From Code to Market Dynamics: Algorithmic Trading Simulation with C++

Saxon Lee

10th of September 2021

Abstract

This paper is a walk-through of a personal C++ project that is being undertaken to overcome what myself and peers sometimes call a "skill issue" and foster competence in quantitative finance and software development. The project involved creating a simulation environment for stock price movements using Geometric Brownian Motion and implementing a Simple Moving Average (SMA) crossover trading strategy. The system back-tests this strategy, providing key performance metrics and data for further analysis, thereby offering a practical platform for learning and experimentation. Return metrics may not be the greatest and my HP Omen doesn't quite match HRT/Optiver/JS computing power but that will not stop me from learning, good morning gamers!

1 Introduction

The main motivation behind this project was a personal drive to overcome perceived skill deficits and cultivate competence by throwing myself into waters that I can somewhat tread in. My goal is to achieve steady growth in the technical domain of quantitative finance by building a tangible application from the ground up. This endeavor represents a step towards transforming theoretical knowledge into practical skills.

The project itself is a C++ application designed to simulate stock market price behavior and test a common algorithmic trading strategy. Specifically, it generates synthetic stock price data using the Geometric Brownian Motion model and then applies a Simple Moving Average (SMA) crossover strategy to make trading decisions. The performance of this strategy is then evaluated using a backtest module. This matters because it provides a controlled environment, a sandbox, to understand complex market dynamics, explore the fundamentals of algorithmic trading, and hone software development practices within a quantitative context, without risking real capital. It's a hands-on approach to learning the interplay between financial theories and their computational implementation.

2 Theory & Background

The project leverages several core concepts from quantitative finance and statistics.

2.1 Geometric Brownian Motion (GBM)

Stock prices are often modeled as following a stochastic process known as Geometric Brownian Motion. The rationale is that percentage returns, rather than absolute price changes, are normally distributed. The GBM model is described by the stochastic differential equation:

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

where:

- S_t is the stock price at time t.

- μ is the drift coefficient, representing the expected rate of return.
- σ is the volatility coefficient, representing the standard deviation of returns.
- dt is an infinitesimal time increment.
- dW_t is a Wiener process or Brownian motion term, $dW_t = \epsilon\sqrt{dt}$ where $\epsilon \sim N(0, 1)$.

This model is widely used for its mathematical tractability and its ability to capture the general random walk nature of stock prices [1]. In the simulation, this is discretized as:

$$S_{t+\Delta t} = S_t \exp\left((\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma\epsilon\sqrt{\Delta t}\right)$$

2.2 Simple Moving Average (SMA)

A Simple Moving Average is a trend-following indicator that calculates the average price of a security over a specified number of periods. It helps to smooth out price data to identify the direction of the trend. An N-day SMA is calculated as:

$$\text{SMA}_N = \frac{P_1 + P_2 + \cdots + P_N}{N}$$

where P_i is the price at period i.

2.3 SMA Crossover Strategy

This project implements a common trading strategy based on the crossover of two SMAs: a short-term SMA and a long-term SMA.

- **Buy Signal**: Generated when the short-term SMA crosses above the long-term SMA. This suggests increasing upward momentum.
- **Sell Signal**: Generated when the short-term SMA crosses below the long-term SMA. This suggests increasing downward momentum.

This strategy aims to capture profits from sustained trends.

2.4 Performance Metrics

To evaluate the strategy, several standard performance metrics are calculated:

- **Total Return**: The overall percentage gain or loss of the portfolio over the backtesting period.
- **Sharpe Ratio**: Measures the risk-adjusted return. It is calculated as the average return earned in excess of the risk-free rate per unit of volatility or total risk [**?**]. A higher Sharpe Ratio generally indicates better performance for the risk taken. The formula used is:

$$\text{Sharpe Ratio} = \frac{R_p - R_f}{\sigma_p}$$

where R_p is the (annualized) portfolio return, R_f is the (annualized) risk-free rate, and σ_p is the (annualized) standard deviation of the portfolio's excess returns.

- **Maximum Drawdown (MDD)**: The largest peak-to-trough decline during a specific period of an investment. It is an indicator of downside risk.

$$\text{MDD} = \max_{t\in(0,T)}\left(\frac{\text{PeakValueBefore}(t) - \text{PortfolioValue}(t)}{\text{PeakValueBefore}(t)}\right)$$

3 Methods

The project is implemented in C++ and structured into several classes and a utility namespace to promote modularity and clarity.

3.1 Code Structure

- **`StockSimulator` Class**: Responsible for generating synthetic stock price data. Its core method, `generateGBMPath`, takes initial price (`S0`), drift (`mu`), volatility (`sigma`), time horizon (`T`), and number of steps (`numSteps`) as input. It uses `std::mt19937` for pseudo-random number generation and `std::normal_distribution` to simulate the stochastic component of GBM.
- **`MovingAverageCalculator` Class**: This class provides a utility function, `calculateSMA`, which computes the Simple Moving Average for a given vector of prices and a specified window size. It efficiently calculates the SMA by sliding the window across the price series.
- **`TradingStrategy` Class**: This class encapsulates the logic for the SMA crossover strategy. Its `generateSignals` method takes a price path and, using the `MovingAverageCalculator`, computes short-term and long-term SMAs. It then compares these SMAs at each step to generate `BUY`, `SELL`, or `HOLD` signals. The class constructor takes the window lengths for the short and long SMAs. Careful checks are included to ensure SMAs are valid (i.e., enough data points exist) before generating signals.
- **`Backtester` Class**: This class simulates the execution of trades based on the signals generated by the `TradingStrategy`. Its `runBacktest` method takes the price path, signals, and initial cash as input. It tracks portfolio value, cash, and shares held over time. It assumes a simple trading logic: buy with all available cash when a BUY signal occurs (if not already holding shares), and sell all shares when a SELL signal occurs (if shares are held). At the end of the simulation, any open positions are liquidated. The class calculates total return, number of trades, Sharpe ratio, and maximum drawdown, returning these in a `BacktestResult` struct.
- **`Utility` Namespace**: Contains helper functions:
  - `writeToCSV`: Exports the price path, SMAs, signals, portfolio values, daily Profit and Loss (PnL), and drawdown series to a CSV file. This allows for external plotting and more detailed analysis. It handles cases where SMA data might not be available for initial periods by writing ”N/A”.
  - `calculateMaxDrawdown`: Computes the maximum drawdown from a series of portfolio values.
  - `calculateDrawdownSeries`: Computes the drawdown percentage at each point in time for the portfolio values.
  - `calculateSharpeRatio`: Calculates the annualized Sharpe ratio, assuming daily returns and a configurable risk-free rate (hardcoded to 2% in this version).
- **`SignalType` Enum**: Defines the possible trading signals: `BUY`, `SELL`, `HOLD`.
- **`main()` Function**: Orchestrates the entire simulation. It sets parameters for the GBM simulation (initial price, drift, volatility, time horizon, steps), trading strategy (SMA window lengths), and backtesting (initial cash). It then instantiates the necessary objects, runs the simulation and backtest, prints summary statistics to the console, and triggers the CSV export.

3.2 Design Choices

- **Object-Oriented Programming**: The use of classes (`StockSimulator`, `MovingAverageCalculator`, `TradingStrategy`, `Backtester`) helps in organizing code, making it more readable, maintainable, and extensible. Each class has a distinct responsibility.
- **Standard C++ Libraries**: The project leverages standard C++ libraries like `<vector>` for dynamic arrays, `<random>` for stochastic simulations, `<cmath>` for mathematical functions, `<numeric>` for `std::accumulate`, `<fstream>` for file I/O, and `<iomanip>` for formatted output. This promotes portability and uses well-tested components.

- **Data Export for External Analysis**: Generating a CSV file (`sma_crossover_backtest.csv`) is a key design choice. It decouples the simulation logic from visualization, allowing users to employ powerful external tools like Python with Matplotlib/Seaborn, R, or spreadsheet software for in-depth analysis and charting of results.
- **Parameterization**: Key parameters for the simulation and strategy are defined in `main()`, making it easy to experiment with different market conditions or strategy configurations.
- **Basic Error Handling**: The code includes some basic checks, such as for file opening errors and division by zero in financial calculations. For instance, SMA values are initialized to 0.0 and treated as "N/A" in the CSV if they fall before the minimum window period, preventing calculations with insufficient data.

4 Results and Insights

The project successfully creates a functional framework for simulating stock price data and backtesting a simple trading strategy. The primary output is twofold:

1. **Console Summary**: The program prints key performance metrics to the console after a simulation run. For the default parameters (`S0=100`, `mu=0.08`, `sigma=0.20`, `T=1 year`, `252 steps`, `shortWindow=10`, `longWindow=30`, `initialCash=$100,000`):
   - Initial and Final Portfolio Value
   - Total Return (%)
   - Number of Trades
   - Sharpe Ratio
   - Maximum Drawdown (%)

   The exact values of these metrics will vary with each run due to the stochastic nature of the GBM simulation, even with fixed parameters, unless a fixed seed is used for the random number generator (which is done by default using `std::random_device{}()`, but could be fixed for reproducibility).
2. **CSV Output (`sma_crossover_backtest.csv`)**: This file contains time-series data for:
   - Day
   - Price
   - Short-term SMA
   - Long-term SMA
   - Signal (BUY, SELL, HOLD)
   - Portfolio Value
   - Daily PnL
   - Drawdown Percent

   This detailed output is crucial for visualizing the strategy's behavior, such as plotting the price series with SMA overlays and buy/sell markers, or observing the portfolio's equity curve and drawdown periods.

Insights Gained:

- The project demonstrates the mechanics of a complete, albeit simplified, algorithmic trading system development cycle: from market simulation to strategy implementation, signal generation, and performance evaluation.

- It provides a tangible way to understand how changing parameters (e.g., SMA window lengths, market volatility σ, or drift μ) can impact strategy performance. For example, one could run multiple simulations to see how often the strategy is profitable under different conditions.
- The process of implementing calculations like the Sharpe Ratio and Maximum Drawdown provides a deeper appreciation for these risk and performance measures.
- The framework itself is an achievement, offering a base upon which more complex strategies or market models could be built.

While this project doesn't aim to find a "holy grail" trading strategy (especially with simulated data and a simple strategy), it serves as an excellent educational tool for exploring quantitative trading concepts.

5 Reflection and Conclusion

This project has been an invaluable learning experience. The journey of conceptualizing, designing, and implementing this trading simulator in C++ has significantly contributed to my understanding of both software engineering principles and fundamental concepts in quantitative finance.

What I Learned:

- **Practical C++ Application**: Beyond textbook exercises, I applied C++ skills to a moderately complex problem, involving OOP design, standard library usage (especially containers, algorithms, and random number generation), and file I/O.
- **Financial Modeling Implementation**: Translating mathematical models like GBM and trading rules like SMA crossovers into working code was a core learning outcome.
- **Quantitative Analysis Basics**: I gained hands-on experience with calculating and interpreting key performance metrics such as Sharpe Ratio and Maximum Drawdown.
- **Importance of Modularity and Debugging**: Structuring the code into logical classes made development and debugging more manageable. The iterative process of testing, identifying bugs (e.g., in SMA calculation edge cases or backtesting logic), and fixing them was a significant part of the learning.

Challenges Encountered:

- **Ensuring Correctness**: Verifying the correctness of financial calculations, especially the iterative updates in the backtester and the SMA computations, required careful attention to detail and systematic testing. For instance, handling the initial periods before SMAs are fully formed, or ensuring the portfolio value updates correctly after trades, needed careful thought.
- **Managing Data Flow**: Coordinating the flow of data (prices, SMAs, signals, portfolio values) between the different classes and ensuring consistency was a non-trivial aspect.
- **Stochastic Nature**: Understanding that results would vary with each run of the GBM simulation (without a fixed seed) and how that impacts the interpretation of a single backtest result was an important realization.

Personal Growth and Future Directions: This project has solidified my interest in the intersection of finance and technology. It has provided me with a foundational C++ codebase and a deeper intuition for how algorithmic strategies are developed and tested. This experience directly addresses my initial goals of overcoming skill gaps and growing my technical competence.

Looking ahead, there are several ways this project could be extended or improved:

- **More Sophisticated Strategies**: Implement other trading strategies (e.g., mean reversion, momentum indicators like RSI or MACD).

- **Risk Management**: Incorporate more advanced risk management rules (e.g., stop-loss orders, position sizing).
- **Parameter Optimization**: Add a module to systematically test a range of strategy parameters (e.g., SMA window lengths) to find optimal settings, though one must be wary of overfitting.
- **Real Market Data**: Adapt the system to use historical market data instead of simulated GBM paths.
- **User Interface**: Develop a simple graphical user interface (GUI) for easier interaction and visualization, perhaps using a library like Qt or by creating a web interface.
- **Enhanced Statistical Analysis**: Include more statistical tests for the significance of results.

In conclusion, this project has been a rewarding endeavor that I have learned a lot from. It serves as nice a stepping stone for further exploration in quantitative trading .

References

[1] Hull, J. C. (2018). *Options, Futures, and Other Derivatives* (10th ed.). Pearson Education.

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