Introductory Computational Finance

Kaiki Ikeda

QUANTNET Baruch PRE-MFE C++ for Financial Engineering

May 2025

Groups A & B: Exact Pricing Methods

Overview

This document outlines the implementation and testing of European Option pricing functionality using the Black Scholes formula. The code is neatly structured into reusable components. Each part of the assignment is addressed by the functions that were created.

a) Implemented the closed form Black Scholes pricing formula for European Call and Put options. Each batch of parameters is processed through the TestBatch function which:

1) Constructs call and put EuropeanOption objects with provided parameters
2) Calculates the theoretical option price using the Price() function
3) Outputs both put and call prices for validation against correct values

The pricing logic is in EuropeanOption cpp and hpp files, while Testing cpp and hpp files are responsible for performing tests cleanly.

Code Design Justification:

- Encapsulation: Option pricing logic is within our EuropeanOption class.
- Modularity: Batch testing logic is separated into Testing.cpp for cleaner application reusability and flow.
- Flexibility: Parameters are passed one by one for clarity and simplicity.

b) Two separate parity functions were implemented:

1) PutCallParityCall() - computes the call price from put

2) PutCallParityPut() - computes the put price from call

3) CheckPutCallParity() - checks that the call and put prices satisfy the parity relationship under a specified tolerance

These functions were put in the TestBatch function to compare parity computed prices directly.

Code Design Justification:

- Clarity: Each parity function has a clear responsibility

- Robustness: CheckPutCallParity to verify the values
- Reusability: Functions in Utilities.cpp/hpp files so it can be accessed between files with an include statement

Output:

==== BATCH 1 ====

Call Price: 2.13337
Put Price: 5.84628
Call from Parity: 2.13337
Put from Parity: 5.84628
Parity Check: PASS

==== BATCH 2 ====

Call Price: 7.96557
Put Price: 7.96557
Call from Parity: 7.96557
Put from Parity: 7.96557
Parity Check: PASS

==== BATCH 3 ====

Call Price: 0.204058
Put Price: 4.07326
Call from Parity: 0.204058
Put from Parity: 4.07326
Parity Check: PASS

==== BATCH 4 ====

Call Price: 92.1757
Put Price: 1.2475
Call from Parity: 92.1757
Put from Parity: 1.2475
Parity Check: PASS

c) To price options in a range of spot prices 10 to 50:

1) Created a utility function called meshArray() that took a start, end, and h as the parameters. This function would return a vector of doubles.

2) Used TestSpotRangePricing() function that:
    a) Iterates each spot price in a given range
    b) instantiates EuropeanOption object for each value
    c) computes the price and stores it in a result vector

Code Design Justification:

- Reusability: meshArray can be called for future parameter sweeps
- Less Concerns: Pricing logic is reused from EuropeanOptions and not reimplemented
- Scalability: Supports any future range of S values with a simple line configuration

Output:

S: 10 Call Price: 2.44845e-26
S: 11 Call Price: 3.76084e-24
S: 12 Call Price: 3.06729e-22
S: 13 Call Price: 1.49274e-20
S: 14 Call Price: 4.73616e-19
S: 15 Call Price: 1.04923e-17
S: 16 Call Price: 1.71328e-16
S: 17 Call Price: 2.1534e-15
S: 18 Call Price: 2.15804e-14
S: 19 Call Price: 1.77503e-13
S: 20 Call Price: 1.22746e-12
S: 21 Call Price: 7.28162e-12
S: 22 Call Price: 3.76946e-11
S: 23 Call Price: 1.72775e-10
S: 24 Call Price: 7.09996e-10
S: 25 Call Price: 2.64421e-09
S: 26 Call Price: 9.00894e-09
S: 27 Call Price: 2.83108e-08
S: 28 Call Price: 8.26534e-08
S: 29 Call Price: 2.2561e-07
S: 30 Call Price: 5.79018e-07
S: 31 Call Price: 1.40424e-06
S: 32 Call Price: 3.23256e-06
S: 33 Call Price: 7.09177e-06

S: 34 Call Price: 1.4881e-05
S: 35 Call Price: 2.99635e-05
S: 36 Call Price: 5.80651e-05
S: 37 Call Price: 0.000108583
S: 38 Call Price: 0.000196421
S: 39 Call Price: 0.000344474
S: 40 Call Price: 0.000586876
S: 41 Call Price: 0.000973115
S: 42 Call Price: 0.00157308
S: 43 Call Price: 0.00248305
S: 44 Call Price: 0.00383262
S: 45 Call Price: 0.00579244
S: 46 Call Price: 0.00858259
S: 47 Call Price: 0.0124814
S: 48 Call Price: 0.0178342
S: 49 Call Price: 0.0250622
S: 50 Call Price: 0.0346705

d) This part builds off of part c by allowing batch processing over any combination of parameters:

1) PriceOptionsBatch(): takes a matrix of parameter vectors each containing the required variables to perform the Black Scholes formula (T, K, sig, r, S, b) and a corresponding vector of strings that has types ("C" for calls and "P" for puts).
2) returns a vector of calculated option prices

Code Design Justification:

- Generality: can batch compute any number of options with any mix of parameters and option types
- Safety: includes validation checks if the number of parameters and the number of types match. An exception will be thrown if invalid.
- Expandable: Can easily be extended in the future to add greeks.

Output:

Option 1(C): Price = 2.13337
Option 2(P): Price = 7.96557
Option 3(C): Price = 0.204058
Option 4(P): Price = 1.2475

Options Sensitivites (The Greeks)

a) The exact analytical formulas for Delta and Gamma for both Calls and puts within the Black-Scholes Model were implemented. Here are the parameters that were used:

- K = 100
- S = 105
- T = 0.5
- r = 0.1
- b = 0
- sig = 0.36

The function TestExactGreeks() in Testing.cpp was created to compute and output the exact values of delta and gamma. These values are checked with known results shown in the document (call delta = 0.5946 and put delta = -0.3566). The output also includes Gamma to showcase how capable the program is and to experiment more.

The EuropeanOption class now has Delta() and Gamma() methods to encapsulate the calculations in an object oriented way.

Output:

[Part A] Exact Greeks for Call:
Delta: 0.594629
Gamma: 0.0134936

[Part A] Exact Greeks for Put:
Delta: -0.356601
Gamma: 0.0134936

b) The delta computation was extended over a monotonically increasing range of spot prices, specifically for the range 10 to 50 with steps of 1. Here are some things that were used:

- Previously we created a meshArray, so we used this to generate our array from the Utility module.
- TestExactDeltaOverSpotRange() function calls the exact Delta method at each spot level
- Results stored in a double vector

The test shows how sensitive the delta value is based on the different spot values and how it changes with respect to S (spot).

Output:

[Part B] Exact Delta over Spot Range for Call:
S = 10, Delta = 2.25551e-19
S = 11, Delta = 6.18174e-18
S = 12, Delta = 1.12536e-16
S = 13, Delta = 1.46673e-15
S = 14, Delta = 1.44882e-14
S = 15, Delta = 1.1336e-13
S = 16, Delta = 7.27491e-13
S = 17, Delta = 3.93787e-12
S = 18, Delta = 1.83921e-11
S = 19, Delta = 7.55191e-11
S = 20, Delta = 2.76878e-10
S = 21, Delta = 9.18317e-10
S = 22, Delta = 2.78591e-09
S = 23, Delta = 7.80377e-09
S = 24, Delta = 2.0348e-08
S = 25, Delta = 4.97346e-08
S = 26, Delta = 1.14647e-07
S = 27, Delta = 2.50577e-07
S = 28, Delta = 5.21714e-07
S = 29, Delta = 1.03903e-06
S = 30, Delta = 1.98667e-06
S = 31, Delta = 3.65877e-06
S = 32, Delta = 6.5092e-06
S = 33, Delta = 1.1216e-05
S = 34, Delta = 1.87624e-05
S = 35, Delta = 3.05351e-05
S = 36, Delta = 4.84406e-05
S = 37, Delta = 7.50367e-05
S = 38, Delta = 0.00011368
S = 39, Delta = 0.000168682
S = 40, Delta = 0.000245471
S = 41, Delta = 0.000350761
S = 42, Delta = 0.000492699

S = 43, Delta = 0.000681019
S = 44, Delta = 0.000927157
S = 45, Delta = 0.00124435
S = 46, Delta = 0.00164771
S = 47, Delta = 0.00215423
S = 48, Delta = 0.00278276
S = 49, Delta = 0.00355398
S = 50, Delta = 0.00449025

c) For multiple option parameter sets, the function TestExactSensitivityBatch() was developed:

- takes a matrix of option parameters
- takes a vector of option types
- takes the sensitivity type, so either delta or gamma

The function SensitivityBatch() calculates Greeks in bulk, this ensures:

- Flexibility for different sensitivity types
- robust parameter size checking
- An upgradeable structure

The final result matrix was traversed and printed out in a way that is easier to see.

Output:

[Part C] Exact Sensitivity Batch:
Option 1 (C, Delta): 0.372483
Option 2 (P, Gamma): 318623
Option 3 (C, Delta): 0.185048
Option 4 (P, Gamma): 49900

d) Divided difference approximation methods for Delta and Gamma were added into the NumericalGreeks hpp and cpp files as DeltaApprox() and GammaApprox(). The following test structures were also created:

- TestNumericalGreeks() - this shows approximate values at a single point
- TestNumericalGreeksOverSpot() - computes approximated delta and gamma over a range of S.
- TestSensitivityBatch(h) - runs the estimate for a matrix filled with parameters.

This was useful for validating the accuracy of approximations for various h values and reinforcing how numerical greeks can be derived when exact formulas are unknown.
Output:

[Part D.1] Numerical Greeks for Call:
Delta (approx): 0.594628
Gamma (approx): 4973.15

[Part D.1] Numerical Greeks for Put:
Delta (approx): -0.356601
Gamma (approx): 3070.69

-- Delta over Spot (Numerical) --

[Part D.2] Numerical Delta over Spot Range for Call:
S = 10, Delta (approx) = 2.30111e-19
S = 11, Delta (approx) = 6.2764e-18
S = 12, Delta (approx) = 1.13869e-16
S = 13, Delta (approx) = 1.48041e-15
S = 14, Delta (approx) = 1.45962e-14
S = 15, Delta (approx) = 1.14044e-13
S = 16, Delta (approx) = 7.31087e-13
S = 17, Delta (approx) = 3.95397e-12
S = 18, Delta (approx) = 1.84548e-11
S = 19, Delta (approx) = 7.57356e-11
S = 20, Delta (approx) = 2.7755e-10
S = 21, Delta (approx) = 9.20215e-10
S = 22, Delta (approx) = 2.79084e-09
S = 23, Delta (approx) = 7.81567e-09
S = 24, Delta (approx) = 2.03748e-08
S = 25, Delta (approx) = 4.97916e-08
S = 26, Delta (approx) = 1.14761e-07
S = 27, Delta (approx) = 2.50796e-07
S = 28, Delta (approx) = 5.22114e-07
S = 29, Delta (approx) = 1.03974e-06
S = 30, Delta (approx) = 1.98785e-06
S = 31, Delta (approx) = 3.66071e-06
S = 32, Delta (approx) = 6.51225e-06
S = 33, Delta (approx) = 1.12207e-05

S = 34, Delta (approx) = 1.87693e-05
S = 35, Delta (approx) = 3.05452e-05
S = 36, Delta (approx) = 4.84549e-05
S = 37, Delta (approx) = 7.50567e-05
S = 38, Delta (approx) = 0.000113707
S = 39, Delta (approx) = 0.000168718
S = 40, Delta (approx) = 0.000245519
S = 41, Delta (approx) = 0.000350822
S = 42, Delta (approx) = 0.000492776
S = 43, Delta (approx) = 0.000681115
S = 44, Delta (approx) = 0.000927275
S = 45, Delta (approx) = 0.0012445
S = 46, Delta (approx) = 0.00164788
S = 47, Delta (approx) = 0.00215443
S = 48, Delta (approx) = 0.002783
S = 49, Delta (approx) = 0.00355426
S = 50, Delta (approx) = 0.00449057

-- Batch Sensitivities (Numerical) --

[Part D.3] Numerical Sensitivity Batch:
Option 1 (C, Delta): 0.372484
Option 2 (P, Gamma): 3186.25
Option 3 (C, Delta): 0.185076
Option 4 (P, Gamma): 499

Perpetual American Options

a) The closed form analytical formulas for Perpetual American call and put options were implemented. The functions were added to the Utility module as PerpetualCall() and PerpetualPut(). Error checking was also added to make sure y1 was greater than 1 and y2 was less than 1 in order to yield a valid solution.

The Utility module isolates mathematical logic from other parts of the application. It also keeps the option pricing hierarchy clean, as perpetual options are conceptually different from European options (they assume infinite expiry and have a unique formula). Furthermore, the formulas can be reused across testing scenarios.

{The answer to part a is written in the project}

b) This part was for single case validation. The formula tested these values:

- K = 100, sig = 0.1, r = 0.1, b = 0.02, S = 110

The output matched the expected values of Call Price: 18.5035 and Put Price: 3.03106.

While object oriented approach was used for EuropeanOptions, perpetual options were implemented procedurally. This is because formulas are closed-form and stateless there might be no need for internal object state or inheritance. Procedural functions felt more lightweight and readable for this type of formula as well.

Output:

Call Price: 18.5035 (Expected: 18.5035)
Put Price : 3.03106 (Expected: 3.03106)

c) Computed perpetual call and put prices over a monotonically increasing spot range from 10 to 50 using mesh size of 1. A helper function meshArray() was used to create this vector.

Each value of S was used to evaluate the formulas above and results were stored in a double vector. Here were the results:

- Call prices increased with S as Put Prices decreased with S

Output:

S: 10 Call: 0.00826235 Put: 9.03489e+06
S: 11 Call: 0.011227 Put: 4.99557e+06
S: 12 Call: 0.0148535 Put: 2.9084e+06
S: 13 Call: 0.0192158 Put: 1.76823e+06
S: 14 Call: 0.0243891 Put: 1.11544e+06
S: 15 Call: 0.03045 Put: 726383
S: 16 Call: 0.0374762 Put: 486308
S: 17 Call: 0.0455465 Put: 333599
S: 18 Call: 0.054741 Put: 233828
S: 19 Call: 0.0651405 Put: 167076
S: 20 Call: 0.076827 Put: 121457
S: 21 Call: 0.0898835 Put: 89678.6
S: 22 Call: 0.104394 Put: 67156
S: 23 Call: 0.120442 Put: 50940.7

S: 24 Call: 0.138115 Put: 39097.9
S: 25 Call: 0.157497 Put: 30334.3
S: 26 Call: 0.178677 Put: 23770.5
S: 27 Call: 0.201742 Put: 18799.2
S: 28 Call: 0.226781 Put: 14995
S: 29 Call: 0.253883 Put: 12055.9
S: 30 Call: 0.283138 Put: 9764.83
S: 31 Call: 0.314637 Put: 7964.02
S: 32 Call: 0.348471 Put: 6537.48
S: 33 Call: 0.384732 Put: 5399.17
S: 34 Call: 0.423512 Put: 4484.6
S: 35 Call: 0.464906 Put: 3745.05
S: 36 Call: 0.509007 Put: 3143.37
S: 37 Call: 0.555908 Put: 2651.05
S: 38 Call: 0.605706 Put: 2246.02
S: 39 Call: 0.658495 Put: 1911.08
S: 40 Call: 0.714373 Put: 1632.76
S: 41 Call: 0.773434 Put: 1400.4
S: 42 Call: 0.835777 Put: 1205.56
S: 43 Call: 0.901499 Put: 1041.49
S: 44 Call: 0.970699 Put: 902.784
S: 45 Call: 1.04347 Put: 785.068
S: 46 Call: 1.11993 Put: 684.8
S: 47 Call: 1.20015 Put: 599.097
S: 48 Call: 1.28425 Put: 525.597
S: 49 Call: 1.37233 Put: 462.36
S: 50 Call: 1.46448 Put: 407.787

d) This is an extension of the implementation with the function TestPerpetualBatch() that takes in a matrix of option parameters and a corresponding vector of option types.

To support batch processing, the pattern from earlier Greeks/Price batch processing was reused. The paramMatrix is a vector of vector<double>, where each inner vector holds the parameters in the form K, sig, r, b, S. Moreover, the parallel “types” vector contains option type C or P.

The batch test:

- loops over matrix
- constructs each option from given parameters

- chooses appropriate formula based on type
- handles exceptions for invalid inputs

Output:

Batch 1 (C):
Perpetual Call Price = 18.5035

Batch 2 (P):
Perpetual Put Price = 10.7569

Batch 3 (C):
Perpetual Call Price = 38.8144

Batch 4 (P):
Perpetual Put Price = 3.42572