Guéant–Lehalle–Fernandez-Tapia Market Making Model and Grid Trading

Overview

Grid trading is straightforward and easy to comprehend, and it excels in high-frequency environments. However, given the intricacies of high-frequency trading, which necessitate comprehensive tick-by-tick simulation with latencies and order fill simulation, optimizing the ideal spread, order interval, and skew can be a challenging task. Furthermore, these values fluctuate over time, especially in response to market conditions, making a fixed setup less than optimal.

To improve grid trading's adaptability, one solution is to combine it with a well-developed market-making model. Let's delve into how this can be achieved.

Guéant–Lehalle–Fernandez-Tapia Market Making Model

This model represents an advanced evolution of the well-known Avellaneda-Stoikov model and provides a closed-form approximation of asymptotic behavior for terminal time T. Simply, this model does not specify a terminal time, which makes it suitable for typical stocks, spot assets, or crypto perpetual contracts. By employing this model, it is anticipated that the half spread and skew will be accurately adjusted according to market conditions.

In this analysis, we will focus on equations (4.6) and (4.7) in Optimal market making and explore how they can be applied to real-world scenarios.

The optimal bid quote depth, δ^{b*}_{approx}, and ask quote depth, δ^{a*}_{approx}, are derived from the fair price as follows:

$$\delta^{b*}_{approx}(q) = \frac{1}{\xi\Delta} log(1 + \frac{\xi\Delta}{k}) + \frac{2q + \Delta}{2} \sqrt{\frac{\gamma\sigma^2}{2A\Delta k}(1 + \frac{\xi\Delta}{k})^{\frac{k}{\xi\Delta}+1}} \tag{4.6}$$

$$\delta^{a*}_{approx}(q) = \frac{1}{\xi\Delta} log(1 + \frac{\xi\Delta}{k}) - \frac{2q - \Delta}{2} \sqrt{\frac{\gamma\sigma^2}{2A\Delta k}(1 + \frac{\xi\Delta}{k})^{\frac{k}{\xi\Delta}+1}} \tag{4.7}$$

Let's introduce c_1 and c_2 and define them by extracting the volatility σ from the square root:

$$c_1 = \frac{1}{\xi\Delta} log(1 + \frac{\xi\Delta}{k})$$
$$c_2 = \sqrt{\frac{\gamma}{2A\Delta k}(1 + \frac{\xi\Delta}{k})^{\frac{k}{\xi\Delta}+1}}$$

Now we can rewrite equations (4.6) and (4.7) as follows:

$$\delta^{b*}_{approx}(q) = c_1 + \frac{\Delta}{2}\sigma c_2 + q\sigma c_2$$
$$\delta^{a*}_{approx}(q) = c_1 + \frac{\Delta}{2}\sigma c_2 - q\sigma c_2$$

As you can see, this consists of the half spread and skew. q represents a market maker's inventory(position).

$$\text{half spread} = C_1 + \frac{\Delta}{2}\sigma C_2$$
$$\text{skew} = \sigma C_2$$
$$\delta^{b*}_{approx}(q) = \text{half spread} + \text{skew} \times q$$
$$\delta^{a*}_{approx}(q) = \text{half spread} - \text{skew} \times q$$

Thus,

$$\text{bid price} = \text{fair price} - (\text{half spread} + \text{skew} \times q)$$
$$\text{ask price} = \text{fair price} + (\text{half spread} - \text{skew} \times q)$$

You can find similarities in what the following two articles describe.
Stochastic Control Theory and High Frequency Trading
How to Market Make Bitcoin Derivatives Lesson 2

Calculating Trading Intensity

To determine the optimal quotes, we need to compute c_1 and c_2. In order to do that, we need to calibrate A and k of trading intensity, as well as calculate the market volatility σ.

Trading intensity is defined as:

$$\lambda = A\exp(-k\delta)$$

We will calibrate these values using market data according to the this article. In order to do that, we need to record market order's arrivals.

Our market maker will react every 100ms, which means they will post or cancel orders at this interval. So, our quotes' trading intensity will be measured in the same time-step. Ideally, we should also account for our orders' queue position; however, to simplify the problem, we will not consider the order queue position in this analysis.

[1]:
```
from numba import njit
from hftbacktest import BUY_EVENT

import numpy as np

@njit
def measure_trading_intensity_and_volatility(hbt):
    tick_size = hbt.depth(0).tick_size
    arrival_depth = np.full(10_000_000, np.nan, np.float64)
    mid_price_chg = np.full(10_000_000, np.nan, np.float64)

    t = 0
    prev_mid_price_tick = np.nan
    mid_price_tick = np.nan

    # Checks every 100 milliseconds.
    while hbt.elapse(100_000_000) == 0:
        #---------------------------------------------------
        # Records market order's arrival depth from the mid-price.
        if not np.isnan(mid_price_tick):
            depth = -np.inf
            for last_trade in hbt.last_trades(0):
                trade_price_tick = last_trade.px / tick_size

                if last_trade.ev & BUY_EVENT == BUY_EVENT:
                    depth = np.nanmax([trade_price_tick - mid_price_tick, depth])
                else:
                    depth = np.nanmax([mid_price_tick - trade_price_tick, depth])
            arrival_depth[t] = depth

        hbt.clear_last_trades(0)

        depth = hbt.depth(0)

        best_bid_tick = depth.best_bid_tick
        best_ask_tick = depth.best_ask_tick

        prev_mid_price_tick = mid_price_tick
        mid_price_tick = (best_bid_tick + best_ask_tick) / 2.0

        # Records the mid-price change for volatility calculation.
        mid_price_chg[t] = mid_price_tick - prev_mid_price_tick

        t += 1
        if t >= len(arrival_depth) or t >= len(mid_price_chg):
            raise Exception
    return arrival_depth[:t], mid_price_chg[:t]
```

Since we're not considering the order's queue position when measuring trading market trades that cross our quote will be counted as executed.

Note: The trading intensity in `out` of `measure_trading_intensity` is incorrectly in half-tick units instead of tick units. Although this fix requires adjusting parameters in all related examples, the example is left unchanged to preserve existing results.

[2]:
```
@njit
def measure_trading_intensity(order_arrival_depth, out):
    max_tick = 0
    for depth in order_arrival_depth:
        if not np.isfinite(depth):
            continue

        # Sets the tick index to 0 for the nearest possible best price
        # as the order arrival depth in ticks is measured from the mid-price
        tick = round(depth / .5) - 1

        # In a fast-moving market, buy trades can occur below the mid-price (and vice versa
for sell trades)
        # since the mid-price is measured in a previous time-step;
        # however, to simplify the problem, we will exclude those cases.
        if tick < 0 or tick >= len(out):
            continue

        # All of our possible quotes within the order arrival depth,
        # excluding those at the same price, are considered executed.
        out[:tick] += 1

        max_tick = max(max_tick, tick)
    return out[:max_tick]
```

Run HftBacktest to replay the market and record order arrival depth and price changes.

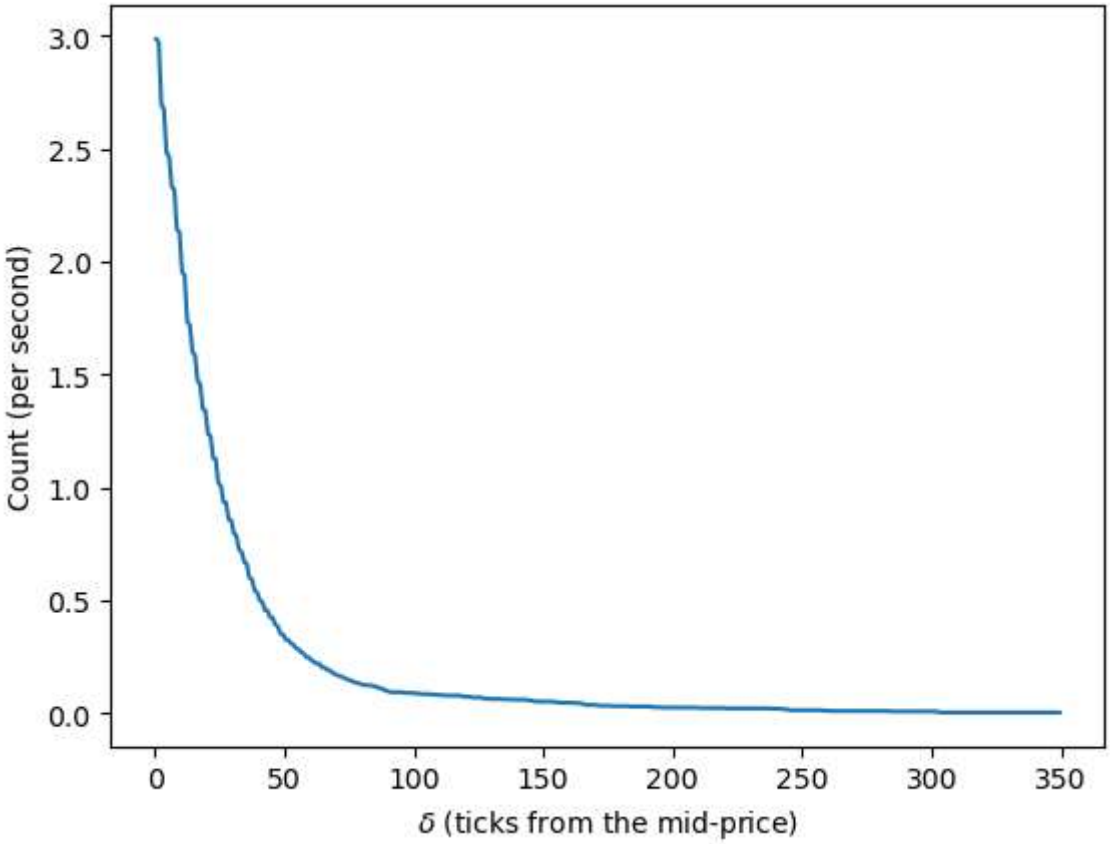
[3]:
```
from hftbacktest import BacktestAsset, ROIVectorMarketDepthBacktest

asset = (
    BacktestAsset()
        .data([
            'data/ethusdt_20221003.npz'
        ])
        .initial_snapshot('data/ethusdt_20221002_eod.npz')
        .linear_asset(1.0)
        .intp_order_latency([
            'latency/feed_latency_20221003.npz'
        ])
        .power_prob_queue_model(2.0)
        .no_partial_fill_exchange()
        .trading_value_fee_model(-0.00005, 0.0007)
        .tick_size(0.01)
        .lot_size(0.001)
        .roi_lb(0.0)
        .roi_ub(3000.0)
        .last_trades_capacity(10000)
)

hbt = ROIVectorMarketDepthBacktest([asset])

arrival_depth, mid_price_chg = measure_trading_intensity_and_volatility(hbt)

_ = hbt.close()
```

Measure trading intensity from the recorded order arrival depth and plot it.

[4]:
```
tmp = np.zeros(500, np.float64)

# Measures trading intensity (lambda) for the first 10-minute window.
lambda_ = measure_trading_intensity(arrival_depth[:6_000], tmp)

# Since it is measured for a 10-minute window, divide by 600 to convert it to per second.
lambda_ /= 600

# Creates ticks from the mid-price.
ticks = np.arange(len(lambda_)) + .5
```

[5]:
```
from matplotlib import pyplot as plt

plt.plot(ticks, lambda_)
plt.xlabel('$ \delta $ (ticks from the mid-price)')
plt.ylabel('Count (per second)')
```

[5]:
```
Text(0, 0.5, 'Count (per second)')
```



Calibrate A and k using linear regression, since by taking the logarithm of both sides of lambda, it becomes $log\lambda = -k\delta + logA$.

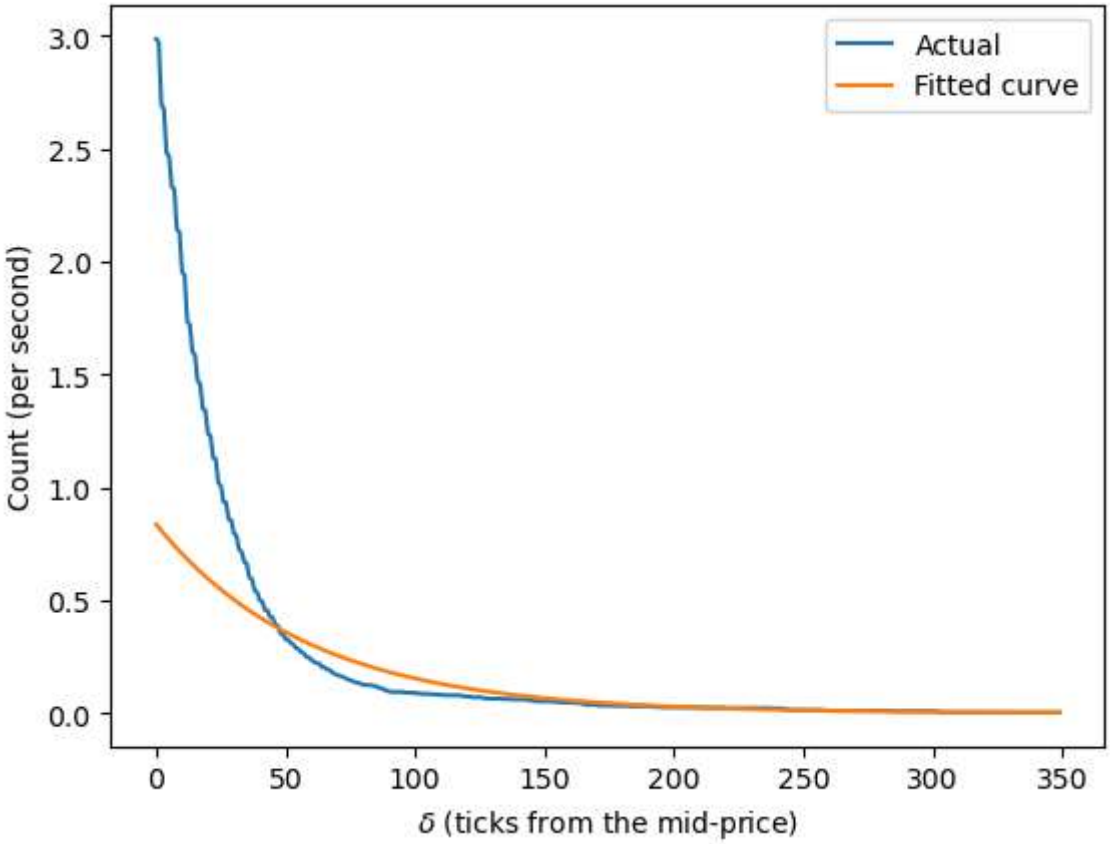
[6]:
```
@njit
def linear_regression(x, y):
    sx = np.sum(x)
    sy = np.sum(y)
    sx2 = np.sum(x ** 2)
    sxy = np.sum(x * y)
    w = len(x)
    slope = (w * sxy - sx * sy) / (w * sx2 - sx**2)
    intercept = (sy - slope * sx) / w
    return slope, intercept
```

[7]:
```
y = np.log(lambda_)
k_, logA = linear_regression(ticks, y)
A = np.exp(logA)
k = -k_

print('A={}, k={}'.format(A, k))
```

```
A=0.8426573649994981, k=0.016958811558646644
```

[8]:
```
plt.plot(lambda_)
plt.plot(A * np.exp(-k * ticks))
plt.xlabel('$ \delta $ (ticks from the mid-price)')
plt.ylabel('Count (per second)')
plt.legend(['Actual', 'Fitted curve'])
```

[8]:
```
<matplotlib.legend.Legend at 0x7fac86b74760>
```



As you can see, the fitted lambda function is not accurate across the entire range. More specifically, it overestimates the trading intensity for the shallow range near the mid-price and underestimates it for the deep range away from the mid-price.

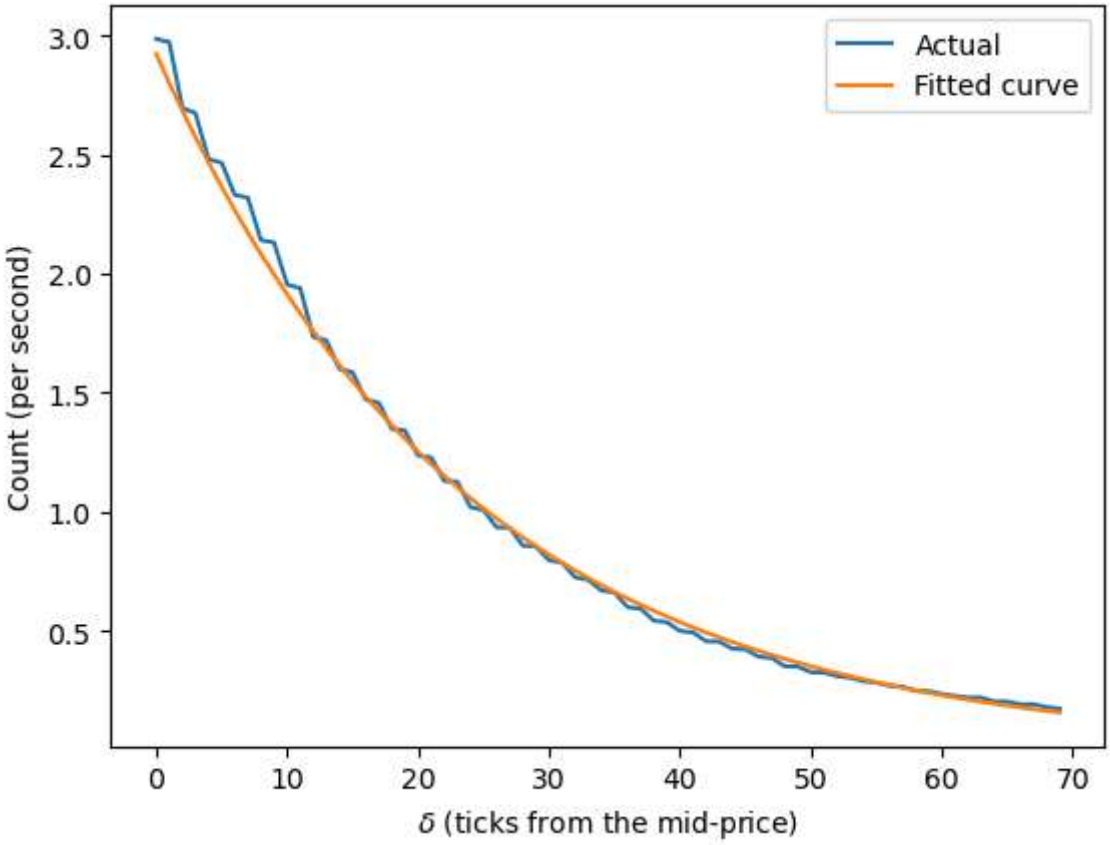
Since our quotes are likely to be placed in the range close to the mid-price, at least under typical market conditions (excluding high volatility conditions), we will refit the function specifically for the nearest range.

[9]:
```
# Refits for the range un to 70 ticks.
x_shallow = ticks[:70]
lambda_shallow = lambda_[:70]

y = np.log(lambda_shallow)
k_, logA = linear_regression(x_shallow, y)
A = np.exp(logA)
k = -k_

print('A={}, k={}'.format(A, k))
```

```
A=2.986162360812285, k=0.04235741115084049
```

[10]:
```
plt.plot(lambda_shallow)
plt.plot(A * np.exp(-k * x_shallow))
plt.xlabel('$ \delta $ (ticks from the mid-price)')
plt.ylabel('Count (per second)')
plt.legend(['Actual', 'Fitted curve'])
```

[10]: `<matplotlib.legend.Legend at 0x7fac86b77070>`



Now, we have a more accurate trading intensity function. Let's see where our quote will be placed.

But before we do that, let's calculate the volatility first.

[11]:
```
# Since we need volatility in ticks per square root of a second and our measurement is every
100ms,
# multiply by the square root of 10.
volatility = np.nanstd(mid_price_chg) * np.sqrt(10)
print(volatility)
```

```
10.725509539115974
```

Compute c_1 and c_2 according to the equations.

[12]:
```
@njit
def compute_coeff(xi, gamma, delta, A, k):
    inv_k = np.divide(1, k)
    c1 = 1 / (xi * delta) * np.log(1 + xi * delta * inv_k)
    c2 = np.sqrt(np.divide(gamma, 2 * A * delta * k) * ((1 + xi * delta * inv_k) ** (k / (xi *
delta) + 1)))
    return c1, c2
```

In the Guéant–Lehalle–Fernandez-Tapia formula, $\Delta = 1$ and $\xi = \gamma$. the value of γ is arbitrarily chosen.

[13]:
```
gamma = 0.05
delta = 1
volatility = 10.69

c1, c2 = compute_coeff(gamma, gamma, delta, A, k)

half_spread_tick = 1 * c1 + 1 / 2 * c2 * volatility
skew = c2 * volatility
print('half_spread_tick={}, skew={}'.format(half_spread_tick, skew))
```

```
half_spread_tick=20.47208533844371, skew=9.76326865029227
```

What does it mean when your quote is positioned 20 ticks away from the mid-price? By analyzing the recorded order arrival depth, you can identify the number of market trades you'll participate in as a market maker, measured in terms of count instead of volume. Additionally, the skew appears to be quite strong, as accumulating just two positions offsets the entire half spread.

[14]:
```
from scipy import stats

# inverse of percentile
pct = stats.percentileofscore(arrival_depth[np.isfinite(arrival_depth)], half_spread_tick)
your_pct = 100 - pct
print('{:.2f}%'.format(your_pct))
```

```
1.86%
```

Approximately 1.86% of market trades per given time-step could execute your quote. Be aware that it's not the percentage of the traded quantity.

Implement a Market Maker using the Model

> **Note:** This example is for educational purposes only and demonstrates effective strategies for high-frequency market-making schemes. All backtests are based on a 0.005% rebate, the highest market maker rebate available on Binance Futures. See Binance Upgrades USDⓈ-Margined Futures Liquidity Provider Program for more details.

In this example, we will disregard the forecast term and assume that the fair price is equal to the mid price, as we can expect the intrinsic value to remain stable in the short term.

[15]:
```
from numba.typed import Dict
from hftbacktest import BUY, SELL, GTX, LIMIT

out_dtype = np.dtype([
    ('half_spread_tick', 'f8'),
    ('skew', 'f8'),
    ('volatility', 'f8'),
    ('A', 'f8'),
    ('k', 'f8')
])

@njit
def glft_market_maker(hbt, recorder):
    tick_size = hbt.depth(0).tick_size
    arrival_depth = np.full(10_000_000, np.nan, np.float64)
    mid_price_chg = np.full(10_000_000, np.nan, np.float64)
    out = np.zeros(10_000_000, out_dtype)

    t = 0
    prev_mid_price_tick = np.nan
    mid_price_tick = np.nan

    tmp = np.zeros(500, np.float64)
    ticks = np.arange(len(tmp)) + 0.5

    A = np.nan
    k = np.nan
    volatility = np.nan
    gamma = 0.05
    delta = 1

    order_qty = 1
    max_position = 20

    # Checks every 100 milliseconds.
    while hbt.elapse(100_000_000) == 0:
        #---------------------------------------------------------
        # Records market order's arrival depth from the mid-price.
        if not np.isnan(mid_price_tick):
            depth = -np.inf
            for last_trade in hbt.last_trades(0):
                trade_price_tick = last_trade.px / tick_size

                if last_trade.ev & BUY_EVENT == BUY_EVENT:
                    depth = np.nanmax([trade_price_tick - mid_price_tick, depth])
                else:
                    depth = np.nanmax([mid_price_tick - trade_price_tick, depth])
            arrival_depth[t] = depth

        hbt.clear_last_trades(0)
        hbt.clear_inactive_orders(0)

        depth = hbt.depth(0)
        position = hbt.position(0)
        orders = hbt.orders(0)

        best_bid_tick = depth.best_bid_tick
        best_ask_tick = depth.best_ask_tick

        prev_mid_price_tick = mid_price_tick
```

```
        mid_price_tick = (best_bid_tick + best_ask_tick) / 2.0

        # Records the mid-price change for volatility calculation.
        mid_price_chg[t] = mid_price_tick - prev_mid_price_tick

        #--------------------------------------------------------
        # Calibrates A, k and calculates the market volatility.

        # Updates A, k, and the volatility every 5-sec.
        if t % 50 == 0:
            # Window size is 10-minute.
            if t >= 6_000 - 1:
                # Calibrates A, k
                tmp[:] = 0
                lambda_ = measure_trading_intensity(arrival_depth[t + 1 - 6_000:t + 1], tmp)
                if len(lambda_) > 2:
                    lambda_ = lambda_[:70] / 600
                    x = ticks[:len(lambda_)]
                    y = np.log(lambda_)
                    k_, logA = linear_regression(x, y)
                    A = np.exp(logA)
                    k = -k_

                # Updates the volatility.
                volatility = np.nanstd(mid_price_chg[t + 1 - 6_000:t + 1]) * np.sqrt(10)

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

        c1, c2 = compute_coeff(gamma, gamma, delta, A, k)

        half_spread_tick = c1 + delta / 2 * c2 * volatility
        skew = c2 * volatility

        reservation_price_tick = mid_price_tick - skew * position

        bid_price_tick = np.minimum(np.round(reservation_price_tick - half_spread_tick),
best_bid_tick)
        ask_price_tick = np.maximum(np.round(reservation_price_tick + half_spread_tick),
best_ask_tick)

        bid_price = bid_price_tick * tick_size
        ask_price = ask_price_tick * tick_size

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

        # Cancel orders if they differ from the updated bid and ask prices.
        order_values = orders.values();
        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.price != bid_price)
                    or (order.side == SELL and order.price != ask_price)
                ):
                    hbt.cancel(0, order.order_id, False)

        # If the current position is within the maximum position,
```

```
        # submit the new order only if no order exists at the same price.
        if position < max_position and np.isfinite(bid_price):
            bid_price_as_order_id = round(bid_price / tick_size)
            if bid_price_as_order_id not in orders:
                hbt.submit_buy_order(0, bid_price_as_order_id, bid_price, order_qty, GTX,
LIMIT, False)
        if position > -max_position and np.isfinite(ask_price):
            ask_price_as_order_id = round(ask_price / tick_size)
            if ask_price_as_order_id not in orders:
                hbt.submit_sell_order(0, ask_price_as_order_id, ask_price, order_qty, GTX,
LIMIT, False)

        #--------------------------------------------------------
        # Records variables and stats for analysis.

        out[t].half_spread_tick = half_spread_tick
        out[t].skew = skew
        out[t].volatility = volatility
        out[t].A = A
        out[t].k = k

        t += 1

        if t >= len(arrival_depth) or t >= len(mid_price_chg) or t >= len(out):
            raise Exception

        # Records the current state for stat calculation.
        recorder.record(hbt)
    return out[:t]
```

[16]:
```
from hftbacktest import Recorder
from hftbacktest.stats import LinearAssetRecord

asset = (
    BacktestAsset()
        .data([
            'data/ethusdt_20221003.npz'
        ])
        .initial_snapshot('data/ethusdt_20221002_eod.npz')
        .linear_asset(1.0)
        .intp_order_latency([
            'latency/feed_latency_20221003.npz'
        ])
        .power_prob_queue_model(2.0)
        .no_partial_fill_exchange()
        .trading_value_fee_model(-0.00005, 0.0007)
        .tick_size(0.01)
        .lot_size(0.001)
        .roi_lb(0.0)
        .roi_ub(3000.0)
        .last_trades_capacity(10000)
)

hbt = ROIVectorMarketDepthBacktest([asset])

recorder = Recorder(1, 5_000_000)

out = glft_market_maker(hbt, recorder.recorder)

hbt.close()

stats = LinearAssetRecord(recorder.get(0)).stats(book_size=30_000)
stats.summary()
```

[16]:
shape: (1, 11)

start	end	SR	Sortino	Return	MaxDrawdown	DailyNumberOfTrades	Da
datetime[μs]	datetime[μs]	f64	f64	f64	f64	f64	
2022-10-03 00:00:00	2022-10-03 23:59:50	-246.379582	-264.130529	-0.020574	0.020601	13579.57171	5

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

[18]:
```
stats.plot()
```

Adjustment factors

It looks like the skew is too strong, which is why the market maker is hesitant to take on the position. To alleviate the skew, you can introduce adjustment factors, adj_1 and adj_2, to the calculated half spread and skew, as follow.

$$\text{half spread}_{adj} = \text{half spread} \times adj_1$$
$$\text{skew}_{adj} = \text{skew} \times adj_2$$

[19]:
```
from numba.typed import Dict

@njit
def glft_market_maker(hbt, recorder):
    tick_size = hbt.depth(0).tick_size
    arrival_depth = np.full(10_000_000, np.nan, np.float64)
    mid_price_chg = np.full(10_000_000, np.nan, np.float64)
    out = np.zeros(10_000_000, out_dtype)

    t = 0
    prev_mid_price_tick = np.nan
    mid_price_tick = np.nan

    tmp = np.zeros(500, np.float64)
    ticks = np.arange(len(tmp)) + 0.5

    A = np.nan
    k = np.nan
    volatility = np.nan
    gamma = 0.05
    delta = 1
    adj1 = 1
    adj2 = 0.05 # Uses the same value as gamma.

    order_qty = 1
    max_position = 20

    # Checks every 100 milliseconds.
    while hbt.elapse(100_000_000) == 0:
        #---------------------------------------------------------
        # Records market order's arrival depth from the mid-price.
        if not np.isnan(mid_price_tick):
            depth = -np.inf
            for last_trade in hbt.last_trades(0):
                trade_price_tick = last_trade.px / tick_size

                if last_trade.ev & BUY_EVENT == BUY_EVENT:
                    depth = np.nanmax([trade_price_tick - mid_price_tick, depth])
                else:
                    depth = np.nanmax([mid_price_tick - trade_price_tick, depth])
            arrival_depth[t] = depth

        hbt.clear_last_trades(0)
        hbt.clear_inactive_orders(0)

        depth = hbt.depth(0)
        position = hbt.position(0)
        orders = hbt.orders(0)

        best_bid_tick = depth.best_bid_tick
        best_ask_tick = depth.best_ask_tick

        prev_mid_price_tick = mid_price_tick
        mid_price_tick = (best_bid_tick + best_ask_tick) / 2.0

        # Records the mid-price change for volatility calculation.
        mid_price_chg[t] = mid_price_tick - prev_mid_price_tick

        #---------------------------------------------------------
        # Calibrates A, k and calculates the market volatility.
```

```
        # Updates A, k, and the volatility every 5-sec.
        if t % 50 == 0:
            # Window size is 10-minute.
            if t >= 6_000 - 1:
                # Calibrates A, k
                tmp[:] = 0
                lambda_ = measure_trading_intensity(arrival_depth[t + 1 - 6_000:t + 1], tmp)
                if len(lambda_) > 2:
                    lambda_ = lambda_[:70] / 600
                    x = ticks[:len(lambda_)]
                    y = np.log(lambda_)
                    k_, logA = linear_regression(x, y)
                    A = np.exp(logA)
                    k = -k_

                # Updates the volatility.
                volatility = np.nanstd(mid_price_chg[t + 1 - 6_000:t + 1]) * np.sqrt(10)

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

        c1, c2 = compute_coeff(gamma, gamma, delta, A, k)

        half_spread_tick = (c1 + delta / 2 * c2 * volatility) * adj1
        skew = c2 * volatility * adj2

        reservation_price_tick = mid_price_tick - skew * position

        bid_price_tick = np.minimum(np.round(reservation_price_tick - half_spread_tick),
best_bid_tick)
        ask_price_tick = np.maximum(np.round(reservation_price_tick + half_spread_tick),
best_ask_tick)

        bid_price = bid_price_tick * tick_size
        ask_price = ask_price_tick * tick_size

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

        # Cancel orders if they differ from the updated bid and ask prices.
        order_values = orders.values();
        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.price_tick != bid_price_tick)
                    or (order.side == SELL and order.price_tick != ask_price_tick)
                ):
                    hbt.cancel(0, order.order_id, False)

        # If the current position is within the maximum position,
        # submit the new order only if no order exists at the same price.
        if position < max_position and np.isfinite(bid_price):
            bid_price_as_order_id = round(bid_price / tick_size)
            if bid_price_as_order_id not in orders:
                hbt.submit_buy_order(0, bid_price_as_order_id, bid_price, orde
LIMIT, False)
        if position > -max_position and np.isfinite(ask_price):
```

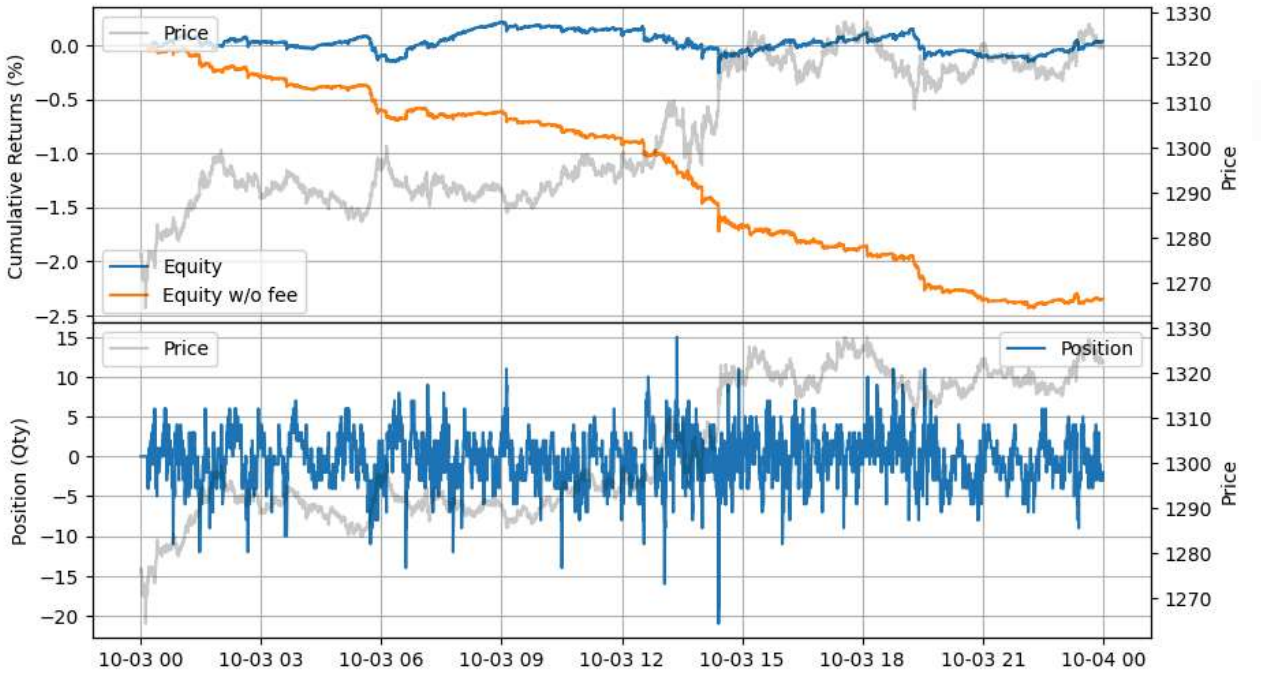
```
            ask_price_as_order_id = round(ask_price / tick_size)
            if ask_price_as_order_id not in orders:
                hbt.submit_sell_order(0, ask_price_as_order_id, ask_price, order_qty, GTX,
LIMIT, False)

        #--------------------------------------------------------
        # Records variables and stats for analysis.

        out[t].half_spread_tick = half_spread_tick
        out[t].skew = skew
        out[t].volatility = volatility
        out[t].A = A
        out[t].k = k

        t += 1

        if t >= len(arrival_depth) or t >= len(mid_price_chg) or t >= len(out):
            raise Exception

        # Records the current state for stat calculation.
        recorder.record(hbt)
    return out[:t]
```

[20]:
```
asset = (
    BacktestAsset()
        .data([
            'data/ethusdt_20221003.npz'
        ])
        .initial_snapshot('data/ethusdt_20221002_eod.npz')
        .linear_asset(1.0)
        .intp_order_latency([
            'latency/feed_latency_20221003.npz'
        ])
        .power_prob_queue_model(2.0)
        .no_partial_fill_exchange()
        .trading_value_fee_model(-0.00005, 0.0007)
        .tick_size(0.01)
        .lot_size(0.001)
        .roi_lb(0.0)
        .roi_ub(3000.0)
        .last_trades_capacity(10000)
)

hbt = ROIVectorMarketDepthBacktest([asset])

recorder = Recorder(1, 5_000_000)

out = glft_market_maker(hbt, recorder.recorder)

hbt.close()

stats = LinearAssetRecord(recorder.get(0)).stats(book_size=30_000)
stats.summary()
```

[20]: shape: (1, 11)

start	end	SR	Sortino	Return	MaxDrawdown	DailyNumberOfTrades	DailyTurnc
datetime[μs]	datetime[μs]	f64	f64	f64	f64	f64	
2022-10-03 00:00:00	2022-10-03 23:59:50	1.202048	1.471295	0.000359	0.004763	10987.271675	477.498

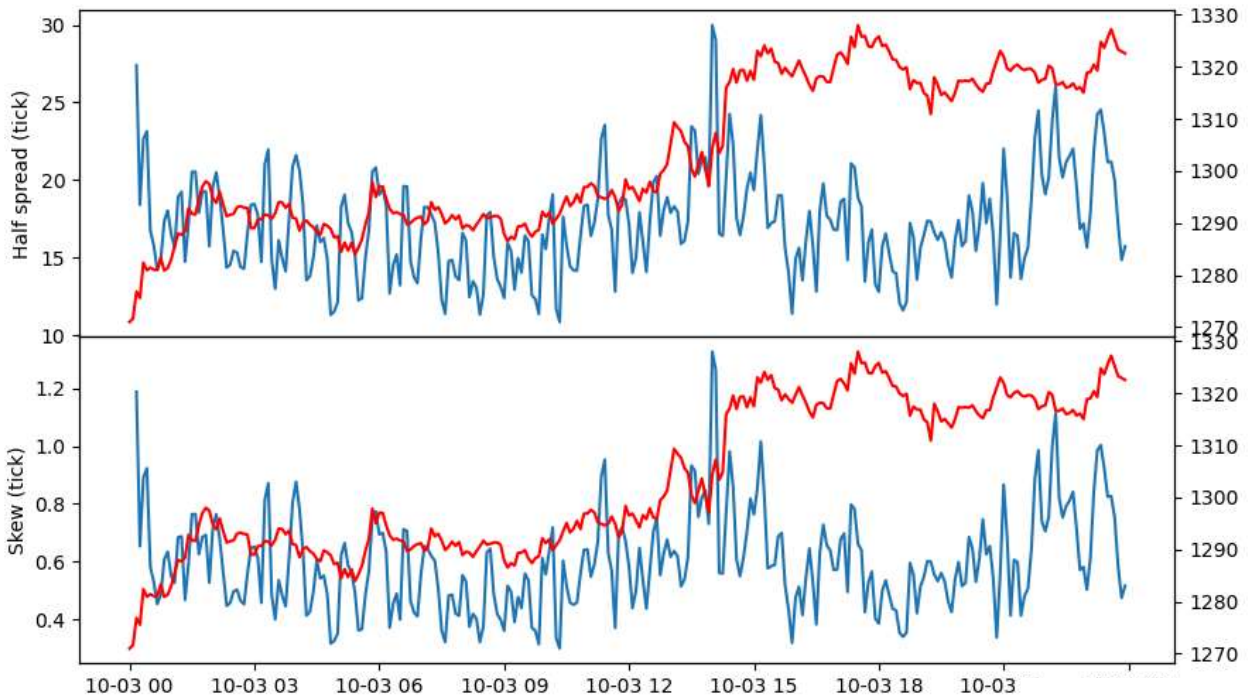
[21]:
```
stats.plot()
```



Improved, but even when accounting for rebates, it can only achieve breakeven at best. As shown below, both the half spread and skew move together, primarily influenced by the c_2 and the market volatility.

[22]:
```
import polars as pl

records = recorder.get(0)
df = pl.DataFrame(out).with_columns(
    pl.Series('timestamp', records['timestamp']),
    pl.Series('price', records['price'])
).with_columns(
    pl.from_epoch('timestamp', time_unit='ns')
)

df = df.group_by_dynamic(
    'timestamp', every='5m'
).agg(
    pl.col('price').last(),
    pl.col('half_spread_tick').last(),
    pl.col('skew').last(),
    pl.col('volatility').last(),
    pl.col('A').last(),
    pl.col('k').last(),
)

fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
fig.subplots_adjust(hspace=0)
fig.set_size_inches(10, 6)

ax1.plot(df['timestamp'], df['half_spread_tick'])
ax1.twinx().plot(df['timestamp'], df['price'], 'r')
ax1.set_ylabel('Half spread (tick)')

ax2.plot(df['timestamp'], df['skew'])
ax2.twinx().plot(df['timestamp'], df['price'], 'r')
ax2.set_ylabel('Skew (tick)')
```

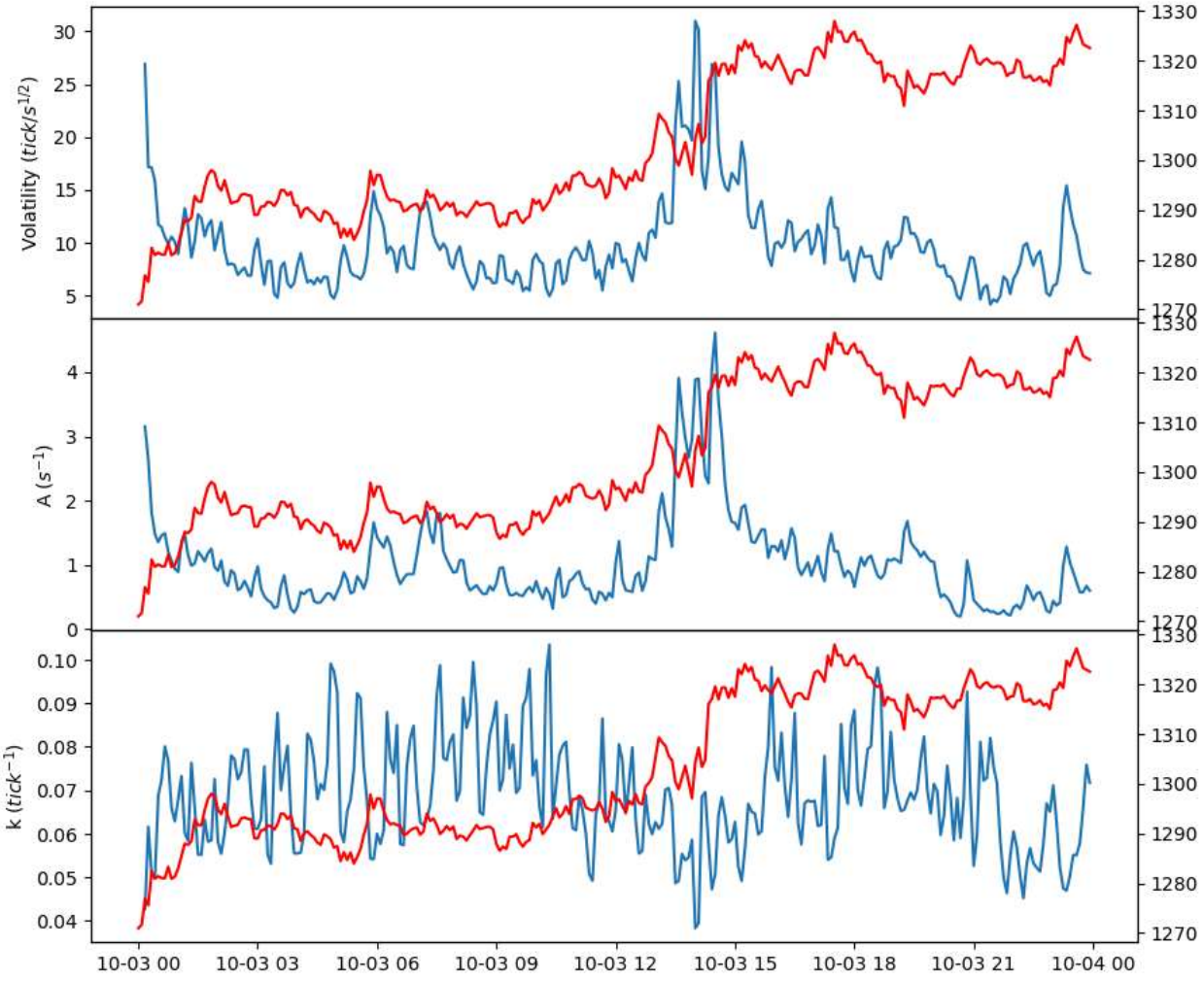
[22]:
```
Text(0, 0.5, 'Skew (tick)')
```



latest

[23]:
```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True)
fig.subplots_adjust(hspace=0)
fig.set_size_inches(10, 9)

ax1.plot(df['timestamp'], df['volatility'])
ax1.twinx().plot(df['timestamp'], df['price'], 'r')
ax1.set_ylabel('Volatility ($ tick/s^{1/2} $)')

ax2.plot(df['timestamp'], df['A'])
ax2.twinx().plot(df['timestamp'], df['price'], 'r')
ax2.set_ylabel('A ($ s^{-1} $)')

ax3.plot(df['timestamp'], df['k'])
ax3.twinx().plot(df['timestamp'], df['price'], 'r')
ax3.set_ylabel('k ($ tick^{-1} $)')
```

[23]:
```
Text(0, 0.5, 'k ($ tick^{-1} $)')
```



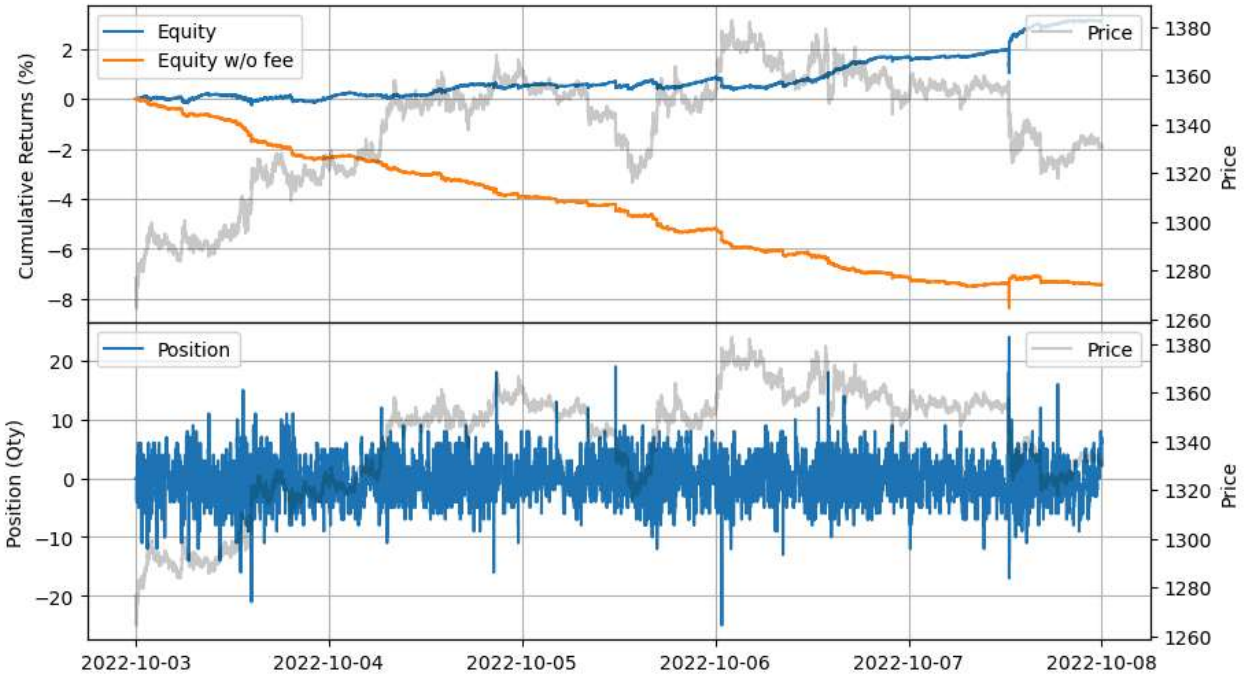
In the 5-day backtest, it's evident that profits are generated through rebates, as a result of maintaining high trading volume by consistently posting quotes.

[24]:
```
asset = (
    BacktestAsset()
        .data([
            'data/ethusdt_20221003.npz',
            'data/ethusdt_20221004.npz',
            'data/ethusdt_20221005.npz',
            'data/ethusdt_20221006.npz',
            'data/ethusdt_20221007.npz'
        ])
        .initial_snapshot('data/ethusdt_20221002_eod.npz')
        .linear_asset(1.0)
        .intp_order_latency([
            'latency/feed_latency_20221003.npz',
            'latency/feed_latency_20221004.npz',
            'latency/feed_latency_20221005.npz',
            'latency/feed_latency_20221006.npz',
            'latency/feed_latency_20221007.npz'
        ])
        .power_prob_queue_model(2.0)
        .no_partial_fill_exchange()
        .trading_value_fee_model(-0.00005, 0.0007)
        .tick_size(0.01)
        .lot_size(0.001)
        .roi_lb(0.0)
        .roi_ub(3000.0)
        .last_trades_capacity(10000)
)

hbt = ROIVectorMarketDepthBacktest([asset])

recorder = Recorder(1, 5_000_000)

out = glft_market_maker(hbt, recorder.recorder)

hbt.close()

stats = LinearAssetRecord(recorder.get(0)).stats(book_size=30_000)
stats.summary()
```

[24]:
shape: (1, 11)

start	end	SR	Sortino	Return	MaxDrawdown	DailyNumberOfTrades	DailyTu
datetime[μs]	datetime[μs]	f64	f64	f64	f64	f64	
2022-10-03 00:00:00	2022-10-07 23:59:50	16.282366	20.682178	0.031145	0.009818	9463.81907	422.4

[25]:
```
stats.plot()
```



Integrating Grid Trading

Creating a grid from the bid and ask prices derived from the Guéant–Lehalle–Fernandez-Tapia market making model.

[26]:
```
from numba.typed import Dict
from numba import uint64

@njit
def gridtrading_glft_mm(hbt, recorder):
    asset_no = 0
    tick_size = hbt.depth(asset_no).tick_size

    arrival_depth = np.full(10_000_000, np.nan, np.float64)
    mid_price_chg = np.full(10_000_000, np.nan, np.float64)

    t = 0
    prev_mid_price_tick = np.nan
    mid_price_tick = np.nan

    tmp = np.zeros(500, np.float64)
    ticks = np.arange(len(tmp)) + 0.5

    A = np.nan
    k = np.nan
    volatility = np.nan
    gamma = 0.05
    delta = 1
    adj1 = 1
    adj2 = 0.05

    order_qty = 1
    max_position = 20
    grid_num = 20

    # Checks every 100 milliseconds.
    while hbt.elapse(100_000_000) == 0:
        #---------------------------------------------------------
        # Records market order's arrival depth from the mid-price.
        if not np.isnan(mid_price_tick):
            depth = -np.inf
            for last_trade in hbt.last_trades(asset_no):
                trade_price_tick = last_trade.px / tick_size

                if last_trade.ev & BUY_EVENT == BUY_EVENT:
                    depth = np.nanmax([trade_price_tick - mid_price_tick, depth])
                else:
                    depth = np.nanmax([mid_price_tick - trade_price_tick, depth])
            arrival_depth[t] = depth

        hbt.clear_last_trades(asset_no)
        hbt.clear_inactive_orders(asset_no)

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

        best_bid_tick = depth.best_bid_tick
        best_ask_tick = depth.best_ask_tick

        prev_mid_price_tick = mid_price_tick
        mid_price_tick = (best_bid_tick + best_ask_tick) / 2.0

        # Records the mid-price change for volatility calculation.
        mid_price_chg[t] = mid_price_tick - prev_mid_price_tick
```

```
        #---------------------------------------------------------
        # Calibrates A, k and calculates the market volatility.

        # Updates A, k, and the volatility every 5-sec.
        if t % 50 == 0:
            # Window size is 10-minute.
            if t >= 6_000 - 1:
                # Calibrates A, k
                tmp[:] = 0
                lambda_ = measure_trading_intensity(arrival_depth[t + 1 - 6_000:t + 1], tmp)
                if len(lambda_) > 2:
                    lambda_ = lambda_[:70] / 600
                    x = ticks[:len(lambda_)]
                    y = np.log(lambda_)
                    k_, logA = linear_regression(x, y)
                    A = np.exp(logA)
                    k = -k_

                # Updates the volatility.
                volatility = np.nanstd(mid_price_chg[t + 1 - 6_000:t + 1]) * np.sqrt(10)

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

        c1, c2 = compute_coeff(gamma, gamma, delta, A, k)

        half_spread_tick = (c1 + delta / 2 * c2 * volatility) * adj1
        skew = c2 * volatility * adj2

        reservation_price_tick = mid_price_tick - skew * position

        bid_price_tick = np.minimum(np.round(reservation_price_tick - half_spread_tick),
best_bid_tick)
        ask_price_tick = np.maximum(np.round(reservation_price_tick + half_spread_tick),
best_ask_tick)

        bid_price = bid_price_tick * tick_size
        ask_price = ask_price_tick * tick_size

        grid_interval = max(np.round(half_spread_tick) * tick_size, tick_size)

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

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

        # Creates a new grid for buy orders.
        new_bid_orders = Dict.empty(np.uint64, np.float64)
        if position < max_position 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.
```

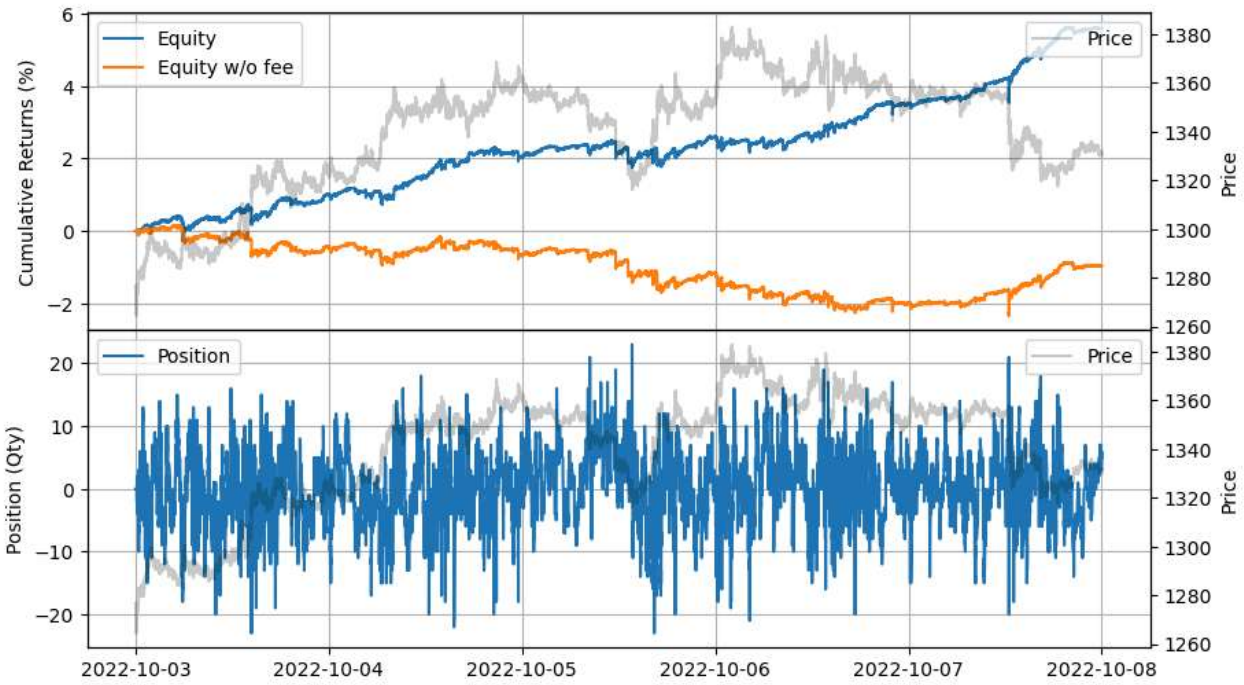
```
        new_ask_orders = Dict.empty(np.uint64, np.float64)
        if position > -max_position and np.isfinite(ask_price):
            for i in range(grid_num):
                ask_price_tick = round(ask_price / tick_size)

                # 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();
        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)

        #------------------------------------------------------
        # Records variables and stats for analysis.

        t += 1

        if t >= len(arrival_depth) or t >= len(mid_price_chg):
            raise Exception

        # Records the current state for stat calculation.
        recorder.record(hbt)
    return out[:t]
```

[27]:
```
asset = (
    BacktestAsset()
        .data([
            'data/ethusdt_20221003.npz',
            'data/ethusdt_20221004.npz',
            'data/ethusdt_20221005.npz',
            'data/ethusdt_20221006.npz',
            'data/ethusdt_20221007.npz'
        ])
        .initial_snapshot('data/ethusdt_20221002_eod.npz')
        .linear_asset(1.0)
        .intp_order_latency([
            'latency/feed_latency_20221003.npz',
            'latency/feed_latency_20221004.npz',
            'latency/feed_latency_20221005.npz',
            'latency/feed_latency_20221006.npz',
            'latency/feed_latency_20221007.npz'
        ])
        .power_prob_queue_model(2.0)
        .no_partial_fill_exchange()
        .trading_value_fee_model(-0.00005, 0.0007)
        .tick_size(0.01)
        .lot_size(0.001)
        .roi_lb(0.0)
        .roi_ub(3000.0)
        .last_trades_capacity(10000)
)

hbt = ROIVectorMarketDepthBacktest([asset])

recorder = Recorder(1, 5_000_000)

out = gridtrading_glft_mm(hbt, recorder.recorder)

hbt.close()

stats = LinearAssetRecord(recorder.get(0)).stats(book_size=30_000)
stats.summary()
```

[27]:
shape: (1, 11)

start	end	SR	Sortino	Return	MaxDrawdown	DailyNumberOfTrades	DailyTu
datetime[μs]	datetime[μs]	f64	f64	f64	f64	f64	
2022-10-03 00:00:00	2022-10-07 23:59:50	19.774661	24.630456	0.055856	0.007438	5878.736082	262.5

[28]:
```
stats.plot()
```



You can see it works even better with other coins as well. In the next example, we will show how to create multiple markets to achieve better risk-adjusted returns.

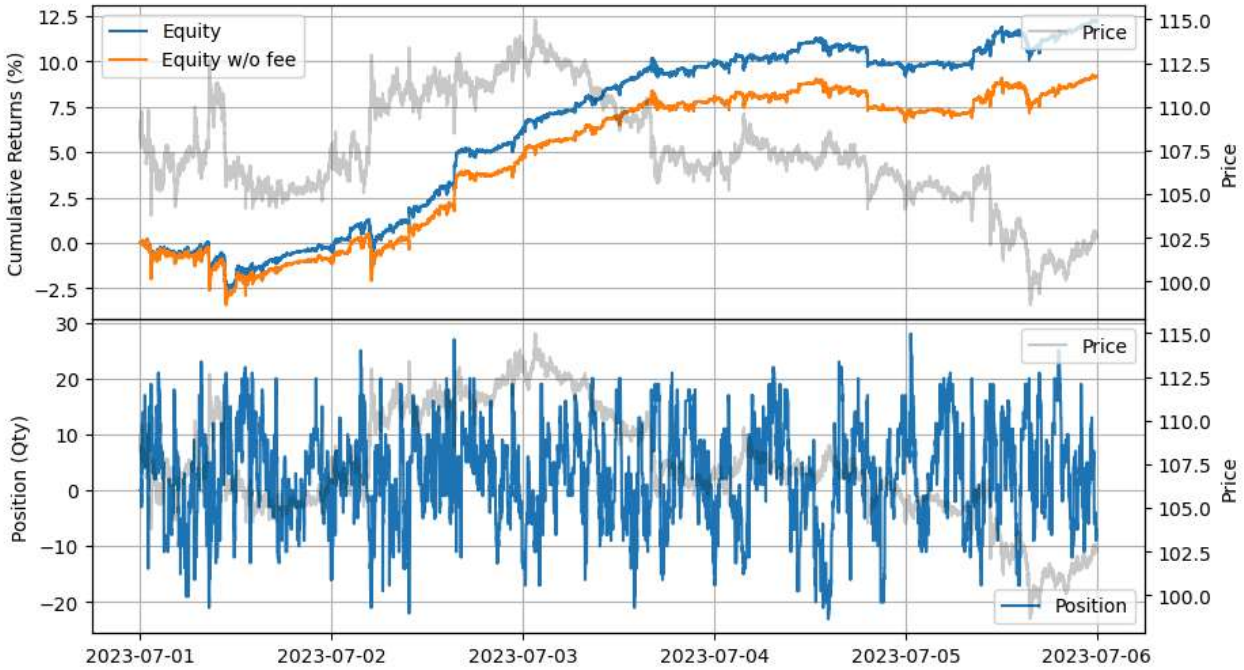
[29]:
```
asset = (
    BacktestAsset()
        .data([
            'data/ltcusdt_20230701.npz',
            'data/ltcusdt_20230702.npz',
            'data/ltcusdt_20230703.npz',
            'data/ltcusdt_20230704.npz',
            'data/ltcusdt_20230705.npz'
        ])
        .initial_snapshot('data/ltcusdt_20230630_eod.npz')
        .linear_asset(1.0)
        .intp_order_latency([
            'latency/feed_latency_20230701.npz',
            'latency/feed_latency_20230702.npz',
            'latency/feed_latency_20230703.npz',
            'latency/feed_latency_20230704.npz',
            'latency/feed_latency_20230705.npz'
        ])
        .power_prob_queue_model(2.0)
        .no_partial_fill_exchange()
        .trading_value_fee_model(-0.00005, 0.0007)
        .tick_size(0.01)
        .lot_size(0.001)
        .roi_lb(0.0)
        .roi_ub(300.0)
        .last_trades_capacity(10000)
)

hbt = ROIVectorMarketDepthBacktest([asset])

recorder = Recorder(1, 5_000_000)

out = gridtrading_glft_mm(hbt, recorder.recorder)

hbt.close()

stats = LinearAssetRecord(recorder.get(0)).stats(book_size=3000)
stats.summary()
```

[29]:

shape: (1, 11)

start	end	SR	Sortino	Return	MaxDrawdown	DailyNumberOfTrades	DailyTurn
datetime[μs]	datetime[μs]	f64	f64	f64	f64	f64	
2023-07-01 00:00:00	2023-07-05 23:59:50	17.17992	23.062973	0.122535	0.032973	3425.879303	122.80

[30]:
```
stats.plot()
```



Wrapping up

Thus far, we have illustrated how to apply the model to a real-world example.

For a more effective market-making algorithm, consider dividing this model into the following categories:

- Half-spread: As shown, the half-spread is a function of trading intensity and market volatility. An exponential function used for trading intensity might not be suitable for the entire range. You could develop a more refined approach to convert trading intensity to half-spread. Additionally, while historical trading intensity and market volatility are utilized here, you could forecast short-term trading intensity and volatility to respond more agilely to changes in market conditions. This might involve strategies that use news, events, liquidity vacuums, and other factors to predict volatility explosions.
- Skew: The skew is also a function of trading intensity and market volatility. In this model, only inventory risk is considered, but you can also account for other risks, particularly when making multiple markets. BARRA is a good example of other risks that can be managed similarly.
- Fair Value Pricing: In this model, the fair price is equal to the mid-price, however, you need to incorporate forecasts such as the micro-price and fair value pricing through correlated assets to enhance the strategy.
- Hedging: Hedging is especially crucial when making multiple markets, as it serves as a valuable tool for managing risks.

We will address a few more topics in upcoming examples.

References

Dealing with the Inventory Risk - A solution to the market making problem
Optimal market making

Knight Capital Group
Stochastic Control Theory and High Frequency Trading

BitMEX Market Making Series
Algo Trading & Market Making
How to Market Make Bitcoin Derivatives Lesson 1
How to Market Make Bitcoin Derivatives Lesson 2

[ ]: