Level 9 HW - Parts A & B

Name: David Leather

Date: April 1st, 2025

Introduction

This report documents implementing and testing a comprehensive option pricing library focusing on European and Perpetual American options as outlined in Level 9 of the course “C++ for Financial Engineering.” The library is designed using object-oriented and generic programming techniques to provide flexible, robust, and extensible option pricing functionality.

Design Overview

The implementation follows a hierarchical class structure with a template-based design to allow for different numeric types:

1. **Base Class**: `OptionContract<NT>` - Abstract base class with standard functionality
2. **European Options**: `EuropeanOption<NT>` (abstract), `EuropeanCall<NT>`, and `EuropeanPut<NT>`
3. **Perpetual American Options**: `PerpetualAmericanOption<NT>` (abstract), `PerpetualAmericanCall<NT>`, and `PerpetualAmericanPut<NT>`
4. **Utility Class**: `MatrixPricer` - For pricing options across ranges of parameters
5. **Helper Class**: `MeshGenerator<NT>` - For generating parameter meshes
6. **Utility Struct**: `NormalDistribution<NT>` - For computing standard normal PDF and CDF using Boost

This design allows easy extension to support additional option types while maintaining a consistent interface. Templates enable working with different numeric types (double, float, etc.) without code duplication in the future.

Part A: European Option Pricing

A.1: Basic Call/Put Pricing and Put-Call Parity

The implementation includes the Black-Scholes formula for European options with a cost-of-carry parameter `b`. Testing with the four provided batches shows that our implementation matches the expected results:

Batch	Parameters	Expected Call	Computed Call	Expected Put	Computed Put	Match?
1	T=0.25, K=65, σ=0.30, r=0.08, S=60	2.13337	2.13337	5.84628	5.84628	Yes
2	T=1.0, K=100, σ=0.2, r=0.0, S=100	7.96557	7.96557	7.96557	7.96557	Yes

3	T=1.0, K=10, σ=0.5, r=0.12, S=5	0.20405	0.20405	4.07326	4.07326	Yes
4	T=30.0, K=100, σ=0.30, r=0.08, S=100	92.17570	92.17570	1.24750	1.24750	Yes

Put-call parity was implemented in two ways:

1. As a conversion mechanism (`putToCall` and `callToPut` static methods)
2. As a verification mechanism (`putCallParityCheck` static method)

Testing verified that all batches satisfy put-call parity, with the checker returning `True` for all test cases.

A.2: Greeks Implementation and Analysis

The Greeks (Delta, Gamma, Vega, and Theta) were implemented using the analytical formulas. For the test case (K=100, S=105, T=0.5, r=0.1, b=0, σ=0.36):

Greek	Put	Call
Δ	-0.3566	0.5946
Γ	0.01349	0.01349
Θ	-8.872	-8.397
Vega	26.78	26.78

The results match the expected values for Delta given in the assignment.

Range of Deltas and Gammas

Using the `MatrixPricer` class, I calculated Delta and Gamma for call options over a range of underlying prices (S = 10 to 50). The results show how these Greeks vary with the underlying price. Delta shows a pronounced increase as the option moves from deep out-of-the-money toward at-the-money.

Divided Differences Method

I implemented divided differences methods to numerically approximate Delta and Gamma:

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

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

Testing with different step sizes (h) shows that an optimal h exists. If h is too small, round-off errors dominate; if h is too large, truncation errors dominate.

Our analysis demonstrates that:

- For Delta calculations, h ≈ 0.001 provides optimal results
- For Gamma calculations, which involve h^2, optimal h is around 0.01-0.001
- Mean absolute error (MAE) is minimized at these values

The table below shows how MAE varies with step size h:

Step Size (h)	Δ MAE	Γ MAE
1e-16	0.0004647	0.0001224
1e-15	0.0004647	88.07
1e-14	0.0007536	4.85e+11
1e-13	0.0001059	3.279e+09
1e-12	1.023e-05	5.933e+07
1e-11	1.017e-06	5.367e+05
1e-10	1.427e-07	5492
1e-09	1.587e-08	32.18
1e-08	7.762e-10	0.7228
1e-07	9.936e-11	0.003609
1e-06	1.473e-11	2.743e-05
1e-05	7.059e-13	3.404e-07
1e-04	1.495e-13	3.129e-09
1e-03	4.566e-12	3.252e-11
1e-02	4.565e-10	4.12e-11
1e-01	4.566e-08	4.091e-09

This demonstrates the trade-off between truncation error and round-off error in numerical differentiation.

Part B: Perpetual American Options

Perpetual American options (options with no expiry date) were implemented according to the provided formulas.

Testing with the provided parameters (K=100, σ=0.1, r=0.1, b=0.02, S=110) yielded:

- Call price: 18.5035 (matching expected)
- Put price: 3.03106 (matching expected)

The `MatrixPricer` class was used to calculate perpetual American option prices across a range of underlying prices (S = 10 to 50) and across a grid of S and K values. The results demonstrate how these prices vary with the parameters.

An interesting observation is that perpetual American put prices are significantly higher for lower underlying prices, as expected, due to the ability to exercise at any time.

Matrix Pricing Implementation

The `MatrixPricer` class provides a flexible framework for computing option prices across ranges of parameters:

1. `priceRange`: Computes option prices/Greeks across a range of a single parameter (typically S)
2. `priceMatrix`: Computes a 2D grid of option prices/Greeks across ranges of two parameters

These methods use functors to set parameters and calculate results, providing maximum flexibility. For example:

```
// Computing Delta across a range of S values
std::vector<double> deltas = MatrixPricer::priceRange(
    fcall,                              // Option object
    10.0, 50.0, 1.0,                        // S range: 10 to 50, step 1
    MatrixPricer::deltaFunctor<EuropeanCall<double>>()  // Functor for Delta
);

// Computing a matrix of Deltas across S and K
std::vector<std::vector<double>> delta_matrix = MatrixPricer::priceMatrix(
    fcall,                              // Option object
    mesh_S,                             // Mesh for S
    mesh_K,                             // Mesh for K
    MatrixPricer::setSFunctor<EuropeanCall<double>>(),  // S setter
    MatrixPricer::setKFunctor<EuropeanCall<double>>(),  // K setter
    MatrixPricer::deltaFunctor<EuropeanCall<double>>()  // Delta calculator
);
```

This design provides a highly flexible framework for exploring option pricing under various parameter combinations.

I initially tried to use Boost's function object class, `boost::function` , but then I has some trouble using the `bind` function to define functor objects. This was especially troublesome for member functions of the parent class, like how `gamma` was defined in the `EuropeanOption<NT>` class and not the `EuropeanCall<NT>` class. To get around this, I ultimately landed on using `std::function` objects with lambda functions, which I know is out of the scope of the class. Once I defined utility functor objects within the `MatrixPricer` class this worked out very nicely and flexible.

Design Decisions

Several key design decisions were made to ensure a robust and flexible implementation:

1. **Template-Based Design**: Using templates allows for different numeric types and promotes code reuse.
2. **Abstract Base Classes**: `OptionContract` , `EuropeanOption` , and `PerpetualAmericanOption` provide common interfaces and implementations.
3. **Clone Pattern**: The `clone()` method creates a polymorphic copy, essential for the divided differences methods.
4. **Strategy Pattern**: Using functors in the `MatrixPricer` class allows for flexible parameter setting and calculation.
5. **Robust Parameter Validation**: Checks for valid inputs (e.g., positive S, K, non-negative T, σ) protect against invalid parameters.
6. **Separation of Concerns**: The pricing logic is separated from mesh generation and matrix computation.

Challenges and Solutions

1. **Divided Differences Accuracy**: Finding the optimal step size h required careful analysis of the trade-off between truncation and round-off errors.
2. **Numerical Stability**: Special cases (e.g., when y_1 or y_2 is close to 1 in perpetual options) require careful handling.

3. **Code Flexibility**: Using templates and function objects added complexity but greatly enhanced flexibility and reusability.