Level 9: Introductory Computational Finance

Goals and Objectives

We discuss various methods to price options:

- Exact (closed) solutions.
- Monte Carlo (MC) method.
- Finite Difference method (FDM).

We discuss the fundamental processes and algorithms that describe how to price options. The focus is on understanding the mathematical and financial fundamentals in `just` enough detail so as to be able to implement option pricing algorithms in C++. We recommend that you consult internet and other sources (for example, www.datasimfinancial.com) in order to deepen your knowledge of these topics.

The goals of these exercises are:

- Understanding the C++ option pricing code for the above methods; create test programs with various data sets.
- Getting used to the code style and learning some basic computational finance.
- Making the code more reusable and improving the design to allow for future extensions. In particular, you'll need to apply the object-oriented and generic programming techniques from previous sections.
- Comparing the above different methods with respect to accuracy, efficiency and stability. Stress testing for the full range of parameters.
- Using STL, Boost and Duffy's data structures.

It is not expected to have the full financial or mathematical background to the enclosed formulae; the focus is on being able to program these formulae in C++.

Groups A&B: Exact Pricing Methods

Important Instructions:

- You will need to encapsulate all functionality (i.e., option pricing, greeks, matrix pricing) into proper classes. You should submit Group A and Group B as a single, comprehensive project that takes all described functionality into account, and presents a unified, well-structured, robust, and flexible design. While you have full discretion to make specific design decisions in this level, your grade for Groups A and B will be based on the overall quality of the submitted code in regards to robustness, flexibility, clarity, code commenting, efficiency, conciseness, taking previously-learned concepts into account, and correctness.
- Your single main() function should fully test each and every aspect of your option pricing classes, to ensure correctness prior to submission. This is of utmost importance.
- All answers to questions, as well as batch test outputs should be outlined in a document. Additionally, and justifications for design decisions should be outlined in the document as well.

A. Exact Solutions of One-Factor Plain Options

In this section, we discuss the exact formulae for plain (European) equity options (with zero dividends) and their sensitivities. These options can be exercised at the expiry time T only. The objectives are:

- Given the exact solution for a given financial quantity (for example, an option price), show how to map it to C++ code.
- Use STL and Boost libraries as often as you can (do not reinvent the wheel). For example, we try to use `std::vector<T>` to model arrays and Boost Random library to generate normal (Gaussian random variates).

The parameters whose values that need to be initialized are:

- T (expiry time/maturity). This is a number, e.g. T = 1 means one year. K (strike price).
- sig (volatility).
- r (risk-free interest rate).
- S (current *stock* price where we wish to price the option).
- C = call option price, P = put option price.

Finally, we note that $n(x)$ is the normal (Gaussian) probability density function and $N(x)$ is the cumulative normal distribution function, both of which are supported in Boost Random.

We give the set of test values for option pricing. We give each set a name so that we can refer to it in later exercises (we call them *batches*).

Batch 1: T = 0.25, K = 65, sig = 0.30, r = 0.08, S = 60 (then C = 2.13337, P = 5.84628).
Batch 2: T = 1.0, K = 100, sig = 0.2, r = 0.0, S = 100 (then C = 7.96557, P = 7.96557).
Batch 3: T = 1.0, K = 10, sig = 0.50, r = 0.12, S = 5 (C = 0.204058, P = 4.07326).
Batch 4: T = 30.0, K = 100.0, sig = 0.30, r = 0.08, S = 100.0 (C = 92.17570, P = 1.24750).

Of course, you can use other data sets; there are many resources available but we recommend you test the above batches at the least.

We now give some mathematical and financial background to the Black-Scholes pricing formula. The formulae apply to option pricing on a range of underlying securities, but we focus on stocks in these exercises.

We introduce the generalized Black–Scholes formula to calculate the price of a call option on some underlying asset. In general, the call price is a function:

$$C = C(S, K, T, r, \sigma) \tag{1}$$

$S =$ asset price
$K =$ strike price
$T =$ exercise (maturity) date
$r =$ risk-free interest rate
$\sigma =$ constant volatility
$b =$ cost of carry.

We can view the call option price C as a vector function because it maps a vector of parameters into a real value. The exact formula for C is given by:

$$C = Se^{(b-r)T}N(d_1) - Ke^{-rT}N(d_2) \tag{2}$$

where $N(x)$ is the standard cumulative normal (Gaussian) distribution function defined by

$$N(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-y^2/2}dy \tag{3}$$

and

$$d_1 = \frac{ln(S/K) + (b+\sigma^2/2)T}{\sigma\sqrt{T}}$$
$$d_2 = \frac{ln(S/K) + (b-\sigma^2/2)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}. \tag{4}$$

The cost-of-carry parameter b has specific values depending on the kind of security in question:

- **b = r** is the Black–Scholes stock option model.
- **b = r - q** is the Morton model with continuous dividend yield q.
- **b = 0** is the Black–Scholes futures option model.
- **b = r - R** is the Garman and Kohlhagen currency option model, where R is the foreign risk-free interest rate.

Thus, we can find the price of a plain call option by using formula (2). Furthermore, it is possible to differentiate C with respect to any of the parameters to produce a formula for the option sensitivities.

The corresponding formula for a put option is:

$$P = Ke^{-rT}N(-d_2) - Se^{b-r)T}N(-d_1).$$

For the case of stock options, you take b = r in your calculations.

There is a relationship between the price of a European call option and the price of a European put option when they have the same strike price K and maturity T. This is called *put-call parity* and is given by the formula:

$$C + Ke^{-rT} = P + S.$$

This formula is used when creating trading strategies.

Answer the following questions:

a) Implement the above formulae for call and put option pricing using the data sets Batch 1 to Batch 4. Check your answers, as you will need them when we discuss numerical methods for option pricing.

b) Apply the put-call parity relationship to compute call and put option prices. For example, given the call price, compute the put price based on this formula using Batches 1 to 4. Check your answers with the prices from part a). Note that there are two useful ways to implement parity: As a mechanism to calculate the call (or put) price for a corresponding put (or call) price, or as a mechanism to check if a given set of put/call prices satisfy parity. The ideal submission will neatly implement both approaches.

c) Say we wish to compute option prices for a monotonically increasing range of underlying values of S, for example 10, 11, 12, …, 50. To this end, the output will be a vector. This entails calling the option pricing formulae for each value S and each computed option price will be stored in a `std::vector<double>` object. It will be useful to write a global function that produces a mesh array of doubles separated by a mesh size h.

d) Now we wish to extend **part c** and compute option prices as a function of **i)** expiry time, **ii)** volatility, or **iii)** any of the option pricing parameters. Essentially, the purpose here is to be able to input a *matrix* (vector of vectors) of option parameters and receive a *matrix* of option prices as the result. Encapsulate this functionality in the most flexible/robust way you can think of.

Option Sensitivities, aka the Greeks

Option sensitivities are the partial derivatives of the Black-Scholes option pricing formula with respect to one of its parameters. Being a partial derivative, a given greek quantity is a measure of the sensitivity of the option price to a small change in the formula's parameter. There are exact formulae for the greeks; some examples are:

$$\begin{aligned}
&\Delta_C \equiv \frac{\partial C}{\partial S} = e^{(b-r)T}N(d_1) \\
&\Gamma_C \equiv \frac{\partial^2 C}{\partial S^2} = \frac{\partial \Delta_C}{\partial S} = \frac{n(d_1)e^{(b-r)T}}{S\sigma\sqrt{T}} \\
&Vega_C \equiv \frac{\partial C}{\partial \sigma} = S\sqrt{T}e^{(b-r)T}n(d_1) \\
&\Theta_C \equiv -\frac{\partial C}{\partial T} = -\frac{S\sigma e^{(b-r)T}n(d_1)}{2\sqrt{T}} - (b-r)Se^{(b-r)T}N(d_1) - rKe^{-rT}N(d_2).
\end{aligned} \tag{5}$$

Answer the following questions:

a) Implement the above formulae for gamma for call and put future option pricing using the data set: K = 100, S = 105, T = 0.5, r = 0.1, b = 0 and sig = 0.36. (exact delta call = 0.5946, delta put = -0.3566).

b) We now use the code in **part a** to compute call delta price for a monotonically increasing range of underlying values of S, for example 10, 11, 12, …, 50. To this end, the output will be a vector and it entails calling the above formula for a call delta for each value S and each computed option price will be store in a `std::vector<double>` object. It will be useful to reuse the above global function that produces a mesh array of double separated by a mesh size h.

c) Incorporate this into your above *matrix pricer* code, so you can input a matrix of option parameters and receive a matrix of either Delta or Gamma as the result.

d) We now use divided differences to approximate option sensitivities. In some cases, an exact formula may not exist (or is difficult to find) and we resort to numerical methods. In general, we can approximate first and second-order derivatives in S by 3-point second order approximations, for example:

$$\Delta = \frac{V(S+h)-V(S-h)}{2h}$$

$$\Gamma = \frac{V(S+h)-2V(S)+V(S-h)}{h^2}.$$

In this case the parameter h is 'small' in some sense. By Taylor's expansion you can show that the above approximations are second order accurate in h to the corresponding derivatives.

The objective of this part is to perform the same calculations as in **parts a** and **b,** but now using divided differences. Compare the accuracy with various values of the parameter h (In general, smaller values of h produce better approximations but we need to avoid *round-offer errors* and subtraction of quantities that are very close to each other). Incorporate this into your well-designed class structure.

B. Perpetual American Options

A European option can only be exercised at the expiry date T and an exact solution is known. An American option is a contract that can be exercised at *any time* prior to T. Most traded stock options are American style. In general, there is no known exact solution to price an American option but there is one exception, namely *perpetual American options*. The formulae are:

$$C = \frac{K}{y_1-1}\left(\frac{y_1-1}{y_1}\frac{S}{K}\right)^{y_1}$$

$$y_1 = \frac{1}{2} - \frac{b}{\sigma^2} + \sqrt{\left(\frac{b}{\sigma^2}-\frac{1}{2}\right)^2 + \frac{2r}{\sigma^2}}$$

for call options and

$$P = \frac{K}{1-y_2}\left(\frac{y_2-1}{y_2}\frac{S}{K}\right)^{y_2}$$

$$y_2 = \frac{1}{2} - \frac{b}{\sigma^2} - \sqrt{\left(\frac{b}{\sigma^2}-\frac{1}{2}\right)^2 + \frac{2r}{\sigma^2}}$$

for put options.

In general, the perpetual price is the time-homogeneous price and is the same as the normal price when the expiry price T tends to infinity. In general, American options are worth more than European options.

Answer the following questions:

a) Program the above formulae, and incorporate into your well-designed options pricing classes.

b) Test the data with K = 100, sig = 0.1, r = 0.1, b = 0.02, S = 110 (check C = 18.5035, P = 3.03106).

c) We now use the code in part a) to compute call and put option price for a monotonically increasing range of underlying values of S, for example 10, 11, 12, …, 50. To this end, the output will be a vector and this exercise entails calling the option pricing formulae in part a) for each value S and each computed option price will be stored in a `std::vector<double>` object. It will be useful to reuse the above global function that produces a mesh array of double separated by a mesh size h.

d) Incorporate this into your above *matrix pricer* code, so you can input a matrix of option parameters and receive a matrix of Perpetual American option prices.

Groups C&D: Monte Carlo Pricing Methods

C. Monte Carlo 101

For this section, all necessary code is provided. You are expected to submit a document with the answers to the below three questions, but not any code. The document should contain a detailed, complete analysis and will be graded on how well it demonstrates understanding of the accuracy and efficiency of Monte Carlo methods in the below context.

We focus on a linear, constant-coefficient, scalar (one-factor) problem. In particular, we examine the case of a one-factor European call option using the assumptions of the original Black Scholes equation. We give an overview of the process.

At the expiry date $t = T$ the option price is known as a function of the current stock price and the strike price. The essence of the Monte Carlo method is that we carry out a *simulation experiment* by finding the solution of a *stochastic differential equation* (SDE) from time $t = 0$ to time $t = T$. This process allows us to compute the stock price at $t = T$ and then the option price using the payoff function. We carry out *M simulations* or *draws* by finding the solution of the SDE and we calculate the option price at $t = T$. Finally, we calculate the discounted average of the simulated payoff and we are done.

Summarizing, the process is:
1. Construct a simulated path of the underlying stock.
2. Calculate the stock price at $t = T$.
3. Calculate the call price at $t = T$ (use the *payoff function*).

Execute steps 1–3 *M* times.
4. Calculate the averaged call price at $t = T$.
5. Discount the price found in step 4 to $t = 0$.

The first step is to replace continuous time by discrete time. To this end, we divide the interval [0, T] (where T is the expiry date) into a number of subintervals as shown in Figure 1. We define *N + 1 mesh points* as follows:

$$0 = t_0 < t_1 < .. < t_n < t_{n+1} < .. < t_N = T.$$

In this case we define a set of *subintervals* (t_n, t_{n+1}) of size $\Delta t_n \equiv t_{n+1} - t_n, \quad 0 \le n \le N-1.$

In general, we speak of a *non-uniform mesh* when the sizes of the subintervals are not necessarily the same. However, in this book we consider a class of finite difference schemes where the N subintervals have the same size (we then speak of a *uniform mesh*), namely $¢t = T=N$.
Having defined how to subdivide $[0,T]$ into subintervals, we are now ready to motivate our finite difference schemes; for the moment we examine the scalar linear SDE with constant coefficients:

$$\begin{aligned} dX &= aXdt + bXdW, \quad a, b \text{ constant} \\ X(0) &= A. \end{aligned} \tag{6}$$

Regarding notation, we do not show the dependence of variable X on t and when we wish to show this dependence we prefer to write $X = X(t)$ instead of the form Xt .

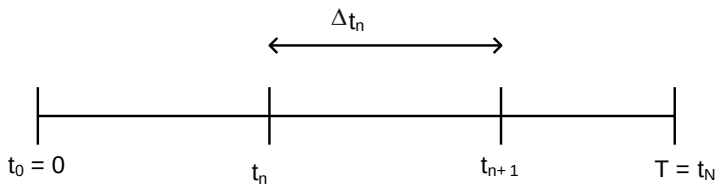


Figure 1 Mesh generation

We write equation (6) as a stochastic integral equation between the (arbitrary) times s and t as follows:

$$X(t) = X(s) + \int_s^t aX(y)dy + \int_s^t bX(y)dW(y), \quad s < t \tag{7}$$

We evaluate equation (7) at two consecutive mesh points and using the fact that the factors *a* and *b* in equation (6) are constant we get the exact identity:

$$\begin{aligned} X(t_{n+1}) &= X(t_n) + \int_{t_n}^{t_{n+1}} aX(y)dy + \int_{t_n}^{t_{n+1}} bX(y)dW(y) \\ &= X(t_n) + a\int_{t_n}^{t_{n+1}} X(y)dy + b\int_{t_n}^{t_{n+1}} X(y)dW(y). \end{aligned} \tag{8}$$

We now approximate equation (8) and in this case we replace the solution *X* of equation (8) by a new discrete function *Y* (that is, one that is defined at mesh points) by assuming that it is constant on each subinterval; we then arrive at the discrete equation:

$$\begin{aligned} Y_{n+1} &= Y_n + aY_n\int_{t_n}^{t_{n+1}} dy + bY_n\int_{t_n}^{t_{n+1}} dW(y) \\ &= Y_n + aY_n\Delta t_n + bY_n\Delta W_n, \end{aligned} \tag{9}$$

where

$$\Delta W_n = W(t_{n+1}) - W(t_n), \quad 0 \le n \le N-1.$$

This is called the (explicit) *Euler-Maruyama scheme* and it is a popular method when approximating the solution of SDEs. In some cases we write the solution in terms of a discrete function X_n if there is no confusion between it and the solution of equations (6) or (7). In other words, we can write the discrete equation (9) in the equivalent form:

$$\begin{cases} X_{n+1} = X_n + aX_n\Delta t_n + bX_n\Delta W_n \\ X_0 \quad = A. \end{cases} \tag{10}$$

In many examples the mesh size is constant and furthermore the Wiener increments are well-known computable quantities:

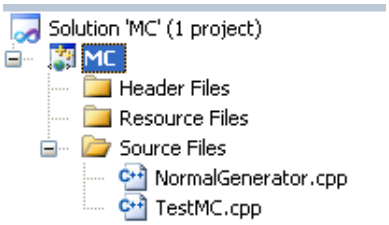
$$\begin{cases} \Delta t_n \quad = \Delta t = T/N, \quad 0 \le n \le N-1 \\ \Delta W_n = \sqrt{\Delta t}z_n, \text{ where } z_n \sim N(0,1). \end{cases}$$

Incidentally, we generate increments of the Wiener process by a random number generator for independent Gaussian pseudo-random numbers, for example Box-Muller, Polar Marsaglia, Mersenne Twister or lagged Fibonacci generator methods. In this book, we focus exclusively on the generators in Boost Random.

We deal almost exclusively with *one-step* marching schemes; in other words, the unknown solution at time level n + 1 is calculated in terms of known values at time level *n*.

Answer the following questions:

a) Study the source code in the file *TestMC.cpp* and relate it to the theory that we have just discussed. The project should contain the following source files and you need to set project settings in VS to point to the correct header files:



Compile and run the program *as is* and make sure there are no errors.

b) Run the MC program again with data from Batches 1 and 2. Experiment with different value of NT (time steps) and NSIM (simulations or draws). In particular, how many time steps and draws do you need in order to get the same accuracy as the exact solution? How is the accuracy affected by different values for NT/NSIM?

c) Now we do some stress-testing of the MC method. Take Batch 4. What values do we need to assign to NT and NSIM in order to get an accuracy to two places behind the decimal point? How is the accuracy affected by different values for NT/NSIM?

D. Advanced Monte Carlo

This section will build upon the provided Monte Carlo code from the previous section, by adding methods to track the accuracy of the MC simulation. You are expected to submit code in addition to a document with the answers to the below questions. The document should contain a detailed, complete analysis and will be graded on how well it presents understanding of the accuracy and efficiency of Monte Carlo methods in the below context.

We wish to add functionality to the Monte Carlo pricer by providing estimates for the *standard deviation* (SD) and *standard error* (SE), defined by:

$$\mathrm{SD} = \sqrt{\frac{\sum C_{T,j}^2 - \frac{1}{M}\left(\sum C_{T,j}\right)^2}{M-1}} \times \exp\left(-rT\right).$$

$$SE = \frac{SD}{\sqrt{M}}$$

where

$C_{T,j}$ = call output price at $t = T$ for the jth simulation, $1 \leq j \leq M$,

M = number of simulations.

Implement this new functionality and test the software for a range of data for call and put options.

Answer the following questions:

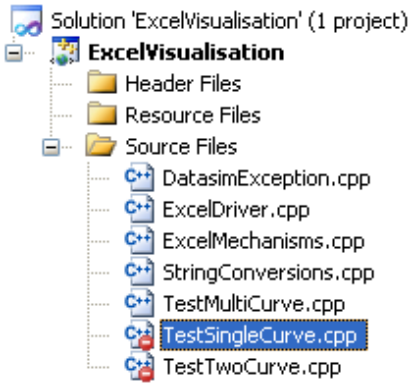
a) Create generic functions to compute the standard deviation and standard error based on the above formulae. The inputs are a vector of size M (M = NSIM), the interest-free rate and expiry time T. Integrate this new code into *TestMC.cpp*. Make sure that the code compiles.

b) Run the MC program again with data from Batches 1 and 2. Experiment with different values of NT (time steps) and NSIM (simulations or draws). How do SD and SE react for these different run parameters, and is there any pattern in regards to the accuracy of the MC (when compared to the exact method)?

E. Excel Visualization

For this section, almost all necessary code is provided. You will need to tweak the code (i.e. the Excel Import paths) to work on your own machine and MS Excel version. Submission should consist of the working code and example Excel output files.

We have developed a small package that allows us to display the output from the above schemes in Excel spreadsheets. We have provided ready-made test programs that you can customize to suit your needs. Make sure that you have the first FOUR files and ONE active test program in your project:



Answer the following questions:

a) Compile and run the sample programs *TestSingleCurve.cpp*, *TestTwoCurve.cpp* and *TestMultipleCurve.cpp*. Make sure that everything compiles and that you get Excel output.

b) We now wish to compute option price for a monotonically increasing range of underlying values of S, for example 10, 11, 12, …, 50. To this end, the output will be a vector and this exercise entails calling the exact option pricing formulae) for each value S and each computed option price will be stored in a

`std::vector<double>` object. It will be useful to write a global function that produces a mesh array of double separated by a mesh size h. Print the output in Excel.

F. Finite Difference Methods (Introduction)

For this section, all necessary code is provided. You will need to tweak the code to work on your own machine and MS Excel version. Submission should consist of the working code, example Excel output files, and a document analyzing the accuracy of FDM versus the exact method. Alternatively, you may choose to modify the code to output to the console (instead of Excel).

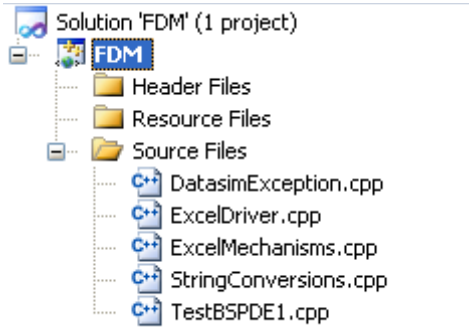
As FDM is a very advanced topic, this section is meant to be a brief taste of how FDM may be implemented using the C++ techniques learned throughout this course. However, students are not expected to understand the intricacies of FDM for option pricing, at this juncture, as that is the topic of a number of advanced MFE-level courses.

The objective is to run the code ***as is*** and is not intended as a course in the finite difference method.

The explicit Euler method is *conditionally stable* by which we mean that the mesh size k in time must be much less than the mesh size h in space (worst-case scenario is that $k = O(h^2)$). The objective of this exercise is to determine what the relationships are. Thus, for various values of h determine the value of k above which the finite difference approximation is no longer accurate.

Answer the following questions:

a) Compile and run the project as in and make sure that you get Excel output. Examine the code and try to get an idea of what is going on. The following files should be in the project:



b) In this exercise we test the FD scheme. We run the programs using the data from Batches1 to 4. Compare your answers with those from the previous exercises. That's all.

Remark: since we are using an explicit method, there is a relationship N = J*J where J is the number of mesh points in space and N is the number of mesh points in time. This is somewhat pessimistic and you can try with smaller values of N.