

# Monte Carlo Final Project

1. Introduction.....	2
2. Overview of Design Patterns .....	3
3. MCMediator .....	5
4. MCSolver .....	6
5. SimulationBuilder .....	7
6. SDE.....	9
7. FDM .....	10
8. RNG.....	11
9. Payoff .....	12
10. StopWatch.....	13
11. Results.....	14

# 1. Introduction

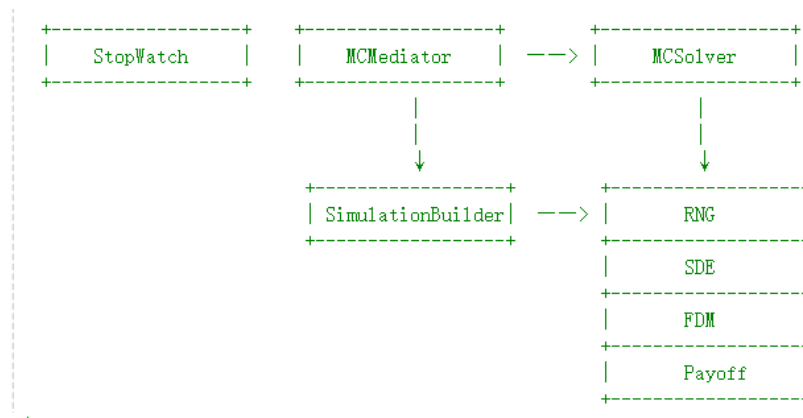
In the field of financial engineering and quantitative finance, **Monte Carlo Simulation** is a widely used numerical method for solving complex financial derivative pricing problems. This method estimates the price of financial products through random sampling and statistical simulation, particularly for complex option pricing problems that cannot be solved analytically. The core idea of Monte Carlo simulation is to generate a large number of random paths to simulate the future behavior of asset prices and calculate the expected payoff of options based on these paths.

This program implements a **Monte Carlo simulation-based option pricing framework**, designed to provide a flexible and efficient pricing tool for various types of financial derivatives, such as European options, Asian options, barrier options, and more. The core components of the program include:

1. **Stochastic Differential Equation (SDE)**: Used to describe the stochastic processes of asset prices, such as Geometric Brownian Motion (GBM), Constant Elasticity of Variance (CEV) model, and Cox-Ingersoll-Ross (CIR) model.
2. **Finite Difference Method (FDM)**: Used for numerically solving stochastic differential equations, supporting Euler method, Milstein method, and drift-adjusted predictor-corrector method.
3. **Random Number Generator (RNG)**: Based on the Mersenne Twister algorithm, it generates high-quality random numbers to ensure the accuracy of simulations.
4. **Payoff**: Define the payoff calculations for different types of options, supporting European options, Asian options, barrier options, and more.
5. **SimulationBuilder**: Used to configure and build the components of the Monte Carlo simulation, providing both interactive user configuration and direct setup methods.
6. **MCSolver**: Responsible for executing the Monte Carlo simulation and computing the expected price of the option.
7. **MCMediator**: Acts as a bridge between the simulation builder and the solver, simplifying the configuration and execution of simulations.

By modularizing these components, the program can flexibly address various option pricing problems and easily extend to new stochastic models, numerical methods, and payoff functions. Additionally, the program provides a **high-precision timer (StopWatch)** to measure the execution time of simulations, helping users evaluate performance under different configurations.

## 2. Overview of Design Patterns



*Figure 1 Overview of the Design*

In this program, we have employed several classic design patterns to build a flexible, extensible, and maintainable Monte Carlo simulation framework. These design patterns not only help organize the code structure effectively but also enhance the modularity and reusability of the system. Below are the main design patterns used in this program and their roles:

### 1. Mediator Pattern

- The Mediator Pattern is used to reduce direct dependencies between multiple classes or objects by introducing a mediator class to coordinate their interactions. In this program, the **MCMediator** class acts as the mediator.
- The **MCMediator** is responsible for coordinating the interaction between **SimulationBuilder** and **MCSolver**. It retrieves simulation configuration information from **SimulationBuilder**, uses this information to initialize **MCSolver**, and finally runs the simulation. In this way, **SimulationBuilder** and **MCSolver** do not need to interact directly but communicate through **MCMediator**.

### 2. Builder Pattern

- The Builder Pattern is a creational design pattern used to construct complex objects step by step. It allows for the flexible construction of objects and can alter their internal representation.
- In this program, the **SimulationBuilder** class acts as the builder. It is responsible for gradually constructing a complex simulation configuration object. **SimulationBuilder** provides multiple *set* methods (e.g., *setSDE*, *setFDM*, *setRNG*, *setPayoff*, etc.), allowing users to configure various components of the simulation (such as SDE, FDM, RNG, and Payoff) step by step. Finally, the *build* method returns a complete configuration object.

### 3. Strategy Pattern

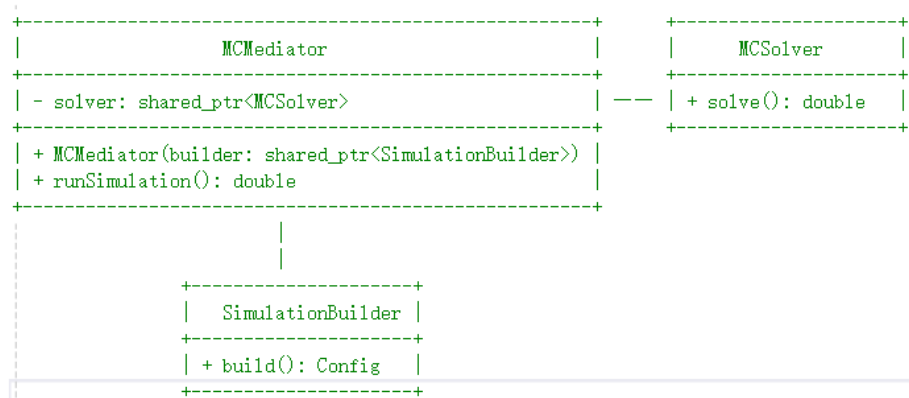
- The Strategy Pattern is used to define a family of algorithms or behaviors, encapsulating them in separate classes so that they can be interchanged. In this program, the **FDM** class hierarchy (including **EulerMethod**, **MilsteinMethod**, and **DriftAdjustedPredictorCorrector**) is a typical application of the Strategy Pattern.

- Users can choose different numerical solution strategies (i.e., different FDM methods) to solve SDEs as needed. Each FDM class implements the `advance` method, providing different numerical solution algorithms.

#### 4. **Factory Method Pattern**

- The Factory Method Pattern is a creational design pattern that defines an interface for creating objects but let subclasses decide which class to instantiate. In this program, the *selectSDE*, *selectFDM*, *selectRNG*, and *selectPayoff* methods in the **SimulationBuilder** class can be seen as factory methods.
- These methods create different object instances (e.g., different SDE, FDM, RNG, and Payoff objects) based on user choices, thereby separating the object creation logic from the usage logic.

### 3. MCMediator



**Figure 2 MCMediator**

The **MCMediator** class acts as an intermediary between the **SimulationBuilder** and the **MCSolver** in the Monte Carlo simulation framework. Its primary responsibility is to simplify the interaction between the builder and the solver, providing a clean and straightforward interface for running Monte Carlo simulations to compute option prices.

#### Key Responsibilities:

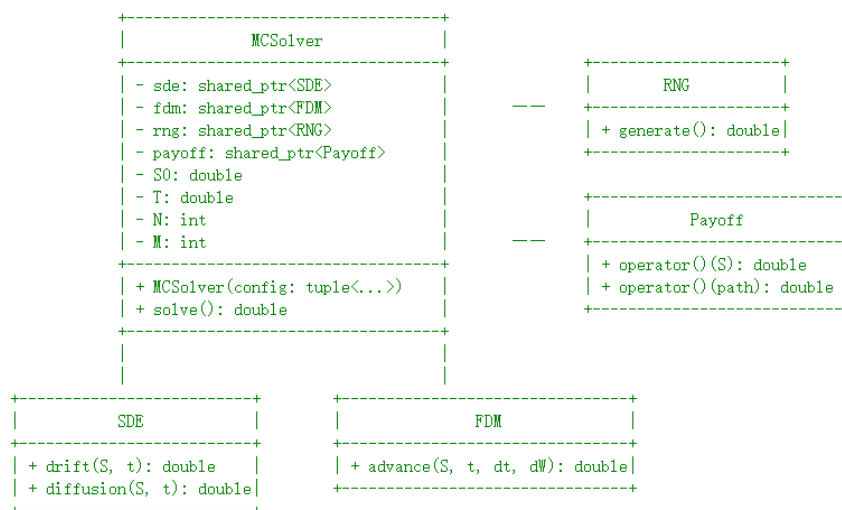
##### 1. Construction of the Monte Carlo Solver:

- The **MCMediator** class takes a **SimulationBuilder** object as input during its construction. It retrieves the simulation configuration (including SDE, FDM, RNG, Payoff, and initial conditions) from the builder using the *build()* method.
- Using this configuration, the **MCMediator** initializes an instance of the **MCSolver**, which is responsible for performing the actual Monte Carlo simulation.

##### 2. Running the Simulation:

- The *runSimulation()* method is the main interface provided by **MCMediator** to execute the Monte Carlo simulation. It delegates the simulation task to the **MCSolver** by calling its *solve()* method.
- The result of the simulation (i.e., the computed option price) is returned to the caller.

# 4. MCSolver



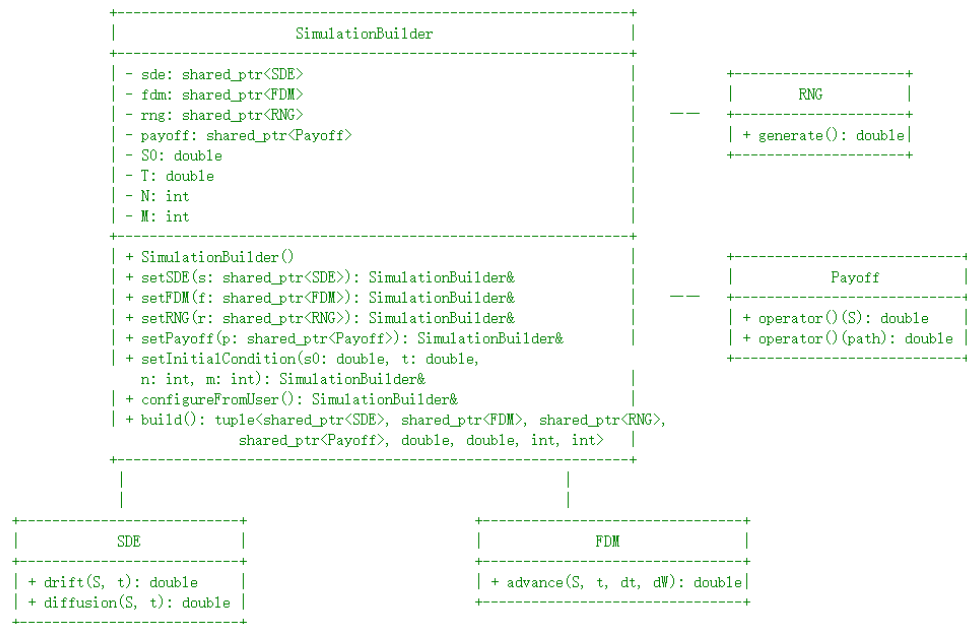
**Figure 3 MCSolver**

The **MCSolver** class is the core component of the Monte Carlo simulation framework, responsible for simulating stochastic differential equations (SDEs) and computing option prices. It integrates components for stochastic modeling (SDE), numerical methods (FDM), random number generation (RNG), and payoff calculations to simulate asset price paths and calculate option values. The solver supports both standard options (e.g., European options) and path-dependent options (e.g., Asian options).

## Key Responsibilities:

- Simulation of Stochastic Differential Equations (SDEs):**
  - The **MCSolver** uses numerical methods (**FDM**) to advance the solution of the SDE over time, simulating asset price paths.
  - It leverages random number generation (**RNG**) to simulate Wiener process increments, which are essential for modeling stochastic processes.
- Option Price Calculation:**
  - The solver calculates the option price by simulating multiple asset price paths and computing the average payoff based on the specified payoff function.
  - It supports both standard options (using the final asset price) and path-dependent options (using the entire price path).
- Validation of Input Parameters:**
  - The **MCSolver** validates the input configuration (**SDE**, **FDM**, **RNG**, **Payoff**, and initial conditions) to ensure all components are properly initialized and the parameters are valid.

## 5. SimulationBuilder



**Figure 4 SimulationBuilder**

The **SimulationBuilder** class is responsible for constructing and configuring the components required for a Monte Carlo simulation. It provides a flexible interface for selecting and setting up the Stochastic Differential Equation (**SDE**), Finite Difference Method (**FDM**), Random Number Generator (**RNG**), and Payoff function, as well as the initial conditions for the simulation. The class supports both direct configuration through setter methods and an interactive user configuration mode.

### Key Responsibilities:

#### 1. Configuration of Simulation Components:

- The **SimulationBuilder** allows users to configure the SDE, FDM, RNG, and Payoff components, which are essential for running a Monte Carlo simulation.
- It provides setter methods (*setSDE*, *setFDM*, *setRNG*, *setPayoff*) for direct configuration of these components.

#### 2. Setting Initial Conditions:

- The class handles the initial conditions for the simulation, such as the initial stock price (**S0**), maturity (**T**), number of time steps (**N**), and number of simulations (**M**).
- The *setInitialCondition* method ensures that these parameters are valid and positive.

#### 3. Interactive User Configuration:

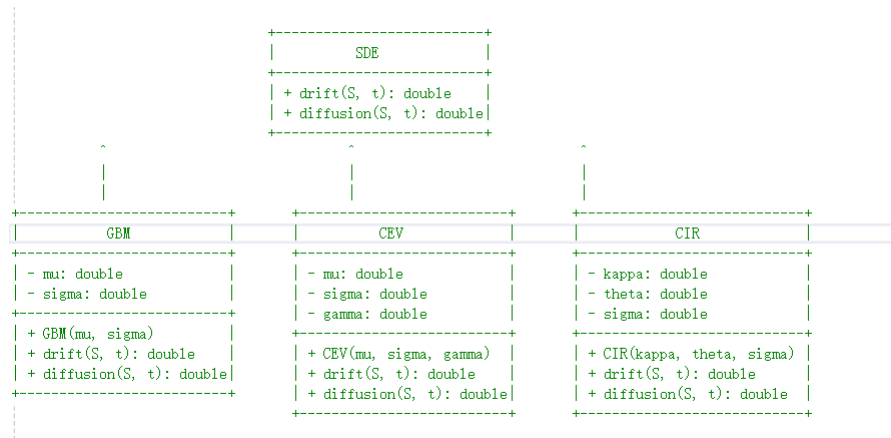
- The *configureFromUser* method provides an interactive way for users to input simulation parameters and select components through a command-line interface.
- It guides the user through selecting the **SDE**, **FDM**, **RNG**, and **Payoff** function, and prompts for necessary parameters like strike price and barrier level.

4. **Building the Simulation Configuration:**

- The build method finalizes the configuration and returns a tuple containing all the components and initial conditions needed to run the simulation.
- It validates that all components and initial conditions are properly set before returning the configuration.



## 6. SDE



**Figure 5 SDE**

The **SDE** (Stochastic Differential Equation) class hierarchy is designed to model various stochastic processes commonly used in financial mathematics. It provides **an abstract base class** (SDE) with virtual methods for computing the drift and diffusion terms of an SDE, along with three derived classes: **GBM** (Geometric Brownian Motion), **CEV** (Constant Elasticity of Variance), and **CIR** (Cox-Ingersoll-Ross).

### Key Responsibilities:

#### 1. Modeling Stochastic Processes:

- The SDE class hierarchy provides a framework for modeling stochastic processes by defining the drift and diffusion terms of the SDE.
- Each derived class (**GBM**, **CEV**, **CIR**) implements specific stochastic models used in financial mathematics.

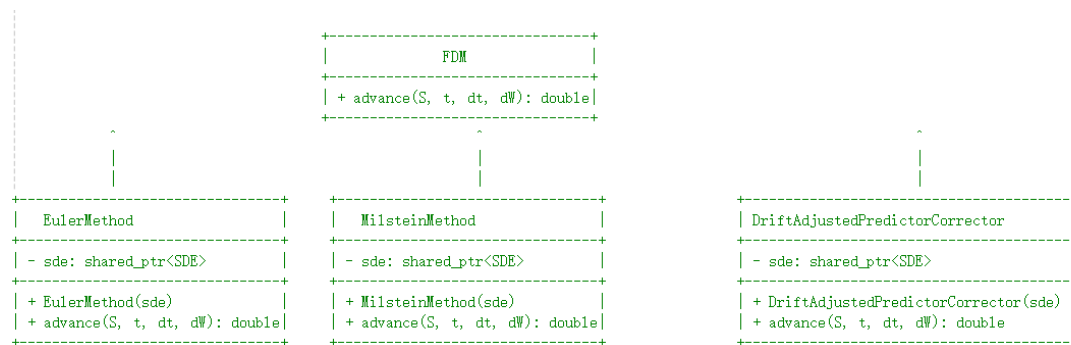
#### 2. Drift and Diffusion Terms:

- The drift method computes the deterministic component of the SDE, representing the expected change in the process over time.
- The diffusion method computes the stochastic component of the SDE, representing the random fluctuations in the process.

#### 3. Support for Multiple Models:

- The hierarchy supports multiple stochastic models, including:
  - GBM**: Models asset prices with constant drift and volatility.
  - CEV**: Models asset prices with a volatility that depends on the asset price raised to a power (gamma).
  - CIR**: Models interest rates with mean reversion and a volatility that depends on the square root of the rate.

# 7. FDM



**Figure 6 FDM**

The **FDM** (Finite Difference Method) class hierarchy provides numerical methods for solving Stochastic Differential Equations (SDEs). It defines **an abstract base class (FDM)** with a virtual method for advancing the solution of an SDE, and three derived classes: **EulerMethod**, **MilsteinMethod**, and **DriftAdjustedPredictorCorrector**. These methods are commonly used in financial mathematics for simulating asset price paths under stochastic models.

## Key Responsibilities:

### 1. Numerical Solution of SDEs:

- The FDM class hierarchy provides numerical methods to advance the solution of an SDE over time.
- Each derived class implements a specific numerical scheme for approximating the solution of the SDE.

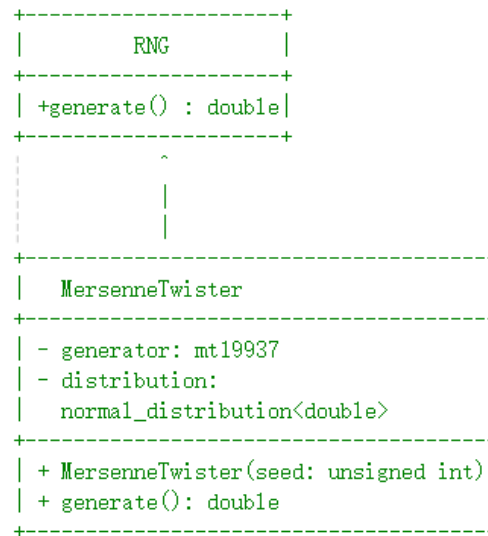
### 2. Advancing the Solution:

- The advance method computes the next state of the system based on the current state (S), time (t), time step (dt), and Wiener process increment (dW).
- This method is essential for simulating asset price paths in Monte Carlo simulations.

### 3. Support for Multiple Numerical Methods:

- The hierarchy supports multiple numerical methods, including:
  - **Euler Method:** A simple and widely used method for approximating SDE solutions.
  - **Milstein Method:** An improved method that includes a correction term for better accuracy.
  - **Drift-Adjusted Predictor-Corrector:** A more advanced method that uses a predictor-corrector approach to improve stability and accuracy.

## 8. RNG



*Figure 7 RNG*

The **RNG** (Random Number Generator) class hierarchy is designed to generate high-quality random numbers for use in simulations, Monte Carlo methods, and other applications requiring stochastic processes. It defines an **abstract base class (RNG)** with a virtual method for generating random numbers, and a derived class (**MersenneTwister**) that implements the Mersenne Twister algorithm, a widely used pseudorandom number generator known for its high-quality random numbers and long period.

### Key Responsibilities:

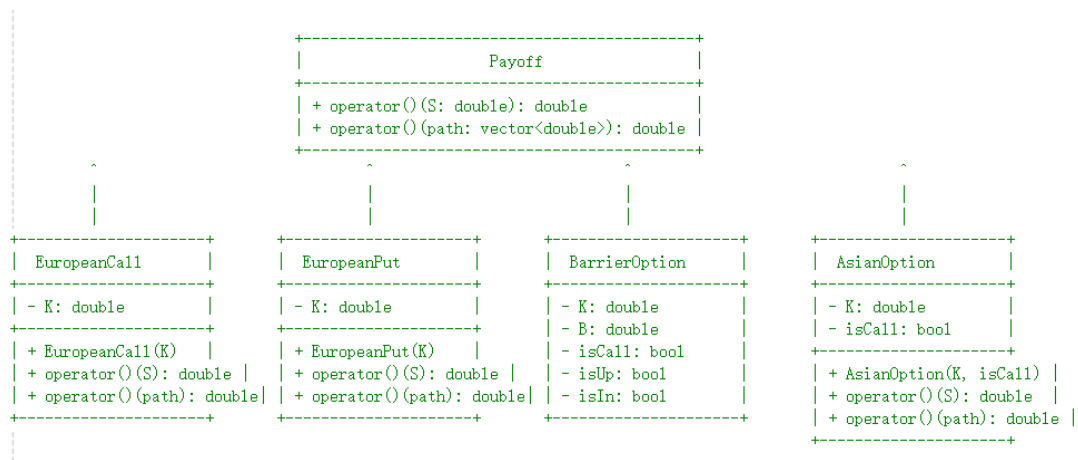
#### 1. Generating Random Numbers:

- The **RNG** class hierarchy provides an interface for generating random numbers, which are essential for simulating stochastic processes such as Wiener process increments in Monte Carlo simulations.
- The generate method returns a random number from a specified distribution (e.g., standard normal distribution).

#### 2. Support for High-Quality Random Numbers:

- The **MersenneTwister** class implements the Mersenne Twister algorithm, which is known for its high-quality random numbers and long period, making it suitable for simulations requiring a large number of random values.

## 9. Payoff



**Figure 8 Payoff**

The **Payoff** class hierarchy is designed to calculate the payoff of various financial options, including standard options (e.g., European options) and path-dependent options (e.g., Asian and Barrier options). It defines an **abstract base class (Payoff)** with virtual methods for calculating the payoff based on a single price or a price path, and derived classes (**EuropeanCall**, **EuropeanPut**, **BarrierOption**, and **AsianOption**) that implement specific payoff calculations for different types of options.

### Key Responsibilities:

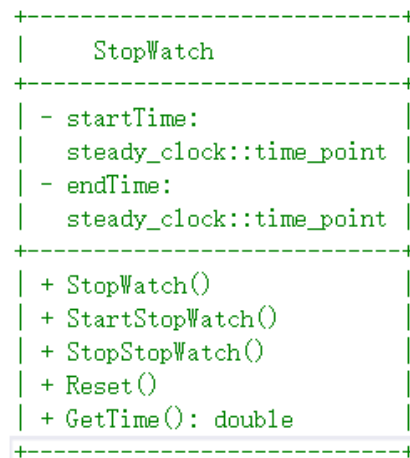
#### 1. Calculating Option Payoffs:

- The **Payoff** class hierarchy provides methods to calculate the payoff of financial options based on the underlying asset's price or price path.
- The *operator()* method is overloaded to handle both single-price payoffs (for standard options) and path-dependent payoffs (for options like Asian and Barrier options).

#### 2. Support for Different Option Types:

- The hierarchy supports a variety of option types, including:
  - European Options:** Payoff depends on the underlying asset's price at maturity.
  - Barrier Options:** Payoff depends on whether the underlying asset's price crosses a specified barrier level.
  - Asian Options:** Payoff depends on the average price of the underlying asset over a specified period.

# 10. Stopwatch



**Figure 9 Stopwatch**

The **StopWatch** class provides a simple and high-resolution timer for measuring elapsed time in C++ applications. It uses the C++11 `<chrono>` library to track time with high precision, making it suitable for performance profiling, benchmarking, and other timing-related tasks. The class supports starting, stopping, resetting the timer, and retrieving the elapsed time in seconds.

## Key Responsibilities:

### 1. Measuring Elapsed Time:

- The **StopWatch** class allows users to measure the time elapsed between two events (e.g., the start and end of a simulation or function execution).
- It provides high-resolution timing using the `std::chrono::steady_clock` from the C++11 standard library.

### 2. Starting and Stopping the Timer:

- The `StartStopWatch` method starts the timer by recording the current time.
- The `StopStopWatch` method stops the timer by recording the current time.

### 3. Resetting the Timer:

- The `Reset` method resets the timer to its initial state, allowing it to be reused for multiple timing measurements.

### 4. Retrieving Elapsed Time:

- The `GetTime` method returns the elapsed time in seconds between the start and stop events.

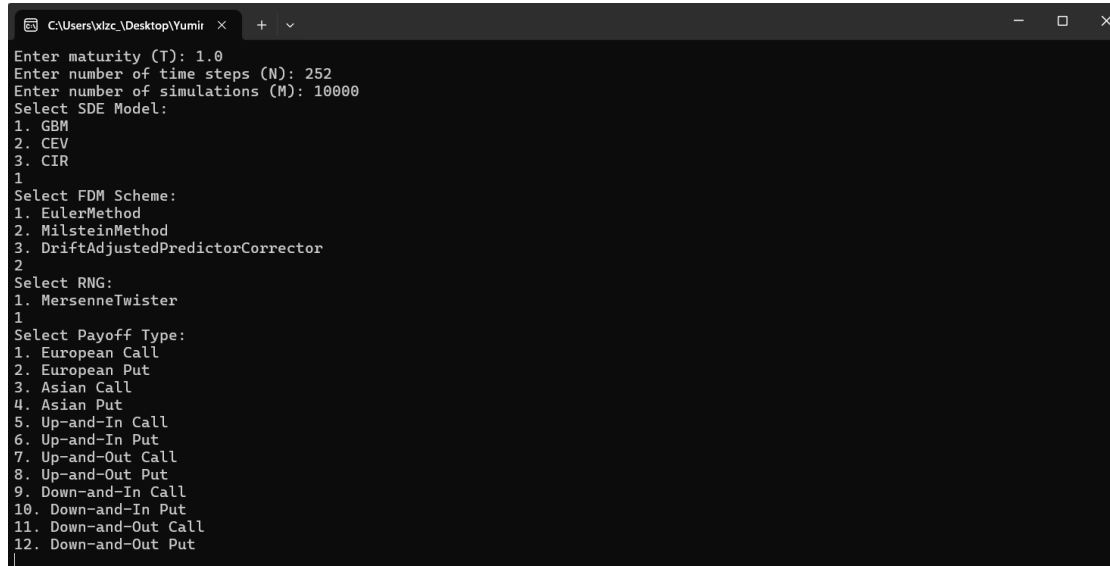
# 11. Results

This test program demonstrates the capabilities of a Monte Carlo simulation framework by testing various configurations of option pricing models.

The program consists of three main test functions, each designed to evaluate different aspects of the Monte Carlo simulation framework:

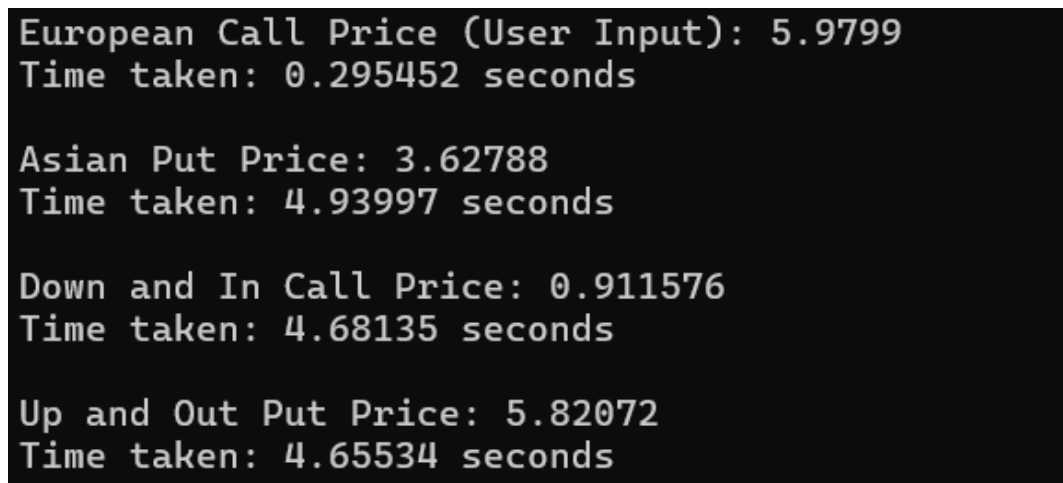
1. **testDifferentOptions:** This function tests the pricing of various option types using the Monte Carlo simulation. It includes examples of a European Call (configured with user input), an Asian Put, a Down-and-In Call, and an Up-and-Out Put. Each option is priced using predefined parameters, and the execution time for each simulation is measured to assess performance. This function demonstrates the framework's ability to handle different payoff structures and option types.
2. **testDifferentFDM:** This function evaluates the impact of different Finite Difference Method (FDM) schemes on the accuracy and performance of the simulation. It compares the Euler method and the Milstein method for pricing a European Call option. By running simulations with both methods, the function highlights the differences in computational efficiency and potential accuracy improvements offered by higher-order FDM schemes.
3. **testDifferentSDE:** This function explores the use of different Stochastic Differential Equation (SDE) models for pricing an Asian Put option. It tests three SDE models: Geometric Brownian Motion (GBM), Cox-Ingersoll-Ross (CIR), and Constant Elasticity of Variance (CEV). Each model is paired with the Euler method for discretization, and the execution time and resulting prices are compared. This function showcases the framework's flexibility in accommodating various stochastic processes for financial modeling.

Here are some actual running results. Figure 10 and Figure 11 display the outcomes of the first test case, `testDifferentOptions`. Users can choose to input parameters interactively through the console or directly set them in the code using the configuration function. Both methods allow the program to run and price the options. The parameter step size for this program is set to 500, and the number of iterations is set to 10,000. Overall, the running efficiency is relatively high.



```
C:\Users\xlzc\Desktop\Yumir >
Enter maturity (T): 1.0
Enter number of time steps (N): 252
Enter number of simulations (M): 10000
Select SDE Model:
1. GBM
2. CEV
3. CIR
1
Select FDM Scheme:
1. EulerMethod
2. MilsteinMethod
3. DriftAdjustedPredictorCorrector
2
Select RNG:
1. MersenneTwister
1
Select Payoff Type:
1. European Call
2. European Put
3. Asian Call
4. Asian Put
5. Up-and-In Call
6. Up-and-In Put
7. Up-and-Out Call
8. Up-and-Out Put
9. Down-and-In Call
10. Down-and-In Put
11. Down-and-Out Call
12. Down-and-Out Put
```

*Figure 10 testDifferentOptions – interact with user*



```
European Call Price (User Input): 5.9799
Time taken: 0.295452 seconds

Asian Put Price: 3.62788
Time taken: 4.93997 seconds

Down and In Call Price: 0.911576
Time taken: 4.68135 seconds

Up and Out Put Price: 5.82072
Time taken: 4.65534 seconds
```

*Figure 11 testDifferentOptions – results*

Figure 12 shows the time taken by two different Finite Difference Methods (FDM), Euler and Milstein. It can be observed that the Euler method is faster, but theoretically, the Milstein method offers higher accuracy. Both methods serve as alternatives to analytical solutions and can accelerate Monte Carlo simulations. Figure 13 presents the results of pricing using different Stochastic Differential Equations (SDE).

```
Testing different FDM methods...
European Call Price (Euler Method): 10.9877
Time taken: 4.68943 seconds

European Call Price (Milstein Method): 11.0091
Time taken: 5.66491 seconds
```

*Figure 12 testDifferentFDM*

```
Testing different SDE methods...
Asian Put Price (GBM): 3.62481
Time taken: 4.8332 seconds

Asian Put Price (CIR): 4.88062
Time taken: 4.84031 seconds

Asian Put Price (CEV): 0.00615157
Time taken: 5.33041 seconds
```

*Figure 13 testDifferentFDM*