Architectural Analysis and Validation of a Gymnasium-Based Order-Driven Market Simulator

[Mudil Goel]

December 29, 2025

1 Market Structure & Matching Logic

The core of the simulator is modeled as an **Order-Driven Market**, a structural choice that aligns with the dominant design of modern global equity exchanges (e.g., NYSE, Nasdaq, Tokyo Stock Exchange). Unlike quote-driven markets, where designated market makers or dealers set buy and sell prices, an order-driven market relies on the interaction of limit orders submitted by market participants to determine the price.

1.1 The Limit Order Book (LOB)

The central data structure of the simulator is the `OrderBook` class. This class maintains two distinct, sorted lists: `bids` (buy orders) and `asks` (sell orders). In adherence to standard microstructure theory, the simulator treats liquidity as a scarce resource.

- **Bid Side:** Sorted in descending order. The highest price a participant is willing to pay is at the "top of the book."
- **Ask Side:** Sorted in ascending order. The lowest price a participant is willing to accept is prioritized.

1.2 Matching Mechanism: Price-Time Priority

To resolve liquidity conflicts, the simulator implements a **Price-Time Priority (FIFO)** algorithm. This is the industry standard for equity markets, designed to incentivize liquidity provision by rewarding traders who improve the best available price or, in the event of a tie, those who arrive earliest.

In the Python implementation, this logic is encapsulated in the `match()` method. The engine continuously monitors the spread. When the inequality $BestBid \geq BestAsk$ is satisfied, a trade occurs. The system executes the trade at the bid price (a simplified crossing rule), records the transaction on the "tape" (the `trades` list), and handles partial fills by decrementing the quantity of the matched orders. This mimics the continuous double auction mechanism found in real-world electronic communication networks (ECNs).

2 The Agent-Environment Loop

A critical architectural requirement for this project was compatibility with Reinforcement Learning (RL) workflows. To achieve this, the simulation is wrapped in a custom `MarketEnvironment` class that adheres to the **Gymnasium (formerly OpenAI Gym)** API standards.

2.1 Separation of Concerns

The architecture enforces a strict separation between the market mechanism and the trader.

- **The Environment (`TradingEnv`):** Acts as the exchange operator. It enforces market rules (e.g., rejecting trades when the book is empty), manages the passage of time, and calculates the results of actions.
- **The Agent:** Represents the trader. The agent submits actions (orders) without direct access to the internal logic of the matching engine.

2.2 Discrete-Time State Transitions

Markets in the real world are continuous, but for the purpose of algorithmic training, the simulator discretizes time into "steps." Each call to the `step()` function represents a micro-moment where the agent assesses the market and submits an order. The state transition follows the Markov Decision Process (MDP) tuple (S, A, R, S'), where the agent observes state S, takes action A (Buy/Sell, Price, Qty), receives reward R (profit/loss), and the market transitions to a new state S'. This standardization allows for the seamless future integration of advanced RL algorithms such as PPO or DQN.

3 Information & State Observation

A defining characteristic of financial markets is **Information Asymmetry**. As noted by O'Hara in *Market Microstructure Theory*, traders almost never possess perfect information about the "true" value of an asset or the hidden intent of other participants.

3.1 Level 2 (L2) Market Data

To replicate this constraint, the simulator provides agents with a restricted view of the world known as "Level 2 Market Data." The `get_l2_state()` function generates an observation vector containing only the prices and quantities of the top N bids and asks.

3.2 Implementation of Partial Observability

Crucially, the agent **cannot** see:

- The identity of other traders (preserving anonymity).
- Orders buried deep in the book (simulating "dark" or off-screen liquidity).
- The private valuation or "alpha" of other agents.

This design forces the agent to infer market dynamics solely from the public flow of orders, mimicking the real-world experience of a quantitative trader staring at a Bloomberg terminal or an order depth ladder.

4 Validation: The Zero-Intelligence Test

To verify the integrity of the market mechanism before introducing complex AI strategies, the system was subjected to a **Zero-Intelligence (ZI) Constraint Test**. This methodology is derived from the seminal paper by Gode and Sunder (1993), *Allocative Efficiency of Markets with Zero-Intelligence Traders*.

4.1 Methodology

The simulation was initialized with a `RandomAgent`—a trader programmed to submit orders with random sides (buy/sell), random price offsets ($P_{last} \pm \delta$), and random quantities. This agent possesses no strategy, no memory, and no intent to maximize profit.

4.2 Results and Analysis

The simulation was run for 100 timesteps. The resulting price path exhibited the following characteristics:

- **Price Discovery:** Despite the random inputs, the transaction prices did not explode to infinity or collapse to zero. They remained within a bounded range (93.0 to 101.0), oscillating around the initial equilibrium price.
- **Non-Trivial Liquidity:** The presence of a fluctuating price path confirms that the `match()` logic successfully identified and executed crossing orders.

4.3 Conclusion

The successful execution of the Zero-Intelligence test confirms the "Allocative Efficiency" of the simulator. It proves that the market structure itself—the funneling of bids and asks through a priority engine—is sufficient to generate a coherent price path. The simulator is therefore validated as a robust environment for the next phase of research: training intelligent agents to exploit these structural dynamics.