From Code to Quotes: Simulating Market Making Dynamics in C++

Saxon Lee

12th of September 2021

Abstract

So here we are, against the advice of my mathematics peers (they love Python) and with good luck wishes from my CS peers I'm giving C++ a go. Here lies the details behind my first extensive C++ project undertaken to bridge skill gaps in quantitative finance and object-oriented programming. Driven by curiosity, the project involved developing a market simulation environment to model a limit order book and the behavior of an adaptive market-making agent. The outcome is a functional simulator that allows for the testing of trading strategies and provides insights into market dynamics, serving as a significant step in my technical growth in this domain. It may not be perfect but we'll give it a go - with the help of a book or two (or more) of course. Edit (Dec 2021): I will likely be on a investment/trading desk this coming year so let's go.

Contents

1 Introduction

My journey into the world of quantitative finance stems from a desire to transform curiosity into competence with direct market feedback and compensation as a potential reward. Recognizing my skill gaps, particularly in C++ application and the nuances of market microstructure, I embark on this project to see what I'm made of. The goal was not just to learn passively, but to actively construct, debug, and refine a system that could bring theoretical concepts to life. This endeavor represents a commitment to steady, hands-on growth in a technical domain that I find deeply fascinating.

This project, at its core, is a C++ application designed to simulate a financial market's limit order book (LOB) and the actions of an automated market-making agent operating within it. The agent employs a dynamic quoting strategy, adjusting its bid and ask orders based on its current inventory, a target inventory level, and a perception of the asset's fair value. Such simulations are vital in quantitative finance for several reasons: they provide a risk-free environment to test and iterate on trading algorithms, offer insights into how different strategies might perform under various market conditions, and help in understanding the complex interplay of liquidity provision, price discovery, and risk management. For me, it was a direct path to understanding these elements from the ground up.

2 Theory & Background

The simulation draws upon several key concepts from market microstructure and algorithmic trading theory.

2.1 Limit Order Book (LOB)

A Limit Order Book is the core mechanism for price discovery in many modern financial markets. It's a record of all outstanding buy (bid) and sell (ask) orders for a specific asset, organized by price level [2]. Orders are typically prioritized first by price (e.g., highest bid, lowest ask) and then by time of arrival (FIFO - First-In, First-Out) within each price level. The simulation's `OrderBook` class meticulously implements this structure, using `std::map` containers to manage orders at discrete price levels and `std::vector<Order>` to maintain time priority.

2.2 Market Making

Market makers are crucial liquidity providers. They simultaneously post bid and ask limit orders, hoping to profit from the bid-ask spread – the difference between the highest price a buyer is willing to pay (best bid) and the lowest price a seller is willing to accept (best ask). The `MarketMakerAgent` class in this project embodies this role. A key challenge for market makers is managing inventory risk: accumulating too much (long) or too little (short) of an asset exposes them to adverse price movements.

2.3 Market Making Strategies

Effective market making involves dynamically adjusting quotes in response to market conditions and inventory levels. This project's agent incorporates:

- **Inventory Skew:** The agent adjusts its bid and ask prices to make it more attractive to trade in a direction that reduces inventory imbalances. If the agent is long (above target inventory), it might lower its bid and ask prices to encourage selling to it and discourage buying from it. This is controlled by parameters like `inventory_skew_factor_`, `skew_power_` (for non-linear response), and `spread_inventory_sensitivity_`.

- **Dynamic Spread and Quantity:** The width of the bid-ask spread and the quantity quoted can also be adjusted based on inventory risk. For instance, a larger inventory deviation might lead to wider spreads or smaller quote sizes to mitigate risk.

These heuristic approaches are inspired by more formal models, such as the Avellaneda-Stoikov model [**?**], which provides a theoretical framework for optimal quoting strategies considering inventory and market volatility.

2.4 Asset Price Dynamics

The `MarketSimulator` class generates a synthetic "fair value" for the traded asset. This underlying price process is modeled using a discrete-time version of an Ornstein-Uhlenbeck process, a common choice for its mean-reverting property [**?**]. The price evolves based on a drift towards a long-term mean (`initial_price_`) and a random shock component scaled by `price_volatility_`:

$$\Delta P_t = \theta(\mu - P_{t-1})\Delta t + \sigma\sqrt{\Delta t}Z_t$$

where P_t is the price at time t, μ is the mean price, θ is the speed of reversion (`mean_reversion_strength_`), σ is the volatility (`price_volatility_`), and Z_t is a standard normal random variable. In the code, Δt is implicitly 1 per simulation step.

2.5 Performance Metrics

To evaluate the agent's strategy, standard performance metrics are calculated, including:

- **Profit and Loss (P&L):** Both realized (from closed trades) and unrealized (from current open positions valued at the current fair value).
- **Sharpe Ratio:** A measure of risk-adjusted return, calculated based on the average P&L per trade and its standard deviation, often annualized.
- **Average Spread Captured:** An indicator of the profitability of the market-making activity per unit traded.

These are computed within the `MarketMakerAgent::get_stats` method.

3 Methods

The simulation is structured as an object-oriented C++ application, promoting modularity and encapsulation. The choice of C++ was driven by its performance characteristics, essential for potentially more complex or high-frequency simulations, and my goal to develop proficiency in the language [3].

3.1 Core Classes and Design

- **`Order` and `Trade`:** These are simple `structs` representing the fundamental data elements: orders placed by participants and trades resulting from matched orders. They store details like price, quantity, side (bid/ask), type (limit/market), timestamp, and unique IDs.
- **`OrderBook`:** This class is the heart of the market.
  - It maintains two `std::maps`: `bids_` (sorted descending by price, `std::greater<double>`) and `asks_` (sorted ascending by price). Each map value is a `std::vector<Order>`, ensuring time priority for orders at the same price level.

- A `std::map<std::string, std::pair<double, size_t>> order_locations_` is used to quickly find an order by its ID for cancellation, storing its price and index within the price level's vector.
- `add_order()`: Adds new limit orders to the book and attempts to match them via `match_limit_orders()`.
- `match_market_order()`: Processes incoming market orders against the resting limit orders on the opposite side.
- `match_limit_orders()`: Checks for and executes crosses between existing bid and ask limit orders (e.g., when a new aggressive limit order is placed).
- `cancel_order()`: Removes an order from the book. The implementation involves swapping the target order with the last order in its price level's vector and then popping, which is efficient.

- **`MarketMakerAgent`:** This class encapsulates the logic of the automated trading agent.
  - `update_quotes(fair_value)`: This is the primary decision-making method. It first cancels existing quotes and then calls `place_quotes`.
  - `place_quotes(fair_value)`: This method implements the dynamic quoting strategy. It calculates bid and ask prices by:
    1. Adjusting a `base_spread_` based on `spread_inventory_sensitivity_` and the agent's deviation from `target_inventory_`.
    2. Applying an inventory skew using `inventory_skew_factor_` and `skew_power_` for non-linear adjustments.
    3. Dynamically sizing quote quantities using `quote_quantity_` and `quantity_inventory_sensitivity_`.

    It then places new limit orders on the `OrderBook`.
  - `on_fill(...)`: Called by the `OrderBook` when one of the agent's orders is (partially or fully) filled. This method updates the agent's `inventory_`, `capital_`, `realized_pnl_`, `avg_entry_price_`, and other statistics like `sum_spread_captured_`.
  - P&L and inventory tracking are core to its state management. `avg_entry_price_` is crucial for calculating realized P&L and current unrealized P&L.

- **`MarketSimulator`:** This class orchestrates the overall simulation.
  - `run_simulation(num_steps)`: Executes the main simulation loop.
  - `run_simulation_step()`: In each step, it:
    1. Updates the asset's fair value using `update_price_model()`.
    2. Instructs the `MarketMakerAgent` to update its quotes.
    3. Potentially generates a random order from other market participants via `generate_random_order()`, which can be a limit or market order.
  - It maintains histories of price, P&L, inventory, and agent quotes for later analysis.

3.2 Event-Driven Flow and Configuration

The simulation operates in discrete time steps. Each step can introduce new information (price changes, new orders), leading to agent reactions and order book updates. This event-driven characteristic mirrors real market behavior. The system is highly configurable via the `SimulationConfig` structure, which allows parameters for the market environment (volatility, order arrival rates) and the agent's strategy to be set either through defaults or a configuration file. This flexibility was a key design choice for enabling experimentation.

4 Results and Insights

This project successfully demonstrates the creation of a functional market simulation environment. The key achievement is not just a running program, but a platform where the intricate dance of market making can be observed and analyzed.

- **Agent Adaptability:** The core insight from observing the agent is its responsiveness. The dynamic quoting logic, which adjusts spread, skew, and quantity based on inventory, clearly shows the agent attempting to manage its risk. When inventory deviates significantly from the target, the agent becomes more conservative or aggressive in specific ways to steer it back, a behavior fundamental to real-world market making.
- **Data-Driven Analysis:** The simulation outputs a CSV file (e.g., `simulation_results_v2.csv`) logging key variables like timestamp, market price, agent P&L, inventory, and the agent's bid/ask quotes. This rich dataset is invaluable. Plotting these time series would allow for visual confirmation of the agent's behavior – for example, observing how its quotes widen or skew as its inventory position changes, or how its P&L evolves over time. I must note that as a newbie to C++ this sometimes does not work but I will try to fix it completely - #SkillIssue.
- **Quantitative Performance Metrics:** The console output at the end of each simulation run provides a snapshot of the agent's performance. Metrics such as Total P&L, Realized P&L, Unrealized P&L, Current Inventory, Max Inventory Held, Total Trades, Average P&L per Trade, P&L Standard Deviation, Sharpe Ratio (annualized), and Average Spread Captured (per unit volume) offer a quantitative basis for evaluating the strategy. While specific values depend heavily on parameter tuning, the ability to generate these metrics is a critical result.
- **Parameter Sensitivity (Qualitative):** Through experimentation (even if not formally documented here), one can observe the impact of the new dynamic parameters. For instance:
  - Increasing `agent_spread_inventory_sensitivity` would likely lead to the agent quoting wider spreads when its inventory is far from the target. This might reduce the frequency of trades but could improve the profitability or reduce the risk of each trade taken under such stressed inventory conditions.
  - A higher `agent_skew_power` would make the price skewing more aggressive for larger inventory imbalances, potentially helping the agent revert to its target inventory more quickly.
  - Adjusting `agent_quantity_inventory_sensitivity` allows control over how much the agent reduces its quoted size when holding risky inventory, directly impacting its market footprint and risk exposure.

  The simulator provides the means to explore these sensitivities systematically.

This project, therefore, serves as a personal laboratory for understanding cause and effect in a simplified market setting. The true "result" is this interactive, observable system itself.

5 Reflection and Conclusion

This project has been an immensely rewarding learning experience. The primary motivation was to develop practical skills, and in that regard, it has been a success.

What I Learned:

- **C++ Proficiency:** I gained significant experience in C++ object-oriented design, memory management (implicitly, through STL containers), file I/O, and the use of standard libraries for data structures and algorithms. The process of structuring the classes and their interactions was a deep dive into practical software engineering.
- **Market Microstructure:** Implementing the LOB and matching engine provided a concrete understanding of price-time priority, the distinction between limit and market orders, and how trades occur.
- **Algorithmic Strategy Design:** Designing the market maker's logic, especially the dynamic adjustments to spread, skew, and quantity, was a fascinating exercise in translating theoretical risk management ideas into code.
- **Simulation Development:** I learned about the challenges of building event-driven simulations, managing state across different components, and generating meaningful data for analysis.

What Was Hard:

- **Debugging Complex Interactions:** The interplay between the `OrderBook`, `MarketMakerAgent`, and `MarketSimulator` sometimes led to subtle bugs that were challenging to trace, particularly in the order matching and P&L accounting logic (e.g., handling partial fills correctly, updating average entry prices).
- **Parameter Tuning:** The system has many parameters. Finding combinations that lead to "sensible" or "interesting" agent behavior and stable market dynamics requires patience and iterative testing. The current set of default parameters represents one such combination, but many others could be explored.
- **P&L Accounting:** Ensuring the realized and unrealized P&L calculations were consistently correct, especially as inventory flipped between long and short, required careful thought and verification.

Contribution to My Growth:
This project has been a significant step in my technical development. It moved concepts from abstract understanding to tangible implementation, building confidence in my ability to tackle complex quantitative problems. The hands-on experience of designing, coding, and debugging a system of this nature is invaluable and has deepened my appreciation for the intricacies of algorithmic trading systems. It has also solidified my interest in pursuing further work in this field. I've tried to structure and format this tex document as best as I can so if there were any hiccups but you stuck through, I appreciate you reading :) !

Future Directions:

- **More Sophisticated Price Models:** Incorporate elements like stochastic volatility or order flow imbalance into the fair value generation.
- **Multiple Agents:** Introduce other types of market participants (e.g., momentum traders, noise traders) to create a more realistic and competitive environment.
- **Advanced Risk Management:** Implement more formal risk controls for the agent, such as Value-at-Risk (VaR) limits or explicit loss limits.

- **Optimization and Calibration:** Use techniques (e.g., parameter sweeps, optimization algorithms) to find optimal parameters for the agent's strategy under different simulated market regimes.
- **Visualization:** Add a simple graphical interface to visualize the order book, agent actions, and P&L in real-time.

Overall, this project has laid a strong foundation for future explorations in quantitative finance and algorithmic trading.

References

[1] Schlögl, E. (2013). *Quantitative Finance: An Object-oriented Approach in C++* (1st ed.). Chapman & Hall/CRC Financial Mathematics Series

[2] Álvaro Cartea, Sebastian Jaimungal, and José Penalva. *Algorithmic and High-Frequency Trading*. Cambridge University Press, 2015.

[3] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.