Market Making with Alpha - Order Book Imbalance

Overview

Order Book Imbalance, also known as Order Flow Imbalance, is a widely recognized microstructure indicator often analyzed alongside trade flow. This concept has several derivatives, including the Micro-Price, VAMP (Volume Adjusted Mid Price), Weighted-Depth Order Book Price, and Static Order Book Imbalance, among others.

- Static Order Book Imbalance

$$Static\ Order\ Book\ Imbalance = \frac{\sum_i^N Q_{bid}^i - \sum_i^N Q_{ask}^i}{\sum_i^N Q_{bid}^i + \sum_i^N Q_{ask}^i}$$

Through standardization, an alternative expression for order imbalance can be obtained, one that does not include the normalization numerator.

$$Standardized\ Order\ Book\ Imbalance = standardize(\sum_i^N Q_{bid}^i - \sum_i^N Q_{ask}^i)$$

- Volume Adjusted Mid Price
  Be aware that price and quantity are cross-multiplied between bid and ask sides.

$$VAMP_{bbo} = \frac{P_{best\ bid} \times Q_{best\ ask} + P_{best\ ask} \times Q_{best\ bid}}{Q_{best\ bid} + Q_{best\ ask}}$$

$$VAMP_N = \frac{\sum_i^N P_{bid}^i \times Q_{ask}^i + \sum_i^N P_{ask}^i \times Q_{bid}^i}{\sum_i^N Q_{bid}^i + \sum_i^N Q_{ask}^i}$$

  where N is usually defined as a percentage of the mid price. For 1% market depth, compute bid side down to mid × 0.99 and ask side up to mid × 1.01.

- Weighted-Depth Order Book Price
  Be aware that price and quantity are multiplied within the same side

latest

$$Weighted - Depth\ Order\ Book\ Price = \frac{\sum_i^N P_{bid}^i \times Q_{bid}^i + \sum_i^N P_{ask}^i \times Q_{ask}^i}{\sum_i^N Q_{bid}^i + \sum_i^N Q_{ask}^i}$$

In this case, N—unlike in VAMP—is defined by a fixed total quantity. For 500 qty of market depth, compute bid side down to its total quantity reaches 500 and ask side up to its total quantity reaches 500. Alternatively, you can use notional value instead of quantity.

- Another variation is to combine VAMP with Weighted-Depth Order Book Price.

$$P_{effective\ bid}^N = \frac{\sum_i^N P_{bid}^i \times Q_{bid}^i}{\sum_i^N Q_{bid}^i}$$

$$P_{effective\ ask}^N = \frac{\sum_i^N P_{ask}^i \times Q_{ask}^i}{\sum_i^N Q_{ask}^i}$$

$$Q_{effective\ bid}^N = \sum_i^N Q_{bid}^i$$

$$Q_{effective\ ask}^N = \sum_i^N Q_{ask}^i$$

$$VAMP_{effective}^N = \frac{P_{effective\ bid}^N \times Q_{effective\ ask}^N + P_{effective\ ask}^N \times Q_{effective\ bid}^N}{Q_{effective\ bid}^N + Q_{effective\ ask}^N}$$

where N is defined just like VAMP.

You can also apply other time-series techniques or statistical adjustments to these derived order book indicators, such as standardization.

Extensive information on these indicators is available online. In the following examples, we begin by testing the standardized order book imbalance and then evaluate the other indicators in subsequent examples.

Reference

- The Micro-Price: A High Frequency Estimator of Future Prices
- Mind the Gaps: Short-Term Crypto Price Prediction
- Market microstructure signals

Note: This example is for educational purposes only and demonstrates effecti for high-frequency market-making schemes. All backtests are based on a 0.00 highest market maker rebate available on Binance Futures. See Binance Upgrades USDⓢ-

latest

Margined Futures Liquidity Provider Program for more details.

latest

```
[1]: import numpy as np

     from numba import njit, uint64
     from numba.typed import Dict

     from hftbacktest import (
         BacktestAsset,
         ROIVectorMarketDepthBacktest,
         GTX,
         LIMIT,
         BUY,
         SELL,
         BUY_EVENT,
         SELL_EVENT,
         Recorder
     )
     from hftbacktest.stats import LinearAssetRecord

     @njit
     def obi_mm(
         hbt,
         stat,
         half_spread,
         skew,
         c1,
         looking_depth,
         interval,
         window,
         order_qty_dollar,
         max_position_dollar,
         grid_num,
         grid_interval,
         roi_lb,
         roi_ub
     ):
         asset_no = 0
         imbalance_timeseries = np.full(30_000_000, np.nan, np.float64)

         tick_size = hbt.depth(0).tick_size
         lot_size = hbt.depth(0).lot_size

         t = 0
         roi_lb_tick = int(round(roi_lb / tick_size))
         roi_ub_tick = int(round(roi_ub / tick_size))

         while hbt.elapse(interval) == 0:
             hbt.clear_inactive_orders(asset_no)

             depth = hbt.depth(asset_no)
             position = hbt.position(asset_no)
             orders = hbt.orders(asset_no)

             best_bid = depth.best_bid
             best_ask = depth.best_ask

             mid_price = (best_bid + best_ask) / 2.0

             sum_ask_qty = 0.0
             from_tick = max(depth.best_ask_tick, roi_lb_tick)
             upto_tick = min(int(np.floor(mid_price * (1 + looking_depth) / tick_size)),
```

latest

```
roi_ub_tick)
        for price_tick in range(from_tick, upto_tick):
            sum_ask_qty += depth.ask_depth[price_tick - roi_lb_tick]

        sum_bid_qty = 0.0
        from_tick = min(depth.best_bid_tick, roi_ub_tick)
        upto_tick = max(int(np.ceil(mid_price * (1 - looking_depth) / tick_size)),
roi_lb_tick)
        for price_tick in range(from_tick, upto_tick, -1):
            sum_bid_qty += depth.bid_depth[price_tick - roi_lb_tick]

        imbalance_timeseries[t] = sum_bid_qty - sum_ask_qty

        # Standardizes the order book imbalance timeseries for a given window
        m = np.nanmean(imbalance_timeseries[max(0, t + 1 - window):t + 1])
        s = np.nanstd(imbalance_timeseries[max(0, t + 1 - window):t + 1])
        alpha = np.divide(imbalance_timeseries[t] - m, s)

        #--------------------------------------------------------
        # Computes bid price and ask price.

        order_qty = max(round((order_qty_dollar / mid_price) / lot_size) * lot_size, lot_size)
        fair_price = mid_price + c1 * alpha

        normalized_position = position / order_qty

        reservation_price = fair_price - skew * normalized_position

        bid_price = min(np.round(reservation_price - half_spread), best_bid)
        ask_price = max(np.round(reservation_price + half_spread), best_ask)

        bid_price = np.floor(bid_price / tick_size) * tick_size
        ask_price = np.ceil(ask_price / tick_size) * tick_size

        #--------------------------------------------------------
        # Updates quotes.

        # Creates a new grid for buy orders.
        new_bid_orders = Dict.empty(np.uint64, np.float64)
        if position * mid_price < max_position_dollar and np.isfinite(bid_price):
            for i in range(grid_num):
                bid_price_tick = round(bid_price / tick_size)

                # order price in tick is used as order id.
                new_bid_orders[uint64(bid_price_tick)] = bid_price

                bid_price -= grid_interval

        # Creates a new grid for sell orders.
        new_ask_orders = Dict.empty(np.uint64, np.float64)
        if position * mid_price > -max_position_dollar and np.isfinite(ask_price):
            for i in range(grid_num):
                ask_price_tick = round(ask_price / tick_size)

                # order price in tick is used as order id.
                new_ask_orders[uint64(ask_price_tick)] = ask_price

                ask_price += grid_interval

        order_values = orders.values();
```

latest

```
        while order_values.has_next():
            order = order_values.get()
            # Cancels if a working order is not in the new grid.
            if order.cancellable:
                if (
                    (order.side == BUY and order.order_id not in new_bid_orders)
                    or (order.side == SELL and order.order_id not in new_ask_orders)
                ):
                    hbt.cancel(asset_no, order.order_id, False)

        for order_id, order_price in new_bid_orders.items():
            # Posts a new buy order if there is no working order at the price on the new grid.
            if order_id not in orders:
                hbt.submit_buy_order(asset_no, order_id, order_price, order_qty, GTX, LIMIT,
False)

        for order_id, order_price in new_ask_orders.items():
            # Posts a new sell order if there is no working order at the price on the new
grid.
            if order_id not in orders:
                hbt.submit_sell_order(asset_no, order_id, order_price, order_qty, GTX, LIMIT,
False)

        t += 1

        if t >= len(imbalance_timeseries):
            raise Exception

        # Records the current state for stat calculation.
        stat.record(hbt)
```

latest

[2]:
```
%%time

roi_lb = 10000
roi_ub = 50000

latency_data = np.concatenate(
[np.load('latency/live_order_latency_{}.npz'.format(date))['data'] for date in range(20230501, 
20230532)]
)

asset = (
    BacktestAsset()
        .data(['data2/btcusdt_{}.npz'.format(date) for date in range(20230501, 20230532)])
        .initial_snapshot('data2/btcusdt_20230430_eod.npz')
        .linear_asset(1.0)
        .intp_order_latency(latency_data)
        .power_prob_queue_model(2)
        .no_partial_fill_exchange()
        .trading_value_fee_model(-0.00005, 0.0007)
        .tick_size(0.1)
        .lot_size(0.001)
        .roi_lb(roi_lb)
        .roi_ub(roi_ub)
)

hbt = ROIVectorMarketDepthBacktest([asset])

recorder = Recorder(1, 30_000_000)

half_spread = 80
skew = 3.5
c1 = 160
depth = 0.025 # 2.5% from the mid price
interval = 1_000_000_000 # 1s
window = 3_600_000_000_000 / interval # 1hour
order_qty_dollar = 50_000
max_position_dollar = order_qty_dollar * 50
grid_num = 1
grid_interval = hbt.depth(0).tick_size

obi_mm(
    hbt,
    recorder.recorder,
    half_spread,
    skew,
    c1,
    depth,
    interval,
    window,
    order_qty_dollar,
    max_position_dollar,
    grid_num,
    grid_interval,
    roi_lb,
    roi_ub
)

hbt.close()
```

```
recorder.to_npz('stats/obi_btcusdt.npz')
```

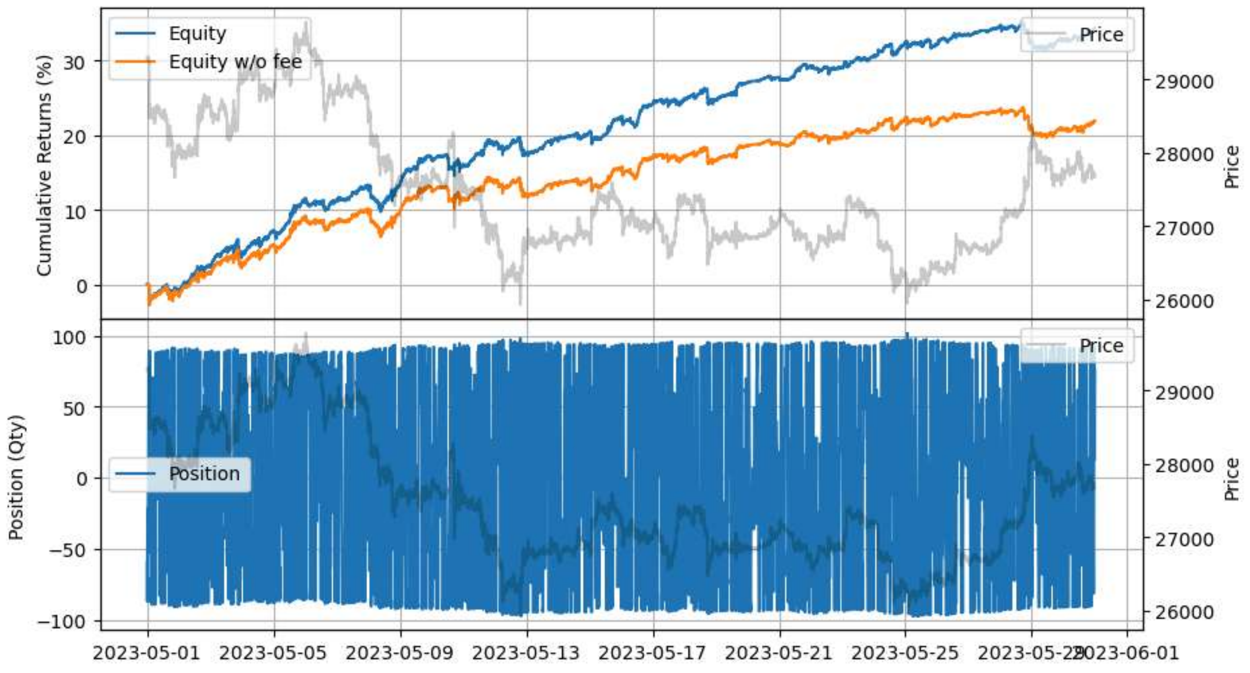
```
CPU times: user 32min 23s, sys: 43.9 s, total: 33min 7s
Wall time: 33min 5s
```

[3]:
```
data = np.load('stats/obi_btcusdt.npz')['0']
stats = (
    LinearAssetRecord(data)
        .resample('5m')
        .stats(book_size=2_500_000)
)
stats.summary()
```

[3]: shape: (1, 11)

start	end	SR	Sortino	Return	MaxDrawdown	DailyNumberOfTrades	DailyTurnc
datetime[μs]	datetime[μs]	f64	f64	f64	f64	f64	
2023-05-01 00:00:00	2023-05-30 23:55:00	10.829336	13.5994	0.342371	0.037249	4119.876838	82.397

[4]:
```
stats.plot()
```



[5]:
```
%%time

roi_lb = 0
roi_ub = 3000

latency_data = np.concatenate(
[np.load('latency/live_order_latency_{}.npz'.format(date))['data'] for date in range(20230501,
20230532)]
)

asset = (
    BacktestAsset()
        .data(['data2/ethusdt_{}.npz'.format(date) for date in range(20230501, 20230532)])
        .initial_snapshot('data2/ethusdt_20230430_eod.npz')
        .linear_asset(1.0)
        .intp_order_latency(latency_data)
        .power_prob_queue_model(2)
        .no_partial_fill_exchange()
        .trading_value_fee_model(-0.00005, 0.0007)
        .tick_size(0.01)
        .lot_size(0.001)
        .roi_lb(roi_lb)
        .roi_ub(roi_ub)
)

hbt = ROIVectorMarketDepthBacktest([asset])

recorder = Recorder(1, 30_000_000)

half_spread = 5
skew = 0.2
c1 = 10
depth = 0.025 # 2.5% from the mid price
interval = 1_000_000_000 # 1s
window = 3_600_000_000_000 / interval # 1hour
order_qty_dollar = 50_000
max_position_dollar = order_qty_dollar * 50
grid_num = 1
grid_interval = hbt.depth(0).tick_size

obi_mm(
    hbt,
    recorder.recorder,
    half_spread,
    skew,
    c1,
    depth,
    interval,
    window,
    order_qty_dollar,
    max_position_dollar,
    grid_num,
    grid_interval,
    roi_lb,
    roi_ub
)

hbt.close()
```

latest

```
recorder.to_npz('stats/obi_ethusdt.npz')
```

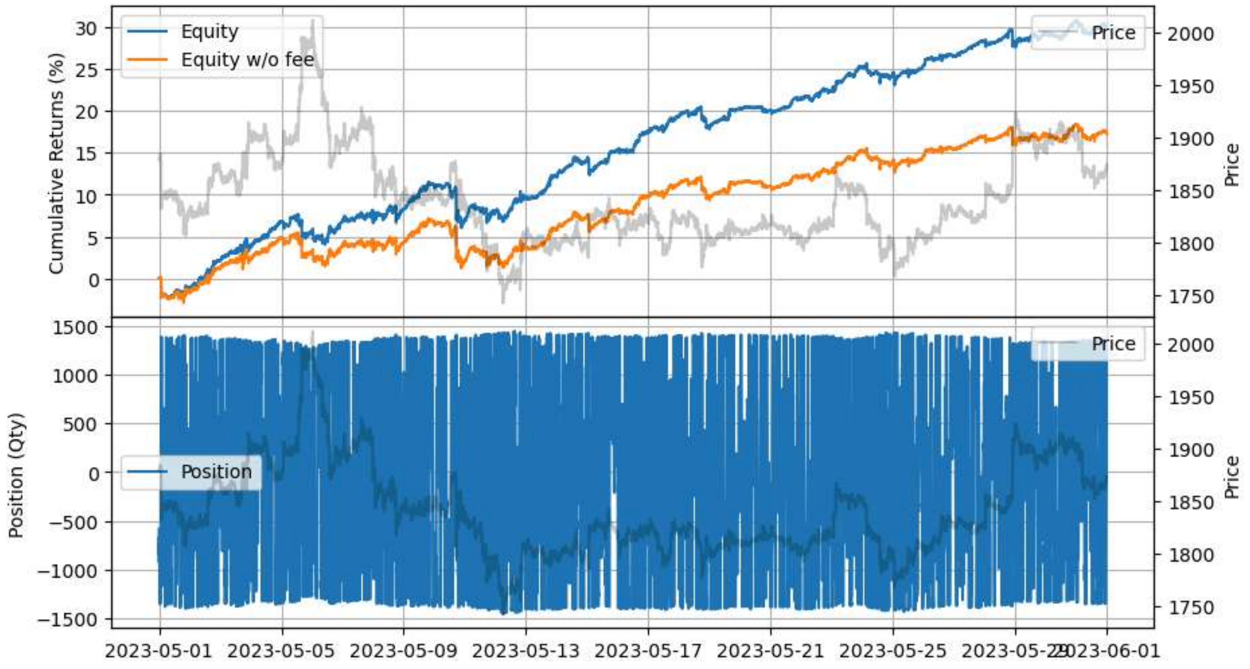
```
CPU times: user 27min 37s, sys: 38.3 s, total: 28min 15s
Wall time: 28min 16s
```

[6]:
```
data = np.load('stats/obi_ethusdt.npz')['0']
stats = (
    LinearAssetRecord(data)
        .resample('5m')
        .stats(book_size=2_500_000)
)
stats.summary()
```

[6]: shape: (1, 11)

start	end	SR	Sortino	Return	MaxDrawdown	DailyNumberOfTrades	DailyTurr
datetime[μs]	datetime[μs]	f64	f64	f64	f64	f64	
2023-05-01 00:00:00	2023-05-31 23:55:00	9.017874	11.140311	0.299582	0.055187	4112.621933	82.25

[7]:
```
stats.plot()
```



Another approach is to generate trading volume to qualify as a market maker and receive rebates. This strategy involves maintaining a high skew and tight spread. While the strategy itself may not be profitable or may incur losses, it can help achieve market maker status.

```
[9]: %%time

roi_lb = 10000
roi_ub = 50000

latency_data = np.concatenate(
[np.load('latency/live_order_latency_{}.npz'.format(date))['data'] for date in range(20230501,
20230532)]
)

asset = (
    BacktestAsset()
        .data(['data2/btcusdt_{}.npz'.format(date) for date in range(20230501, 20230532)])
        .initial_snapshot('data2/btcusdt_20230430_eod.npz')
        .linear_asset(1.0)
        .intp_order_latency(latency_data)
        .power_prob_queue_model(2)
        .no_partial_fill_exchange()
        .trading_value_fee_model(-0.00005, 0.0007)
        .tick_size(0.1)
        .lot_size(0.001)
        .roi_lb(roi_lb)
        .roi_ub(roi_ub)
)

hbt = ROIVectorMarketDepthBacktest([asset])

recorder = Recorder(1, 30_000_000)

half_spread = 10
skew = 2
c1 = 20
depth = 0.001 # 0.1% from the mid price
interval = 500_000_000 # 500ms
window = 600_000_000_000 / interval # 10min
order_qty_dollar = 25_000
max_position_dollar = order_qty_dollar * 20
grid_num = 1
grid_interval = hbt.depth(0).tick_size

obi_mm(
    hbt,
    recorder.recorder,
    half_spread,
    skew,
    c1,
    depth,
    interval,
    window,
    order_qty_dollar,
    max_position_dollar,
    grid_num,
    grid_interval,
    roi_lb,
    roi_ub
)

recorder.to_npz('stats/obi_vg_btcusdt.npz')
```

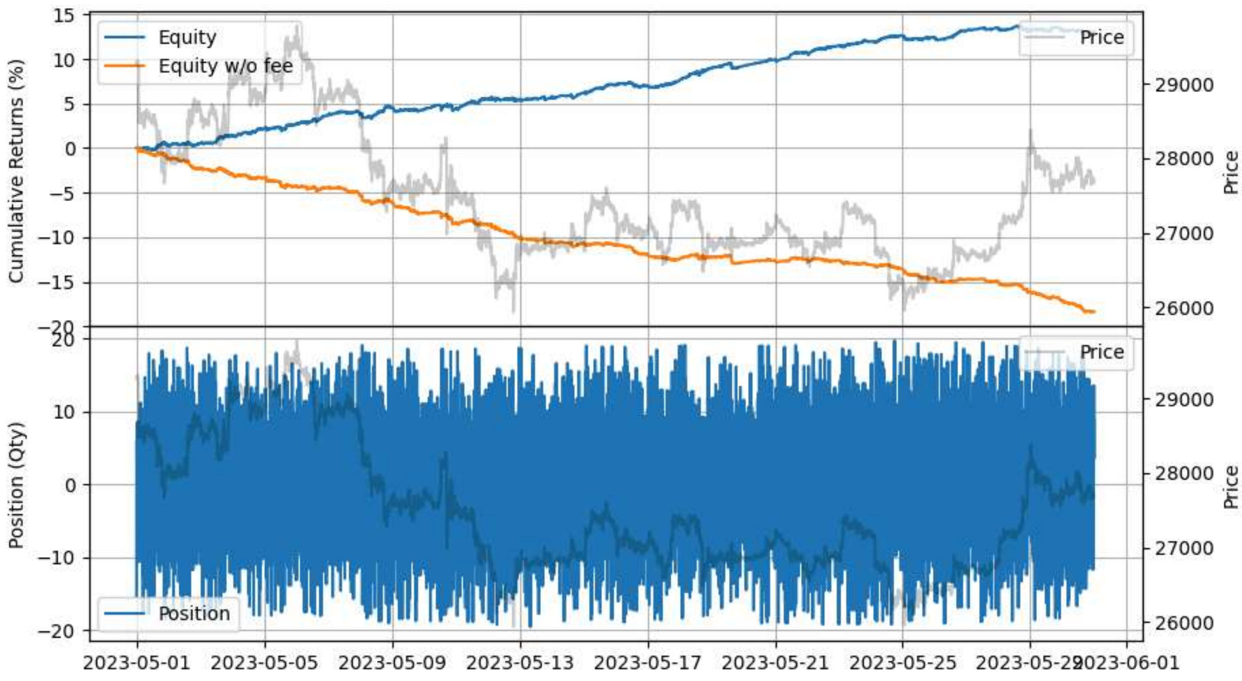
```
CPU times: user 30min 13s, sys: 44.1 s, total: 30min 57s
Wall time: 31min 2s
```

[10]:
```
data = np.load('stats/obi_vg_btcusdt.npz')['0']
stats = (
    LinearAssetRecord(data)
        .resample('5m')
        .stats(book_size=1_000_000)
)
stats.summary()
```

[10]: shape: (1, 11)

start	end	SR	Sortino	Return	MaxDrawdown	DailyNumberOfTrades	DailyTurno
datetime[μs]	datetime[μs]	f64	f64	f64	f64	f64	
2023-05-01 00:00:00	2023-05-30 23:55:00	14.0088	17.366641	0.129901	0.00928	8368.135201	209.203

[11]:
```
stats.plot()
```



Update on Backtesting Results for 2025

Updated backtesting results for February 2025 using the same parameters as in May 2023, except for the `power_prob_queue_model` parameter, which was changed from 2 to 3 to reflect market changes and a more challenging fill in the queue.

[13]:
```
import datetime

start_date = datetime.datetime.strptime('20250101', '%Y%m%d')
end_date = datetime.datetime.strptime('20250228', '%Y%m%d')
```

latest

[14]:
```
%%time

roi_lb = 50000
roi_ub = 150000

latency_data = []
date = start_date
while date <= end_date:
    latency_data.append('latency/order_latency_{}.npz'.format(date.strftime('%Y%m%d')))
    date += datetime.timedelta(days=1)

data = []
date = start_date
while date <= end_date:
    data.append('data2/btcusdt_{}.npz'.format(date.strftime("%Y%m%d")))
    date += datetime.timedelta(days=1)

asset = (
    BacktestAsset()
        .data(data)
        .initial_snapshot('data2/btcusdt_20241231_eod.npz')
        .linear_asset(1.0)
        .intp_order_latency(latency_data)
        .power_prob_queue_model(3)
        .no_partial_fill_exchange()
        .trading_value_fee_model(-0.00005, 0.0007)
        .tick_size(0.1)
        .lot_size(0.001)
        .roi_lb(roi_lb)
        .roi_ub(roi_ub)
)

hbt = ROIVectorMarketDepthBacktest([asset])

recorder = Recorder(1, 30_000_000)

half_spread = 80
skew = 3.5
c1 = 160
depth = 0.025 # 2.5% from the mid price
interval = 1_000_000_000 # 1s
window = 3_600_000_000_000 / interval # 1hour
order_qty_dollar = 50_000
max_position_dollar = order_qty_dollar * 50
grid_num = 1
grid_interval = hbt.depth(0).tick_size

obi_mm(
    hbt,
    recorder.recorder,
    half_spread,
    skew,
    c1,
    depth,
    interval,
    window,
    order_qty_dollar,
    max_position_dollar,
    grid_num,
    grid_interval,
```
latest

```
    roi_lb,
    roi_ub
)

hbt.close()

recorder.to_npz('stats/obi_btcusdt_2025.npz')
```

```
CPU times: user 1h 56min 52s, sys: 8min 33s, total: 2h 5min 25s
Wall time: 1h 29min 36s
```

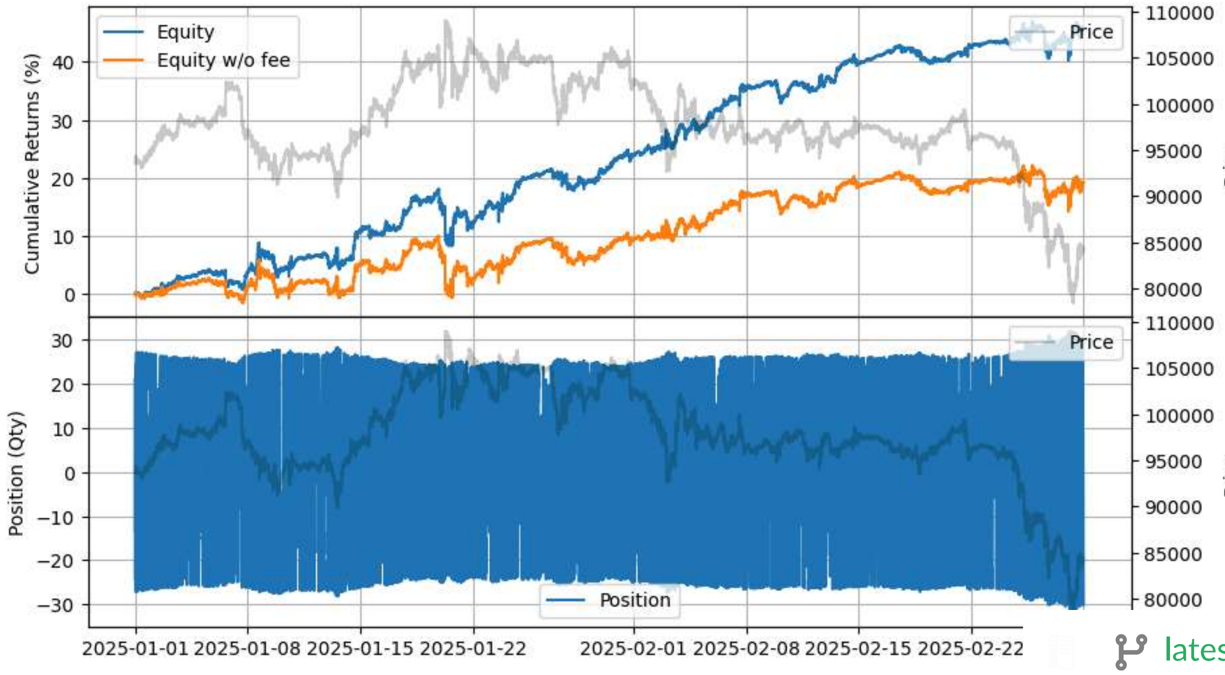
You can see from the following report that the order book imbalance continues to work consistently. However, the return per trade drops from 0.0139% to 0.0086%, including the 0.005% rebates. This again highlights the importance of rebates and the corresponding fee structure for market makers.

[16]:
```
data = np.load('stats/obi_btcusdt_2025.npz')['0']
stats = (
    LinearAssetRecord(data)
        .resample('5m')
        .stats(book_size=2_500_000)
)
stats.summary()
```

[16]: shape: (1, 11)

start	end	SR	Sortino	Return	MaxDrawdown	DailyNumberOfTrades	DailyTurn
datetime[μs]	datetime[μs]	f64	f64	f64	f64	f64	
2025-01-01 00:00:00	2025-02-28 23:55:00	5.369875	7.198137	0.459649	0.097893	4533.741393	90.675

[17]:
```
stats.plot()
```



latest

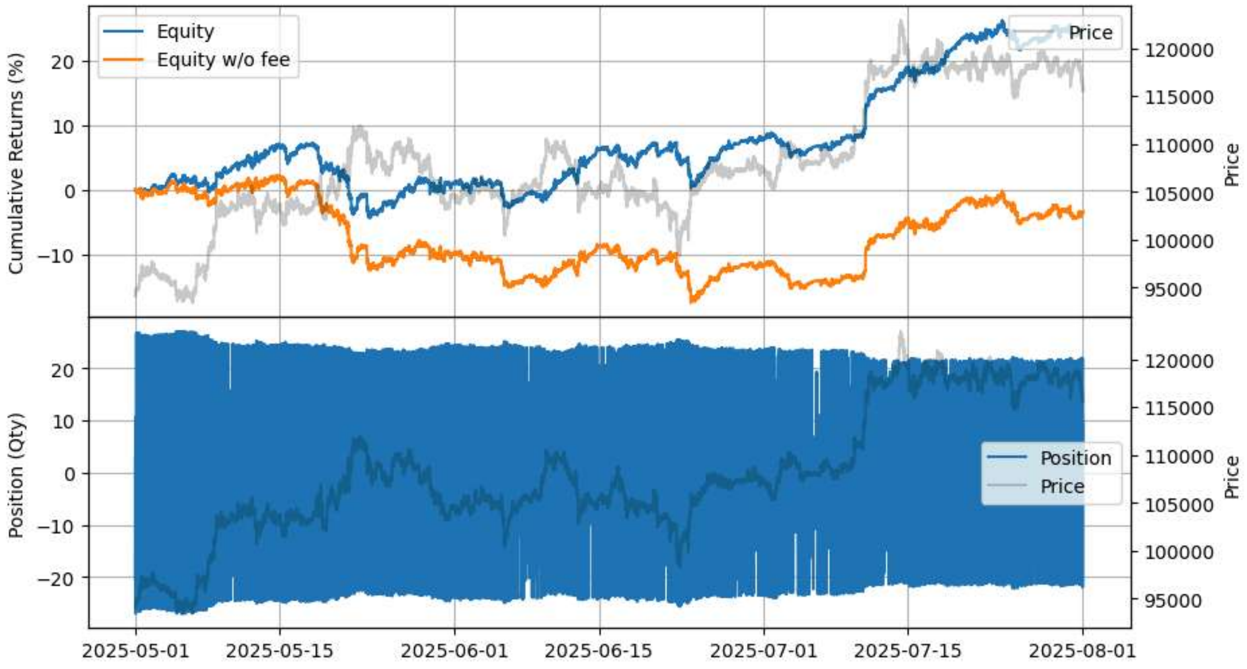
```
[2]: data = np.load('stats/obi_btcusdt_202505.npz')['0']
     stats = (
         LinearAssetRecord(data)
             .resample('5m')
             .stats(book_size=2_500_000)
     )
     stats.summary()
```

[2]: shape: (1, 11)

start	end	SR	Sortino	Return	MaxDrawdown	DailyNumberOfTrades	DailyTurno
datetime[μs]	datetime[μs]	f64	f64	f64	f64	f64	
2025-05-01 00:00:00	2025-07-31 23:55:00	3.037498	4.129756	0.250298	0.115695	3095.899453	61.918

```
[3]: stats.plot()
```

[3]:


Equity
Equity w/o fee
Price
Cumulative Returns (%)
Price
Position
Price
Position (Qty)
Price