Algo_LatencyArb

Sunday, April 27, 2025 10:32 AM

Parameter Adjustment Algorithm

You want to **adjust X** based on the **magnitude of volumeImbalance**. Here's a flexible, production-style approach:

Adjustment Function

```
def adjust_parameter_X(volume_imbalance, base_X, k):
  """
  Adjusts X based on volume imbalance.
  - base_X: the baseline parameter value
  - k: sensitivity factor
  """
  # e.g., X increases linearly with the absolute imbalance
  # You can use other functions (exponential, piecewise, etc.) as needed
  return base_X + k * abs(volume_imbalance)
```

In practice, you might want to cap X within bounds.

```
def adjust_parameter_X(self, product_id, counterparty_type):
    imbalance = self.get_volume_imbalance(product_id, counterparty_type)
    # Example adjustment: linear with cap
    X = self.base_X + self.k * abs(imbalance)
    return min(max(X, 0.5), 5.0)  # Clamp X between 0.5 and 5.0
```

$EMA_{new} = \alpha \times value_{new} + (1-\alpha) \times EMA_{old}$

α (smoothing factor) defines how quickly old values lose influence (typical: 0.01–0.2).
In your context, each trade updates the EMA of volumeImbalance.

This is often called a **continuous-time EMA** and is **ideal for irregularly-timed events** (like trades that don't arrive exactly every second). Here's how and why it works, plus sample code.

Why Time-Based EMA?

- In financial trading, trades can occur at any moment.
- If you use a fixed alpha (as in regular EMA), the "decay" rate is tied to trade frequency, not wall-clock time.
- **Time-based EMA** ensures that the impact of a trade decays at a consistent rate over real time, regardless of how often trades arrive.

$\alpha = 1 - \exp(-\Delta t / \tau)$

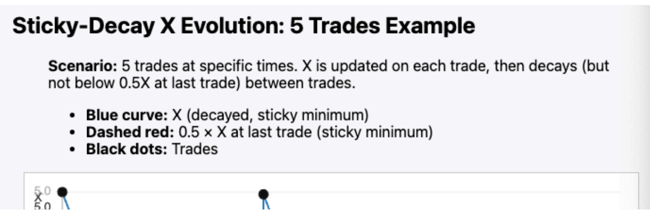
$EMA_{new} = \alpha \times x_{new} + (1-\alpha) \times EMA_{old}$

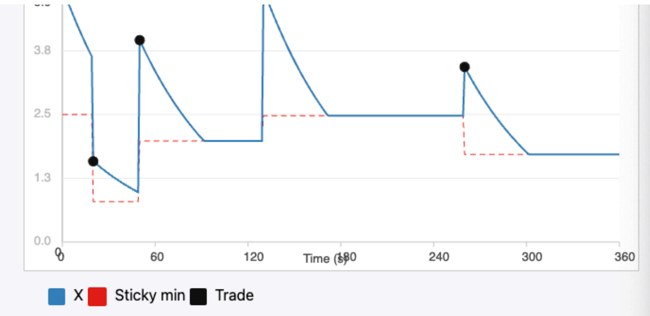
Key Points

- **(\tau)**: Higher τ means slower decay (old trades matter longer); lower τ means faster decay (recent trades matter more).
- **Handles irregular intervals**: If many seconds pass between trades, alpha is higher (old EMA "forgets" faster).

Retail Market-making Algorithm

- On each trade
  - update imbalance (via time-based EMA or running sum)
  - Compute a new X:
    - X new = k * imbalance
- Between trades
  - If there's no new activity, X should decay over time
  - However, X cannot decay below a "sticky minimum", which is 0.5 x the last nonzero X
- Implementation plan
  - Track last trade time and last adjusted X
  - On each trade
    - Update the running sum of imbalance
    - Compute new X
    - Set the new sticky minimum: Xmin = 0.5 * Xnew
  - On each X query (even without new trades):
    - Decay X using the time-based EMA (with no new input, so input = 0)
    - Clamp X to Xmin





step_num = floor(EMA imbalance / threshold).

- If step_num increases, increment X by k for each step up.
- If step_num decreases, decrement X by k for each step down.
- sticky_min = 0.5 * abs(X) * Math.sign(X)
- Between trades, **decay X exponentially towards zero but do not cross the sticky minimum** (sticky min is always half the magnitude of the last X after a step).

1. Initialization

- Set parameters:
  - k: the increment/decrement per threshold crossed (e.g., 1.0)
  - threshold: the imbalance threshold for each step (e.g., 50)
  - tau: EMA decay time constant (e.g., 60 seconds)
  - sticky_factor: fraction for sticky minimum (e.g., 0.5)

  Initialize for each trading pair (product_id, counterparty_type):
  - ema_imbalance = 0
  - X = 0
  - last_step_num = 0
  - sticky_min = 0
  - last_update_time = None

2. On Each Trade

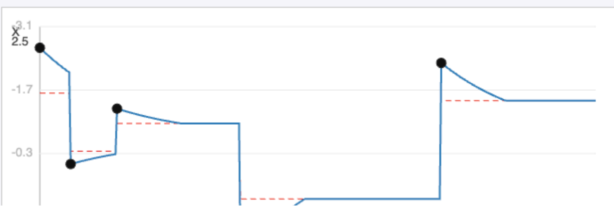
- **2.1. Decay EMA and X to the current trade's timestamp** (if any time has passed):
  - $\mathrm{dt} = \mathrm{trade.time} - \mathrm{last\_update\_time}$
  - $\mathrm{ema\_imbalance} \leftarrow \mathrm{ema\_imbalance} \times e^{-\mathrm{dt}/\tau}$
  - $X \leftarrow X \times e^{-\mathrm{dt}/\tau}$
  - Clamp X to not cross the sticky minimum:
    - If $X > 0$: $X \leftarrow \max(X, \mathrm{sticky\_min})$
    - If $X < 0$: $X \leftarrow \min(X, \mathrm{sticky\_min})$
- **2.2. Update EMA imbalance with the new trade:**
  - $\mathrm{ema\_imbalance} \leftarrow \mathrm{ema\_imbalance} + \mathrm{trade.qty}$
- **2.3. Calculate the new step number (can be negative):**
  - $\mathrm{step\_num} = \mathrm{floor}(\mathrm{ema\_imbalance}/\mathrm{threshold})$
- **2.4. If the step number changed (up or down):**
  - $\mathrm{steps\_change} = \mathrm{step\_num} - \mathrm{last\_step\_num}$
  - $X \leftarrow X + \mathrm{steps\_change} \times k$
  - $\mathrm{sticky\_min} \leftarrow \mathrm{sticky\_factor} \times |X| \times \mathrm{sign}(X)$
  - $\mathrm{last\_step\_num} \leftarrow \mathrm{step\_num}$
- **2.5. Update last update time:**
  - $\mathrm{last\_update\_time} = \mathrm{trade.time}$

3. Between Trades (when you want to check X at any time)

- Decay EMA and X to the current time as in step 2.1.
- Clamp X to sticky minimum as above.

Scenario: X increases or decreases by **k** every time the signed EMA imbalance crosses a multiple of the threshold (50).

- **Blue curve:** X (stepwise, sticky minimum)
- **Dashed red:** Sticky min (always 0.5 × |X| × sign(X) after last step)
- **Black dots:** Trades
- X can go negative if imbalance reverses direction!



Code Implementation

Here's how you can **implement this logic in Pandas** as a function that adds the computed z, level_num, and sticky_min columns to your DataFrame for each counterpartyType, following your description.

Step-by-Step Approach

1. **Sort** your DataFrame by counterpartyType and timeBin.
2. **For each counterpartyType**, process the group in time order:
   - Initialize variables:
     - z = 0
     - level_num = 0
     - sticky_min = 0
     - last_level_num = 0
     - last_update_time = None
   - For each row:
     - Compute dt = time difference in seconds from last_update_time (set dt = 0 for the first row).
     - Decay z as:
       z = z * exp(-dt / tau)
     - Clamp z to sticky_min:
       - If z > 0: z = max(z, sticky_min)
       - If z < 0: z = min(z, sticky_min)
     - Compute new level_num = floor(volumeImbalance / threshold)
     - Compute level_change = level_num - last_level_num
     - Update z = z + level_change * k
     - Clamp z to [min_z, max_z]
     - Update sticky_min = sticky_factor * z
     - Save values to new columns.
     - Update last_level_num, last_update_time.

Python/Pandas Implementation

Assume your DataFrame is called df and has columns:

- "timeBin" (as string or pd.Timestamp)
- "counterpartyType"
- "volumeImbalance" (numeric)

```
import pandas as pd
import numpy as np
def add_algo_live_settings(
    df,
    tau=600,
    k=1,
    threshold=1_000_000,
    sticky_factor=0.7,
    max_factor=10,
    min_factor=10,
    time_format='%H:%M:%S'
):
    # Ensure timeBin is datetime
    if not np.issubdtype(df['timeBin'].dtype, np.datetime64):
        df = df.copy()
        df['timeBin'] = pd.to_datetime(df['timeBin'], format=time_format)

    df = df.sort_values(['counterpartyType', 'timeBin']).reset_index(drop=True)
    df['algo_z'] = 0.0
    df['algo_level_num'] = 0
    df['algo_sticky_min'] = 0.0
max_z = k * max_factor
    min_z = -k * min_factor
# Process each counterpartyType group
    for cpty, group_idx in df.groupby('counterpartyType').groups.items():
        idxs = list(group_idx)
        z = 0.0
        sticky_min = 0.0
        last_update_time = None
        last_level_num = 0
for i in idxs:
            row = df.loc[i]
            t = row['timeBin']
            volumeImb = row['volumeImbalance']
            # dt in seconds
            if last_update_time is None:
                dt = 0
            else:
                dt = (t - last_update_time).total_seconds()
            # Decay z
            z = z * np.exp(-dt / tau)
```

```
        # Clamp z to sticky min
        if z > 0:
            z = max(z, sticky_min)
        elif z < 0:
            z = min(z, sticky_min)
        # Level
        level_num = int(np.floor(volumeImb / threshold))
        level_change = level_num - last_level_num
        # Update z by steps
        z = z + level_change * k
        # Clamp z within allowed range
        z = max(min(z, max_z), min_z)
        # Update sticky min
        sticky_min = sticky_factor * z
        # Store
        df.at[i, 'algo_z'] = z
        df.at[i, 'algo_level_num'] = level_num
        df.at[i, 'algo_sticky_min'] = sticky_min
        # Update state
        last_level_num = level_num
        last_update_time = t
return df
# Example usage:
# df = pd.DataFrame({...}) # as described in your format
# df = add_algo_live_settings(df)
# print(df)
```

What This Does

- **Adds** algo_z, algo_level_num, and algo_sticky_min columns.
- **Handles** time decay, sticky minimum, and step-wise increments per counterparty.
- **Works** for any DataFrame with your described columns.

```
import pandas as pd
import numpy as np

def add_algo_live_settings(
    df,
    tau=600,
    k=1,
    threshold=1_000_000,
    sticky_factor=0.7,
    max_factor=10,
    min_factor=10,
    time_format='%H:%M:%S'
):
    # Ensure timeBin is datetime
    if not np.issubdtype(df['timeBin'].dtype, np.datetime64):
        df = df.copy()
        df['timeBin'] = pd.to_datetime(df['timeBin'], format=time_format)

    df = df.sort_values(['counterpartyType', 'timeBin']).reset_index(drop=True)
    df['algo_z_old'] = 0.0
    df['algo_z_new'] = 0.0
    df['algo_level_num'] = 0
    df['algo_sticky_min'] = 0.0

    max_z = k * max_factor
    min_z = -k * min_factor

    # Process each counterpartyType group
    for cpty, group_idx in df.groupby('counterpartyType').groups.items():
        idxs = list(group_idx)
        z = 0.0
        sticky_min = 0.0
        last_update_time = None
        last_level_num = 0

        for i in idxs:
            row = df.loc[i]
            t = row['timeBin']
            volumeImb = row['volumeImbalance']
            # dt in seconds
            if last_update_time is None:
                dt = 0
            else:
                dt = (t - last_update_time).total_seconds()
            # Decay z
            z_decayed = z * np.exp(-dt / tau)
            # Clamp z to sticky min before stepping
            if z_decayed > 0:
                z_clamped = max(z_decayed, sticky_min)
            elif z_decayed < 0:
```

```
            z_clamped = min(z_decayed, sticky_min)
        else:
            z_clamped = z_decayed
        # Save z_old
        df.at[i, 'algo_z_old'] = z_clamped
        # Level
        level_num = int(np.floor(volumeImb / threshold))
        level_change = level_num - last_level_num
        # Step z
        z_stepped = z_clamped + level_change * k
        # Clamp z within allowed range
        z_new = max(min(z_stepped, max_z), min_z)
        # Update sticky min
        sticky_min_new = sticky_factor * z_new
        # Save results
        df.at[i, 'algo_z_new'] = z_new
        df.at[i, 'algo_level_num'] = level_num
        df.at[i, 'algo_sticky_min'] = sticky_min_new
        # Update state for next row
        z = z_new
        sticky_min = sticky_min_new
        last_level_num = level_num
        last_update_time = t

    return df

# Example usage:
# df = pd.DataFrame({...}) # as described in your format
# df = add_algo_live_settings(df)
# print(df[['timeBin', 'counterpartyType', 'volumeImbalance', 'algo_z_old', 'algo_z_new', 'algo_level_num',
'algo_sticky_min']])
```

Absolutely! **Tau** (the decay time constant) controls how quickly your variable (e.g., `z`) "forgets" past values and responds to new changes.

Understanding Tau

- **Tau is in seconds** (if your time units are seconds).
- The formula for exponential decay is:
  [
  z_{\text{new}} = z_{\text{old}} \times \exp\left(-\frac{dt}{\tau}\right)
  ]
- **Rule of thumb:** After a time equal to tau, the variable decays to about 37% of its previous value (since (e^{-1} \approx 0.37)).
- Typical interpretation:
  - "Tau is the time it takes for the signal to reduce to 37% of its value, if no new changes happen."

How to select tau for a target duration

If you want the signal to "mostly forget" after 3 hours:

- 3 hours = 3 × 60 × 60 = **10,800 seconds**
- Set **tau = 10,800**.

Or, if you want to "mostly forget" after N hours:

- Set `tau = N * 3600` (N = number of hours)

If you want 50% decay after T hours

If you want **half-life** (i.e., decays to 50% in T hours), compute tau as:
[
\tau = \frac{T}{\ln(2)} \approx 1.44 \times T
]
where T is the duration in seconds you want 50% decay.

Examples

- **Tau for 3 hours (for 37% decay in 3 hours):**
  `tau = 3 * 60 * 60  # 10800 seconds`
- **Tau for 3 hours half-life (50% decay in 3 hours):**
  `tau = 3 * 60 * 60 / np.log(2)  # ~15577 seconds`

Summary Table

Decay Type	Target Time (hours)	Formula for tau	tau (seconds) for 3h
37% decay (1/e)	3	`tau = 3*3600`	10,800
50% decay (half-life)	3	`tau = 3*3600/ln(2)`	~15,577

How to choose?

- **If you want "signal is mostly gone after 3 hours":** use 37% decay, so `tau = 10,800`.
- **If you want "signal halves after 3 hours":** use half-life, so `tau = 15,577`.
- **For "forgetting" most of the signal after 3 hours, tau = 10,800 is typical.**

In Practice

Just set:
```
tau = 3 * 60 * 60  # for 3 hours in seconds
```

Reasons to use decay (gradual reduction) rather than instant reset for your Z charge (price margin adjustment)

1. Market Impact Persists Beyond a Single Trade

- **Order flow creates pressure:** A surge in demand (or supply) often leads to continued directional flow. Decaying Z means your price continues to “remember” and counteract the earlier imbalance, helping balance the market even after the initial trades.

2. Smooth Price Adjustments

- **Avoids abrupt jumps:** If you instantly reset Z, your price could snap back sharply, leading to unstable or erratic pricing. Decay ensures smoother transitions, which is less confusing for clients and reduces the risk of overreacting to transient moves.

3. Recognizes Latent Risk

- **Risk remains after the flow:** If you’ve bought inventory to meet big buy orders, you’re left with more risk. Decaying Z lets you unwind this risk gradually, keeping some compensation in price until the risk really fades, rather than pretending it's instantly gone.

4. Mitigates Adverse Selection

- **Protects against quick reversals:** If you immediately remove Z, opportunistic traders could exploit the quick reversion, causing you to sell too cheaply (or buy too expensively) if the flow reverses. Decay prevents this by keeping a memory of recent activity.

5. Captures Market Memory

- **Markets have inertia:** Large order flows can have lasting effects as market participants digest and respond over time. Decay in Z means your pricing “remembers” recent events, even if activity briefly pauses.

6. Theoretical Foundation

- **Exponential decay is a classic model** for “forgetting” in both markets and control theory; it provides a tunable, time-weighted memory that is mathematically robust and intuitive.

7. Allows for Measured Rebalancing

- **Gradual normalization:** If the flow truly dries up, Z will return to zero, but only gradually. This gives liquidity time to rebalance and avoids overcompensating in the other direction.

Approach	**What Happens**	**Result**
Instant reset	Z goes to zero quickly	Price can “snap” back, losing memory of risk/flow
Decay	Z fades gradually	Smooth, risk-aware, market-memory adaptive pricing

In Short

> **Decay lets your pricing “remember” recent flow, helping you manage risk, avoid sharp price swings, and respond more intelligently to the true liquidity and risk profile of the market.**

8. Inventory Management & Risk Control

- **Inventory matters:** As a market maker in listed derivatives, your price charge (Z) shouldn’t only react to recent flow—**it should also reflect your current inventory**. If you have accumulated a large long (or short) position as a result of market flows, you are exposed to risk.
- **Gradual decay aligns with real risk:** Instantly resetting Z would ignore the reality that you’re still holding, say, a lot of contracts to sell. By letting Z decay slowly, you ensure your prices continue to reflect the risk and cost of carrying this inventory, and you keep incentivizing counterparties to help you rebalance, even if the active flow pauses.
- **Prevents “dumping” inventory at poor prices:** If Z were removed instantly, you might end up selling your remaining inventory too cheaply (or buying back too expensively), especially if the market reverses or liquidity dries up.