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.