Option Pricing Framework: Exact Solutions and Finite Difference Methods (FDM)

Aidan Richer

August 2025

T his document focuses on the explanation and solutions for the implementation and testing of exact and Finite Difference Method (FDM) pricing methods for both European and Perpetual options using an object-oriented design in C++. The core structure is built around the base Option class, from which EuropeanCall, EuropeanPut, PerpAmericanCall, and PerpAmericanPut inherit, with contract parameters encapsulated in an OptionData class. This class is then used as a private member function m_data within each derived option class. These classes implement pricing and sensitivity functions such as Price(), Delta(), Gamma(), PutCallParity(), and numerical divided-difference approximations for option greek approximations. To extend functionality, utility components like MeshArray() were created to generate underlying spot price grids, as well as matrix-based classes (MatrixParameters and PricingMatrix) which handle batch computations of option prices and option greeks across structured datasets. At the beginning of the document is the UML diagram, which outlines the relationships and behaviors between the classes, followed by the implementation rationale for certain functionality in the program. After this, the document covers the implementation and testing of Monte Carlo simulations, including pricing accuracy across different NT and NSIM values, stress testing different option parameters, and the addition of functions to compute standard deviation and standard error for simulation output. Finally, the document introduces an introduction to a Finite Difference Method for computing option prices and compares the results to the exact solutions across batches of option parameters, highlighting both its strengths and the situations where stability conditions affect accuracy.

1 UML Diagram

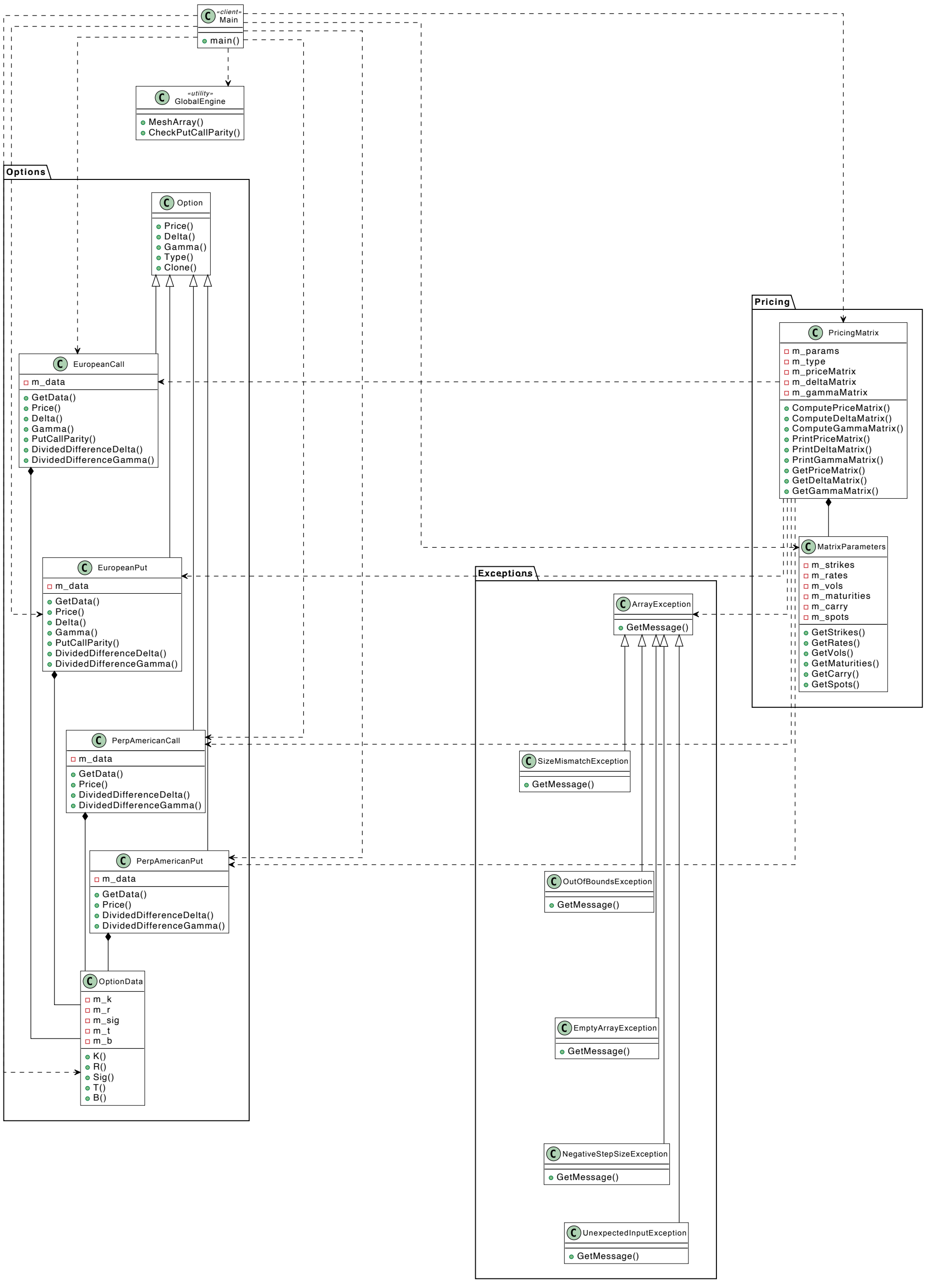

Figure 1: UML Diagram for Option Pricing Framework.

2 Option Pricing Framework

European Options

Pricing Formulas

Implementing the formulae for European options and test on 4 batches of data.

Implementation

To implement the pricing methods for European options, I created two European option classes (call and put) which derived from the base class Option. Each of these classes were created identically, they both contain the private member m_data which contains option contract parameters constructed from the OptionData class. This way, EuropeanCall and EuropeanPut can easily pull from m_data and compute seamlessly. The core methods implemented within these classes were Price(), Delta(), Gamma(), PutCallParity(), DividedDifferenceDelta(), DividedDifferenceGamma(), which were mostly implemented later in this project. Implementation for these functions require the boost C++ library for the boost normal distribution.

Price() functions for EuropeanCall and EuropeanPut classes

```
double EuropeanCall::Price(const double U) const
{ // return the price of the European call option
    if (U <= 0.0)
    {
        throw std::invalid_argument("Underlying spot price U must be positive.");
    }

    normal_distribution<> myNormal;
    // boost normal distribution
    double d1 = (log(U / m_data.K()) + (m_data.B() + 0.5 * m_data.Sig() * m_data.Sig()) * m_data.T()) / (m_data.Sig() * sqrt(m_data.T()));
    double d2 = d1 - m_data.Sig() * sqrt(m_data.T());

    // return call price
    return U * exp((m_data.B() - m_data.R()) * m_data.T()) * cdf(myNormal, d1) - m_data.K() * exp(-m_data.R() * m_data.T()) * cdf(myNormal, d2);
}

double EuropeanPut::Price(const double U) const
{ // return the price of the European put option
    if (U <= 0.0)
    {
        throw std::invalid_argument("Underlying spot price U must be positive.");
    }

```

```
    normal_distribution<> myNormal;
    // boost normal distribution
    double d1 = (log(U / m_data.K()) + (m_data.B() + 0.5 * m_data.Sig() * m_data.Sig()) * m_data.T()) / (m_data.Sig() * sqrt(m_data.T()));
    double d2 = d1 - m_data.Sig() * sqrt(m_data.T());

    // return put price
    return m_data.K() * exp(-m_data.R() * m_data.T()) * cdf(myNormal, -d2) - U * exp((m_data.B() - m_data.R()) * m_data.T()) * cdf(myNormal, -d1);
}
```

Usage in main()

```
vector<OptionData> batches =
{
    OptionData(65.0, 0.08, 0.30, 0.25, 0.08),
    OptionData(100.0, 0.0, 0.2, 1.0, 0.0),
    OptionData(10.0, 0.12, 0.50, 1.0, 0.12),
    OptionData(100.0, 0.08, 0.30, 30.0, 0.08),
};

vector<double> a_spots = { 60.0, 100.0, 5.0, 100.0 };

for (size_t i = 0; i < batches.size(); ++i)
{
    EuropeanCall call(batches[i]);
    EuropeanPut put(batches[i]);

    cout << "Batch " << (i + 1) << ": " << endl;
    cout << "Call = " << call.Price(a_spots[i]) << endl;
    cout << "Put = " << put.Price(a_spots[i]) << endl;
}
```

Terminal Output

```
Batch 1:
Call = 2.13337
Put = 5.84628
Batch 2:
Call = 7.96557
Put = 7.96557
Batch 3:
Call = 0.204058
Put = 4.07326
Batch 4:
Call = 92.1757
Put = 1.2475
```

Put-Call Parity

Checking for put-call parity.

Implementation

In my EuropeanPut and EuropeanCall class, I implemented a function PutCallParity() that returns the price of the opposite option type. For example, the EuropeanPut class returns the corresponding call price, and vice versa. This function is then utilized in a global function, CheckPutCallParity(), which verifies whether the observed market price and the implied price are consistent. In doing so, it confirms that the put–call parity relationship holds.

Global CheckPutCallParity() function

```
void CheckPutCallParity(const EuropeanCall& call, const EuropeanPut& put, double U)
{ // verify if parity holds between a put and call option
const OptionData& c = call.GetData();
const OptionData& p = put.GetData();

if (c.K() == p.K() && c.R() == p.R() && c.Sig() == p.Sig() && c.T() == p.T() && c.B() == p.B())
{ // parameters are equal, can check for parity
double callParity = put.PutCallParity(U);
// implied call price
double putParity = call.PutCallParity(U);
// implied put price
double callMkt = call.Price(U);
// market call price
double putMkt = put.Price(U);
// market put price

if (round(1000 * callParity) == round(1000 * call.Price(U)) && round(1000 * putParity) == round(1000 * put.Price(U)))
{ // parity holds
cout << "Put-Call Parity holds: No-arbitrage.\n"
     << "Call price: " << callMkt << "\n"
     << "Put price: " << putMkt << endl;
}
else
{ // parity is violated
cout << "Put-Call Parity is violated: Arbitrage opportunity detected.\n"
     << "Call price: " << callMkt << "\n"
     << "Put price: " << putMkt << endl;
    }
}
    else
    { // parameters are not equal, output error
```

```
        cout << "The option parameters are not equivalent." <<
            endl;
    }
}
```

Usage in main()

```
cout << "Applying Put-Call Parity Relationship: " << endl;
for (size_t i = 0; i < batches.size(); ++i)
{
    EuropeanCall call(batches[i]);
    EuropeanPut put(batches[i]);

    cout << "Batch " << (i + 1) << ": " << endl;
    // call global CheckPutCallParity() function
    CheckPutCallParity(call, put, a_spots[i]);
}
```

Terminal Output

```
Applying Put-Call Parity Relationship:
Batch 1:
Put-Call Parity holds: No-arbitrage.
Call price: 2.13337
Put price: 5.84628
Batch 2:
Put-Call Parity holds: No-arbitrage.
Call price: 7.96557
Put price: 7.96557
Batch 3:
Put-Call Parity holds: No-arbitrage.
Call price: 0.204058
Put price: 4.07326
Batch 4:
Put-Call Parity holds: No-arbitrage.
Call price: 92.1757
Put price: 1.2475
```

Mesh Array Pricing

Compute option prices over a monotonically increasing mesh array of underlying spot prices.

Implementation

To answer this question, I first had to create a MeshArray() function which was also a global utility function. The function was fairly straightforward, it takes input parameters start, step size, and vector size to create the mesh array. This function makes for seamless reusability throughout the remainder of the program.

Global MeshArray() function

```
template <typename T>
vector<T> MeshArray(const T start, const T step_size_h, const
    size_t vector_size)
{
    vector<T> mesh;
    // create the mesh array
    for (size_t i = 0; i < vector_size; ++i)
    {
        mesh.push_back(start + i * step_size_h);
    }

    return mesh;
}
```

Usage in main()

```
vector<double> underlying_prices = MeshArray<double>(60.0, 10.0,
    5.0);
OptionData batch1 = batches[0];
// using batch 1 data
EuropeanCall call(batch1);
EuropeanPut put(batch1);
vector<double> call_prices;
// vector to store call prices
vector<double> put_prices;
// vector to store put prices

for (vector<double>::const_iterator itr = underlying_prices.begin
    (); itr != underlying_prices.end(); ++itr)
{
    call_prices.push_back(call.Price(*itr));
    put_prices.push_back(put.Price(*itr));
}

cout << "Spot\tCall\tPut" << endl;
for (size_t i = 0; i < underlying_prices.size(); ++i)
{
    cout << underlying_prices[i] << "\t" << call_prices[i] << "\t
       " << put_prices[i] << endl;
}
```

Terminal Output

```
Testing option prices over a mesh of underlying prices for Batch
    1:
Spot    Call    Put
60      2.13337 5.84628
```

```
70       7.90027 1.61319
80       16.5879 0.300857
90       26.3282 0.0411543
100      36.2916 0.00448099
```

Matrix Pricing

Input a matrix of option parameters and receive a matrix of option prices.

Implementation

From discussions on the forums, it appeared that the task involved inputting a matrix of option parameters and receiving a one-dimensional matrix (vector) of option prices as the output. To implement this, I introduced two classes: MatrixParameters and PricingMatrix. This design mirrors the structure of my option classes. In that setup, the OptionsData class stores the contract parameters, which are then passed to specific option types through private member functions (m_data). Similarly, MatrixParameters holds matrices (vectors of vectors) of contract parameters, which are passed to PricingMatrix. MatrixParameters also has two different parameter constructors, making it very flexible. It can construct from matrices directly or take in vectors to construct a singular matrix. The PricingMatrix class can then compute option price matrices over these parameters for a given contract type. To answer this question, I used the vector constructor in MatrixParameters as I was just passing one-dimensional vectors of option parameters to construct a matrix. Below is the ComputePriceMatrix() function in the PricingMatrix class.

ComputePriceMatrix() function

```cpp
void PricingMatrix::ComputePriceMatrix()
{
    // access private member m_params from MatrixParameters
    const std::vector<std::vector<double>>& strikes = m_params.GetStrikes();
    const std::vector<std::vector<double>>& rates = m_params.GetRates();
    const std::vector<std::vector<double>>& vols = m_params.GetVols();
    const std::vector<std::vector<double>>& maturities = m_params.GetMaturities();
    const std::vector<std::vector<double>>& carry = m_params.GetCarry();
    const std::vector<std::vector<double>>& spots = m_params.GetSpots();

    // verify outer matrix dimenstions are equal
    if (strikes.size() != rates.size() || strikes.size() != vols.size() || strikes.size() != maturities.size() || strikes.size() != carry.size() || strikes.size() != spots.size())
    {
        throw SizeMismatchException();
```

```
    }

    // verify inner matrix dimensions per row are equal
    for (size_t i = 0; i < strikes.size(); ++i)
    {
        if (rates[i].size() != strikes[i].size() || vols[i].size() != strikes[i].size() || maturities[i].size() != strikes[i].size() || carry[i].size() != strikes[i].size() || spots[i].size() != strikes[i].size())
        {
            throw SizeMismatchException();
        }
    }

    // resize price matrix
    m_priceMatrix.clear();
    m_priceMatrix.resize(strikes.size());
    // rows

    for (size_t i = 0; i < strikes.size(); ++i)
    {
        m_priceMatrix[i].resize(strikes[i].size());
        // columns

        for (size_t j = 0; j < strikes[i].size(); ++j)
        {
            OptionData optionData(strikes[i][j], rates[i][j], vols[i][j], maturities[i][j], carry[i][j]);
            double spot = spots[i][j];

            if (m_type == "EuropeanCall")
            {
                EuropeanCall opt(optionData);
                m_priceMatrix[i][j] = opt.Price(spot);
            }
            else if (m_type == "EuropeanPut")
            {
                EuropeanPut opt(optionData);
                m_priceMatrix[i][j] = opt.Price(spot);
            }
            else if (m_type == "PerpAmericanCall")
            {
                PerpAmericanCall opt(optionData);
                m_priceMatrix[i][j] = opt.Price(spot);
            }
            else if (m_type == "PerpAmericanPut")
            {
                PerpAmericanPut opt(optionData);
                m_priceMatrix[i][j] = opt.Price(spot);
            }
            else
```

```
                {
                    throw UnexpectedInputException();
                }
            }
        }
}
```

Usage in main()

```
try
{
    cout << "Computing European Call and Put option Price Matrices: " << endl;

    // vectors must be of same size for mesh array function
    size_t matrix_col_length = 10;

    // create vectors of parameters
    vector<double> spots = MeshArray(50.0, 10.0, matrix_col_length);
    vector<double> strikes = MeshArray(60.0, 10.0, matrix_col_length);
    vector<double> maturities = MeshArray(0.1, 0.1, matrix_col_length);
    vector<double> vols = MeshArray(0.1, 0.05, matrix_col_length);
    vector<double> rates = MeshArray(0.1, 0.02, matrix_col_length);
    vector<double> carry = MeshArray(0.1, 0.02, matrix_col_length);

    // use MatrixParameters class vector parameter constructor
    MatrixParameters vector_params(strikes, rates, vols, maturities, carry, spots);
    // use PricingMatrix class parameter constructor
    PricingMatrix call_matrix(vector_params, "EuropeanCall");
    PricingMatrix put_matrix(vector_params, "EuropeanPut");

    // compute price matrices
    call_matrix.ComputePriceMatrix();
    put_matrix.ComputePriceMatrix();

    // print price matrices
    call_matrix.PrintPriceMatrix();
    put_matrix.PrintPriceMatrix();

} catch (const ArrayException& e) {
    cout << "Error: " << e.GetMessage() << endl;
} catch (...) {
    cout << "Unexpected error." << endl;
```

```
}
```

Terminal Output

```
Computing European Call and Put option Price Matrices:
EuropeanCall Price Matrix:
0.00000   0.04273   0.90509   3.26619   6.98458   11.91441   17.98779   25.16721   33.41924   42.70351
EuropeanPut Price Matrix:
9.40299   8.38273   7.61468   7.68664   8.37770   9.47565   10.86043   12.45710   14.20989   16.07107
```

European Option Greeks

Delta and Gamma Implementation

Implement delta and gamma formulae and test using the provided data set:
$K = 100$, $S = 105$, $T = 0.5$, $r = 0.1$, $b = 0$, sig $= 0.36$
(exact delta call = 0.5946, exact delta put = -0.3566)

Implementation

To implement the functionality required in this question, all I had to do was add the Delta() and Gamma() methods to my EuropeanCall and EuropeanPut classes. Very straightforward, just implementing the correct formulae and returning either delta or gamma.

Delta() function in EuropeanCall class

```cpp
double EuropeanCall::Delta(const double U) const
{ // return the delta of the European call option
    if (U <= 0.0)
    {
        throw std::invalid_argument("Underlying spot price U must be positive.");
    }

    normal_distribution<> myNormal;
    // boost normal distribution
    double d1 = (log(U / m_data.K()) + (m_data.B() + 0.5 * m_data.Sig() * m_data.Sig()) * m_data.T()) / (m_data.Sig() * sqrt(m_data.T()));

    // return delta
    return exp((m_data.B() - m_data.R()) * m_data.T()) * cdf(myNormal, d1);
}
```

Usage in main()

```
OptionData partials_call(100.0, 0.1, 0.36, 0.5, 0.0);
OptionData partials_put(100.0, 0.1, 0.36, 0.5, 0.0);
double partials_strike = 105.0;
EuropeanCall call_greeks(partials_call);
EuropeanPut put_greeks(partials_put);

cout << "Computing Delta and Gamma for European options: " << endl;
cout << "European Call Delta: " << call_greeks.Delta(partials_strike) << endl;
cout << "European Call Gamma: " << call_greeks.Gamma(partials_strike) << endl;
cout << "European Put Delta: " << put_greeks.Delta(partials_strike) << endl;
cout << "European Put Gamma: " << put_greeks.Gamma(partials_strike) << endl;
```

Terminal Output

```
Computing Delta and Gamma for European options:
European Call Delta: 0.59463
European Call Gamma: 0.01349
European Put Delta: -0.35660
European Put Gamma: 0.01349
```

Mesh Array Greeks Pricing

Compute the call option delta over a monotonically increasing mesh array of underlying spot prices.

Implementation

Since I previously created the MeshArray() function, I was able to simply reuse it in this case. Thus, there was really no new program design in this question. I also chose to calculate both call option delta and gamma as well because I read that we should in the forum.

Usage in main()

```
vector<double> spot_mesh = MeshArray(105.0, 5.0, 5.0);
// create mesh array of spot prices
vector<double> call_deltas;
// vector to store deltas
vector<double> call_gammas;
// vector to store gammas

// compute and store deltas and gammas in their respective vector
```

```
for (vector<double>::const_iterator itr = spot_mesh.begin(); itr != spot_mesh.end(); ++itr)
{
    call_deltas.push_back(call_greeks.Delta(*itr));
    call_gammas.push_back(call_greeks.Gamma(*itr));
}

cout << "Computing European call option deltas and gammas for a monotonically increasing mesh of spot prices:" << endl;
cout << "Spot\t\tDelta\tGamma" << endl;
for (size_t i = 0; i < spot_mesh.size(); ++i)
{
    cout << spot_mesh[i] << "\t" << call_deltas[i] << "\t" << call_gammas[i] << endl;
}
```

Terminal Output

```
Computing European call option deltas and gammas for a monotonically increasing mesh of spot prices:
Spot            Delta   Gamma
105.00000       0.59463 0.01349
110.00000       0.65831 0.01195
115.00000       0.71397 0.01031
120.00000       0.76149 0.00870
125.00000       0.80120 0.00721
```

Greeks Matrix Pricing

Incorporate this into the matrix pricing code, so we are able to receive a matrix of deltas or gammas as the result.

Implementation

My implementation for this solution was done identically to how the ComputePriceMatrix() function was done, however instead of computing price, we compute delta or gamma. So I made two separate functions ComputeDeltaMatrix() and ComputeGammaMatrix() to handle this functionality. In main(), I then tested the matrix parameter constructor in the MatrixParameters class to pass in matrices of contract parameters and receive matrices of both delta and gamma as a result.

Usage in main()

```
// create matrices of option parameters (5 by 5 matrix)
vector<vector<double>> strikes =
{{60.0, 65.0, 70.0, 75.0, 80.0},{85.0, 90.0, 95.0, 100.0, 105.0},
{110.0, 115.0, 120.0, 125.0, 130.0},{135.0, 140.0, 145.0, 150.0, 155.0},{160.0, 165.0, 170.0, 175.0, 180.0}};
vector<vector<double>> rates =
```

```
{{0.01, 0.02, 0.03, 0.04, 0.05},{0.06, 0.07, 0.08, 0.09,
    0.10},{0.01, 0.02, 0.03, 0.04, 0.05},{0.06, 0.07, 0.08, 0.09,
    0.10},{0.01, 0.02, 0.03, 0.04, 0.05}};
vector<vector<double>> vols =
{{0.10, 0.15, 0.20, 0.25, 0.30},{0.12, 0.18, 0.22, 0.28,
    0.32},{0.10, 0.15, 0.20, 0.25, 0.30},{0.12, 0.18, 0.22, 0.28,
    0.32},{0.10, 0.15, 0.20, 0.25, 0.30}};
vector<vector<double>> maturities =
{{0.1, 0.2, 0.3, 0.4, 0.5},{0.6, 0.7, 0.8, 0.9, 1.0},{0.1, 0.2,
    0.3, 0.4, 0.5},{0.6, 0.7, 0.8, 0.9, 1.0},{0.1, 0.2, 0.3, 0.4,
    0.5}};
vector<vector<double>> carry =
{{0.01, 0.01, 0.01, 0.01, 0.01},{0.02, 0.02, 0.02, 0.02,
    0.02},{0.01, 0.01, 0.01, 0.01, 0.01},{0.02, 0.02, 0.02, 0.02,
    0.02},{0.01, 0.01, 0.01, 0.01, 0.01}};
vector<vector<double>> spots =
{{50.0, 55.0, 60.0, 65.0, 70.0},{75.0, 80.0, 85.0, 90.0,
    95.0},{100.0, 105.0, 110.0, 115.0, 120.0},{125.0, 130.0,
    135.0, 140.0, 145.0},{150.0, 155.0, 160.0, 165.0, 170.0}};

// use MatrixParameters matrix constructor
MatrixParameters matrixParams(strikes, rates, vols, maturities,
    carry, spots);
PricingMatrix callMatrix(matrixParams, "EuropeanCall");
PricingMatrix putMatrix(matrixParams, "EuropeanPut");

try
{
    cout << "Computing European Call Delta Matrix:" << endl;
    // call ComputeDeltaMatrix() function
    callMatrix.ComputeDeltaMatrix();
    callMatrix.PrintDeltaMatrix();

    cout << "\nComputing European Call Gamma Matrix: " << endl;
    // call ComputeGammaMatrix() function
    callMatrix.ComputeGammaMatrix();
    callMatrix.PrintGammaMatrix();

    cout << "\nComputing European Put Delta Matrix:" << endl;
    putMatrix.ComputeDeltaMatrix();
    putMatrix.PrintDeltaMatrix();

    cout << "\nComputing European Put Gamma Matrix: " << endl;
    putMatrix.ComputeGammaMatrix();
    putMatrix.PrintGammaMatrix();

} catch (const ArrayException& e) {
    cout << "Error: " << e.GetMessage() << endl;
} catch (...) {
    cout << "Unexpected error." << endl;
}
```

Terminal Output

```
Computing European Call Delta Matrix:
EuropeanCall Delta Matrix:
   0.00000   0.00760   0.09203   0.20913   0.30248
   0.11794   0.26039   0.33352   0.39650   0.42836
   0.00151   0.09785   0.23676   0.33214   0.39422
   0.25098   0.36020   0.40720   0.44733   0.46675
   0.02311   0.19214   0.31682   0.38973   0.43560

Computing European Call Gamma Matrix:
EuropeanCall Gamma Matrix:
   0.00000   0.00568   0.02508   0.02784   0.02324
   0.02815   0.02648   0.02111   0.01537   0.01206
   0.00155   0.02451   0.02554   0.01982   0.01490
   0.02710   0.01867   0.01408   0.01006   0.00794
   0.01153   0.02626   0.02025   0.01458   0.01074

Computing European Put Delta Matrix:
EuropeanPut Delta Matrix:
  -1.00000  -0.99040  -0.90199  -0.77894  -0.67771
  -0.85834  -0.70522  -0.61962  -0.54245  -0.49475
  -0.99849  -0.90015  -0.75725  -0.65593  -0.58597
  -0.72531  -0.60540  -0.54594  -0.49162  -0.45637
  -0.97689  -0.80587  -0.67720  -0.59834  -0.54459

Computing European Put Gamma Matrix:
EuropeanPut Gamma Matrix:
   0.00000   0.00568   0.02508   0.02784   0.02324
   0.02815   0.02648   0.02111   0.01537   0.01206
   0.00155   0.02451   0.02554   0.01982   0.01490
   0.02710   0.01867   0.01408   0.01006   0.00794
   0.01153   0.02626   0.02025   0.01458   0.01074
```

Divided Difference Method

Implement the divided difference method to approximate option sensitivities (delta and gamma). This will be done through numerical methods, specifically, the divided difference method to approximate delta and gamma.

Implementation

Implementation of the DividedDifferenceDelta() and DividedDifferenceGamma() functions was encapsulated in each European option class using the provided formulae. I used certain error handling for input parameter h to make sure that it wasn't too small for computation. Other than that, the implementation was very simple and I tested it against different values of h to see how accuracy changed. I then compared these results to the results obtained earlier in parts a) and b).

DividedDifferenceDelta() function in EuropeanCall class

```cpp
double EuropeanCall::DividedDifferenceDelta(const double U, const double h) const
{ // delta approximation using the divided difference method
    if (abs(Price(U + h) - Price(U - h)) <= std::pow(2, -53))
    {
        std::cout << "H␣is␣too␣small␣for␣accurate␣computation." << std::endl;
        return 0;
    }

    // return approximated delta
    return (Price(U + h) - Price(U - h)) / (2.0 * h);
}
```

Usage in main()

```cpp
// use the same option data as in part a) for comparison
OptionData divided_difference_call_params(100.0, 0.1, 0.36, 0.5, 0.0);
OptionData divided_difference_put_params(100.0, 0.1, 0.36, 0.5, 0.0);

// set spot price
double divided_difference_strike = 105.0;

// pass params to EuropeanCall and EuropeanPut objects
EuropeanCall divided_difference_call(divided_difference_call_params);
EuropeanPut divided_difference_put(divided_difference_put_params);

// to compare part a) results
cout << "Computing␣approximate␣Delta␣and␣Gamma␣for␣European␣options␣using␣Divided␣Differences:␣" << endl;

cout << "European␣Call␣Delta␣(h␣=␣1.0):␣" << divided_difference_call.DividedDifferenceDelta(divided_difference_strike, 1) << endl;

cout << "European␣Call␣Delta␣(h␣=␣1e-4):␣" << divided_difference_call.DividedDifferenceDelta(divided_difference_strike, 1e-4) << endl;

cout << "European␣Call␣Gamma␣(h␣=␣1.0):␣" << divided_difference_call.DividedDifferenceGamma(divided_difference_strike, 1) << endl;

```

```cpp
cout << "European Call Gamma (h = 1e-4): " << divided_difference_call.DividedDifferenceGamma(divided_difference_strike, 1e-4) << endl;

cout << "European Put Delta (h = 1.0): " << divided_difference_put.DividedDifferenceDelta(divided_difference_strike, 1) << endl;

cout << "European Put Delta (h = 1e-4): " << divided_difference_put.DividedDifferenceDelta(divided_difference_strike, 1e-4) << endl;

cout << "European Put Gamma (h = 1.0): " << divided_difference_put.DividedDifferenceGamma(divided_difference_strike, 1) << endl;

cout << "European Put Gamma (h = 1e-4): " << divided_difference_put.DividedDifferenceGamma(divided_difference_strike, 1e-4) << endl;

// to compare part b) results
cout << "\nComputing approximate European Call Delta and Gamma for a monotonically increasing mesh of spot prices: " << endl;

vector<double> divided_difference_mesh = MeshArray(105.0, 5.0, 5.0);
// same mesh array of spot prices as part b)
vector<double> mesh_deltas;
// vector to store computed deltas
vector<double> mesh_gammas;
// vector to store computed gammas

// calculate and store in respective vector
for (vector<double>::const_iterator itr = divided_difference_mesh.begin(); itr != divided_difference_mesh.end(); ++itr)
{
    mesh_deltas.push_back(divided_difference_call.DividedDifferenceDelta(*itr, 1e-4));

    mesh_gammas.push_back(divided_difference_call.DividedDifferenceGamma(*itr, 1e-4));
}

// output vectors
cout << "Spot\t\tDelta\tGamma" << endl;
for (size_t i = 0; i < divided_difference_mesh.size(); ++i)
{
    cout << divided_difference_mesh[i] << "\t" << mesh_deltas[i] << "\t" << mesh_gammas[i] << endl;
}
```

Terminal Output

```
Computing approximate Delta and Gamma for European options using
   Divided Differences:
European Call Delta (h = 1.0): 0.59458
European Call Delta (h = 1e-4): 0.59463
European Call Gamma (h = 1.0): 0.01349
European Call Gamma (h = 1e-4): 0.01349
European Put Delta (h = 1.0): -0.35665
European Put Delta (h = 1e-4): -0.35660
European Put Gamma (h = 1.0): 0.01349
European Put Gamma (h = 1e-4): 0.01349

Computing approximate European Call Delta and Gamma for a
   monotonically increasing mesh of spot prices:
Spot            Delta   Gamma
105.00000       0.59463 0.01349
110.00000       0.65831 0.01195
115.00000       0.71397 0.01031
120.00000       0.76149 0.00871
125.00000       0.80120 0.00721
```

Compare with Exact Solutions

```
Originial part a) output:
European Call Delta: 0.59463
European Call Gamma: 0.01349
European Put Delta: -0.35660
European Put Gamma: 0.01349

Original part b) output:
Spot            Delta   Gamma
105.00000       0.59463 0.01349
110.00000       0.65831 0.01195
115.00000       0.71397 0.01031
120.00000       0.76149 0.00870
125.00000       0.80120 0.00721
```

Takeaway: As h gets smaller, accuracy improves in delta and gamma approximation, which is expected. I also seem to have found the perfect h value for approximation through experimenting, which was found to be $h = 1e^{-4}$. This gave me the exact same results as those computed in parts a) and b).

2.1 Perpetual American Options

Pricing Formulas

Implementing proper formulae in the program to price Perpetual American options.

Implementation

The implementation for this type of option mirrored how I implemented European options. I created a class for both a PerpAmericanCall and PerpAmericanPut, which are derived from the base class Option. In these classes, they contain the private member function m_data, which contains contract parameters from the class OptionData used for pricing methods. The core methods implemented in these classes are Price(), DividedDifferenceDelta(), and DividedDifferenceGamma().

Price() function for PerpAmericanCall and PerpAmericanPut classes

```
double PerpAmericanCall::Price(const double U) const
{ // return the price of the perpetual american call option
    if (U <= 0.0)
    {
        throw std::invalid_argument("Underlying spot price U must be positive.");
    }

    double sigma_squared = m_data.Sig() * m_data.Sig();

    double y1 = 0.5 - (m_data.B() / sigma_squared) + std::sqrt(std::pow((m_data.B() / sigma_squared - 0.5), 2.0) + (2.0 * m_data.R() / sigma_squared));

    double call_rhs = ((y1 - 1.0) / y1) * (U / m_data.K());

    // return call price
    return (m_data.K() / (y1 - 1.0)) * std::pow(call_rhs, y1);
}

double PerpAmericanPut::Price(const double U) const
{ // return the price of the perpetual american put option
    if (U <= 0.0)
    {
        throw std::invalid_argument("Underlying spot price U must be postive.");
    }

    double sigma_squared = m_data.Sig() * m_data.Sig();

    double y2 = 0.5 - (m_data.B() / sigma_squared) - std::sqrt(std::pow((m_data.B() / sigma_squared - 0.5), 2.0) + (2.0 * m_data.R() / sigma_squared));

```

```
    double put_rhs = ((y2 - 1.0) / y2) * (U / m_data.K());

    // return put price
    return (m_data.K() / (1.0 - y2)) * std::pow(put_rhs, y2);
}
```

Testing Pricing Classes

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

Usage in main()

```
cout << "Computing Perpetual American Call and Put Prices:" <<
    endl;

OptionData perpetual_options_data(100.0, 0.1, 0.1, 0.0, 0.02);
// set T = 0
PerpAmericanCall perp_call(perpetual_options_data);
PerpAmericanPut perp_put(perpetual_options_data);

cout << "Perpetual American Call Price: " << perp_call.Price
    (110.0) << endl;
// spot price = 110.0
// Call price: 18.50350
cout << "Perpetual American Put Price: " << perp_put.Price(110.0)
     << endl;
// spot price = 110.0
// Put price: 3.03106
```

Terminal Output

```
Computing Perpetual American Call and Put Prices:
Perpetual American Call Price: 18.50350
Perpetual American Put Price: 3.03106
```

Mesh Array Pricing

Compute the call and put option price over a monotonically increasing mesh array or underlying spot prices.

Implementation

For this implementation, I was able to reuse my global MeshArray() function created earlier. Thus, there was no further program development needed.

Usage in main()

```cpp
cout << "Computing␣Perpetual␣American␣Call␣and␣Put␣Prices␣for␣a␣monotonically␣increasing␣mesh␣of␣spot␣prices:␣" << endl;

vector<double> perp_options_mesh = MeshArray(110.0, 5.0, 5.0);
// create the mesh array
vector<double> perp_call_prices;
// vector to store call prices
vector<double> perp_put_prices;
// vector to store put prices

// calculate and store in respective vector, we will be using same option data parameters as used in Question a) and b)
for (vector<double>::const_iterator itr = perp_options_mesh.begin(); itr != perp_options_mesh.end(); ++itr)
{
    perp_call_prices.push_back(perp_call.Price(*itr));
    perp_put_prices.push_back(perp_put.Price(*itr));
}

// output calculated vectors
cout << "Spot\t\tCall\t\tPut" << endl;
for (size_t i = 0; i < perp_options_mesh.size(); ++i)
{
    cout << perp_options_mesh[i] << "\t" << perp_call_prices[i] << "\t" << perp_put_prices[i] << endl;
}
```

Terminal Output

```
Computing Perpetual American Call and Put Prices for a monotonically increasing mesh of spot prices:
Spot            Call            Put
110.00000       18.50350        3.03106
115.00000       21.34806        2.29919
120.00000       24.48045        1.76467
125.00000       27.91596        1.36913
130.00000       31.67005        1.07287
```

Matrix Pricing

Incorporate this into the matrix pricing code. This way you can input a matrix of option parameters and receive a matrix of Perpetual American option prices.

Implementation

The implementation for this was very easy due to the flexibility in my ComputePriceMatrix() function. All that was required was the addition of an else-if statement for the string type to compute a Perpetual American option (PerpAmericanCall or PerpAmericanPut). I also added the functionality to compute matrices of approximated deltas and gammas for these options using the divided difference method. This required passing through the parameter h which I hard coded to be $h = 1e^{-1}$ due to the accuracy I had after experimenting with different values. This can also easily be changed by going into the header file and adjusting the h parameter. A preview of the implementation is below by showing the ComputeDeltaMatrix() function.

ComputeDeltaMatrix() function

```cpp
void PricingMatrix::ComputeDeltaMatrix(const double h)
{ // compute a delta matrix for a given matrix of option parameters
    const std::vector<std::vector<double>>& strikes = m_params.GetStrikes();
    const std::vector<std::vector<double>>& rates = m_params.GetRates();
    const std::vector<std::vector<double>>& vols = m_params.GetVols();
    const std::vector<std::vector<double>>& maturities = m_params.GetMaturities();
    const std::vector<std::vector<double>>& carry = m_params.GetCarry();
    const std::vector<std::vector<double>>& spots = m_params.GetSpots();

    // verify outer matrix dimenstions are equal
    if (strikes.size() != rates.size() || strikes.size() != vols.size() || strikes.size() != maturities.size() || strikes.size() != carry.size() || strikes.size() != spots.size())
    {
        throw SizeMismatchException();
    }

    // verify inner matrix dimensions per row are equal
    for (size_t i = 0; i < strikes.size(); ++i)
    {
        if (rates[i].size() != strikes[i].size() || vols[i].size() != strikes[i].size() || maturities[i].size() != strikes[i].size() || carry[i].size() != strikes[i].size() || spots[i].size() != strikes[i].size())
        {
```

```
                throw SizeMismatchException();
            }
        }

        // resize delta matrix
        m_deltaMatrix.clear();
        m_deltaMatrix.resize(strikes.size());

        for (size_t i = 0; i < strikes.size(); ++i)
        {
            m_deltaMatrix[i].resize(strikes[i].size());

            for (size_t j = 0; j < strikes[i].size(); ++j)
            {
                OptionData optionData(strikes[i][j], rates[i][j],
                   vols[i][j], maturities[i][j], carry[i][j]);
                double spot = spots[i][j];

                // find string type and compute

                if (m_type == "EuropeanCall")
                {
                    EuropeanCall opt(optionData);
                    m_deltaMatrix[i][j] = opt.Delta(spot);
                }
                else if (m_type == "EuropeanPut")
                {
                    EuropeanPut opt(optionData);
                    m_deltaMatrix[i][j] = opt.Delta(spot);
                }
                else if (m_type == "PerpAmericanCall")
                {
                    PerpAmericanCall opt(optionData);
                    m_deltaMatrix[i][j] = opt.DividedDifferenceDelta(
                       spot, h);
                }
                else if (m_type == "PerpAmericanPut")
                {
                    PerpAmericanPut opt(optionData);
                    m_deltaMatrix[i][j] = opt.DividedDifferenceDelta(
                       spot, h);
                }
                else
                {
                    throw UnexpectedInputException();
                }
            }
        }
}
```

Usage in main()

```cpp
// create matrices of option parameters (5 x 5 matrices)
// b < r for the computation to work *Important
vector<vector<double>> perp_strikes =
{{50.0, 55.0, 60.0, 65.0, 70.0},{75.0, 80.0, 85.0, 90.0, 95.0},{100.0, 105.0, 110.0, 115.0, 120.0},{125.0, 130.0, 135.0, 140.0, 145.0},{150.0, 155.0, 160.0, 165.0, 170.0}};
vector<vector<double>> perp_rates =
{{0.10, 0.10, 0.10, 0.10, 0.10},{0.12, 0.12, 0.12, 0.12, 0.12},{0.14, 0.14, 0.14, 0.14, 0.14},{0.16, 0.16, 0.16, 0.16, 0.16},{0.18, 0.18, 0.18, 0.18, 0.18}};
vector<vector<double>> perp_vols =
{{0.10, 0.15, 0.20, 0.25, 0.30},{0.12, 0.17, 0.22, 0.27, 0.32},{0.14, 0.19, 0.24, 0.29, 0.34},{0.16, 0.21, 0.26, 0.31, 0.36},{0.18, 0.23, 0.28, 0.33, 0.38}};
// T = 0 for perpetual American options
vector<vector<double>> perp_maturities =
{{0.0, 0.0, 0.0, 0.0, 0.0},{0.0, 0.0, 0.0, 0.0, 0.0},{0.0, 0.0, 0.0, 0.0, 0.0},{0.0, 0.0, 0.0, 0.0, 0.0},{0.0, 0.0, 0.0, 0.0, 0.0}};
vector<vector<double>> perp_carry =
{{0.01, 0.02, 0.03, 0.04, 0.05},{0.01, 0.02, 0.03, 0.04, 0.05},{0.01, 0.02, 0.03, 0.04, 0.05},{0.01, 0.02, 0.03, 0.04, 0.05},{0.01, 0.02, 0.03, 0.04, 0.05}};
vector<vector<double>> perp_spots =
{{50.0, 55.0, 60.0, 65.0, 70.0},{75.0, 80.0, 85.0, 90.0, 95.0},{100.0, 105.0, 110.0, 115.0, 120.0},{125.0, 130.0, 135.0, 140.0, 145.0},{150.0, 155.0, 160.0, 165.0, 170.0}};

// use MatrixParameters matrix constructor
MatrixParameters perp_matrix_params(perp_strikes, perp_rates, perp_vols, perp_maturities, perp_carry, perp_spots);

PricingMatrix perp_call_matrix(perp_matrix_params, "PerpAmericanCall");
PricingMatrix perp_put_matrix(perp_matrix_params, "PerpAmericanPut");

try
{
    cout << "Computing Perpetual American Call Price Matrix:" << endl;
    perp_call_matrix.ComputePriceMatrix();
    perp_call_matrix.PrintPriceMatrix();

    cout << "\nComputing Perpetual American Put Price Matrix:" << endl;
    perp_put_matrix.ComputePriceMatrix();
    perp_put_matrix.PrintPriceMatrix();

```

```
    cout << "\nApproximating Perpetual American Call Deltas Using the Divided Difference Method (h = 0.0001):" << endl;
    perp_call_matrix.ComputeDeltaMatrix();
    perp_call_matrix.PrintDeltaMatrix();

    cout << "\nApproximating Perpetual American Call Gammas Using the Divided Difference Method (h = 0.0001):" << endl;
    perp_call_matrix.ComputeGammaMatrix();
    perp_call_matrix.PrintGammaMatrix();

    cout << "\nApproximating Perpetual American Put Deltas Using the Divided Difference Method (h = 0.0001):" << endl;
    perp_put_matrix.ComputeDeltaMatrix();
    perp_put_matrix.PrintDeltaMatrix();

    cout << "\nApproximating Perpetual American Put Gammas Using the Divided Difference Method (h = 0.0001):" << endl;
    perp_put_matrix.ComputeGammaMatrix();
    perp_put_matrix.PrintGammaMatrix();

} catch (const ArrayException& e) {
    cout << "Error: " << e.GetMessage() << endl;
} catch (...) {
    cout << "Unexpected error." << endl;
}
```

Terminal Output

```
Computing Perpetual American Call Price Matrix:
PerpAmericanCall Price Matrix:
   5.27344    9.64471   15.00000   21.41474   28.98287
   8.16222   13.51932   19.74407   26.86789   34.93090
  11.29627   17.45767   24.39484   32.11503   40.63138
  14.67627   21.52479   29.07451   37.31877   46.25577
  18.29429   25.74780   33.84046   42.55740   51.88807

Computing Perpetual American Put Price Matrix:
PerpAmericanPut Price Matrix:
   3.34898    5.20830    7.39202    9.89313   12.70628
   5.78085    8.24727   11.02933   14.11311   17.48878
   8.59507   11.61025   14.93059   18.53791   22.41934
  11.73922   15.25816   19.07065   23.15616   27.49980
  15.17657   19.16301   23.43087   27.95844   32.72960

Approximating Perpetual American Call Deltas Using the Divided Difference Method (h = 1e-4):
PerpAmericanCall Delta Matrix:
   0.42187    0.45905    0.50000    0.54503    0.59464
   0.42364    0.45562    0.49016    0.52733    0.56724
   0.42581    0.45415    0.48436    0.51641    0.55030
```

```
   0.42815   0.45378   0.48084   0.50926   0.53902
   0.43055   0.45407   0.47872   0.50442   0.53114

Approximating Perpetual American Call Gammas Using the Divided
   Difference Method (h = 1e-4):
PerpAmericanCall Gamma Matrix:
   0.02531   0.01350   0.00833   0.00549   0.00371
   0.01634   0.00966   0.00640   0.00449   0.00324
   0.01179   0.00749   0.00521   0.00381   0.00287
   0.00907   0.00608   0.00439   0.00331   0.00257
   0.00726   0.00508   0.00378   0.00292   0.00231

Approximating Perpetual American Put Deltas Using the Divided
   Difference Method (h = 1e-4):
PerpAmericanPut Delta Matrix:
  -0.33490  -0.32155  -0.30800  -0.29441  -0.28086
  -0.33001  -0.31754  -0.30491  -0.29226  -0.27968
  -0.32574  -0.31398  -0.30210  -0.29023  -0.27843
  -0.32192  -0.31076  -0.29951  -0.28829  -0.27714
  -0.31845  -0.30780  -0.29709  -0.28642  -0.27583

Approximating Perpetual American Put Gammas Using the Divided
   Difference Method (h = 1e-4):
PerpAmericanPut Gamma Matrix:
   0.04019   0.02570   0.01797   0.01329   0.01022
   0.02324   0.01620   0.01202   0.00930   0.00742
   0.01560   0.01148   0.00886   0.00707   0.00578
   0.01140   0.00872   0.00692   0.00565   0.00470
   0.00881   0.00693   0.00562   0.00467   0.00395
```

3 Monte Carlo Simulation

Monte Carlo Simulation on Batch 1 and 2

Batch 1 Parameters: T = 0.25, K = 65, sig = 0.30, r = 0.08, S = 60
Batch 2 Parameters: T = 1.0, K = 100, sig = 0.2, r = 0.0, S = 100

Batch 1: European Call Option

Table 1: Monte Carlo Pricing Results vs. Exact Solution

NT (steps)	NSIM (sims)	Monte Carlo Call	Exact Call	Diff
100	100,000	2.13043	2.13337	0.00294
250	250,000	2.14390	2.13337	-0.01053
500	500,000	2.12530	2.13337	0.00807
500	1,000,000	2.13071	2.13337	0.00266
500	10,000,000	2.13391	2.13337	5.4×10^{-4}

Batch 1: European Put Option

Table 2: Monte Carlo Pricing Results vs. Exact Solution

NT (steps)	NSIM (sims)	Monte Carlo Put	Exact Put	Diff
100	100,000	5.87321	5.84628	-0.02693
250	250,000	5.85586	5.84628	-0.00958
500	500,000	5.85493	5.84628	-0.00865
500	1,000,000	5.84125	5.84628	0.00503
500	10,000,000	5.84542	5.84628	8.6×10^{-4}

Batch 2: European Call Option

Table 3: Monte Carlo Pricing Results vs. Exact Solution

NT (steps)	NSIM (sims)	Monte Carlo Call	Exact Call	Diff
100	100,000	7.94362	7.96557	0.02195
250	250,000	7.98572	7.96557	-0.02015
500	500,000	7.94180	7.96557	0.02377
500	1,000,000	7.96142	7.96557	0.00415
500	10,000,000	7.96866	7.96557	-0.00309

Batch 2: European Put Option

Table 4: Monte Carlo Pricing Results vs. Exact Solution

NT (steps)	NSIM (sims)	Monte Carlo Put	Exact Put	Diff
100	100,000	8.00790	7.96557	-0.04233
250	250,000	7.98104	7.96557	-0.01547
500	500,000	7.97869	7.96557	-0.01312
500	1,000,000	7.95663	7.96557	0.00894
500	10,000,000	7.96539	7.96557	1.8×10^{-4}

Takeaways: After experimenting with different NT and NSIM values, to achieve near-perfect accuracy, I found that NT = 500 and NSIM = 10,000,000 obtained values within 4 decimal places. However, computational efficiency is definitely a factor to consider when running these tests, as higher NSIM values drastically slowed down the program, and lower NSIM values were still able to produce prices within 2 decimal places. I found that ultimately increasing NSIM reduces the random fluctuations in the results, while increasing NT reduces the step-size error. Low values of either one can lead to a computed price with a larger margin of error, but using high NT and NSIM gives results that match the exact solution very closely, as expected.

Stress Testing

Stress Testing Batch 4 Parameters: T = 30.0, K = 100.0, sig = 0.30, r = 0.08, S = 100.0

Aiming to get within 2 decimal places with the most efficient NT and NSIM values.

Table 5: Stress Testing Monte Carlo Accuracy

NT (steps)	NSIM (sims)	MC Call	MC Put	Exact Call	Exact Put
50	50,000	84.695	1.35766	92.17570	1.24750
75	75,000	89.4762	1.31453	92.17570	1.24750
100	100,000	89.4248	1.29604	92.17570	1.24750
150	150,000	91.474	1.27424	92.17570	1.24750
200	200,000	92.2352	1.26430	92.17570	1.24750
250	250,000	91.8867	1.26552	92.17570	1.24750
300	500,000	92.049	1.26727	92.17570	1.24750
350	750,000	91.4314	1.26304	92.17570	1.24750
450	1,000,000	91.7719	1.25421	92.17570	1.24750
500	5,000,000	91.6479	1.25478	92.17570	1.24750
500	10,000,000	91.6058	1.25594	92.17570	1.24750
1000	10,000,000	91.8100	1.25167	92.17570	1.24750
1000	50,000,000	91.6124	1.2513	92.17570	1.24750

Takeaways: Performing the stress test on batch 4 clearly shows that the time-to-maturity is the main factor driving variance in our simulations. In earlier batches with T = 0.5 and T = 1.0, we were able to obtain much more accurate prices. Even with NT up to 1,000 and NSIM in the tens of millions, the computed prices for the call and put did not consistently reach two-decimal accuracy. This is expected, as the extremely long maturity amplifies the variance of the underlying asset, making it harder to produce an accurate price. Not to mention, these computations take very long to process and are almost not worth it in terms of efficiency. With time-to-maturities as long as T = 30, you are almost better off choosing NT ≤ 500 and NSIM $\leq 1,000,000$ as you still produce similar results with much better efficiency.

Standard Deviation and Standard Error

Implementing functions to compute the standard deviation and standard error of Monte Carlo simulation.

StandardDeviation() function

```cpp
double StandardDeviation(const std::vector<double>& NSIM, double
    r, double T)
{ // compute the standard deviation of Monte Carlo results
    // check for at least 2 NSIM
    if (NSIM.size() < 2) {
    return 0.0;
    }

    double sum = 0.0;
    double sum_squared = 0.0;

    for (std::vector<double>::const_iterator itr = NSIM.begin();
        itr != NSIM.end(); ++itr)
    {
        sum += *itr;
        sum_squared += (*itr) * (*itr);
    }

    double mean_squared = (sum * sum) / static_cast<double>(NSIM.
        size());
    double variance = (sum_squared - mean_squared) / static_cast<
        double>(NSIM.size() - 1);

    // return standard deviation
    return sqrt(variance) * exp(-r * T);
}
```

StandardError() function

```cpp
double StandardError(const std::vector<double>& NSIM, double r,
    double T)
{ // compute the standard error of Monte Carlo results
    // error handling
    if (NSIM.size() == 0) {
        return 0.0;
    }

    // call StandardDeviation() function
    double sd = StandardDeviation(NSIM, r, T);

    return sd / sqrt(static_cast<double>(NSIM.size()));
}
```

Comparing the standard deviation and standard error for simulation results on batches 1 and 2. Recall:
Batch 1 Parameters: T = 0.25, K = 65, sig = 0.30, r = 0.08, S = 60
Batch 2 Parameters: T = 1.0, K = 100, sig = 0.2, r = 0.0, S = 100

NT	**NSIM**	**MC Call**	**Exact Call**	**SD**	**SE**
100	100,000	2.13043	2.13337	4.51298	0.0142713
250	250,000	2.14390	2.13337	4.54275	0.0090855
500	500,000	2.12530	2.13337	4.51365	0.00638326
500	1,000,000	2.13071	2.13337	4.51286	0.00451286
500	10,000,000	2.13391	2.13337	4.51504	0.00142778

Table 6: Batch 1: Monte Carlo vs Exact Option Call Prices with SD and SE

NT	**NSIM**	**MC Put**	**Exact Put**	**SD**	**SE**
100	100,000	5.87321	5.84628	6.05775	0.0191563
250	250,000	5.85586	5.84628	6.05028	0.0121006
500	500,000	5.85493	5.84628	6.05373	0.00856126
500	1,000,000	5.84125	5.84628	6.04743	0.00604743
500	10,000,000	5.84542	5.84628	6.04884	0.00191281

Table 7: Batch 1: Monte Carlo vs Exact Option Put Prices with SD and SE

NT	**NSIM**	**MC Call**	**Exact Call**	**SD**	**SE**
100	100,000	8.00790	7.96557	10.4359	0.0330012
250	250,000	7.98104	7.96557	10.4132	0.0208264
500	500,000	7.94180	7.96557	13.1421	0.0185857
500	1,000,000	7.96142	7.96557	13.1421	0.0131421
500	10,000,000	7.96866	7.96557	13.1481	0.0041578

Table 8: Batch 2: Monte Carlo vs Exact Option Call Prices with SD and SE

NT	**NSIM**	**MC Put**	**Exact Put**	**SD**	**SE**
100	100,000	8.00790	7.96557	10.4359	0.0330012
250	250,000	7.98104	7.96557	10.4132	0.0208264
500	500,000	7.97869	7.96557	10.4208	0.0147373
500	1,000,000	7.95663	7.96557	10.4052	0.0104052
500	10,000,000	7.96539	7.96557	10.4071	0.00329101

Table 9: Batch 2: Monte Carlo vs Exact Option Put Prices with SD and SE

Takeaways: Comparing the results of batches 1 and 2, we can see that the SD increases proportionally with the time-to-maturity (T = 0.25 for batch 1 and T = 1.0 for batch 2). Furthermore, as expected, the SE decreases as NSIM increases, which is also reflected in the smaller differences between the Monte Carlo prices and the exact prices.

4 Introduction to Finite Difference Methods

Test the Finite Difference Method using the data from batches 1 to 4. Compare the answers with the answers from the previous exercises. Recall batches 1–4 parameters:
Batch 1: T = 0.25, K = 65, sig = 0.30, r = 0.08, S = 60 (Call = 2.13337, Put = 5.84628).
Batch 2: T = 1.0, K = 100, sig = 0.2, r = 0.0, S = 100 (Call = 7.96557, Put = 7.96557).
Batch 3: T = 1.0, K = 10, sig = 0.50, r = 0.12, S = 5 (Call = 0.204058, Put = 4.07326).
Batch 4: T = 30.0, K = 100.0, sig = 0.30, r = 0.08, S = 100.0 (Call = 92.17570, Put = 1.24750).

Table 10: Batch 1: FDM vs Exact Put Prices

N	J	FDM Put	Exact Put	Difference
10000-1	$5 \times K$	5.84207	5.84628	0.00421
100000-1	$10 \times K$	5.84523	5.84628	0.00105
1000000-1	$10 \times K$	5.84522	5.84628	0.00106

Table 11: Batch 2: FDM vs Exact Put Prices

N	J	FDM Put	Exact Put	Difference
10000-1	$5 \times K$	7.96321	7.96557	0.00236
100000-1	$10 \times K$	7.96496	7.96557	6.1×10^{-4}
1000000-1	$10 \times K$	7.96495	7.96557	6.2×10^{-4}

Table 12: Batch 3: FDM vs Exact Put Prices

N	J	FDM Put	Exact Put	Difference
10000-1	$5 \times K$	4.07128	4.07326	0.00198
100000-1	$10 \times K$	4.07275	4.07326	5.1×10^{-4}
1000000-1	$10 \times K$	4.07275	4.07326	5.1×10^{-4}

Table 13: Batch 4: FDM vs Exact Put Prices

N	J	FDM Put	Exact Put	Difference
10000-1	$5 \times K$	nan	1.24750	n/a
100000-1	$5 \times K$	nan	1.24750	n/a
1000000-1	$5 \times K$	1.19586	1.24750	0.05164
10000000-1	$5 \times K$	1.19586	1.24750	0.05164

Takeaways: After experimenting with different N and J values I noticed that as N gets larger, the FDM prices get closer to the exact values in batches 1–3. In batch 4, the method didn't work properly at first and gave strange results, but this was because the values of N and J weren't set quite right. Once they are chosen more carefully, the results improve, although very long maturities still make things harder for the method.