A Trade Order Executor for Reducing Transaction Costs

HangukQuant[1, 2*]

October 26, 2022

[1] [2]

Abstract

We build an improved order executor using the MetaAPI SDK developed in our earlier work [1]. The order executor takes into consideration execution (market spread) costs, slippage, and transaction commissions in relation to a predetermined positional inertia to attempt execution at the cheapest cost at limit/market. We also perform logging, which will allow us to analyze the trade logs in future work to see if our execution strategies are indeed cost-saving.

[1]***1**: hangukquant@gmail.com, hangukquant.substack.com

[2]***2**: DISCLAIMER: the contents of this work are not intended as investment, legal, tax or any other advice, and is for informational purposes only. It is illegal to make unauthorized copies, forward to an unauthorized user or to post this article electronically without express written consent by HangukQuant.

1 Introduction

As before, the same caveat applies - this is a result of weekly musings - there are bound to be mistakes. Please email all errors/issues to *hangukquant@gmail.com* or tweet me at *@HangukQuant*.

In the previous version of the order executor, we built a barebone executor. However, this was quite lacking and was just a prototype - in particular the executor was not robust to network and server issues and fairly simplistic in its logic, only taking into account one or two variables. We want to evolve that further - leading us to build a better, more advanced MetaAPI brokerage service SDK to make communication with our APIs more error tolerant and efficient. Primarily, we did this with better error handling, doing rate limiting with a custom asynchronous credit semaphore to throttle our APIs. We also tried to be more efficient with our data requests, such as using streaming socket connections to synchronize our local copy of the position and order state with the terminal state. Using these techniques, we managed to reduce our overall credit cost for the API services we use (MetaAPI), allowing us to handle thousands of trades simultaneously. We refer the readers on implementation and discussion to the previous paper. [1, 4]. The free version of the paper will contain some implementation details and code. The full version of the paper for paid readers will contain full implementation details and code files.

For readers who have not yet grasped the importance of trade cost reduction can refer to our work on Costful Trading here. [2] To summarize, we saw that capturing the spread is part of an essential execution strategy and can help to improve post-cost returns and reduce the penalty cost attribution. This can be performed with passive order execution with orders that sit in the order book and provide liquidity as opposed to taking liquidity. However, the deeper our price sits in the order book, the lower the probability of our order getting processed. Hence, there is a tradeoff in the risk of us not fulfilling our target portfolios against saving transaction costs. This risk-saving tradeoff can be visualized as a efficient trading curve, much like the formulations of Markowitz. [6, 7].

If our target portfolio is not as latency sensitive, and where positive expectancy of our returns are immune to noise of the alpha signal to a certain degree - a simple approach of sending limit orders and substituting them for a market order beyond some pre-determined execution span might suffice. Additionally, where our strategy is more cost sensitive than tracking sensitive - we can opt out the market order and instead incur the tracking error. An additional note is that there is also the concept of slippage. There are 3 different prices associated with execution, first is the price p_1 obtained when computing trade signals, second is p_2 we observe when we peek at the order book for quotes, and p_3 the filled price of our executor. How should we then execute our strategies, taking into account short term market variance (noise), signal variance (another form of noise), slippage, execution costs, spread and inertia?

1.1 A Note of Precaution

There is absolutely no warranty or guarantee implied with this product. Use at your own risk. I provide no guarantee that it will be functional, destructive or constructive in any sense of the word. Use at your own risk. Trading is a risky operation.

1.2 Desired Behavior

Let's consider first what it is that we want to build, and then go ahead and build it. The desired behavior is that our order executor takes in a bunch of order schema from our trading system that are basically 'instructions' or 'information' about the current state of our portfolio, against the target state of our portfolio, as well as other meta information about the contracts we are trading so that we can perform the right actions for our execution strategy. Such a schema may look something like this:

```
"USDCAD%CAD": {
  "fx_base": "USD",
  "fx_quote": "CAD",
  "fx_margins": "USD",
  "size_contract": 100000,
  "digits": 5,
  "volume_max": 100,
  "volume_min": 0.01,
  "swap_long_pct": -4.9,
  "swap_short_pct": -3.5,
  "trade_mode": "full",
  "asset_class": "fx",
  "backtest_spread": 0,
  "backtest_swap_long": 0,
  "backtest_swap_short": 0,
  "commission": 0,
  "pos_current": 0.0,
  "pos_target": -0.24352299725026555,
  "close_signal": 1.38811
}
```

which can easily be represented as a *dict* object in Python. These dictionaries will essentially be the communication protocol between our trading system (whatever it may be) and what we are discussing today, which is the order executor.

The order executor will then take a bunch of these order instructions, and then a brokerage service class and execution window as a parameter. The brokerage service class should implement the methods that allow us to make the trades, by communicating with a foreign API or equivalent. We want to give each of the orders the specified execution window to attempt to get filled, and at the end of the execution window clean up any unfilled tickets and force fill trades that are sufficiently 'urgent'. This sounds like a fairly reasonable series of events, at least in the high level hierarchy of abstraction. We want to zoom in further into the execution strategy, which are the parts 'attempt to get filled' and 'force filling'.

Let us recall our quote on positional inertia from [2]

''

The first is to recognise that even after the aggregation of signals, there will be significant noise in our signal estimates. Our signals are approximating the true, underlying stochastic alpha distribution. The underlying alpha distribution also moves from point to point, but not in a smooth way. If our signals capture the noise in the stochasticity of the underlying alpha that we are approximating, we would accordingly be eating costs in terms of turnover. We want to only take changes in our *positions if the distance between the probability distributions of our estimates of the underlying alpha is sufficiently large. That is, we want to implement positional inertia to avoid frequent trading and act only if the change in positions is significantly different from the previous estimate.*

''

Without even considering any serious mathematics, let us just consider some simple evolution in portfolio target position.

```
position A ----------------------------------------> position B
```

At any of the '-' positions, the target position is ! = A but not so much so that the positional inertia is overcome. Only when we reach position B is our tracking error (from our theoretical backtest computing the trade signals) sufficiently large that we want to adjust our actualized portfolio holdings. However, in most cases the positions do not go from A to B instantaneously, and follow (albeit rough) path. Note that our tracking error is relative to the backtest, which is our approximation of the theoretical alpha. In practice, we are approximating the backtest, which is an iterated approximation of the approximation itself. If we can realise backtested returns, that would be sufficiently fantastic.

Suppose it takes 5 days or (other units/candles you are trading on) for position A to go to position B, like this

```
A ----- > B
```

and each day we would observe a price. Suppose at position A is 'long 5 AAPL' as we expect a price increase, and at B it is 'long 8 AAPL' as we accumulate into the trade. The positional adjustment is long 3 AAPL in the vanilla imlementation, but the price path could be something like

```
5 65667 > 8
```

in target positions. Although over a longer period of time we might assume there is some edge, in practise the price fluctuations over each day is close to a coin flip. Suppose we did not make any trades in the meantime (since our positional inertia shows no urgency), when we reach position B at 8 contracts our portfolio marks our position as 'urgent' and call for the need to get 3 contracts. Since this is an urgent operation, if it does not get filled, we are likely to need to make a force fill at market, which is costly!

The thing is, if we were able to switch our positions for *free*, we would just do continuous rebalancing and trade frequently. This concept of 'free' trade is relative to the computed signal price, which is the price we used in our backtest. There is actually a fairly high probability that this would be realised in practise, since market prices fluctuate almost randomly in the short term. There is almost a 50% probability that when we look at the quotes it would be more favorable than when we computed the signal.

```
5 656*67 > 8
```

Suppose we are able to make the jump to 6 contracts on the day indicated by *, it would be wise for us to make the jump, since a free rebalancing is almost always a positive. When we reach 8 contracts, we would be 'less urgent' as before, and being less urgent is less costly since we have more time to hit our target position without forcing the trade at market and incurring the spread. The more time we have to realise our target position, the more patient we can be in being a liquidity provider and be compensated for this. We shall see how this concept of 'free rebalancing' comes into play during the discussions on our execution strategy.

2 Implementation

2.1 Order Executor

First, our order executor will take in a list of specifications for our trade orders, as we desmonstrated earlier with the order schema.

```python
import pytz
import json
import asyncio
import pandas as pd
from datetime import datetime
from pprint import pprint

from brokerage_service.darwinex.executor.order import Order

class OrderExecutor:

    def __init__(self, ticker_specs, broker_service, positional_inertia,
db_service=None, execution_window=60 * 15):
        self.ticker_specs = ticker_specs
        self.broker_service = broker_service
        self.execution_window = execution_window
        self.db_service = db_service
        self.orders = [
            Order.build_order(
                broker_service=broker_service,
                inst=k,
                inst_specs=v,
                inertia_level=positional_inertia
            ) for k,v in ticker_specs.items()
        ]
        self.batchids = []
        self.ticker_to_batch = {}
```

Listing 1: order_executor.py

This order executor should work regardless of which brokerage we are using, as long as the schema is consistent. Regardless of whether we are trading futures contracts, CFDs, trading on IBKR or Saxo - the schema contains common data that we require for a trade order. The order executor should not be concerned with facts other than that.

We will separate our hierarchy of execution details and logging into 3 steps. First, there is the execution *window*, which are ALL the trades received by the order executor to handle in that round. Next, there are *batches*, which are groups of *orders* executed together. Let's further elaborate on this.

Suppose we have a thousand trades, and we execute them all together. One of the things that we do in our order execution is to peek at the bid-ask and then bet limit to capture the spread. Now imagine the asyncio event loop, which queues up coroutines to run. If we have thousands of trades in bursts trying to execute at the same time, there we be significant latency between when we getting our quote and getting to submit our order request, increasing the probability that our bid or ask is no longer valid at submission. We want to handle this burst rate, while also maintaining the concurrency for efficiency. Additionally, batching them gives us better safety in terms of trade execution, in that a batch failure does not affect other batches. We are effectively reducing execution dependencies, in case one of our trade execution crashes this should not kill the remaining trades. Last but not least, batching them allows us to better analyze trade logs as we have more granular execution log data. The execution window gives us an overall view of how our execution strategy went, while the order logs give us play-by-play analysis of the execution details that went down in production.

To execute all the orders in our executor, we first batch them and work on them in asynchronous bursts.

```
async def execute_orders(self, batchsize=30):
    self.execution_start = datetime.now(pytz.utc)
    batch_executions = []
    for i in range(0, len(self.orders), batchsize):
        batch = self.orders[i : i + batchsize]
        batch_task = asyncio.create_task(self._execute_batch(batch))
        batch_executions.append(batch_task)
        await asyncio.sleep(30) #dispatch new batch every 30 seconds

    await asyncio.gather(*batch_executions)
    self.execution_end = datetime.now(pytz.utc)
    await self.log_execution_window()

async def _execute_batch(self, batch):
    batch_start = datetime.now(pytz.utc)
    batch_logs = [{"datetime" : batch_start, "message" : f"batch id {batch_start.
   timestamp()} started"}]
    for item in batch:
        self.ticker_to_batch[item.get_ticker()] = batch_start.timestamp()
    try:
        for item in batch:
            print(item)

        ######### << the core of our execution logic, the rest is just logging
        tasks = [asyncio.create_task(order.execute_order()) for order in batch]
        batch_logs.append({"datetime" : datetime.now(pytz.utc), "message" : "batch init exec"})
        done, pending = await asyncio.wait(
            tasks, timeout=self.execution_window
        )
        for pending_task in pending:
            pending_task.cancel()
        batch_logs.append(
            {"datetime" : datetime.now(pytz.utc), "message" : "batch init force"}
        )
        tasks = [asyncio.create_task(order.force_order()) for order in batch]
        await asyncio.gather(*tasks)
        #########

        batch_logs.append({"datetime" : datetime.now(pytz.utc), "message" : "batch done force"})
    except Exception as err:
        batch_logs.append({"datetime" : datetime.now(pytz.utc), "message" : f"batch error {err}"})
    finally:
        batch_end = datetime.now(pytz.utc)
        batch_logs.append({"datetime" : datetime.now(pytz.utc), "message" : "batch finished"})
        await self.log_execution_orders(batchstamp=batch_start.timestamp(), batch=batch)
        await self.log_execution_batch(batchstamp=batch_start.timestamp(), batch=batch, batch_logs=batch_logs)
    return
```

Listing 2: order_executor.py

Other than the execution logic, we will also maintain code to use our database service programmed in previous work [3, 5] to get our log execution data on our Mongo instance. In particular, we will use both time series service to submit execution logs on the window level, while using document service for the batch level and order level. Note that we have also allowed a allow_disjoint=True option with the time series service, which allows our database to remove the safety check for having contiguous time-series data. We do not need the contiguous safety check for the execution logs. The edit to our source code in db_service.py is minimal.

```python
async def log_execution_orders(self, batchstamp, batch):
    assert(self.db_service)
    docdatas = []
    doc_identifiers = []
    for item in batch:
        ticker = item.get_ticker()
        doc_identifier = {"type": "execution_log", "order_stamp": f"{batchstamp}/{ticker}"}
        market_spread_penalty, exec_spread_penalty = await item.get_executor_performance()
        docdata = {
            "ticker": item.get_ticker(),
            "pre": item.get_start_pos(),
            "post": item.get_end_pos(),
            "target": item.get_target_pos(),
            "comm": item.get_exec_comm(),
            "inertia": item.get_inertia(),
            "peek_ba": item.get_peek_ba(),
            "ord_snaps": item.get_order_snapshots(),
            "pos_snaps": item.get_position_snapshots(),
            "received_price": item.get_received_price(),
            "slippage": item.get_slippage_penalty(),
            "market_spread_penalty": market_spread_penalty,
            "execution_spread_penalty": exec_spread_penalty,
            "tickets_and_fills": await item.get_tickets_and_fills(),
            "order_logs": item.dump_logs()
        }
        docdatas.append(docdata)
        doc_identifiers.append(doc_identifier)

    await self.db_service.asyn_batch_insert_docs(
        dtype="log", dformat="docexec", dfreq="o", docdatas=docdatas, doc_identifiers=doc_identifiers, metalogs=[item.get_ticker() for item in batch]
    )
    return
```

```

async def log_execution_batch(self, batchstamp, batch, batch_logs):
    assert(self.db_service)
    doc_identifier = {"type": "execution_log", "batch_stamp": batchstamp}
    batch_tickers = [item.get_ticker() for item in batch]
    docdata = {
        "batch_logs": batch_logs,
        "batch_tickers": batch_tickers
    }

    await self.db_service.asyn_insert_docs(
        dtype="log", dformat="docexec", dfreq="o", docdata=docdata, doc_identifier=doc_identifier
    )
    return

async def log_execution_window(self):
    assert(self.db_service)
    series_metadata = {"exec_level" : "window"}
    series_identifier = {"type": "execution_log", **series_metadata}
    records = []
    for order in self.orders:
        market_spread_penalty, exec_spread_penalty = await order.get_executor_performance()
        records.append(
            {
                "datetime": self.execution_start,
                "duration": (self.execution_end - self.execution_start).total_seconds(),
                "ticker": order.get_ticker(),
                "comm": order.get_exec_comm(),
                "pre" : order.get_start_pos(),
                "post": order.get_end_pos(),
                "target": order.get_target_pos(),
                "inertia": order.get_inertia(),
                "peek_ba": order.get_peek_ba(),
                "received_price": order.get_received_price(),
```

```python
                "slippage": order.get_slippage_penalty(),
                "market_spread_penalty": market_spread_penalty,
                "execution_spread_penalty": exec_spread_penalty,
                "fills": await order.get_tickets_and_fills(),
                "batch": self.ticker_to_batch[order.get_ticker()]
            }
        )

    df = pd.DataFrame(records).reset_index(drop=True)
    await self.db_service.asyn_insert_timeseries_df(
        dtype="log",
        dformat="tsexec",
        dfreq="o",
        df=df,
        series_metadata=series_metadata,
        series_identifier=series_identifier,
        allow_disjoint=True
    )
    return
```

Listing 3: order_executor.py

That's it! We now need to make the 'Order' object, which is the core of our execution logic. The Order logic is the class that interacts directly with the API - note that so far in the executor class we do not know or care what broker we are using, because this is 'hidden' and abstracted away. We can work with any Order object that implements the interface of the Order class. Let's take a look at our Order class.

2.2 Order

For the order class, we will just discuss the order logic in prose, and let readers peruse the source code through the pdf/text files at their own. This is because the code is somewhat long, and due to logging texts everywhere - it can be somewhat verbose. We will hence discuss the code, and allow readers to access the code files, which I believe many of you prefer anyway!

The first thing we want to do in the Order class is to see if our positional inertia is overriden. This determines the urgency at which we want to execute. If it is overriden, then we want to hit the target position at the end regardless of the slippage or costs - but first we want to try to capture the spread. If it is not overriden, then we want to see the price at which the signal was computed. We take a look at the quote, and if the quote plus any additional execution and spread costs are still more favorable then the price that we used to compute our signal, we go ahead and try to rebalance even if the positional change is not significant. This is effectively the concept of free rebalancing.

In the execute_order portion of the code (refer to the code files), this is the section that does what we just described (yes, the pdf renders it rather messily so I'm sending the code files).

```python
prev = round(self.pos_start_contracts, 5)
target = round(self.rounded_target_contracts, 5)
overriden = Order.is_overriden(self.inertia_level, prev, target)
posdiff = target - prev
self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : f"posdiff : {posdiff} overriden {overriden}"})
if posdiff == 0:
    self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : "finished execution with posdiff = 0"})
    return

if (self.trade_allowed == "long" and posdiff < 0 and target < 0) or \
   (self.trade_allowed == "short" and posdiff > 0 and target > 0) or \
   (self.trade_allowed == "close" and (np.sign(prev) != np.sign(target) or np.abs(target) > np.abs(prev))):

    self.log_buffer.append({
        "datetime" : datetime.now(pytz.utc),
        "message" : f"finished execution with posdiff {posdiff}, prev {prev} target {target} and trade allowed {self.trade_allowed}"
    })
    return

bid, ask = await self.broker_service.get_last_tick(symbol=self.ticker)
self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : f"bid-ask received = {bid}-{ask}"})
self.peek_ba = (bid, ask)
```

```

if not bid or not ask:
    self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : "
    finished execution with null bid-ask"})
    self.done = True
    return

if not overriden:
    self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : "
    position inertia not overriden"})
    if posdiff > 0 and ask + self.execution_commission < self.received_price:
        self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : f
    "make limit at bid from {prev} -> {target}"})
        await self.make_limit_at(open=bid, prev=prev, target=target)
    if posdiff < 0 and bid - self.execution_commission > self.received_price:
        self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : f
    "make limit at ask from {prev} -> {target}"})
        await self.make_limit_at(open=ask, prev=prev, target=target)
if overriden:
    self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : "
    position inertia overriden"})
    if posdiff > 0:
        self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : f
    "make limit at bid from {prev} -> {target}"})
        await self.make_limit_at(open=bid, prev=prev, target=target)
    if posdiff < 0:
        self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : f
    "make limit at ask from {prev} -> {target}"})
        await self.make_limit_at(open=ask, prev=prev, target=target)
```

Listing 4: orders.py::execute_order (snapshot of code section)

Interestingly enough, in order to close existing positions with a limit order, the MetaTrader terminal only allows us to use take profit orders. Additionally, this needs to be applied to the whole order, so that we cannot fractionally close an open position with limits using take profits. To overcome this, we submit orders of minimal volume, so that when closing fractional positions at limit we can just choose a subset of the orders that constitute a position.

On the other hand, at the end of the execution window, we want to clean up all our take profit limits and existing open limit orders. We then want to force the trades that are favorable OR urgent. This is similar in logic to the limit orders, but we make no attempt to force orders that are less favorable then the received price when they are not urgent. This is captured by the logic:

```python
if not overriden:
    self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : "position inertia not overriden"})
    if posdiff > 0 and ask + self.execution_commission < self.received_price:
        self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : f"make market market from {prev} -> {target}"})
        await self.make_market(prev=prev, target=target)
    if posdiff < 0 and bid - self.execution_commission > self.received_price:
        self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : f"make market market from {prev} -> {target}"})
        await self.make_market(prev=prev, target=target)
if overriden:
    self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : "position inertia overriden"})
    if posdiff > 0:
        await self.make_market(prev=prev, target=target)
    if posdiff < 0:
        await self.make_market(prev=prev, target=target)
    self.log_buffer.append({"datetime" : datetime.now(pytz.utc), "message" : f"make market market from {prev} -> {target}"})
```

Listing 5: order.py::force_order (another code section)

3 Concluding Remarks

The remainder of the implementation details will be left to the reader for self-study. Any questions related to the implementation will be answered in the comments!

References

[1] HangukQuant. A Barebone Implementation of an Order Executor. https://hangukquant.

substack.com/p/a-barebone-order-executor-w-code.

[2] HangukQuant. Costful Trading. https://hangukquant.substack.com/p/costful-trading-w-code.

[3] HangukQuant. Design and Implementation of a Quant Database. https://hangukquant.substack.com/p/117-pages-design-and-implementation?utm_source=substack.

[4] HangukQuant. Developing a Private MetaAPI Python Lib for MetaTrader Trading. https://open.substack.com/pub/hangukquant/p/developing-a-private-metaapi-python.

[5] HangukQuant. Improving our Database Service. https://hangukquant.substack.com/p/improving-our-database-service.

[6] Markowitz, H. Portfolio selection. *The Journal of Finance 7*, 1 (1952), 77–91.

[7] Nevmyvaka, Y., Kearns, M., Papandreou, M., and Sycara, K. Electronic trading in order-driven markets: Efficient execution. vol. 2005, pp. 190– 197.