Queue Reactive for Fast Markets

Table of contents

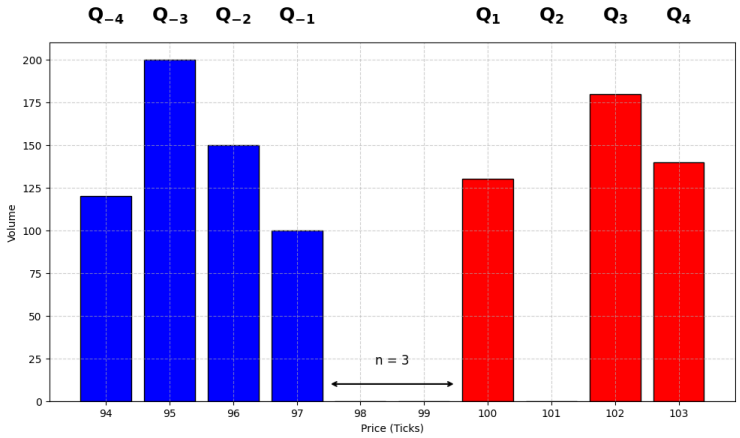
Queue Reactive Model

Model defintion

At any time t, the order book $\text{LOB}(t)$ is projected onto a state $f(\text{LOB}) = (\text{Imb}, n)$ where:

$$\text{Imb} = \frac{Q^B - Q^A}{Q^B + Q^A}$$

and n is the bid ask spread in ticks. The order book is comprised of different levels 1 and -1 are respectively the best ask and bid regardless of the spread between the two, and the levels $\pm l$ for $l \in \{2, ...\}$ are $l-1$ ticks away from the best ask/bid.



The possible events are L : limits orders, C : cancelations, T : market orders and $+$: add in spread orders. Each event e is a combination of an event type, a side B:bid/A:ask and a level (or an available queue for add in spread).

At time t when the current state of the order book is (Imb, n) the next event is distributed acording to :

$$\Delta t = \min_{e \in \mathcal{E}} \Delta t^e \ , \ e^* = \text{argmin}_{e \in \mathcal{E}} \Delta t^e \text{ where } \Delta t^e \sim \text{Exp}\left(\lambda^e(\text{Imb}, n)\right)$$

Where $\mathcal{E}$ is the set of all possible events. Or alternatively if we consider:

$$\Lambda(\text{Imb}, n) = \sum_{e \in \mathcal{E}} \lambda^e(\text{Imb}, n) \text{ and } q^e = \frac{\lambda^e(\text{Imb}, n)}{\Lambda(\text{Imb}, n)}$$

$$\text{Then } \Delta t \sim \text{Exp}(\Lambda(\text{Imb}, n)) \text{ and } e^* \sim \sum_{e \in \mathcal{E}} q^e \delta_e$$

Estimation

We define t_k as the k-th timestamp, and: $\Delta t_k = t_k - t_{k-1}$. Let $\mathcal{T} = \{(t_k, e_k, \text{Imb}_k, n_k)\}_{k \in \mathbb{N}}$ be our dataset.

Let $\mathcal{K}^e(\text{Imb}, n)$ be the set of indices k such that the event e occurred at time t_k and the previous state was (Imb, n):

$$\mathcal{K}^e(\text{Imb}, n) = \{k : e_k = e, f(\text{LOB}(t_{k-1})) = (\text{Imb}, n)\} \text{ and } \mathcal{K}(\text{Imb}, n) = \bigcup_{e \in \mathcal{E}} \mathcal{K}^e(\text{Imb}, n)$$

$$\Delta t_k \mid k \in \mathcal{K}(\text{Imb}, n) \sim \text{Exp}(\Lambda(\text{Imb}, n)) \implies \Lambda(\text{Imb}, n) = \left(\mathbb{E}[\Delta t_k \mid k \in \mathcal{K}(\text{Imb}, n)]\right)^{-1}$$

$$\hat{\Lambda}(\text{Imb}, n) = \left(\frac{1}{\#\mathcal{K}(\text{Imb}, n)} \sum_{e \in \mathcal{E}} \sum_{k \in \mathcal{K}^e(\text{Imb}, n)} \Delta t_k^e \right)^{-1} \text{ and } \hat{q}^e = \frac{\#\mathcal{K}^e(\text{Imb}, n)}{\#\mathcal{K}(\text{Imb}, n)}$$

Practical Considerations

The imbalance is seperated into 10 bins of equal width, we identify each bin by it's left value.

Since we mainly care about large tick stocks and to not have to estimate many statistics, I chose to only consider up to level 2 i.e: the best bid/ask and a tick beyond that, and only for a spread of 1 tick ($n = 1$). If the spread is above 1 tick the only possible event is an add in spread.

We also added random volumes for the different events, the volumes are bucketed according to the median event size per level.

Add in Spread

When the spread is equal to 2 ticks, there is only one available queue for an add in spread, this makes the estimation easy since we only have to estimate the side of the add in spread per Imbalance bucket. Since for the bulk of historical data for large tick stocks the spread is equal to 1 and 2 I chose to only estimate the add in spread probabilities for spread equal to 2 ticks. When sampling, if the spread ever goes beyond that we draw a side for the add in spread according to the statistics for spread equal to 2 and draw a queue at random from the available or empty queues, this rough approximation seems reasonable to me since it doesn't happen all that often anyways.

Random Volumes

As mentionned before, volumes are bucketed according to median event size per level i.e: we take the smallest integer greater or equal to the volumes divided by median event size.

As we said before an event is basically a tuple of event type, event side, and event level (1 or 2 in this case) (level being somewhat irrelevant for add in spread since it is chosen at random). When estimating a volume distribution, for limit/cancel/market orders for each Imbalance bin, event type, event side and event level we estimate the empirical distribution of volumes that we sample from at each draw.

For add in spread events the volume distribution is a bit different. It is first of all not conditionned on Imbalance, and when estimating the empirical distribution we don't just consider the volume of the limit order that constitues the add in spread but the cumulative volume of all orders in the level that was added as long as there are no events in other queues. e.g:

event	side	price	size	Q_-2	Q_-1	P_-2	P_-1	P_1	P_2	Q_1	Q_2
Create_Bid	B	100	80	150	200	98	99	101	102	120	100
Add	B	100	60	200	80	99	100	101	102	120	100
Can	B	100	50	200	140	99	100	101	102	120	100
Add	B	100	100	200	90	99	100	101	102	120	100
Can	A	101	60	200	190	99	100	101	102	120	100
Add	A	102	80	200	190	99	100	101	102	60	100

In this example the add in spread event or more specifically `Create_Bid` in this case has an effective volume so to say of 190 and that's the volume that is considered when computing the empirical distribution of event sizes for add in spread.

When sampling I chose to keep track of 4 price levels, when the mid price shifts and we discover new levels, their volumes are drawn from an empirical distribution also.

Also In my data I have a `Trade_All` event that I kind of dropped after introducing random volumes.

Data

I work with databento MBP10 data (L2). With a seperate dataframe for each stock/day, I first converted this data to a "queue reactive" friendly format that looks something like this :

ts_event	event	side	size	price	Q_-2	Q_-1	P_-2	P_-1	P_1	P_2	Q_1	Q_2
10:00:00.083	Add	A	800	1682	1300	2108	1679	1680	1681	1682	512	880
10:00:01.837	Can	A	109	1681	1300	2108	1679	1680	1681	1682	512	1680
10:00:01.837	Add	A	400	1681	1300	2108	1679	1680	1681	1682	403	1680
10:00:01.837	Add	A	200	1681	1300	2108	1679	1680	1681	1682	803	1680

```
| 10:00:01.837    | Can    | A    |  300 |  1681 | 1300 | 2108 | 1679 | 1680 | 1681 | 1682 |1003 | 1680 |
| 10:00:02.081    | Can    | A    |  200 |  1681 | 1300 | 2108 | 1679 | 1680 | 1681 | 1682 | 703 | 1680 |
| 10:00:02.088    | Add    | A    |  100 |  1681 | 1300 | 2108 | 1679 | 1680 | 1681 | 1682 | 503 | 1680 |
| 10:00:02.088    | Add    | A    |  100 |  1681 | 1300 | 2108 | 1679 | 1680 | 1681 | 1682 | 603 | 1680 |
| 10:00:02.088    | Add    | A    |  200 |  1681 | 1300 | 2108 | 1679 | 1680 | 1681 | 1682 | 703 | 1680 |
| 10:00:02.113    | Can    | A    |  200 |  1681 | 1300 | 2108 | 1679 | 1680 | 1681 | 1682 | 903 | 1680 |
```

The timestamps are nanonesecond precision but I cut them here for visibility. I also chose to cut out 30 minutes from the beginning and end of each trading day to avoid the openning and the close.

From each of these dataframes I extract some summary statistics that when aggregated allow me to estimate the intensities using any combination of dates I want.

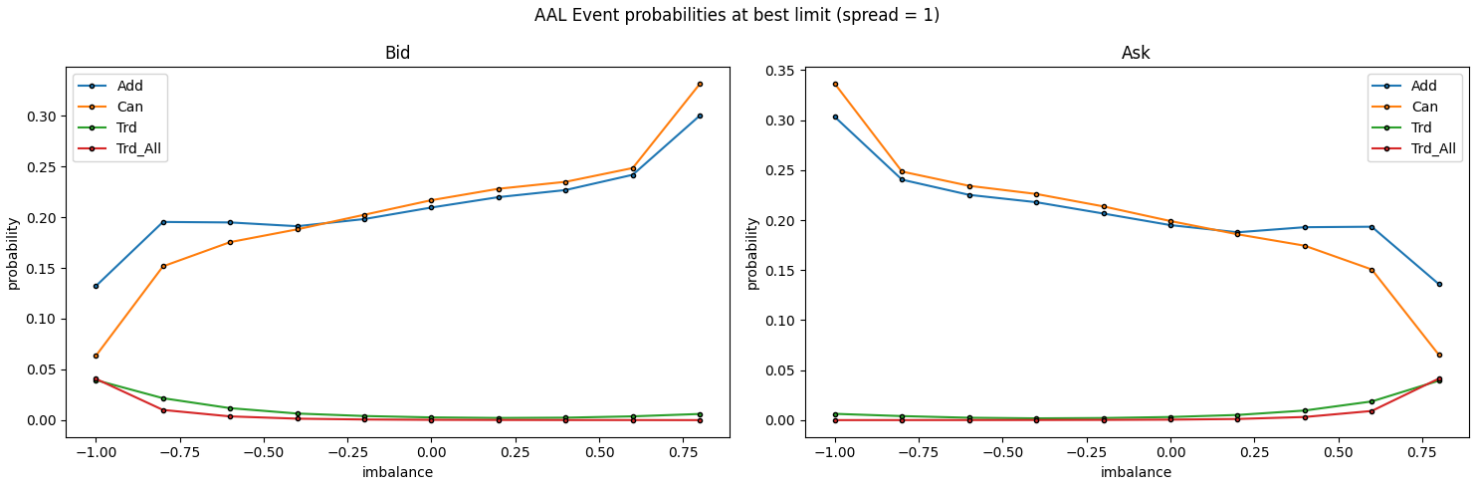
```
$ ctree /home2/lobib/QR/v1/estimations/ | head
/home2/lobib/QR/v1/estimations/
|-- CXW
|   |-- xnas-itch
|   |   |-- 2024-10-08.csv
|   |   |-- 2024-10-30.csv
|   |   |-- 2024-10-15.csv
|   |   |-- 2024-08-30.csv
|   |   |-- 2024-08-22.csv
|   |   |-- 2024-08-01.csv
|   |   |-- 2024-10-03.csv
```

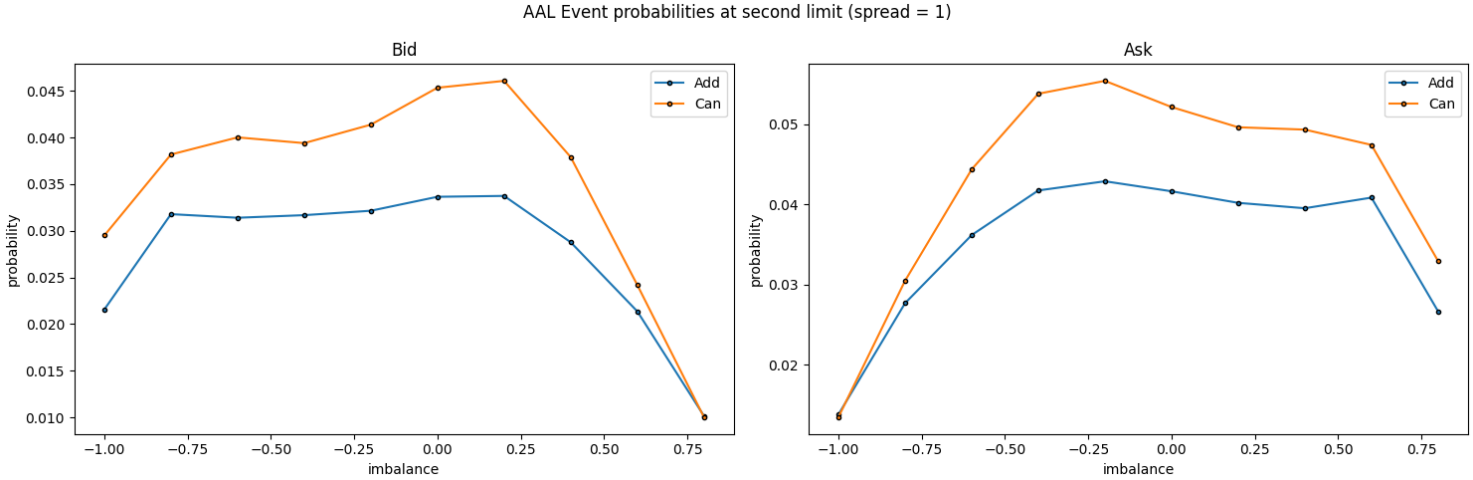
```
$ csvlook /home2/lobib/QR/v1/estimations/CXW/xnas-itch/2024-10-08.csv | head
| imb_left | spread | event  | side | q_nbr | time     | size | ts_delta | count | date       |
|----------|--------|--------|------|-------|----------|------|----------|-------|------------|
| -0.20    | 1      | Add    | B    | -1    | 15:00:00| 839  | 3.28e9   | 9     | 2024-10-08 |
| -1.00    | 2      | Can    | A    | 1     | 12:00:00| 598  | 5.57e5   | 5     | 2024-10-08 |
| -0.20    | 3      | Add    | A    | 2     | 14:00:00| 100  | 2.20e7   | 1     | 2024-10-08 |
| 0.00     | 2      | Add    | B    | -2    | 15:00:00| 100  | 8.44e7   | 1     | 2024-10-08 |
| 0.60     | 1      | Can    | B    | -1    | 14:30:00| 600  | 3.56e9   | 5     | 2024-10-08 |
| 0.40     | 2      | Add    | B    | -1    | 12:00:00| 400  | 1.45e10  | 4     | 2024-10-08 |
| 0.60     | 3      | Can    | B    | -3    | 14:30:00| 100  | 1.04e9   | 1     | 2024-10-08 |
| -0.40    | 2      | Create | A    | 0     | 10:30:00| 100  | 7.63e5   | 1     | 2024-10-08 |
```

I added a time component here to assess the parameters stability w.r.t time of day.

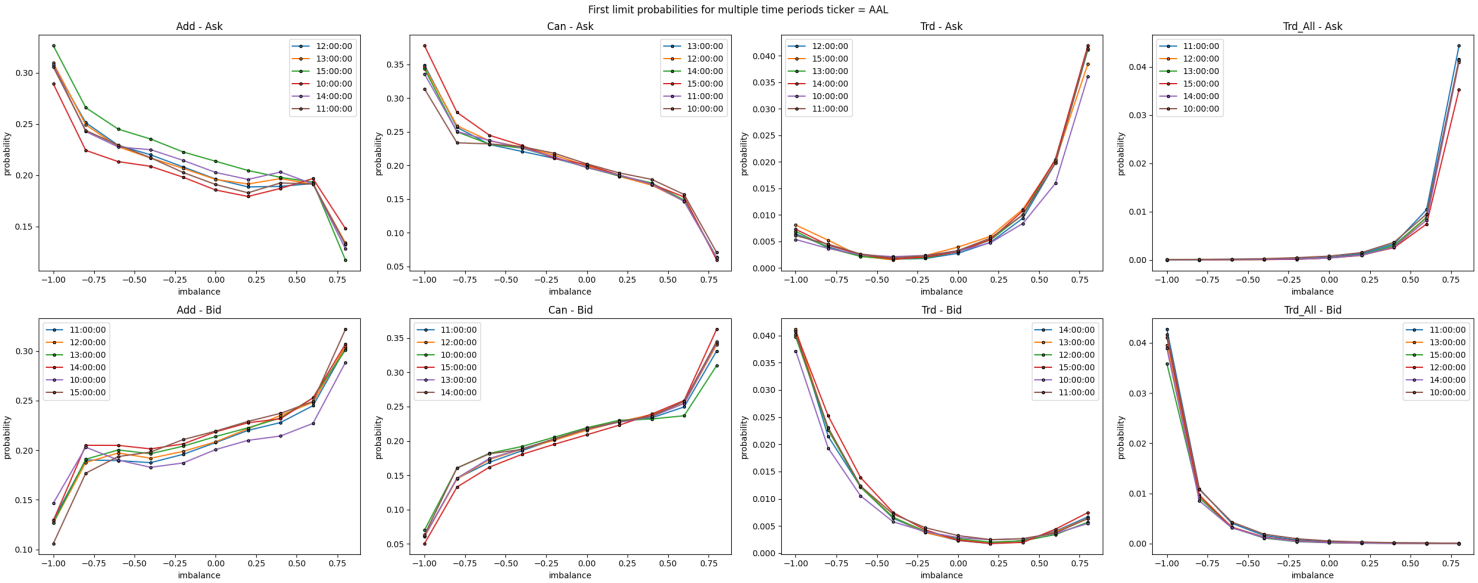
- `imb_left`: is the left value of the imbalance bin.
- `event`: takes the following values `{"Add", "Can", "Trd", "Create_Bid", "Create_Ask"}`.
- `time`: is the time of day in 30 minute bins, I added that in case we wanted to make estimations per time of day.
- `q_nbr`: is counted from the mid price, itself being queue number 0 in case it is a valid price.
- `size`: is the sum of the sizes of all the events that share the same other parameters. Useful for computing average event size later.
- `ts_delta`: is the sum of Δt_k for all events that share the same other parameters.

Here is an example of event probabilities for the stock `AAL`:





We can even plot them by time of day as mentionned :



For the volumes I still parse the entire "queue reactive" data, I should probably compute invariant distributions per day and aggregate them like with intensities.

Implementation

In my implementation, I chose to encode all the above logic into a `QRModel` object, that given the current state of the order book $\text{LOB}(t)$ one can sample the next event and Δt.

```python
# src/aqr.py
class QRModel(ABC):
    @abstractmethod
    def sample_deltat(self, lob: LimitOrderBook) -> float:
        raise NotImplementedError

    @abstractmethod
    def sample_order(self, lob: LimitOrderBook, alpha: float) -> Order:
        raise NotImplementedError
```

My implementation of the imbalance/spread is as follows :

```python
# src/aqr.py
class ImbalanceQR(QRModel):
    def __init__(
        self,
        deltat_distrib: dict[tuple[int, int], ExpDistribution],
        probabilities: pl.DataFrame,
        event_distrib: dict[tuple[int, int], DiscreteDistribution],
        order_volumes_distrib: dict[tuple[str, int], dict[tuple[int, int], DiscreteDistribution]],
        imbalance_bins: np.ndarray,
        rng: np.random.Generator = np.random.default_rng(1337),
    ) -> None:
        self.deltat_distrib = deltat_distrib
        self.probabilities = probabilities
        self.event_distrib = event_distrib
        self.imbalance_bins = imbalance_bins
        self.order_volumes_distrib = order_volumes_distrib
        self.rng = rng

    def map_imbalance_bin(self, imbalance: float) -> float:
        ...

    def sample_deltat(self, lob: LimitOrderBook) -> int:
        ...

    def _adjust_probabilities(self, eps: float, exclude_cancels: bool = True) -> pl.DataFrame:
        ...

    def adjust_event_distrib(self, eps: float, exclude_cancels: bool = True) -> None:
        ...

    def sample_order(self, lob: LimitOrderBook, alpha: Alpha)-> Order:
        ...

    def _create_order(self, order: tuple[str, str, int], lob: LimitOrderBook, alpha: float) -> Order:
        ...
```

- `imbalance_bins` : is just an array of the left values of each imbalance bin and is used to map a float in $[-1, 1]$ to the corresponding bin.
- `deltat_distrib` : maps (imbalance_bin, spread) to the corresponding exponential disrtibution (`Distribution` is a thin wrapper around `numpy` random functionnality). Since I chose to discard spreads beyond 2 when estimating statistics, the spread falls back to 2 if greater.

```python
# src/aqr.py/ImbalanceQR
def sample_deltat(self, lob: LimitOrderBook) -> int:
        state = (self.map_imbalance_bin(lob.imbalance), min(lob.spread, 2))
        return self.deltat_distrib[state].sample().item()
```

- `event_distrib` : maps (imbalance_bin, spread) to the corresponding discrete distribution of possible events. Each sample is a `tuple[str, str, int]` when unpacked contains `order_type, order_side, order_queue_nbr`.
- `probabilities` : is basically the dataframe respresentation of `event_distrib` in fact `event_distrib` is parsed from this dataframe. The reason I keep it is when we will later need to dynamically change the event probabilities according to an `alpha` the dataframe can be quickly adjusted and we can rebuild `event_distrib`. I will get into the adjustement later on.

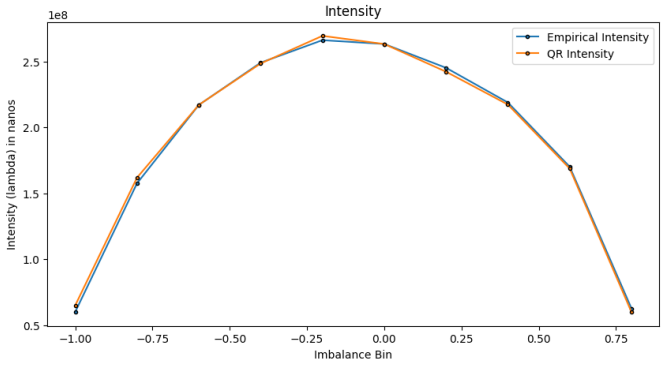
```python
# src/aqr.py/ImbalanceQR
def sample_order(self, lob: LimitOrderBook, alpha: Alpha) -> Order:
    if lob.spread > 1:
        state = (self.map_imbalance_bin(lob.imbalance), 2)
        order_type, order_side, order_queue = (
            self.event_distrib[state].sample().flatten()
        )
        if lob.spread % 2 == 0:
            available_queues = np.arange(-(lob.spread // 2) + 1, lob.spread // 2)
        else:
            available_queues = np.hstack(
                (
                    np.arange(-(lob.spread // 2), 0),
                    np.arange(1, lob.spread // 2 + 1),
                )
            )
        order_queue = self.rng.choice(available_queues)
        order_size = self.order_volumes_distrib[(order_type.item(), Side[order_side].value)][state].sample().ite
    else:
        state = (self.map_imbalance_bin(lob.imbalance), lob.spread)
        order_type, order_side, order_queue = self.event_distrib[state].sample().flatten()
        order_size = self.order_volumes_distrib[(order_type.item(), int(order_queue))][state].sample().item()

    order = (order_type, order_side, order_queue, order_size)
    return self._create_order(order, lob, alpha.value)
```

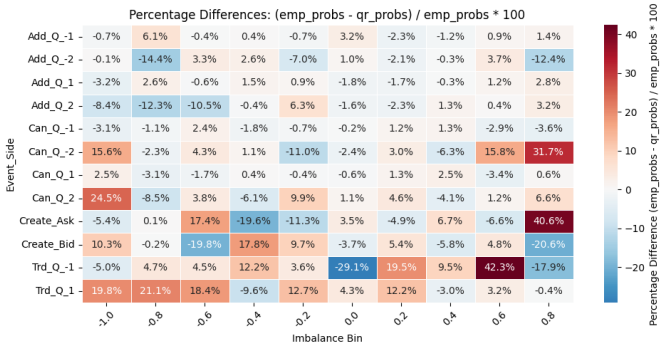
As mentionned earlier, if the spread is greater or equal to 2, we sample an add in spread according to the statistics of $(\text{Imb}, 2)$ and choose an available queue at random.

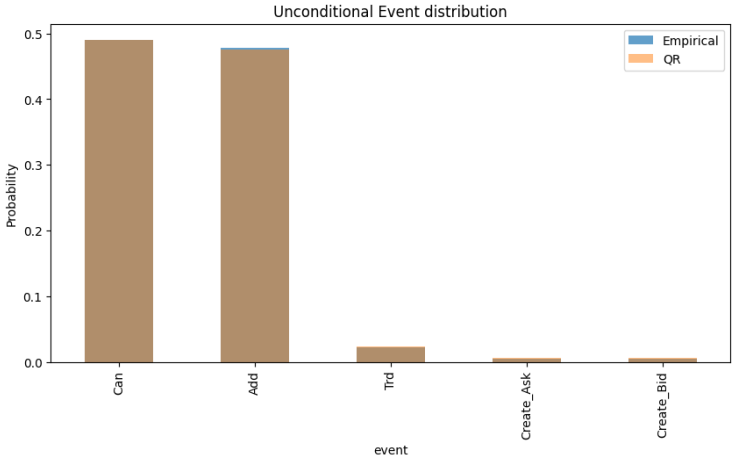
- `order_volumes_distrib` : maps (order_type, order_queue) to a another dictionnary that maps (imbalance, spread) to the corresponding stationnary distribution. order_queue for add in spread is 1 when it's and ask and -1 when it's a bid, also since the volumes are not conditionned on imbalance all the distributions for an add in spread are the same but for the sake of consistency we kept the same structure as with the conditioned ones.

Goodness of fit

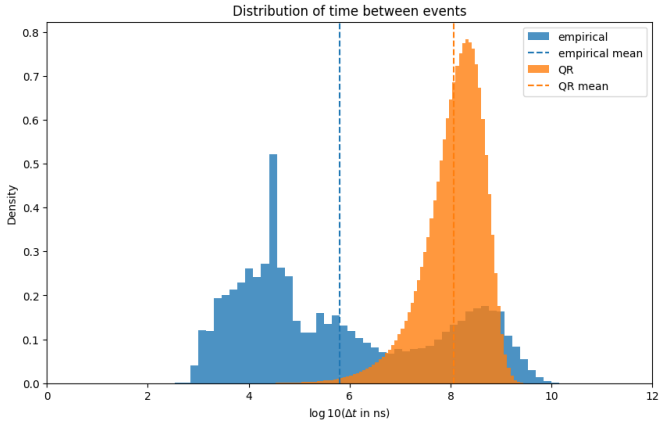
The standard QR model naturally fits well intensities and event probabilities (conditionnaly on the imbalance) :

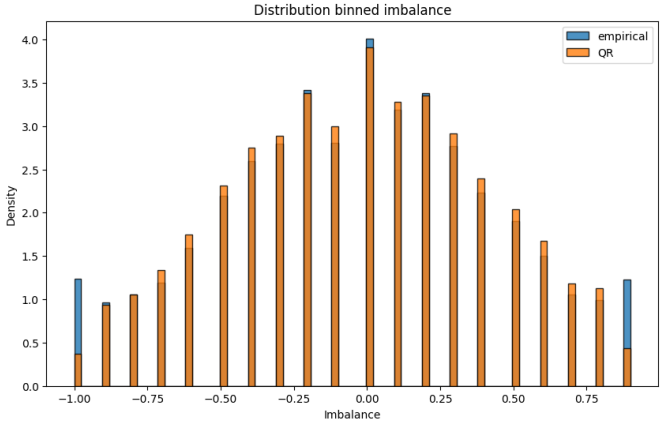




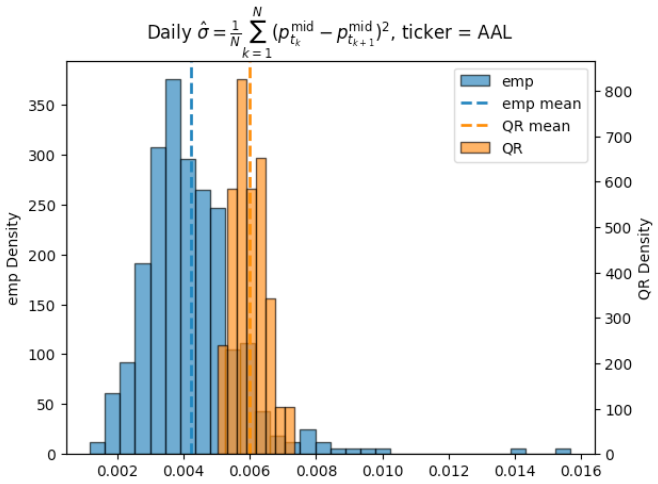


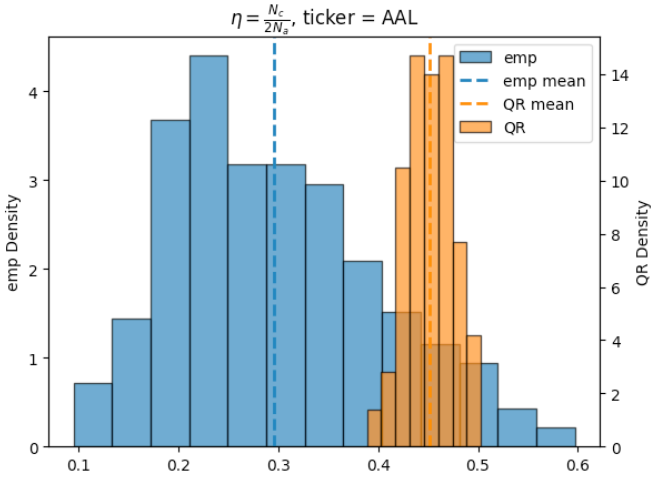
But it struggle to capture the distribution of inter time events Δt, even on average since the distribution of the imbalance is not the same:

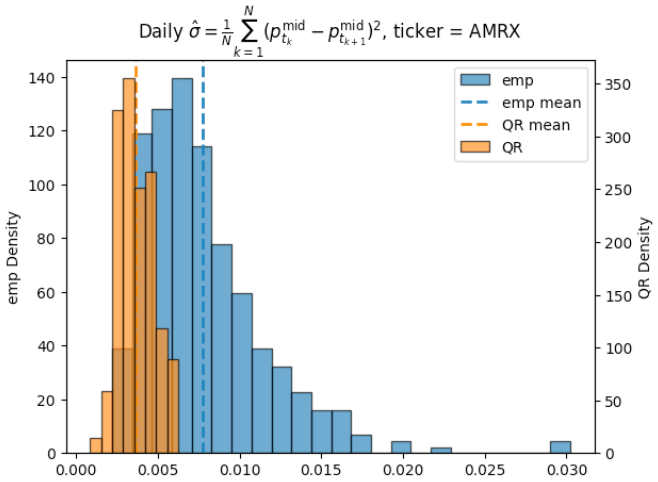


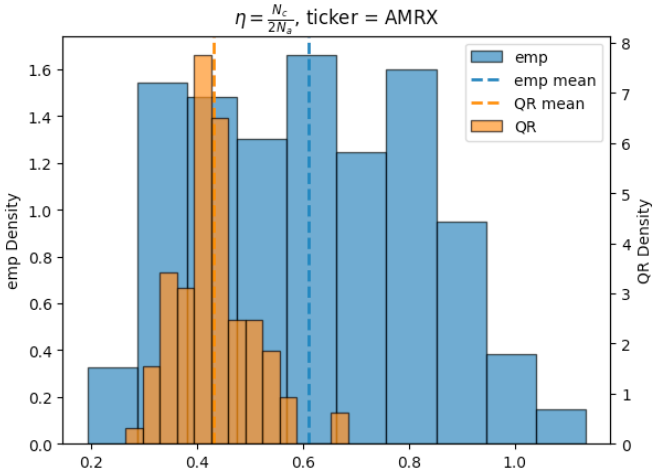


One of the problems of the standard queue reactive model at least in the first paper was that it underestimated volatility but I have found that for some stocks it's the opposite:









We should probably estimate this at scale for a large universe to draw a significant conclusion.

Race Markets Model

Model definition

Alpha and races

In addition to the regular Queue Reactive model, there is now an external process α_t and races are triggered with a certain probability $p(\text{LOB}_t, \alpha_t)$. There may also be "random" races that are triggered at random times following some totally independant process that we shall call here t_{ext}.

Travel time

Also now we account for the time it takes orders to reach the matching engine, and make it's way back to the market participants. Mainly when an order is sent at time t it reaches the matching engine at time $t + l_1$, for EUREX $l_1 = 911 + 271 + 665$ns. Then there is a matching engine latency (random or constant?) of around 30 micros, and it should be generally higher when the first order is a market order as market orders induce more delay. Then this order makes it's way back to all market participants at $t + l_1 + \delta + l_2$, except if rejected. In the case of EUREX $l_2 = 919 + 911$ns

Sampling

At time t

- Order book state seen from the market participant: LOB_t.
- Current α of the market participant: α_t.
- Order queue with a pending order that should reach the market participants at t_{queue}. If the queue is empty $t_{\text{queue}} = \infty$.
- t_{ext} timestamp of the next random race
- t_α timestamp of the next jump in α (independant of the orderbook)

Information processing

- Draw $t_{\text{reg}} = t + \Delta t$, Δt being drawn according to the queue reactive statistics.
- if $t_{\text{reg}} < t_\alpha$ `and` $t_{\text{reg}} < t_{\text{queue}}$ `and` $t_{\text{reg}} < t_{\text{ext}}$ then:
  - Set $t = t_{\text{reg}}$.
  - Sample a regular order according to the Queue Reactive statistics.
  - Append it to the order queue with a $t_{\text{queue}} = t + l_1 + \delta + l_2$.
  - Go back to information processing.
- else if $t_\alpha < t_{\text{ext}}$ `and` $t_\alpha < t_{\text{queue}}$ then:
  - Update the value of α.
  - Adjust the Queue Reactive statistics.
  - Set $t = t_\alpha$.
  - With probability $p(\text{LOB}_t, \alpha_t)$ go to Trigger a Race.
  - Sample the next t_α.
  - Go back to information processing.
- else if $t_{\text{ext}} < t_{\text{queue}}$ then:
  - Set $t = t_{\text{ext}}$.
  - With probability p_{ext} go to Trigger a Race.
  - Sample the next t_{ext}
  - Go back to information processing
- else :
  - Set $t = t_{\text{queue}}$.
  - Update LOB_t according to the new order.
  - If the order is executed succesfully and not rejected, with probability $p(\text{LOB}_t, \alpha_t)$ go to Trigger a Race.
  - Go back to information processing.

Trigger a Race at t

- N aggressive orders are sent. If the race is external the side of the orders is random, or it can be negatively correlated to the imbalance. Otherwise the aggressive orders are sent according to the imbalance.
- The orders are sent to the matching engine. We sample random matching engine latencies between each order following some distribution γ. The orders are sent to the order queue with timestamps t_{queue} which are equal to :

$$t + l_1 + \delta + \sum_{j=1}^{i-1} \gamma_j + l_2.$$

Practical choices

External race

For the external races, I settled on an independant poisson process $t_{\text{ext}} \sim \text{Exp}(\lambda_{\text{ext}})$, where λ_{ext} is around a few seconds (too little ? too much ?). This race will be sent to the ask with probability $\frac{1+\rho\times\text{Imb}_t}{2}$ where ρ is a free parameter between $[-1, 1]$. Also when t_{ext} hits a race is triggered with some probability p_{ext}, I chose it constant for convenience.

Alpha

We chose an alpha that is partly imbalance, partly some external process of the form :

$$\alpha_t = \beta \text{Imb}_t + \eta \varepsilon_t, \quad \beta \approx 0.8, \ \eta \approx 1$$

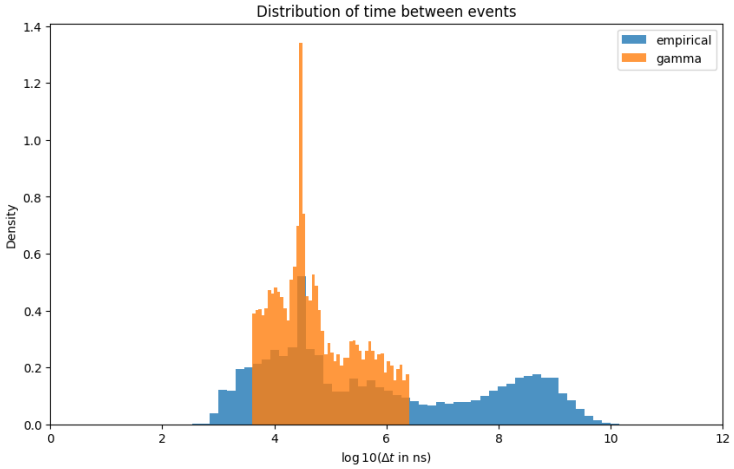
Where :

$$T \sim \text{Exp}(\lambda_\alpha), \ \varepsilon_{t+T} \sim \frac{\delta_{-1} + \delta_1}{2} \times \text{Exp}(\lambda_\varepsilon), \ m = -0.15, \ M = 0.15, \ \lambda_\varepsilon \approx 10 \text{ seconds}, \ \lambda_\varepsilon \approx 0.15$$

We bias the probability of market orders in the ask/bid by $(1+\varepsilon)/(1-\varepsilon)$

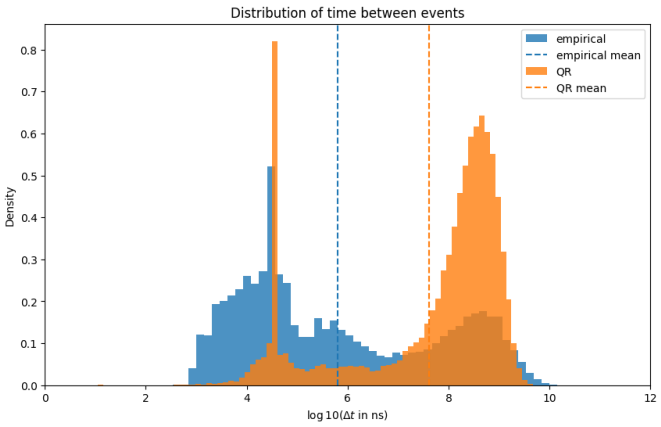
Gamma

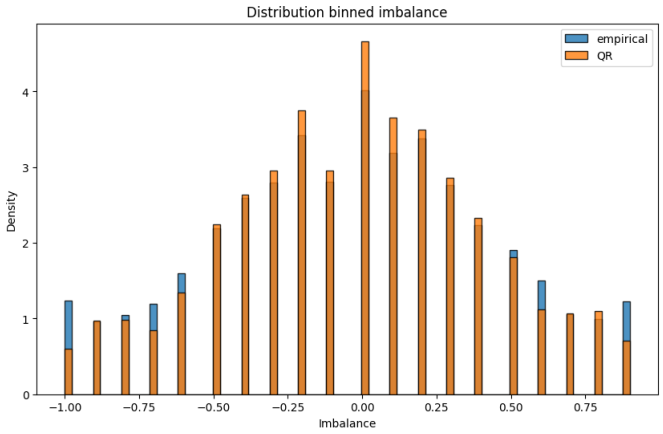
The distribution γ represents the times between events within a race and is the main way we can recover fast events we observe empirically. An easy way to estimate it and what I have done until now is to just take the empirical Δt distribution up to some quantile to try to fit the empirical Δt distribution, e.g:



I think this is a very convenient, but ultimately wrong way of doing this. One of it's biggest flaws is that doesn't account for the peak arround 30 micros which is approximately the round trip time of an order in the NASDAQ which is already taken into account by $l_1 + \delta + l_2$.

Below is a simulation example with races. We can see that this gamma distribution helps fill the gaps in the Δt distribution, but it overestimates the 30 micros peak.





Marketable limit orders

When examining the data, most of the fast events are limit orders. One of the possible explanations is that most are marketable limit orders, so we made all trades in races marketable limit orders.

Race parameters

First parameter is the probability $p(\text{LOB}_t, \alpha_t)$. At first I just gave it a constant value if the alpha is above some threshold in absolute value, but after discussions we settles on an increasing probability starting from some threshold for α. e.g:

$$p(\text{LOB}_t, \alpha_t) = \min\left(\mathbb{1}_{|\alpha_t| \geqslant \alpha_{\text{lower}}}\left(p_{\min} + \frac{|\alpha_t| - \alpha_{\text{lower}}}{\alpha_{\text{upper}} - \alpha_{\text{lower}}}(p_{\max} - p_{\min})\right), p_{\max}\right)$$

With $\alpha_{\text{lower}} \approx 0.7$, $\alpha_{\text{upper}} \approx 1$ and $p_0 \approx 0.4$.

The second parameter of interest is the number of orders in a race which should be increasing when the signal is stronger. It is drawn from a geometric distribution where the parameter θ is decreasing with a stronger singal, e.g:

$$\theta = \max\left(\theta_{\max} + \mathbb{1}_{|\alpha_t| \geqslant \alpha_{\text{lower}}} \frac{|\alpha_t| - \alpha_{\text{lower}}}{\alpha_{\text{upper}} - \alpha_{\text{lower}}}(\theta_{\min} - \theta_{\max}), \theta_{\min}\right)$$

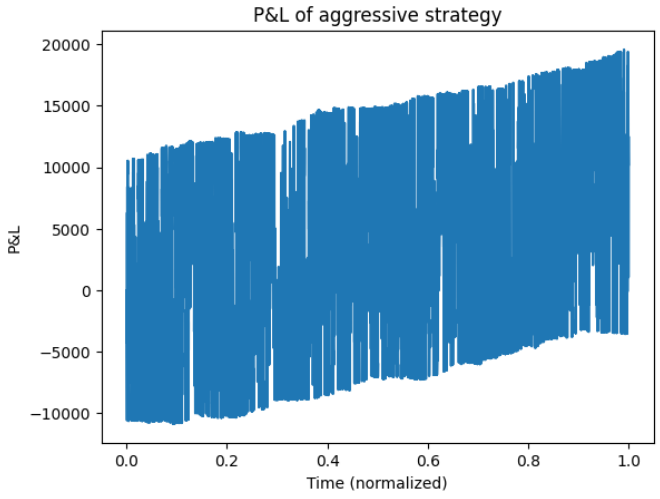
With $\theta_{\max} \approx 0.4$ and $\theta_{\min} \approx 0.1$. Also we draw marketable limit orders and cancel orders with respective probabilities of around 0.65/0.35.

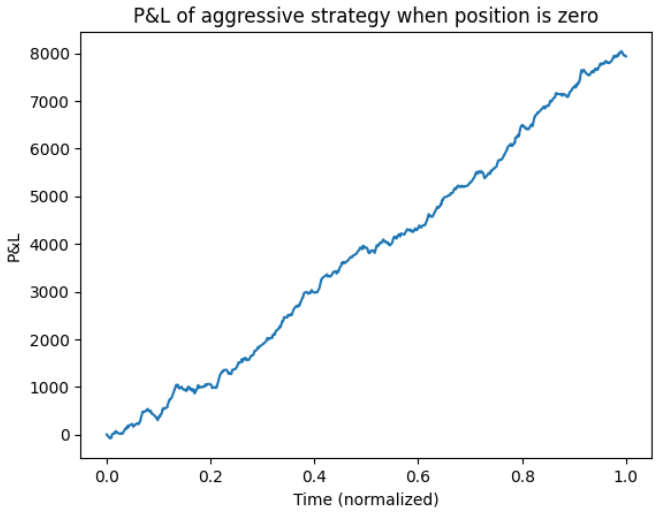
For the volumes of the race orders, we don't condition them on anything I just estimate a stationnary distribution of order size for simplicity. Not sure if it's the best way to go about it but wanted to avoid bloating the model.

One dilemma we had is wether these parameters should really depend on the enitre α or just the ε part.

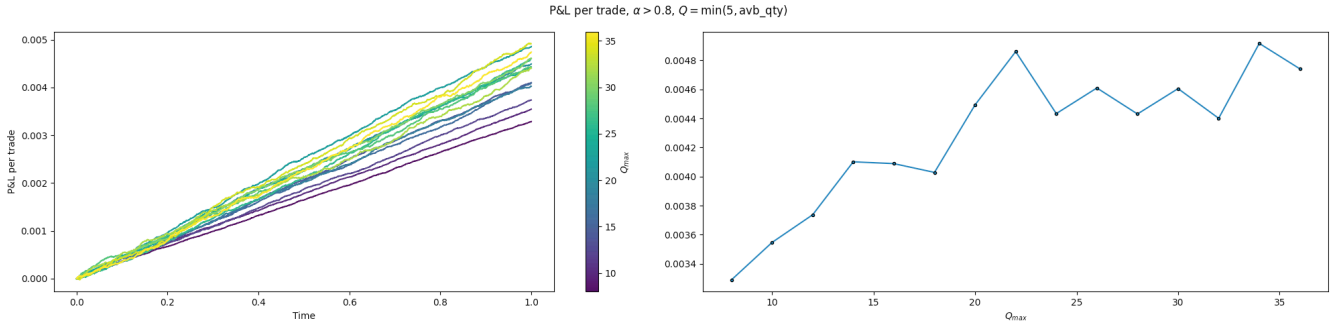
Aggressive Strategy

One of the things we are interested in is the evolution of the $P\&L$ of a liquidity taking strategy with respect to the maximum volume of every trade or the maximum allowed position. The strategy we implement is very simple, if the α (or just ε ?) is above some threshold in absolute value, we send a market order to the corresponding side of volume $\min(V_{\max}, V_{\text{available}})$ as long as the position does not surpass some value $Q_{\max}$ in absolute value. A reasonable approximation for the $P\&L$ we settled on is to compute the cumulative dollar value of our position at the times where we have 0 inventory divided by the total number of transactions.

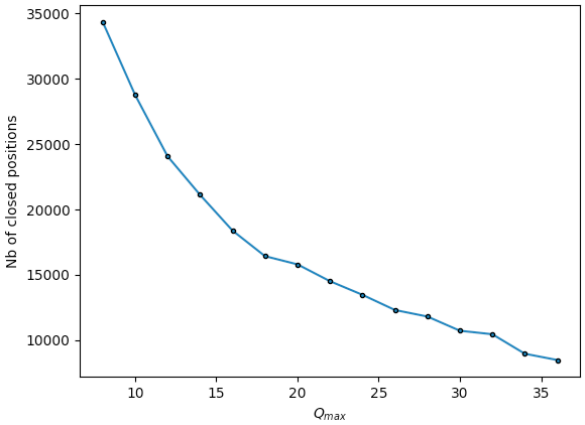




In the example below we vary $Q_{\max}$ with $\alpha_{\text{threshold}} = 0.8$ and $V_{\max}$.



One of the problems is that the more $Q_{\max}$ increases the less times we observe a position equal to 0 which makes the estimate noisy. The numbers below are from the same example as above.



Our goal is for the $P\&L$ to plateau or even decrease above some $Q_{\max}$ threshold but I think we should perform some very long and extensive simulations to draw signigicant conclusions, varying the different parameters $\alpha_{\text{threshold}}, Q_{\max}, V_{\max}$.

Market impact

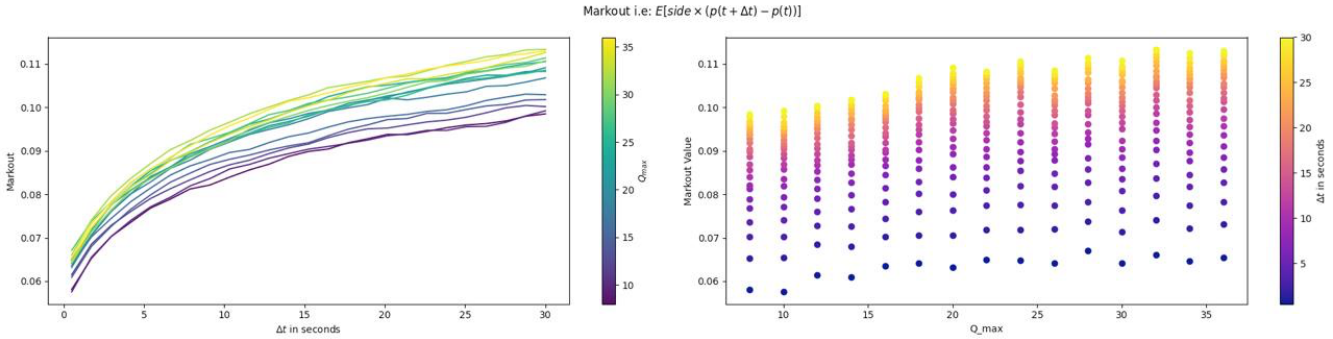
Buy market orders drive the price up and vice versa for sell market orders, this effect is implicit in the QR statistics since a buy/sell market order will drive the imbalance up/down adjusting market order probabilities. But this impact decays over time in $\frac{1}{\sqrt{t}}$. In order to account for this decay, we introduce:

$$\phi_t = \sum_k \frac{\xi_k \psi}{\sqrt{t - t_k}}$$

Where t_k are the timestamps of previous market orders (cancels too ?), ξ_k is the side -1/1 for bid/ask and ψ is some constant coefficient. We bias the queue reactive probabilities of bid/ask market orders by $(1 - \phi_t)/(1 + \phi_t)$. I have not yet played around with this feature so I am not sure what value to give to ψ.

I also chose to limit the number of previous order taken into account in the impact, this is controlled by a `max_size` parameter.

When plotting the markout of an aggressive strategy apparently it should be falt-ish for the first few seconds then increasing which is not the case as seen below, and hopefully this parameter adjusts this.



Also a point worth mentionning, when I computed this markout, I didn't let the model purely play out after an aggressive strategy sends an order, meaning I could have sent an order and then disabled the strategy to see what happens purely with the model but it was simpler to just let the strategy play out. I am not sure if this makes a difference or not but maybe worth exploring.

Implementation

The Race Markets Model is implemented through the `AQR` (Alpha Queue Reactive) class.

```python
# src/aqr/alpha_qr.py
class AQR:
    def __init__(
        self,
        qr_model: QRModel,
        alpha: Alpha,
        matching_engine: MatchingEngine,
        race_model: Race,
        external_event_model: ExternalEvent,
        impact_model: SQRTImpact,
        trader: Trader,
        rng: np.random.Generator = np.random.default_rng(1337),
    ) -> None:
        self.qr_model = qr_model
        self.alpha = alpha
        self.matching_engine = matching_engine
        self.race_model = race_model
        self.external_event_model = external_event_model
        self.impact_model = impact_model
        self.trader = trader
        self.order_queue = OrderQueue()
        self.rng = rng

    def sample(self, lob: LimitOrderBook, max_ts: int, buffer: Buffer) -> None:
        ...
```

The main components are:

- **`qr_model`**: Base Queue Reactive model for regular order sampling
- **`alpha`**: Alpha process implementation (imbalance-based with jumps)
- **`matching_engine`**: Handles order processing with latency simulation
- **`race_model`**: Manages race event generation
- **`external_event_model`**: Handles external random races
- **`impact_model`**: Market impact model with square-root decay
- **`trader`**: Represents market participant behavior
- **`order_queue`**: Priority queue for pending orders with travel times

Sampling Loop Implementation

The core sampling loop implements the event-driven simulation described in the model definition. Let's walk through each step in detail:

```
def sample(self, lob: LimitOrderBook, max_ts: int, buffer: Buffer) -> None:
    # Initialise state
    self.order_queue.clear()
    self.alpha.eps = None
    self.alpha.initialise(lob)
    self.trader.curr_pos = 0
    curr_ts, sequence = 0, 0
    t_external_event = self.external_event_model.sample_deltat()
    t_alpha = self.alpha.sample_jump()

    while curr_ts <= max_ts:
        self.qr_model.adjust_event_distrib(
            self.alpha.eps if self.alpha.eps else 0,
            self.impact_model.value,
            exclude_cancels=True,
        )
        t_reg = (
            self.qr_model.sample_deltat(lob)
            if not self.order_queue.has_regular_order
            else float("inf")
        )
        t1 = self.order_queue.dt

        sample_regular_order = (
            curr_ts + t_reg < t_alpha
            and curr_ts + t_reg < t_external_event
            and curr_ts + t_reg < t1
            and not self.order_queue.has_regular_order
        )
        alpha_jump = t_alpha < t_external_event and t_alpha < t1
        external_race = t_external_event < t1

        if sample_regular_order:
            curr_ts += t_reg
            reg_order = self.qr_model.sample_order(lob, self.alpha)
            self.matching_engine.process_regular_order(reg_order, curr_ts)
            self.order_queue.append_order(reg_order, regular_order=True)
            continue
        elif alpha_jump:
            curr_ts = t_alpha
            self.alpha.update_eps()
            self.alpha.compute_value()
            t_alpha += self.alpha.sample_jump()
        elif external_race:
            curr_ts = t_external_event
            p_external_event = self.external_event_model.probability(lob)
            if self.rng.uniform() < p_external_event:
                external_event_orders = (
                    self.external_event_model.sample_orders(
                        lob, self.alpha.value
                    )
```

```
            )
            self.matching_engine.process_race(
                external_event_orders, curr_ts
            )
            self.order_queue.append_race(external_event_orders)
        t_external_event += self.external_event_model.sample_deltat()
        continue
    else:
        curr_ts = t1
        pending_order = self.order_queue.pop_order()
        if pending_order.action == "Trd_Add":
            market_order, limit_order = process_marketable_limit_order(
                lob, pending_order
            )

            market_order = lob.process_order(market_order)
            buffer.record(
                lob, self.alpha, market_order, self.trader, sequence
            )
            limit_order = lob.process_order(limit_order)
            buffer.record(
                lob, self.alpha, limit_order, self.trader, sequence
            )
            sequence += 1  # both have the same sequence
            if not market_order.rejected:
                self.impact_model.update(
                    market_order.dt, market_order.side.value
                )
            if limit_order.trader_id == 0 and not limit_order.rejected:
                self.trader.can_trade += 1
        else:
            pending_order = lob.process_order(pending_order)
            buffer.record(
                lob, self.alpha, pending_order, self.trader, sequence
            )
            sequence += 1
            if pending_order.rejected:  # No new information
                continue
            if pending_order.action == "Trd":
                self.impact_model.update(
                    pending_order.dt, pending_order.side.value
                )
            if pending_order.trader_id == 0:
                self.trader.can_trade += 1
            if pending_order.trader_id == 1:
                self.trader.curr_pos += (
                    pending_order.size
                    if pending_order.side == Side.A
                    else -pending_order.size
                )

        self.alpha.update_imbalance(lob)
        self.alpha.compute_value()

race_orders = []
```

```
        p_trader = self.trader.probability(lob, self.alpha) * (
            self.order_queue.trader_order_count == 0
        )
        if self.rng.uniform() < p_trader:
            order = self.trader.order(lob, self.alpha)
            race_orders.append(order)
            self.trader.can_trade = -self.trader.cooldown + 1

        p_race = self.race_model.probability(lob, self.alpha)
        if self.rng.uniform() < p_race:
            orders = self.race_model.sample_race(lob, self.alpha)
            race_orders.extend(orders)

        if len(race_orders) > 0:
            shuffled_race_orders = self.rng.permutation(
                race_orders
            ).tolist()
            self.matching_engine.process_race(
                shuffled_race_orders, curr_ts
            )
            self.order_queue.append_race(shuffled_race_orders)
```

During the first step of the main loop, queue reactive probabilities are adjusted to account for the decaying impact and alpha's random epsilon:

```
self.qr_model.adjust_event_distrib(
    self.alpha.eps if self.alpha.eps else 0,
    self.impact_model.value,
    exclude_cancels=True,
)
```

Then t_{reg} is sampled following QR statistics, t_1 is the next order to be executed in the queue:

```
t_reg = (
    self.qr_model.sample_deltat(lob)
    if not self.order_queue.has_regular_order
    else float("inf")
)
t1 = self.order_queue.dt
```

Following this initial step is an if else statement that decides which event comes first. If one of these events does not include a race a `continue` statement brings us back to the beginning of the loop. Otherwise here is the race sampling logic:

```
race_orders = []

# Aggressive strategy part
p_trader = self.trader.probability(lob, self.alpha) * (
    self.order_queue.trader_order_count == 0
)
if self.rng.uniform() < p_trader:
    order = self.trader.order(lob, self.alpha)
    race_orders.append(order)
    self.trader.can_trade = -self.trader.cooldown + 1

# Fast markets race part
p_race = self.race_model.probability(lob, self.alpha)
if self.rng.uniform() < p_race:
```

```
    orders = self.race_model.sample_race(lob, self.alpha)
    race_orders.extend(orders)

if len(race_orders) > 0:
    shuffled_race_orders = self.rng.permutation(
        race_orders
    ).tolist() # Orders are shuffled that way the strategy is not always first
    self.matching_engine.process_race(
        shuffled_race_orders, curr_ts
    )
    self.order_queue.append_race(shuffled_race_orders)
```

Travel Time and Matching Engine

The matching engine implements the travel time model with components l_1, δ, and l_2:

```
# src/aqr/matching_engine.py
class MatchingEngine:
    def process_regular_order(self, order: Order, curr_ts: int) -> None:
        """Process regular order with travel time l1 + delta + l2"""
        order.ts = curr_ts
        order.xt = curr_ts + self.l1
        order.dt = order.xt + self.delta.sample().item() + self.l2

    def process_race(self, orders: List[Order], curr_ts: int) -> None:
        """Process race orders with inter-arrival times gamma"""
        gammas = self.gamma.sample(n=len(orders))
        gammas.sort()
        gammas = gammas.cumsum()
        gammas[0] = 0
        gammas += self.delta.sample()

        for order, gamma in zip(orders, gammas):
            order.ts = curr_ts
            order.xt = curr_ts + self.l1
            order.dt = order.xt + gamma.item() + self.l2
```

Order Queue Management

The `OrderQueue` uses a priority queue to efficiently manage orders by their arrival timestamps:

```
# src/aqr/matching_engine.py
class OrderQueue:
    def __init__(self) -> None:
        self.heap = []  # Min-heap: (dt, insertion_order, order)
        self.regular_order_count = 0
        self.trader_order_count = 0
        self._insertion_counter = 0

    @property
    def dt(self) -> int | float:
        """Timestamp of the order with smallest dt in the queue"""
        return self.heap[0][0] if self.heap else float("inf")

    def append_order(self, order: Order, regular_order: bool = False) -> None:
        heapq.heappush(self.heap, (order.dt, self._insertion_counter, order))
        self._insertion_counter += 1
```

```
        if regular_order:
            self.regular_order_count += 1

    def append_race(self, orders: Iterable[Order]) -> None:
        for order in orders:
            heapq.heappush(self.heap, (order.dt, self._insertion_counter, order))
            self._insertion_counter += 1
            if order.trader_id == 1:
                self.trader_order_count += 1
```

At the beginning speed wise my implementation was fine, but as I added on more features it became pretty slow so I am thinking of ways to speed things up. Unfortunately I don't see a way around the python loop. I tried at one point doing it in `JAX` but it took me alot of time and whenever I wanted to add a new feature I had to make breaking changes that would force me to rewrite a big portion of it and it just wasn't worth it.