Final Project for Advanced C++11/C++14 Certificate – Level 7

Monte Carlo Option Pricing Application Analysis

Pavlos Sakoglou

1. Project Description

The Monte Carlo Option Pricing application is used in computational finance to approximate prices of financial derivatives (options) given a set of parameters as input, either determined by the user in real-time, either having this set of parameters fixed, or fed into the program from another system.

The current application follows a classification logic, that is, it groups the model parameters into data aggregates and then uses them into the pricing algorithm that implements the Monte Carlo simulation, which prices them. When the pricing is done, the outcome along with the simulation data are sent into a management information class that computes certain statistics on the pricing method that was used, in order to help the user to evaluate the whole process. Finally, the data and the newly computed statistics are send to an output class, with which the user can choose to print them on the console, or to save them in a .txt file, or in an excel document.

In case the user wants to price multiple derivatives at once, the application provides an extra feature that allows the user to choose once the model parameters and select multiple option contracts to price. The application saves the results of each pricing process, which will then be printed all together in the same way, by either printing in the console consecutively, or creating multiple files.

2. System Components

FDM_SDE class

The user can choose between three numerical approximations for the pricing process, namely, the general Geometric Brownian Motion, the Explicit Euler Approximation, and the Milstein Approximation. All methods work equally well, however, in regards to computational speed and convergence, the General Geometric Brownian Motion is the fastest one, with the Explicit Euler close second, and finally the Milstein approximation, which appears to have slightly slower computational convergence.

RNG class

The random generation processes used in the application are the Default Random Engine and the Mersenne Twister Engine, coupled with a standard normal distribution variate. The user can choose either engine in run-time, or fix a random engine before the compilation. For more random generation processes, the application is easy to extend, namely, the developer needs to add an extra method by modifying the RNG class and its user interface appropriately.

Payoff class

The option contracts that are subject to pricing in this application are the following:

- European Call/Put
- Asian Call/Put
- Knock-out Call/Put
- Knock-in Call/Put

The process of extending the application into pricing more option contracts, is again simple. The user has to define the extra payoff functions and modify the corresponding user interface, in Payoff class. Alternatively, the user can hard code a new payoff and pass it as argument either in the pricing class Pricer, or via Payoff class setters.

Input class

The input class provides a basic interface, user-interactive, for option data selection in run-time, along with basic construction and initialization methods, one setter and getter for group data retrieve, and of course the option data parameters as public members, so that to be accessible without the need of extra member functions.

Pricer class

The Pricer class is a templated class, using as template parameters the four classes that are described above: RNG, Input, FDM_SDE, Payoff. Its purpose is to use user-determined input, either in run time or pre-determined, and use it to price accordingly the derivative as per the appropriate model. Therefore, it implements a general pricer function, that can also handle the Asian options by averaging the stock price in [0,T] and use it in the payoff if certain conditions are satisfied.

Moreover, the Pricer has setters for the model parameters, but also implements a get() method that calls the user-interactive interface of the parameter classes, and determines all the model data. Then, it initializes its members appropriately and feeds them to the pricing algorithms accordingly. When the process is over, it gathers the output data into tuples and provides getter function so that they can be used in other classes such as MIS and Output.

MIS class

This class serves only for managerial decisions and produces statistics given the simulation data. It computes the mean, max and min prices of the random process, standard deviation and standard error, etc. Moreover, it computes the exact prices of the underlying derivative and compares it with the approximated price. If the approximated price is close enough to the exact price, it indicates “true” as a decision, otherwise “false”. The user then can decide if the simulation is successful or satisfying enough.

Furthermore, MIS class provides a stopwatch method that measures the processing time of the pricing algorithm, providing useful information on a system performance scale.

Output class

This class takes input from MIS and Pricer class and provides a user-interactive interface for printing method selection. The printing options are the console, a text file, and an excel file. Moreover, in case of multiple option pricing, there is an appropriate printing interface that iterates a list of multiple Pricer and MIS output data, and prints them consecutively in one of the above methods.

Builder class

The builder class servers as a mediator method that puts the whole process together. It is a meta-programming class with three methods, namely, an informative message method, a run() method, that allows the user to select the pricing model and parameters, as well as the number of options to be priced, the pricing methods, the MIS methods, and finally the output methods.

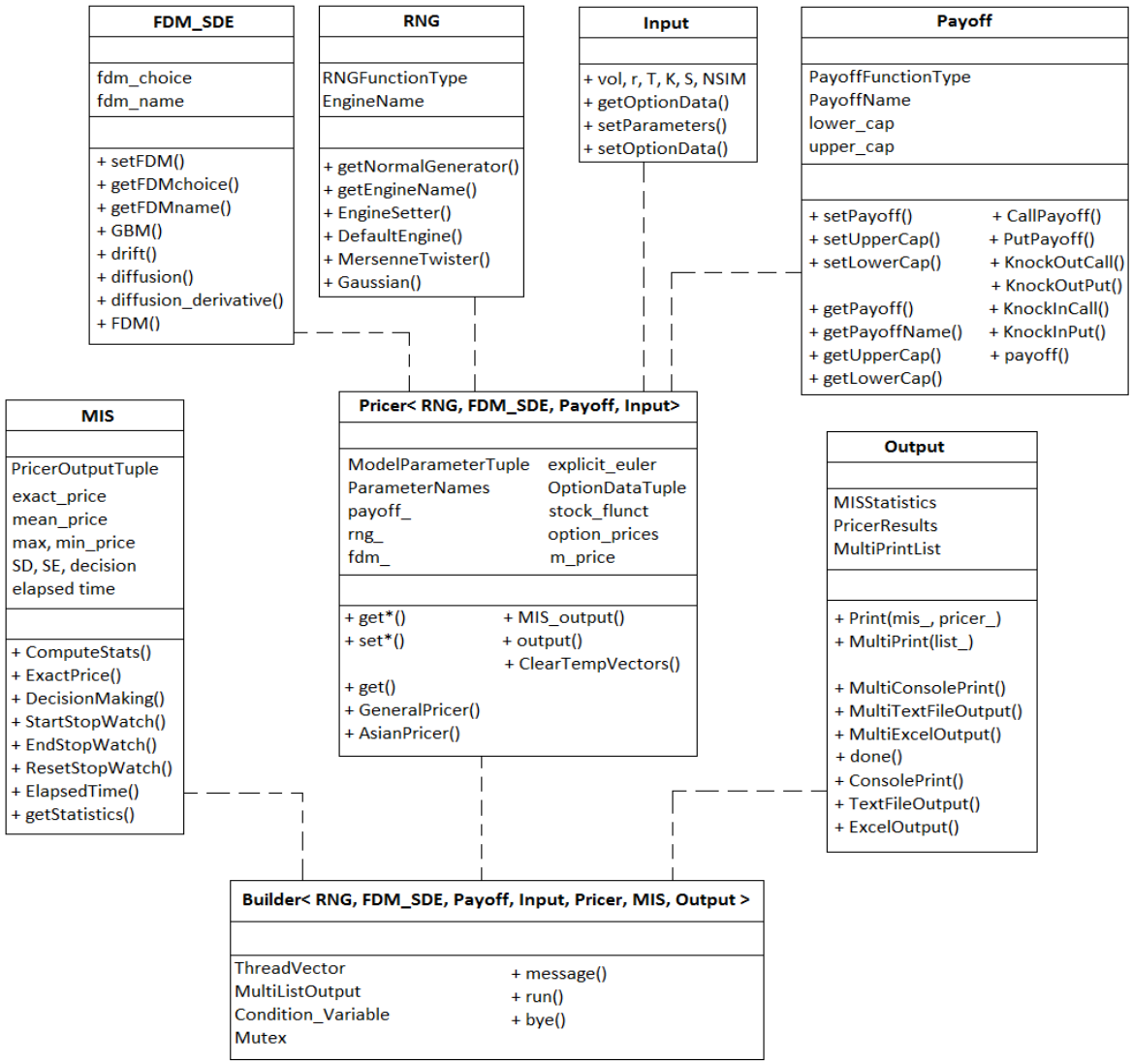
The motivation for a builder class is to keep the main() function cleaner. As for performance issues, we gain very little by implementing the functionality directly in main() without a mediator method. One would argue that the program becomes “heavier” with extra classes, however, the next-generation pattern is quite appropriate for such code design. Finally, the Builder provides a bye() method that indicates the end of the program, after the output print is completed.

3. Design and domain architecture

The system architecture is based entirely on meta-programming and next generation pattern, where the data are interchanged between the classes via function wrappers, tuples and information containers, and direct calls of static and inline functions. Thus, there are no class hierarchies and the object creation and copying is kept at the minimum.

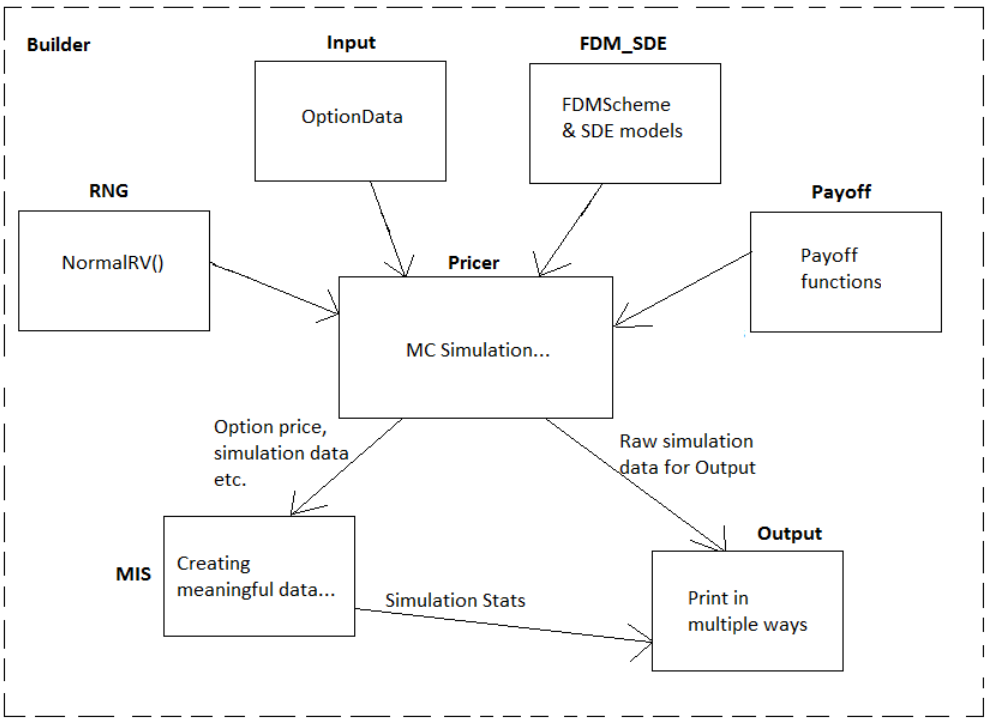
The builder is based on the concept of MAN domain architecture since its purpose is the object creation and data assembly between the system's classes. The Pricer, however, is based on the concept of RAT, since it tracks and allocates information in run-time (optionally with fixed data) through the system. As a result, the UML diagram is as follows:

UML diagram



Furthermore, the data flow diagram below describes how the information is distributed throughout the system's components:

Data flow diagram



4. Complexity Analysis

The algorithms that are used are all of linear or constant complexity, namely, for-loops, STL algorithms, function calls, etc.

There are no recursive calls nowhere in the code, and thus the system's total complexity is the sum of $O(n_i)$, i = 1, …, k; where k = number of algorithms and processes, excluding the simulation loops. That claim can be justified since all the algorithms we are using, i.e. clear(), max/min_element(), for_each(), etc. are all of linear complexity. To our favor, push_back() calls are of constant complexity.

To that result, we estimate an O(NSIM) * O(NSteps) complexity for the simulation, and probably other processing factors that might cause delays.

Therefore, the aggregate complexity of our system is close to the following result:

$$f(m,l) = a \times O(1) + b \times O(m) \times O(l) + \sum_{i=1}^{k} O(n_i) + error$$

Where a, b, and error and real constants, and m and l are the number of simulations and steps respectively.

5. System Performance and Optimization

The system performance is well-mannered for the underlying design. To achieve faster results, we need to hard code all the components in main(), but that won't necessarily guarantee better performance, since there is a trade-off between multiple object creation and non-flexibility, and speed.

To optimize the code, I used the following techniques:

- Switch-case instead of if-statements, since it breaks after a true-evaluated case and doesn't check further.

- Static and inline functions (trade-off between time and space). Since this is an option pricing application, speed matters more than memory. Moreover, static prevents invocation of *this* pointer.

- Initializer lists and explicit constructors

- STL containers and algorithms

Sample results of this optimization are the following:

Before optimization	**NSIM = 1,000,000, NSteps = 500**
GBM	15.271725 seconds
Explicit Euler	2604.7156749 seconds
Milstein	2803.5772064 seconds

After optimization	**NSIM = 1,000,000, NSteps = 500**
GBM	2.91236 seconds
Explicit Euler	101.404 seconds
Milstein	126.36 seconds

6. Output analysis

Below several approximations for all options that are subject to MCS in this application. Next to each price is the measured processing time of the simulation. Both random generation engines where used, create completely random simulated stock paths.

European Call			
Option Data	S = 60, K = 65, Vol = 0.3, r = 0.08, T = 0.25 European Call Option exact price: **$ 2.13293**		
Approximate Price	**Explicit Euler Approximation**		
		NSteps	
NSIM	100	500	1000
10.000	2.1989	2.06382	2.24252
100.000	2.1286	2.12418	2.17034
1.000.000	2.15955	2.13507	2.13646

European Put			
Option Data	S = 60, K = 65, Vol = 0.3, r = 0.08, T = 0.25 European Put Option exact price: **$ 5.84584**		
Approximate Price	**Explicit Euler Approximation**		
		NSteps	
NSIM	100	500	1000
10.000	5.79343	5.78207	5.79623
100.000	5.84185	5.83612	5.84554
1.000.000	5.85526	5.85598	5.85561

Asian Call			
Option Data	S = 60, K = 65, Vol = 0.3, r = 0.08, T = 0.25 Asian Call Option exact price: **$ 2.13293**		
Approximate Price	**Explicit Euler Approximation**		
		NSteps	
NSIM	100	500	1000
10.000	2.1139	2.10315	2.16743
100.000	2.13442	2.12411	2.11017
1.000.000	2.15461	2.13626	2.1426

Asian Put			
Option Data	S = 60, K = 65, Vol = 0.3, r = 0.08, T = 0.25 Asian Put Option exact price: **$ 5.84584**		
Approximate Price	**Explicit Euler Approximation**		
	NSteps		
NSIM	100	500	1000
10.000	5.83546	5.85451	5.83265
100.000	5.82118	5.86165	5.85727
1.000.000	5.85357	5.84296	5.84568

Knock-out Call			
Option Data	S = 60, K = 65, Vol = 0.3, r = 0.08, T = 0.25, Upper Cap = 100 Knock-Out Call Option exact price: **$ 2.13293**		
Approximate Price	**Explicit Euler Approximation**		
	NSteps		
NSIM	100	500	1000
10.000	2.19165	2.1172	2.2146
100.000	2.14589	2.11183	2.10843
1.000.000	2.13024	2.12023	2.11948

Knock-out Put			
Option Data	S = 60, K = 65, Vol = 0.3, r = 0.08, T = 0.25, Upper Cap = 100 Knock-Out Put Option exact price: **$ 5.84584**		
Approximate Price	**Explicit Euler Approximation**		
	NSteps		
NSIM	100	500	1000
10.000	5.95381	5.79044	5.86231
100.000	5.8057	5.84573	5.83748
1.000.000	5.85785	5.83474	5.84035

Knock-in Call			
Option Data	S = 60, K = 65, Vol = 0.3, r = 0.08, T = 0.25, Lower Cap = 63 Knock-in Call Option exact price: **$ 2.13293**		
Approximate Price	**Explicit Euler Approximation**		
	NSteps		
NSIM	100	500	1000
10.000	2.109.3	2.10332	2.2243
100.000	2.15821	2.14201	2.13251
1.000.000	2.15934	2.13658	2.14076

Knock-in Put			
Option Data	S = 60, K = 65, Vol = 0.3, r = 0.08, T = 0.25, Lower Cap = 63 Knock-in Put Option exact price: **$ 5.84584**		
Approximate Price	**Explicit Euler Approximation**		
	NSteps		
NSIM	100	500	1000
10.000	0.07619	0.0727847	0.0712272
100.000	0.0764799	0.076479	0.0762153
1.000.000	0.077148	0.0770821	0.0778065

As it is expected from the theory, the law of large numbers in Monte Carlo simulation confirms that ***the larger the number of simulations, the closer we get to the exact price, asymptotically***. If the approximated prices are *epsilon* close to the exact price, we feel comfortable to proceed.

***** For simulation tests with Milstein approximation look at "example output" folder.**

7. Extreme scenarios and limitations

Depending the processing power and ram memory of the machine this application is going to run, there are limitations in use, for practical reasons. Although that theoretically the program can handle multiple threads, I do not advise to use this application for extravagant amount of iterations, since the containers will reach the limit and the program will crash.
However, for extremely big simulations, namely NSIM > 10,000,000 and NSteps > 5,000, it is advised to run through theTestPlainMC test file, which has limited output options, but is lighter in regards to memory and does not use containers for the option price fluctuations. The results will come faster and will be equally accurate. For ordinary use however, the BulderMC file is more convenient since it doesn't require hard coding.

8. Project Cost and time allocation

The time resources needed for the completion of this project from scratch, are listed below, using many heuristics and established methodology. This work, however, is entirely original, and no particular resources were required to build this application, not even concerning the simulation itself.

- Understanding the project/problem: 2-3 hours
- Understanding what I needed to do, i.e. brainstorming, drawing designs, etc.: 7-10 hours
- Determining an almost final design: 10-15 hours
- Review and examine the design: 2-4 hours
- Implementing the design: 6-8 hours
- Fix bugs and make the code work as expected: 5-8 hours
- Test the code and run simulations to confirm exactness: 5-7 hours
- Optimizing the code: 10-15 hours
- Documentation and cleaning the code: 3-5 hours
- Final check: 2-3 hours

Total: ~ 65 hours worked