High Frequency Trading Toolkit (Documentation)

Hugo Dolan

Running Algos

Algorithms: (algo.py, bp.py, vol_algo.py)

Contains the entry point for all trading systems which may be called from the terminal (ctrl-c can be used to halt execution at any point):

```
python algo_name.py
```

API Connection (sources.py)

Connecting to the RIT API is fairly straightforward, you will need the API Key from the client and the endpoint for any future API. You then must replace the default values in the configs/api_config.json file with the correct values for your setup:

```
# api_config.json
{
    "api_access": {
        "RIT": {
            "headers": {
                "X-API-KEY": “YOUR API KEY"
            },
            "endpoint": “http://localhost:9999/v1"

        }
    }
}
```

Note future functionality for this JSON file may contain algorithm presets / parameters to remove necessity for hardcoding of values. Once this is complete you will need to initialise an API object which is passed in as an initialisation parameter into many other components of HFToolkit.

```
from sources import API
API_CONFIG = './configs/api_config.json'

api = API(API_CONFIG)
```

Fetching Data (securities.py)

The majority of data streaming from Rotman Interactive Trader (RIT) is security specific and thus must be accessed via the Security (Or Options) class. To retrieve data for a specific security simply create an instance of security, by specifying the ticker symbol (eg. BEAR).

```
from security import Security
sec = Security(‘BEAR’, api, is_currency=False)
```

To retrieve information from the server regarding the security sec.poll() must be called to update the security's state. Alternatively the Security class is multithread enabled and simply calling sec.start() is sufficient to start the security polling in the background, outside the main execution thread, which is often desirable for complex trading systems.

```
sec.poll() # Synchronous and must be called as frequently as required (200ms
recommended)
sec.start() # Asynchronous alternative to sec.poll() and called only once.

sec.shutdown() # It is a good idea to call this when your trading algo finishes to
clean up the security thread. This applies to the asynchronous implementation only.
```

There is a wide range of built in functions with the security class designed to save time. These include:

```
from time import time

mid = sec.get_midprice() # Current Mid Price (float)
best_bid, best_ask = sec.get_best_bid_ask() # Current Best Bid / Best Ask (float)

spread = sec.get_bid_ask_spread() # Current Bid Ask Spread (float)
slippage = sec.get_average_slippage() # Average historical slippage (float)

pct_return = sec.get_net_returns(60) # Computes percentage return for last 60 seconds
ohlc_df = sec.get_ohlc_history(freq="100ms") # Computes Open High Low Close data frame
for all time at the specified sample frequency.
```

The list is not limited to what is detailed above, there are also some more specialist / custom metrics such as Volume Probability of Informed Trading (VPIN), Volatility, Order Imbalance, Volume Weighted Average Price (VWAP) which have been implemented, the way of calling these functions is not yet entire consistent but typically they may be accessed through sec.indicators['indicator_name']. It is recommended to take a look in securities.py for specific implementations. A word of warning that accessing of these custom functions is not 100% stable (due to computational delays to obtain representative samples) so it is worth checking if "indicator_name" in self.indicators, before using.

If you wish to add additional functionality or state to the security class you should do so by either:

a) Creating a subclass (the Options subclass in security.py is a good example of this)
b) Adding additional functions to security (Avoid this if at all possible)

Trading Time Synchronisation

Often in any trading system we wish to know what tick we are on in RIT case time. Typically we want to loop through our main trading logic whilst the case is within some specified time range (5 seconds after case start until 5 seconds before the end). This can be achieved easily using the TradingTick generator.

```
from execution import TradingTick
CASE_END = 295 # Tick at which case finishes

for t in TradingTick(CASE_END, self.api):
  if t % 5 == 0:
    print("The current case time is: %s" % t)

  # … Run Main Trading Logic
```

Execution Managers

So far we've covered how to retrieve data from the server and establish a main logic loop, but to build any trading system we need to be able to execute orders and perform any necessary position management / hedging etc.

We introduce the execution manager which can be instantiated as follows (we include an example compiling everything we have covered so far for completeness).

```
from execution import ExecutionManager
from sources import API

# Constants
API_CONFIG = './configs/api_config.json'
api = API(API_CONFIG)

# Securities
tickers = [“BEAR”,”BULL”,”RITC”]
securities = {}

for ticker in tickers:
  securities[ticker] = Security(ticker, api, is_currency=False)

# Execution Manager
em = ExecutionManager(api, tickers, securities)
em.start() # Must be called to ensure orders are monitored for execution

# Trading Logic
for t in TradingTick(CASE_END, self.api):
  if t % 5 == 0:
    print(“The current case time is: %s“ % t)
  # … Run Main Trading Logic
```

There is a wide variety of helpful functions available within the execution manager. The general flow typically involves creating orders -> executing orders -> tracking -> hedging.

```
ticker = “BEAR”
midprice = securities[ticker].get_midprice()
target_price = midprice + 2

market_order = em.create_order(ticker, “MARKET”, “BUY”, 1000) # Simple Order Creation
limit_order = em.create_order(ticker, “LIMIT”, “SELL”, 1000, price=target_price)

# Actually Execute the orders
orders = [market_order, limit_order] # Note orders is always a list of orders
order_ids = em.execute_orders(orders, “ARBITRAGE”)

# The second argument is a ‘source’ which is essentially a tag used by execute orders
to monitor specific positions risk. The current implementation of this is quite bespoke
so if you want to add additional sources its best to subclasss ExecutionManager and
override the method.

# Order_ids is a list which can be used to track orders if necessary for example:
sleep(5)
mo_transacted = em.is_order_transacted(order_ids[0])
lo_transacted = em.is_order_transacted(order_ids[1])
arb_complete = mo_transacted and lo_transacted

if lo_transacted == False:
  fill_qty = em.get_order_filled_qty(order_ids[1])
  print(“Limit Order Partially Filled: %s / 1000“ % fill_qty)
```

A word of warning, the list of orders returned is subject to:

a) Size of orders submitted
b) The risk rules of the securities being traded

As large orders which exceed the maximum order size will automatically be split into smaller orders which will be queued up and fed to the RIT client at the maximum rate permitted by the API. Similarly if an order exceeds the risk rules it will be rejected. This current implementation is clearly not ideal and thus a future iteration of HFToolkit would see a redesign of this interface. As such it is best to view what is returned by execute_orders as a list of successfully transacted order ids. Thus in practice it is recommended that you design your trading system to ensure you will not make trades which could breach gross / net limits.

Some other useful functionality built in to the execution manager includes:

```
# Retrieves / Cancels any specified open orders
em.pull_orders(order_ids)

# Gets the current position sizes net / gross
net, gross = em.get_net_gross_positions()

# Checks if orders within risk (Executed by default in execute_orders)
can_trade = em.can_execute_orders(orders)

# Contains custom logic for hedging positions such as currency risk for variety of
sources (see code for full details)
em.hedge_position("MARKET_MAKER")

# Tender
em.accept_tender(tender) # where tender is the JSON object received from RIT API
em.decline_tender(tender)

# Order splitting (for orders exceeding limits, executed by default in execute_orders)
em.split_order(order)
```

Note that by default the Execution Manager watches the status of all your orders in the background (on a separate thread) so that any partially filled or transacted orders will update your current net / gross position.

Bespoke Executions Managers

In practice most trading systems may have more complex needs than what was detailed above, such as:

- Simultaneous execution of several positions to ensure correct hedging
- Entering / Exiting large positions with minimal market impact and slippage
- Forecasting / Pricing of derivatives

These needs can be handled by a second layer of dedicated execution managers. Some existing implementations include:

OptimalTenderExecutor

```
opt_ex = OptimalTenderExecutor(em, 'RITC', num_large_orders = 3,
num_proceeding_small_orders = 10, large_to_small_order_size_ratio = 5,
vpin_threshold=0.6, risk_aversion=0.005)

opt_ex.add_tender_order(order)
```

The idea of the optimal tender executor is to neutralise a large position resulting from accepting a tender in an efficient manner. We initialise opt_ex only once and then every time we receive a tender on ticker 'RITC' we essentially layer on another tender. The opt_ex then handles the neutralisation of any position, executing a series of larger orders interleaved by smaller orders. The orders are executed at regular intervals determined by the Optimal Execution Horizon (Easley, O'Hara, 2016), which are optimised to reduce information leakage to the market. We define risk aversion and Volume Probability of Informed Trade thresholds which are hyper-parameters of the model which determine the volume in which an order should be hidden and whether a market or limit order should be placed as a function of market order flow toxicity. One of the great advantages of using opt_ex is that should a new tender arrive before the old tender can be fully offloaded, the model automatically accounts for this and offloads the net position after the new tender has been accepted.

OptimalArbitrageExecutor

```
trading_size = 10000

opt_arb_ex = OptimalArbitrageExecutor(em, ['BEAR','BULL','RITC'], trading_size,
num_large_orders = 3, num_proceeding_small_orders = 10, large_to_small_order_size_ratio
= 5, vpin_threshold=0.6, risk_aversion=0.005)

opt_arb_ex.open_arbitrage_position(cointegration_coeff, leg_1_dir, leg_2_dir)

opt_arb_ex.close_arbitrage_position()
```

The Optimal Arbitrage Executor handles the opening and closing of statistical arbitrage positions. It requires the specification of a coefficient of co-integration for the second leg of the spread and then the direction in which to buy or sell on each leg. (Clearly the direction (BUY / SELL) of the second leg must be opposite to that of the first, but this has been left explicit).

The internals of this work reasonably similar to that of the optimal tender executor, relying on entry to positions using limit orders (falling back on a market order after a respecified duration) which are again split intelligently and optimally for very large trading_sizes. When close position is called this gets out of the position quickly using market orders. One of the great advantages of using this class is that hedging of positions across multiple securities is all handled simultaneously and internally.

MIT License