.

# Dependencies:

* pandas
* sqlite3

# Class Function Descriptions:

|  |  |  |  |
| --- | --- | --- | --- |
| **Function Name** | **Inputs** | **Output** | **Description** |
| Constructor | Database Name (str) | None | DB\_Operator is constructed by connecting to an existing Database in the current working directory |
| create\_new\_table | table\_name (str), columns (list), var\_type (str) | None | Creates a new table in the connected database. Table will use the table name provided. Created table will have corresponding columns as provided in the list. Content of table will be of type provided, default value for var\_type is ‘text’ |
| DF\_to\_new\_table | Dataframe (pd.df), tablename (str) | None | This function either creates a new table with the tablename provided and contents from the DataFrame provided, or it appends the dataframe contents to the bottom of an existing table is the table already exists. |
| read\_entire\_table | tablename (str) | pd.df | This function reads an entire table named by tablename, and returns all contents of the table in the form of a pandas dataframe |
| Select\_rows | tablename (str),  find (list),  by (str) | pd.df | This function will go to the table named by ‘*tablename’*, look for the column as index by ‘*by’,* and track down the rows where the values matches the elements in ‘*find’*. ‘*by’* has a default value of ‘Asset’ because it is most commonly used as unique IDs (ticker names). |
| disconnect | None | None | Disconnect with the Database. |

* 1. **Generate\_Signal**

***“Signal\_Pool.py*” – Documentation**

# Description:

This file contains a Generate\_Signal class that performs the function of taking in price and Factor information to generate Technical trading signal as well as machine learning based trading signal.

# Dependencies:

* pandas
* numpy
* copy
* sklearn

# Class Function Descriptions:

|  |  |  |  |
| --- | --- | --- | --- |
| Function Name | Inputs | Output | Description |
| Constructor | Price\_Data (pd.df),  Factor\_Pool (pd.df) | None | Signal Generator is constructed by simply recording Price and Factor info and create an empty output signal pool |
| MACD | Short (int),  Long (int),  Signal\_period (int) | None | Standard MACD signal generating process. I won’t described in much details here other than just saying when MACD is positive and cross signal line downwards it’s a selling signal and vice versa. Then this signal is recorded in Signal Pool |
| Bollinger\_Band | MA\_period (int),  Band\_Width (int) | None | Standard Bollinger Band signal generating process. Short when crossing the upper bound downward and vice versa.  Same as MACD signal is added to the pool |
| RSI | MA\_period (int),  Low (int),  High (int) | None | Standard RSI generating process, short when RSI > High and buy when RSI < Low, default low and high are 30 and 70.  Signal added to the pool |
| EMA | Fast (int),  Slow (int) | None | Long when fast EMA cross Slow EMA upward and short when it crosses downward.  Signal added to the pool |
| Accum\_Dist | Period (int),  Threshold (double) | None | Calculate the accumulated distribution indicator. Then turn it into trading signal by creating a Benchmark as rolling window standard deviation of AD, then we long if the daily change of AD is greater than threshold \* Benchmark, and short if AD is smaller than (-1)\*threshold\*Benchmark |
| Machine\_Learning\_  Models | Buffer (pd.df),  Refit (int),  Use\_length (int),  Look\_Forward (int),  Feature\_period (int) | None | This function trains several machine learning models to predict the forward looking return (of Look\_Forward days). The models are by default trained on historical price and volume information of the all assets lagged (Buffer). Features are also enriched by aggregating these historical returns/ volumes by (Feature\_period). Feature enrichment is done by calling the \_Get\_ML\_Features function. However, these factors can be overwritten by the factors provided in class construction. All models are refitted every (Refit) days using (Use\_length) number of observations every refit.  Currently I included 5 Machine learning models including: LASSO, Elastic Net, Neural Network, Support Vector Machine and Random Forest |
| Generate\_All\_Signals | Buffer (pd.df)  Refit\_freq (int) | pd.df | This is a wrapper function of all above signals and it returns the signal pool that contains all 10 above stated signals. Both arguments are passed along to generate the ML signals |
| \_Get\_ML\_Features | Period (int)  Buffer (pd.df)  Feature\_time (int) | Pd.df | This is an internal function that takes care of turning factors/historical prices/volumes into the features used in the machine learning signal generation. |

* 1. **Risk\_Advisor Documentation**

***“Risk\_Analytic\_App.py*” – Documentation**

# Description:

This file contains a Risk\_Advisor class that provide risk metrics calculation. The Risk Advisor is designed to be able to work on both a single time point portfolio holding and a dynamic portfolio that evolves over time. This design ensures Risk Advisor smoothly connects to other parts in PARIS system including Back\_Testor, Live\_Trading\_Assistant and actual historical trades database.

# Dependencies:

* scipy
* numpy
* pandas
* matplotlib
* itertools
* DB\_Operator

# Class Function Descriptions:

|  |  |  |  |
| --- | --- | --- | --- |
| **Function Name** | **Inputs** | **Output** | **Description** |
| Constructor | Portfolio  (pd.df/dict),  \_Price (pd.df),  \_Volume(pd.df),  Graphic (bool),  Benchmark (list),  Factors (list),  Holding\_info (dict),  Test\_day (str),  frequency (str),  Volume\_Multiplier (dict) | None | Risk\_Advisor can be constructed in different ways by providing different arguments:   * When no argument is provided at all, it reads historical trades and price/volume info from databases and conduct historical analysis * When a single day’s holding\_info is provided, user can either specify date of holding and that day’s information will be read from database; or user can choose to input price/volume to override historical data. This design is to supports hypothetical position in live trading assistant. * When a time-series portfolio holding information is provided it will be constructed to reflect how these risk metrics evolved over time. i.e when the Risk Advisor is constructed in Back\_Testor   Graphic argument decides whether to show a graphic output or statistics in a dataframe.  Volume Multiplier is used to turn lots into units of underlying assets as streaming hear-backs from MT4 (maybe other brokers too) are in lots instead of units. |
| gross\_exp | None | Print (str)  or  line plot (plt)  or  DataFrame  (pd.df) | On single time point (i.e. some past day or real time advising): this function prints a sentence describing gross exposure.  On time series advising: this function plots how gross exposure evolved over time (gross exposure as percentage). |
| net\_exp | None | Print (str)  or  line plot (plt)  or  DataFrame  (pd.df) | On single time point (i.e. some past day or real time advising): this function prints a sentence describing net exposure.  On time series holding: this function plots how net exposure evolved over time (net exposure as percentage). |
| corr\_table | None | pd.df | This function calculates the correlation between assets invested in the portfolio and returns a dataframe of all the correlation information |
| corr\_history | window (int) | line plot (plt)  or  DataFrame (pd.df) | This function calculates the correlation between all pairs of investments held and plotted them alongside the Benchmark return, to show how they have evolved under different market conditions. |
| port\_vol\_monetary | None | Print (str)  or  line plot (plt)  or  DataFrame  (pd.df) | This function calculates the monetary volatility of the portfolio holding based on holding value and historical variance covariance matrix.  Then it returns a sentence describing this information if assessing single time point, or show how monetary vol evolved over time if assessing a time series portfolio. |
| port\_vol\_percentage | None | Print (str)  or  line plot (plt)  or  DataFrame  (pd.df) | This function calculates the percent volatility of the portfolio holding based on holding weights and historical variance covariance matrix.  Then it returns a sentence describing this information if assessing single time point or show how percentage vol evolved over time if assessing a time series portfolio. |
| Calculate\_VaR | Percentile (int) | Print (str)  or  line plot (plt)  or  DataFrame  (pd.df) | This function calculates value at risk of the portfolio at the percentile specified. Percentile has a default value of 95. Assuming return follows historical covariance matrix and normal distribution.  Then it returns a sentence describing this information if assessing single time point or show how VaR evolved over time if assessing a time series portfolio. |
| Empirical\_VaR | Percentile (int) | Print (str) | This function calculates value at risk of the portfolio using actual historical performance at the percentile. i.e. it checks the 95% actual worst day if we had held this portfolio over time. Percentile has a default value of 95.  This function only works when we deal with a single day’s holding. |
| Get\_Top\_Positions | Number (int) | Print (str)  or  line plot (plt)  or  DataFrame  (pd.df) | This function ranks the portfolio holdings’ absolute values and return top N, as specified by Number input, assets with their weights.  Then it returns a sentence describing this information if assessing single time point or line plot to show how top positions evolved over time if assessing a time series portfolio. |
| Expected\_Shortfall | Percentile (int) | Print (str)  or  line plot (plt)  or  DataFrame  (pd.df) | This function calculates Expected Shortfall of the portfolio at the percentile specified. Expected shortfall means the average loss if a tail event happens (exceeding the percentile specified). Percentile has a default value of 95. Assuming return follows historical covariance matrix and normal distribution.  Then it returns a sentence describing this information if assessing single time point or show how Expected Shortfall evolved over time if assessing a time series portfolio. |
| Liquidity | days (int) | Print (str)  or  line plot (plt)  or  DataFrame  (pd.df) | This function looks at the past n (specified by days input) bars trading volume and calculate our portfolio holding as a percentage of per bar volume.  Then it returns a sentence describing this information if assessing single time point or show how liquidity evolved over time if assessing a time series portfolio. |
| Liquidity\_Dry\_Case | Percentage (int) | Print (str) | This function calculates liquidity not using past n day average volume but rather using the x% (specified as Percentage input) worst case scenario in the history.  This function only works when we deal with a single day’s holding. |
| Risk\_Contribution | Latest (bool) | bar plot (plt)  or  pie plot (plt)  or  DataFrame  (pd.df) | This function takes a portfolio and calculates the percentage contribution of each its constituent assets towards the portfolio’s volatility.  On graphic mode, when deal with a single time point, if all the contributions are positive then it will show a pie plot of the %contribution. It’s because I find pie plots most intuitive, but if there are negative contributors to volatility (benefit from holding them) then this function will return a bar plot.  When the Latest argument is set to true, Risk Contribution will work on the streaming portfolio holding information instead of looking for the holding of any historical days  On non-graphic mode (i.e. time series portfolio hard to show), this function returns dataframe. |
| All\_Live\_Printers | None | None | This is a wrapper function that just calls above listed functions. It’s only purpose is for the Live\_Trading\_Assistant’s code to be simplified without need to call 10+ functions |

* 1. **Performance Advisor Documentation**

***“Performance\_Analytics\_App.py*” – Documentation**

Description:

This file contains a Performance Advisor class which performs performance appraisals to inform users of how good their market time and stock picking skills are. Letting users know whether their returns come from a wide range of factor loadings or skill value added (alphas). Similar to the Risk Advisor, Performance Advisor is designed to have the capability of running on 1. Historical trades, 2. Hypothetical positions. 3. Streaming live portfolio, and it is connected to other parts in PARIS system including Back\_Testor, Live\_Trading\_Assistant and actual historical trades database.

Dependencies:

* Risk\_Advisor
* DB\_Operator
* All other dependencies Risk\_Advisor has

Class Function Descriptions:

|  |  |  |  |
| --- | --- | --- | --- |
| **Function Name** | **Inputs** | **Output** | **Description** |
| Constructor | Graphic (bool),  All other input options available for Risk Advisor constructor | None | Performance Advisor is an inherited class from Risk Advisor and it inherits the constructor function too. Performance Advisor can also be constructed in same three ways as described in previous documentations. It has its own Graphic choice to alter its output types. |
| Basic\_chart | None | Line plot (plt) | This function plots the portfolio equity and benchmark on a line chart. |
| Exposure\_vs\_Benchmark | style (str) | styled plot  (plt) | This function plots gross and net exposure over time, alongside benchmark. Style of this plot can be changed, its default value is line plot.  Reason for change of style is because portfolio may not have holding all the time then line plot will be messier.  This plot indicates market timing. |
| Fully\_Invest\_Performance | Check (str),  Show\_current (bool) | Line plot (plt) | This function calculates the portfolio value assuming we are always investing 100% capital and only adjust asset allocation through time.  This plot indicates stock selecting. |
| \_Calculate\_Factor\_  Loadings | Use\_bars (int)  Rolling (int) | None | This function calculates factor loading of our portfolio with respect to the factors named in Advisor construction.  It uses a rolling window regression approach with window size= use\_bar and rolling frequency= rolling  Then it stores the calculated factor loadings (betas) internal the Advisor class. This is an internal function to assist other functions in advisor app. |
| Return\_Attribution | None | Line plot (plt)  And  Dataframe (pd.df) | This function attributes historical returns to different factors loadings and calculate the alphas generated by investors. It returns a dataframe aggregating this information over time and plots how these returns evolved over time. |
| Show\_Loading | None | Line plot (plt)  Or  Dataframe | This function either returns factor loadings in a dataframe or plots it in a line plot. |

* 1. **Back\_Testor Documentation**

***“Back\_Testor.py*” – Documentation**

# Description:

This file contains a Backtestor class which serves the purpose of backtesting trading strategies based on the signals generated from the Generate\_Signal class. Backtestor is connected to Risk Advisor and Performance Advisor to assess risk and performance metrics of the strategy. Backtestor runs on data stored in the database.

# Dependencies:

* Pandas
* numpy
* matplotlib
* copy
* Risk\_Advisor
* Performance\_Advisor
* DB\_Operator
* Any further dependencies of Risk and Performance advisors or DB\_Operator

# Class Function Descriptions:

|  |  |  |  |
| --- | --- | --- | --- |
| Function Name | Inputs | Output | Description |
| Constructor | Asset (list),  Frequency (str),  Signal\_use (list/str),  start (str),  end (str),  Starting\_Equity | None | User can choose which asset to trade on, what period to trade on, and the frequency of trading.  Upon its construction, the Backtestor will go read the recorded price/ volume/ signal information from the historical database. Then it sets out several adjustable parameters including primary\_signal, weighting method, Risk cap etc. All of which can be modified in following functions. |
| Set\_Signal\_Rule | Primary (str),  Agree (int) | None | This function sets the rule of combining signal. It allows to specify which signal is primary and how many non-primary signal needs to agree before we take a trade.  Primary signal has same importance as non-primary signals combined. i.e. if Primary signals says to long we’d go long unless non-primary signals all agree to short. |
| Set\_Weight\_Method | Method (str) | None | This function sets rules about how to calculate values to trade when we made a trade decision. |
| Set\_Margin\_Rate | Margin (dict) | None | This function informs Back\_Testor the margin rate of all tradables through a dictionary. This is used later on to calculate the margin required to ensure we don’t run out of free cash. |
| Set\_Risk\_Cap | Value\_Cap (double),  Indiv\_Value\_Cap(double),  Volatility\_Cap (double),  Indiv\_Vol\_Cap (double) | None | This function allows users to set the value cap of investing in individual asset level and portfolio level. It also allows set the volatility cap on individual level and portfolio level |
| \_Signal\_Combine | None | None | This function combines signals based on the rule specified by user through above function. By default there’s no primary signal and no agree required. |
| \_Calculate\_Weight | Trade\_Info (pd.df),  Time (str),  Count (str),  Original\_Holding | None | This is an internal function that calculates today’s trading amount based on last day’s position holding and today’s new signal.  For example if we have long position and have a short signal then we close long position. Otherwise we add to the short position using the weight method named above, e.g. ‘Equal\_Weight’ that spends 1/20 of portfolio equity to trade.  After new positions calculated we run the new position through exposure control to ensure it ticks all risk criteria |
| \_Exposure\_Control | Holding (list),  Time (str),  Count (int),  Period (int) | Holding (list) | This is also an internal function that ensures all the risk caps set by user are met.  The calculation is based on last day’s equity and the variance covariance matrix of the past (Period) amount of days.  If any cap is exceeded, this function will downscale today’s trade to ensure it falls under risk radar again. Then it returns the new position to be executed. |
| run | None | None | This is the main user function that iterates through time and calculate every day’s holding, run them through exposure check, calculate returns and move on to the next day.  After portfolio holding is built for the whole period, internal risk advisor and performance advisor are constructed to assess the performance and risk of the named strategy. |
| Risk\_Demos | None | Pd.df  and  Bar plot  and  Line Plot | This is a wrapper function for the user to quickly see all the risk metrics of the tested strategy |
| Performance\_Demos | None | Pd.df  and  Bar plot  and  Line Plot | This is a wrapper function for the user to quickly see all the performance metrics of the tested strategy |

* 1. **Live\_Trading\_App Documentation**

***“Live\_Trading\_App.py*” – Documentation**

# Description:

This file contains a Live\_Trade\_App class that handles communication with the MT4 platform including requesting and receiving current position holdings, subscribe and collect live market data, trade executions. Live Trading App connects with Risk Advisor and Performance Advisor to give a live-updating demonstration of the portfolio risk metrics and return attributions. It also connects with the database to add latest price data into the database.

# Dependencies:

* pandas
* matplotlib
* threading
* time
* IPython
* Random
* DWX\_ZeroMQ\_Connector
* Risk\_Advisor
* Performance\_Advisor
* DB\_Operator
* Any further dependencies of Risk and Performance advisors or DB\_Operator

# Class Function Descriptions:

|  |  |  |  |
| --- | --- | --- | --- |
| Function Name | Inputs | Output | Description |
| Constructor | Pool\_Limit (int),  Update\_frequency(int),  \_Time\_Out (int),  \_Refresh (double),  \_TimeZone (int) | None | Upon constructing the live\_traing\_app class I set several parameters that will be applied through different functions.  (Pool Limit) means the size limit of live data pool, before it is recorded and cleared.  (Update\_Frequency) is the frequency of requesting portfolio holding and recalculating all risk/performance metrics.  When requests are sent these requests are refreshed per (\_Refresh) frequency if we didn’t hear back. Then they would be timed out at (\_Time\_Out) limit to prevent infinite waiting. Price information heard back will be adjusted to our local (\_TimeZone) before they are used and recorded to the historical database |
| \_Current\_Holding | Every (int) | None | This function takes care of requesting portfolio holding per (every) seconds, and record these holding, as well as request a lots/units translator and request portfolio equity level. These information are recorded and used by Risk/Performance Advisor.  This is an internal function. |
| \_Risk\_Performance\_Monitor | Every (int) | Print (str)  And  DataFrame (pd.df)  And  Pie Chart (plt)  Or  Bar Chart (plt) | This function calls the Risk and Performance Advisors to be constructed and the metrics to be shown. Details of what these metrics are is documented in other documentations.  All live demonstrations are then updated per (Every) seconds. |
| \_Baby\_Strategy | Every (int) | None | This Function is a toy trading strategy that’s there to trade randomly. I wrote it to show all other parts of Live\_Trading\_App works smoothly. |
| Trade\_Execute | Action (str),  Symbol (str),  \_type (int),  Lots (double, optional),  Price (double, optional),  Strategy\_Number (int, optional),  Tkt (int, optional),  SL(double, optional),  TP(double, optional) | None | This function executes trades of the given (Action : OPEN/CLOSE) on the given (Symbol), at given (\_type: long = 0/short = 1), with any other optional specifications. (lots, take profit, stop loss, limit price etc.)  Then this function feeds back whether the execution has been successful. |
| \_Price\_Buffering | Symbol (list),  Stop (bool) | None | This is an internal function that send request to either start live price subscription or terminate subscription (Stop), of Assets we’re interested in (Symbol). |
| \_Snapshots | symbol (list),  every (int) | None | This function takes a snapshot of streaming price and volume per (every) seconds, add it to the price buffer. When buffer reach the (Pool\_Limit) size it is added to the database. |
| Print\_Price\_Buffer | None | Pd.df | This function returns the price buffer |
| \_Check\_Connection\_status | None | None | This is an internal function that’s called when the Live\_Trading\_App was constructed to ensure all three sockets are successfully connected to the MT4 trading platform |
| Run | None | None | This is the main user function. This function spins off four threads to serve four purposes: 1. Constantly updating portfolio holding information. 2. Constantly outputting the updated risk and performance metrics as the holdings change. 3. A trading bot that trades with random strategy. 4. Constantly hearing the live streaming price and add it to the database. |