Authors:
Antoine BERGIER
Nils LELORIEUX
Alexandre ROGOJAN

Course: C++ Project

Professor: Miss DUMITRESCU

December 29, 2025

Contents

1 Description, Usage, and Structure

1.1 Project Description

This project implements a C++ pricer for path-dependent Asian options using Monte Carlo simulations. It calculates the price of Asian Call and Put options based on the arithmetic average of the underlying asset's price path, modeled via Geometric Brownian Motion.

The main capabilities of the software include the valuation of Asian Call and Put options, and the computation of the full suite of Greeks: Delta, Vega, Gamma, Rho, and Theta. It also features integrated plotting of price surfaces using `Gnuplot`. The simulation is optimized using the C++ standard library's random number generation and vector arithmetic.

1.2 Usage Instructions

The project is designed to be easily executed on Windows environments without complex local dependencies (other than the included MinGW compiler and Gnuplot).

To run the project, locate the file 'run_project.bat' in the root directory and execute it (either by double-clicking or running it from the command line).

```
> run_project.bat
```

This script automatically checks for the compiler, compiles the project using 'g++' with static linking, and launches the 'pricer.exe' executable.

Once running, the application offers two modes:

- **Manual Mode**: Enter parameters manually (Spot, Rate, Volatility, etc.).
- **Real-time Mode**: Enter a stock ticker (e.g., AAPL). The program connects to the Yahoo Finance API to fetch the latest spot price and historical volatility automatically.

Finally, follow the on-screen prompts to configure simulation steps and select the Greeks to visualize.

1.3 Project Structure

The codebase is organized into logical modules to separate concerns:

`Math/` Contains the core mathematical models.

- `StochasticModel.hpp/cpp`: Implements the Geometric Brownian Motion path generation and the Monte Carlo engine.
- `Greeks.hpp/cpp`: Functions to calculate Delta, Vega, Gamma, Rho, and Theta using finite difference methods.

`MarketData/` Handles external data retrieval.

- `MarketData.hpp`: Implements the HTTP client and JSON parsing for Yahoo Finance.

`Instruments/` Defines the financial instruments.

- `Payoff.hpp/cpp`: Payoff functions for Asian Call/Put (and European variants).

`Graphics/` Handles visualization.

- `plot.hpp/cpp`: Interface to Gnuplot to render 3D surfaces and dashboards.
- `gnuplot/`: Contains the local Gnuplot binary.

`App/` Encapsulates the application logic.

- `PricerApp.hpp/cpp`: Manages the application lifecycle, user input handling, and orchestration of the Monte Carlo simulation and plotting.

`main/` Application entry point.

- `main.cpp`: A lightweight wrapper that instantiates and runs the `PricerApp`.

2 Review of Problems Encountered

(This section describes the challenges faced during development.)

One of the main challenges was coordinating the work across the team. Working simultaneously on the same codebase led to occasional merge conflicts, and integrating different features required regular communication to ensure architectural consistency.

Configuring the build system to work portably across different Windows machines was also difficult. We resolved this by including a local MinGW distribution and using a batch script for automatic path configuration. Additionally, piping data correctly to Gnuplot for 3D surface rendering required careful formatting of the data streams.

Finally, computing Θ (Theta) and ρ (Rho) using finite difference methods initially produced inaccurate results. For Theta, choosing the right time step dt was tricky: too large a step introduced bias, while too small a step amplified Monte Carlo noise. For Rho, the low sensitivity of the price to interest rates made calculations unstable. Use of variance reduction techniques (identical random seeds for perturbed paths) helped, though these Greeks remain more sensitive to noise than Delta or Vega.

3 General Architecture

(This section details the software design decisions and technical justifications.)

The architecture of this project was driven by three primary goals: performance, portability, and simplicity. We wanted a codebase that is straightforward to compile on any Windows machine while being efficient enough to run heavy Monte Carlo simulations.

3.1 Global Architecture & Portability

The project relies on a modular structure with clear separation of concerns: the Math module handles the stochastic calculus, Instruments defines the financial contracts, and Graphics manages visualization.

A key challenge in C++ development is distributing executables that run on any machine. To solve this, we used MinGW64 with static linking flags (`-static`). This ensures that all necessary C++ runtime libraries are bundled directly into the executable, preventing the common "DLL missing" errors that often plague academic projects when moved between computers.

3.2 Mathematical Engine: Precision & Performance

At the heart of the pricer lies the Monte Carlo engine. Financial simulations require generating billions of random numbers, and the quality of these numbers directly impacts the pricing accuracy. We explicitly rejected the standard C `rand()` function because of its known statistical flaws and short period. Instead, we implemented the Mersenne Twister 19937 engine (`std::mt19937`), which provides the high-quality randomness essential for robust financial modeling.

Performance was also a critical consideration. In our implementation of `StochasticModel.cpp`, we pay close attention to memory management. Specifically, we allocate the path vector *outside* the main simulation loop and pass it by reference. This avoids the distinct overhead of allocating and deallocating memory millions of times, which would otherwise significantly slow down the pricing process.

3.3 Financial Instruments: Function Pointers vs OOP

When designing the financial instruments, we had to choose between a classic Object-Oriented approach (with inheritance and polymorphism) and a more functional C-style approach.

We opted for function pointers. While polymorphism is a powerful tool, virtual function calls incur a small CPU cost due to vtable lookups. In a tight loop running millions of iterations, these small costs accumulate. By using function pointers for our payoffs, we achieve a direct and faster execution path. This also keeps our code structure simpler and lighter compared to a deep class hierarchy.

3.4 Visualization: Lightweight Inter-Process Communication

Finally, for visualization, we wanted to avoid the complexity of linking heavy GUI libraries like Qt, which can be difficult to configure. Instead, we used a lightweight inter-process communication strategy: Piping.

Our program spawns a `Gnuplot` subprocess and sends commands via standard input. To optimize performance, we use Gnuplot's "Data Blocks" feature. We transmit the simulation data only once, store it in memory within Gnuplot, and then issue multiple commands to render different views. This approach gives us powerful 3D visualization capabilities with minimal code and zero external library dependencies.

3.5 Market Data Module: Windows API & JSON

To enable real-time features without bloating the project with heavy dependencies like `libcurl`, we implemented a lightweight HTTP client using the native Windows `WinInet` API. This allows the application to directly fetch JSON data from Yahoo Finance.

For parsing the response, we utilize the header-only `nlohmann/json` library. This combination ensures that the "automatic mode" works out-of-the-box on any standard Windows development environment, maintaining our goal of maximum portability.