

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

Aidan Richer

August 2025

This 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 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 Greeks across structured datasets. The program is organized into modular sections corresponding to the Group A&B questions, with each section demonstrating class implementations, usage examples in `main()`, and terminal output validation (solutions). At the beginning of the document is the UML diagram, which outlines the relationships and behaviors between the classes, followed by the solutions and implementation rationale for certain functionality in the program.

1 UML Diagram

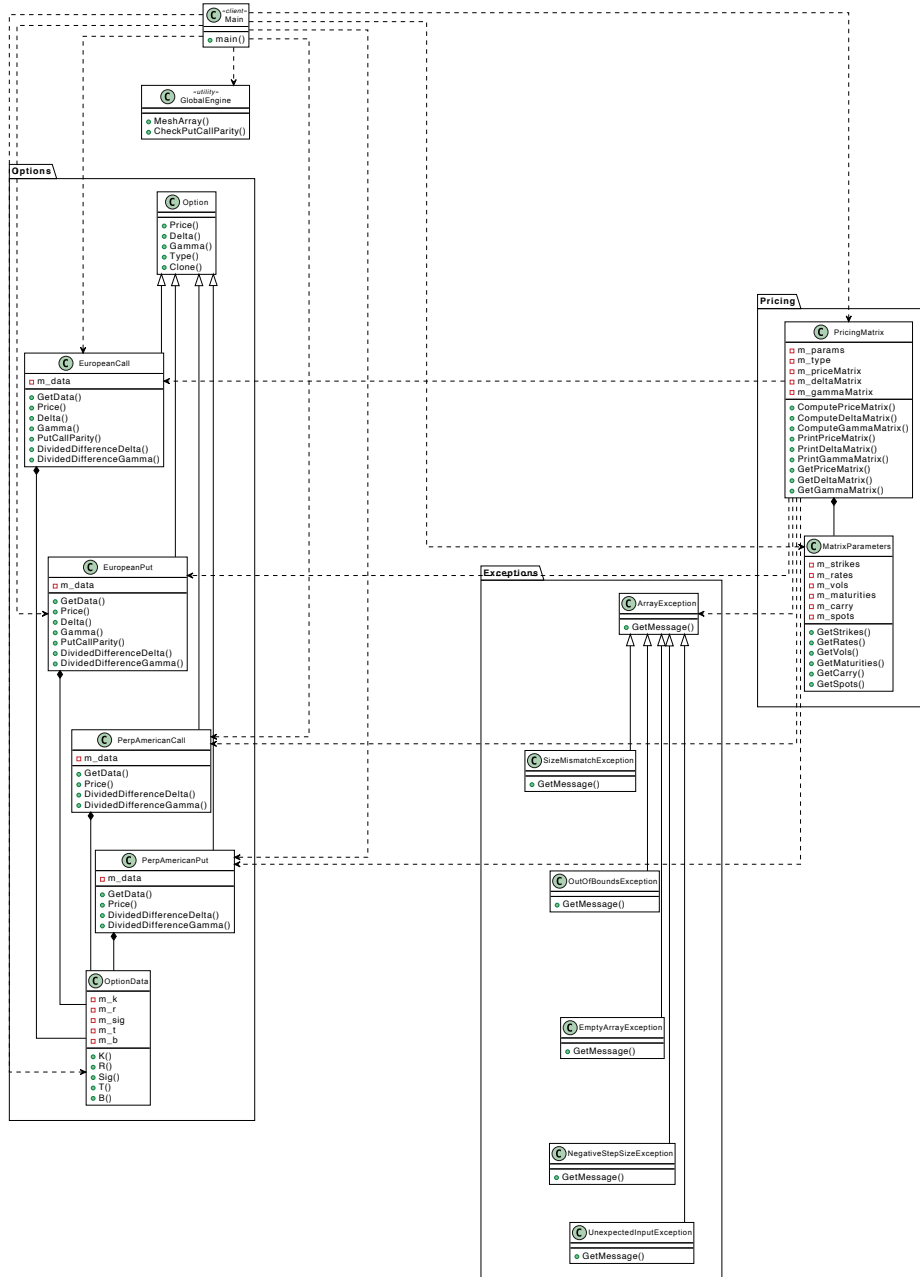


Figure 1: UML Diagram for Option Pricing Framework.

2 Questions

Group A, Section 1: European Options

Question (a)

Implement 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

```
1 double EuropeanCall::Price(const double U) const
2 { // return the price of the European call option
3     if (U <= 0.0)
4     {
5         throw std::invalid_argument("Underlying spot price U must
6             be positive.");
7     }
8
9     normal_distribution<> myNormal;
10    // boost normal distribution
11    double d1 = (log(U / m_data.K()) + (m_data.B() + 0.5 * m_data
12        .Sig() * m_data.Sig()) * m_data.T()) / (m_data.Sig() *
13        sqrt(m_data.T()));
14    double d2 = d1 - m_data.Sig() * sqrt(m_data.T());
15
16    // return call price
17    return U * exp((m_data.B() - m_data.R()) * m_data.T()) * cdf(
18        myNormal, d1) - m_data.K() * exp(-m_data.R() * m_data.T())
19        * cdf(myNormal, d2);
20 }
21
22 double EuropeanPut::Price(const double U) const
23 { // return the price of the European put option
24     if (U <= 0.0)
25     {
26         throw std::invalid_argument("Underlying spot price U must
27             be positive.");
28     }
29 }
```

```

24     normal_distribution<> myNormal;
25     // boost normal distribution
26     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()));
27     double d2 = d1 - m_data.Sig() * sqrt(m_data.T());
28
29     // return put price
30     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);
31 }

```

Usage in main()

```

1 vector<OptionData> batches =
2 {
3     OptionData(65.0, 0.08, 0.30, 0.25, 0.08),
4     OptionData(100.0, 0.0, 0.2, 1.0, 0.0),
5     OptionData(10.0, 0.12, 0.50, 1.0, 0.12),
6     OptionData(100.0, 0.08, 0.30, 30.0, 0.08),
7 };
8
9 vector<double> a_spots = { 60.0, 100.0, 5.0, 100.0 };
10
11 for (size_t i = 0; i < batches.size(); ++i)
12 {
13     EuropeanCall call(batches[i]);
14     EuropeanPut put(batches[i]);
15
16     cout << "Batch_" << (i + 1) << ":\n" << endl;
17     cout << "Call_=" << call.Price(a_spots[i]) << endl;
18     cout << "Put_=" << put.Price(a_spots[i]) << endl;
19 }

```

Terminal Output

```

1 Batch 1:
2 Call = 2.13337
3 Put = 5.84628
4 Batch 2:
5 Call = 7.96557
6 Put = 7.96557
7 Batch 3:
8 Call = 0.204058
9 Put = 4.07326
10 Batch 4:
11 Call = 92.1757
12 Put = 1.2475

```

Question (b)

Check 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

```
1 void CheckPutCallParity(const EuropeanCall& call, const
   EuropeanPut& put, double U)
2 { // verify if parity holds between a put and call option
3   const OptionData& c = call.GetData();
4   const OptionData& p = put.GetData();
5
6   if (c.K() == p.K() && c.R() == p.R() && c.Sig() == p.Sig() && c.T
       () == p.T() && c.B() == p.B())
7   { // parameters are equal, can check for parity
8     double callParity = put.PutCallParity(U);
9     // implied call price
10    double putParity = call.PutCallParity(U);
11    // implied put price
12    double callMkt = call.Price(U);
13    // market call price
14    double putMkt = put.Price(U);
15    // market put price
16
17    if (round(1000 * callParity) == round(1000 * call.Price(U)) &&
        round(1000 * putParity) == round(1000 * put.Price(U)))
18    { // parity holds
19      cout << "Put-Call Parity holds: No-arbitrage.\n"
20           << "Call price:" << callMkt << "\n"
21           << "Put price:" << putMkt << endl;
22    }
23    else
24    { // parity is violated
25      cout << "Put-Call Parity is violated: Arbitrage opportunity
           detected.\n"
26           << "Call price:" << callMkt << "\n"
27           << "Put price:" << putMkt << endl;
28    }
29  }
30  else
31  { // parameters are not equal, output error
```

```

32         cout << "The option parameters are not equivalent." <<
           endl;
33     }
34 }

```

Usage in main()

```

1  cout << "Applying Put-Call Parity Relationship:" << endl;
2  for (size_t i = 0; i < batches.size(); ++i)
3  {
4      EuropeanCall call(batches[i]);
5      EuropeanPut put(batches[i]);
6
7      cout << "Batch " << (i + 1) << ": " << endl;
8      // call global CheckPutCallParity() function
9      CheckPutCallParity(call, put, a_spots[i]);
10 }

```

Terminal Output

```

1 Applying Put-Call Parity Relationship:
2 Batch 1:
3 Put-Call Parity holds: No-arbitrage.
4 Call price: 2.13337
5 Put price: 5.84628
6 Batch 2:
7 Put-Call Parity holds: No-arbitrage.
8 Call price: 7.96557
9 Put price: 7.96557
10 Batch 3:
11 Put-Call Parity holds: No-arbitrage.
12 Call price: 0.204058
13 Put price: 4.07326
14 Batch 4:
15 Put-Call Parity holds: No-arbitrage.
16 Call price: 92.1757
17 Put price: 1.2475

```

Question (c)

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

```
1 template <typename T>
2 vector<T> MeshArray(const T start, const T step_size_h, const
   size_t vector_size)
3 {
4     vector<T> mesh;
5     // create the mesh array
6     for (size_t i = 0; i < vector_size; ++i)
7     {
8         mesh.push_back(start + i * step_size_h);
9     }
10
11     return mesh;
12 }
```

Usage in main()

```
1 vector<double> underlying_prices = MeshArray<double>(60.0, 10.0,
   5.0);
2 OptionData batch1 = batches[0];
3 // using batch 1 data
4 EuropeanCall call(batch1);
5 EuropeanPut put(batch1);
6 vector<double> call_prices;
7 // vector to store call prices
8 vector<double> put_prices;
9 // vector to store put prices
10
11 for (vector<double>::const_iterator itr = underlying_prices.begin
   (); itr != underlying_prices.end(); ++itr)
12 {
13     call_prices.push_back(call.Price(*itr));
14     put_prices.push_back(put.Price(*itr));
15 }
16
17 cout << "Spot\tCall\tPut" << endl;
18 for (size_t i = 0; i < underlying_prices.size(); ++i)
19 {
20     cout << underlying_prices[i] << "\t" << call_prices[i] << "\t"
   " << put_prices[i] << endl;
21 }
```

Terminal Output

```
1 Testing option prices over a mesh of underlying prices for Batch
   1:
2 Spot      Call      Put
3 60        2.13337  5.84628
```

4	70	7.90027	1.61319
5	80	16.5879	0.300857
6	90	26.3282	0.0411543
7	100	36.2916	0.00448099

Question (d)

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

```

1 void PricingMatrix::ComputePriceMatrix()
2 {
3     // access private member m_params from MatrixParameters
4     const std::vector<std::vector<double>>& strikes = m_params.
        GetStrikes();
5     const std::vector<std::vector<double>>& rates = m_params.
        GetRates();
6     const std::vector<std::vector<double>>& vols = m_params.
        GetVols();
7     const std::vector<std::vector<double>>& maturities = m_params.
        GetMaturities();
8     const std::vector<std::vector<double>>& carry = m_params.
        GetCarry();
9     const std::vector<std::vector<double>>& spots = m_params.
        GetSpots();
10
11     // verify outer matrix dimensions are equal
12     if (strikes.size() != rates.size() || strikes.size() != vols.
        size() || strikes.size() != maturities.size() || strikes.
        size() != carry.size() || strikes.size() != spots.size())
13     {
14         throw SizeMismatchException();

```



```

15     }
16
17     // verify inner matrix dimensions per row are equal
18     for (size_t i = 0; i < strikes.size(); ++i)
19     {
20         if (rates[i].size() != strikes[i].size() || vols[i].size()
21             != strikes[i].size() || maturities[i].size() !=
22             strikes[i].size() || carry[i].size() != strikes[i].
23             size() || spots[i].size() != strikes[i].size())
24         {
25             throw SizeMismatchException();
26         }
27     }
28
29     // resize price matrix
30     m_priceMatrix.clear();
31     m_priceMatrix.resize(strikes.size());
32     // rows
33     for (size_t i = 0; i < strikes.size(); ++i)
34     {
35         m_priceMatrix[i].resize(strikes[i].size());
36         // columns
37         for (size_t j = 0; j < strikes[i].size(); ++j)
38         {
39             OptionData optionData(strikes[i][j], rates[i][j],
40                                     vols[i][j], maturities[i][j], carry[i][j]);
41             double spot = spots[i][j];
42
43             if (m_type == "EuropeanCall")
44             {
45                 EuropeanCall opt(optionData);
46                 m_priceMatrix[i][j] = opt.Price(spot);
47             }
48             else if (m_type == "EuropeanPut")
49             {
50                 EuropeanPut opt(optionData);
51                 m_priceMatrix[i][j] = opt.Price(spot);
52             }
53             else if (m_type == "PerpAmericanCall")
54             {
55                 PerpAmericanCall opt(optionData);
56                 m_priceMatrix[i][j] = opt.Price(spot);
57             }
58             else if (m_type == "PerpAmericanPut")
59             {
60                 PerpAmericanPut opt(optionData);
61                 m_priceMatrix[i][j] = opt.Price(spot);
62             }
63             else

```

```

62         {
63             throw UnexpectedInputException();
64         }
65     }
66 }
67 }

```

Usage in main()

```

1  try
2  {
3      cout << "Computing European Call and Put option Price
         Matrices:" << endl;
4
5      // vectors must be of same size for mesh array function
6      size_t matrix_col_length = 10;
7
8      // create vectors of parameters
9      vector<double> spots = MeshArray(50.0, 10.0,
        matrix_col_length);
10     vector<double> strikes = MeshArray(60.0, 10.0,
        matrix_col_length);
11     vector<double> maturities = MeshArray(0.1, 0.1,
        matrix_col_length);
12     vector<double> vols = MeshArray(0.1, 0.05, matrix_col_length)
        ;
13     vector<double> rates = MeshArray(0.1, 0.02, matrix_col_length
        );
14     vector<double> carry = MeshArray(0.1, 0.02, matrix_col_length
        );
15
16     // use MatrixParameters class vector parameter constructor
17     MatrixParameters vector_params(strikes, rates, vols,
        maturities, carry, spots);
18     // use PricingMatrix class parameter constructor
19     PricingMatrix call_matrix(vector_params, "EuropeanCall");
20     PricingMatrix put_matrix(vector_params, "EuropeanPut");
21
22     // compute price matrices
23     call_matrix.ComputePriceMatrix();
24     put_matrix.ComputePriceMatrix();
25
26     // print price matrices
27     call_matrix.PrintPriceMatrix();
28     put_matrix.PrintPriceMatrix();
29
30 } catch (const ArrayException& e) {
31     cout << "Error:" << e.GetMessage() << endl;
32 } catch (...) {
33     cout << "Unexpected error." << endl;

```

34 }

Terminal Output

```
1 Computing European Call and Put option Price Matrices:
2 EuropeanCall Price Matrix:
3 0.00000    0.04273    0.90509    3.26619    6.98458    11.91441
   17.98779    25.16721    33.41924    42.70351
4 EuropeanPut Price Matrix:
5 9.40299    8.38273    7.61468    7.68664    8.37770    9.47565
   10.86043    12.45710    14.20989    16.07107
```

Group A, Section 2: European Options

Question (a)

Implement delta and gamma formulae and test using the provided data set:

$K = 100$, $S = 105$, $T = 0.5$, $r = 0.1$, $b = 0$, $\text{sig} = 0.36$

(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

```
1 double EuropeanCall::Delta(const double U) const
2 { // return the delta of the European call option
3     if (U <= 0.0)
4     {
5         throw std::invalid_argument("Underlying spot price U must
6             be positive.");
7     }
8     normal_distribution<> myNormal;
9     // boost normal distribution
10    double d1 = (log(U / m_data.K()) + (m_data.B() + 0.5 * m_data
11        .Sig() * m_data.Sig()) * m_data.T()) / (m_data.Sig() *
12        sqrt(m_data.T()));
13    // return delta
14    return exp((m_data.B() - m_data.R()) * m_data.T()) * cdf(
    myNormal, d1);
}
```

Usage in main()

```
1 OptionData partials_call(100.0, 0.1, 0.36, 0.5, 0.0);
2 OptionData partials_put(100.0, 0.1, 0.36, 0.5, 0.0);
3 double partials_strike = 105.0;
4 EuropeanCall call_greeks(partial_call);
5 EuropeanPut put_greeks(partial_put);
6
7 cout << "Computing Delta and Gamma for European options:" <<
    endl;
8 cout << "European Call Delta:" << call_greeks.Delta(
    partials_strike) << endl;
9 cout << "European Call Gamma:" << call_greeks.Gamma(
    partials_strike) << endl;
10 cout << "European Put Delta:" << put_greeks.Delta(
    partials_strike) << endl;
11 cout << "European Put Gamma:" << put_greeks.Gamma(
    partials_strike) << endl;
```

Terminal Output

```
1 Computing Delta and Gamma for European options:
2 European Call Delta: 0.59463
3 European Call Gamma: 0.01349
4 European Put Delta: -0.35660
5 European Put Gamma: 0.01349
```

Question (b)

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()

```
1 vector<double> spot_mesh = MeshArray(105.0, 5.0, 5.0);
2 // create mesh array of spot prices
3 vector<double> call_deltas;
4 // vector to store deltas
5 vector<double> call_gammas;
6 // vector to store gammas
7
8 // compute and store deltas and gammas in their respective vector
```

```

9  for (vector<double>::const_iterator itr = spot_mesh.begin(); itr
    != spot_mesh.end(); ++itr)
10 {
11     call_deltas.push_back(call_greeks.Delta(*itr));
12     call_gammas.push_back(call_greeks.Gamma(*itr));
13 }
14
15 cout << "Computing European call option deltas and gammas for a
    monotonically increasing mesh of spot prices:" << endl;
16 cout << "Spot\t\tDelta\tGamma" << endl;
17 for (size_t i = 0; i < spot_mesh.size(); ++i)
18 {
19     cout << spot_mesh[i] << "\t" << call_deltas[i] << "\t" <<
        call_gammas[i] << endl;
20 }

```

Terminal Output

```

1 Computing European call option deltas and gammas for a
    monotonically increasing mesh of spot prices:
2 Spot          Delta    Gamma
3 105.00000      0.59463  0.01349
4 110.00000      0.65831  0.01195
5 115.00000      0.71397  0.01031
6 120.00000      0.76149  0.00870
7 125.00000      0.80120  0.00721

```

Question (c)

Incorporate this into the matrix pricer 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()

```

1 // create matrices of option parameters (5 by 5 matrix)
2 vector<vector<double>> strikes =
3 {{60.0, 65.0, 70.0, 75.0, 80.0},{85.0, 90.0, 95.0, 100.0, 105.0},
4 {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}};
5 vector<vector<double>> rates =

```

```

6  {{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}}};
7  vector<vector<double>> vols =
8  {{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}}};
9  vector<vector<double>> maturities =
10 {{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}}};
11 vector<vector<double>> carry =
12 {{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}}};
13 vector<vector<double>> spots =
14 {{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}}};
15
16 // use MatrixParameters matrix constructor
17 MatrixParameters matrixParams(strikes, rates, vols, maturities,
   carry, spots);
18 PricingMatrix callMatrix(matrixParams, "EuropeanCall");
19 PricingMatrix putMatrix(matrixParams, "EuropeanPut");
20
21 try
22 {
23     cout << "Computing European Call Delta Matrix:" << endl;
24     // call ComputeDeltaMatrix() function
25     callMatrix.ComputeDeltaMatrix();
26     callMatrix.PrintDeltaMatrix();
27
28     cout << "\nComputing European Call Gamma Matrix:" << endl;
29     // call ComputeGammaMatrix() function
30     callMatrix.ComputeGammaMatrix();
31     callMatrix.PrintGammaMatrix();
32
33     cout << "\nComputing European Put Delta Matrix:" << endl;
34     putMatrix.ComputeDeltaMatrix();
35     putMatrix.PrintDeltaMatrix();
36
37     cout << "\nComputing European Put Gamma Matrix:" << endl;
38     putMatrix.ComputeGammaMatrix();
39     putMatrix.PrintGammaMatrix();
40
41 } catch (const ArrayException& e) {
42     cout << "Error:" << e.GetMessage() << endl;
43 } catch (...) {
44     cout << "Unexpected error." << endl;
45 }

```

Terminal Output

```
1 Computing European Call Delta Matrix:
2 EuropeanCall Delta Matrix:
3     0.00000    0.00760    0.09203    0.20913    0.30248
4     0.11794    0.26039    0.33352    0.39650    0.42836
5     0.00151    0.09785    0.23676    0.33214    0.39422
6     0.25098    0.36020    0.40720    0.44733    0.46675
7     0.02311    0.19214    0.31682    0.38973    0.43560
8
9 Computing European Call Gamma Matrix:
10 EuropeanCall Gamma Matrix:
11     0.00000    0.00568    0.02508    0.02784    0.02324
12     0.02815    0.02648    0.02111    0.01537    0.01206
13     0.00155    0.02451    0.02554    0.01982    0.01490
14     0.02710    0.01867    0.01408    0.01006    0.00794
15     0.01153    0.02626    0.02025    0.01458    0.01074
16
17 Computing European Put Delta Matrix:
18 EuropeanPut Delta Matrix:
19    -1.00000   -0.99040   -0.90199   -0.77894   -0.67771
20    -0.85834   -0.70522   -0.61962   -0.54245   -0.49475
21    -0.99849   -0.90015   -0.75725   -0.65593   -0.58597
22    -0.72531   -0.60540   -0.54594   -0.49162   -0.45637
23    -0.97689   -0.80587   -0.67720   -0.59834   -0.54459
24
25 Computing European Put Gamma Matrix:
26 EuropeanPut Gamma Matrix:
27     0.00000    0.00568    0.02508    0.02784    0.02324
28     0.02815    0.02648    0.02111    0.01537    0.01206
29     0.00155    0.02451    0.02554    0.01982    0.01490
30     0.02710    0.01867    0.01408    0.01006    0.00794
31     0.01153    0.02626    0.02025    0.01458    0.01074
```

Question (d)

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

```
1 double EuropeanCall::DividedDifferenceDelta(const double U, const
2 double h) const
3 { // delta approximation using the divided difference method
4   if (abs(Price(U + h) - Price(U - h)) <= std::pow(2, -53))
5   {
6       std::cout << "H is too small for accurate computation."
7       << std::endl;
8       return 0;
9   }
10
11   // return approximated delta
12   return (Price(U + h) - Price(U - h)) / (2.0 * h);
13 }
```

Usage in main()

```
1 // use the same option data as in part a) for comparison
2 OptionData divided_difference_call_params(100.0, 0.1, 0.36, 0.5,
3 0.0);
4 OptionData divided_difference_put_params(100.0, 0.1, 0.36, 0.5,
5 0.0);
6
7 // set spot price
8 double divided_difference_strike = 105.0;
9
10 // pass params to EuropeanCall and EuropeanPut objects
11 EuropeanCall divided_difference_call(
12 divided_difference_call_params);
13 EuropeanPut divided_difference_put(divided_difference_put_params)
14 ;
15
16 // to compare part a) results
17 cout << "Computing approximate Delta and Gamma for European
18 options using Divided Differences: " << endl;
19
20 cout << "European Call Delta (h=1.0): " <<
21 divided_difference_call.DividedDifferenceDelta(
22 divided_difference_strike, 1) << endl;
23
24 cout << "European Call Delta (h=1e-4): " <<
25 divided_difference_call.DividedDifferenceDelta(
26 divided_difference_strike, 1e-4) << endl;
27
28 cout << "European Call Gamma (h=1.0): " <<
29 divided_difference_call.DividedDifferenceGamma(
30 divided_difference_strike, 1) << endl;
```



```

21 cout << "European_Call_Gamma(h=1e-4):" <<
    divided_difference_call.DividedDifferenceGamma(
        divided_difference_strike, 1e-4) << endl;
22
23 cout << "European_Put_Delta(h=1.0):" <<
    divided_difference_put.DividedDifferenceDelta(
        divided_difference_strike, 1) << endl;
24
25 cout << "European_Put_Delta(h=1e-4):" <<
    divided_difference_put.DividedDifferenceDelta(
        divided_difference_strike, 1e-4) << endl;
26
27 cout << "European_Put_Gamma(h=1.0):" <<
    divided_difference_put.DividedDifferenceGamma(
        divided_difference_strike, 1) << endl;
28
29 cout << "European_Put_Gamma(h=1e-4):" <<
    divided_difference_put.DividedDifferenceGamma(
        divided_difference_strike, 1e-4) << endl;
30
31 // to compare part b) results
32 cout << "\nComputing approximate European Call Delta and Gamma
    for a monotonically increasing mesh of spot prices:" << endl;
33
34 vector<double> divided_difference_mesh = MeshArray(105.0, 5.0,
    5.0);
35 // same mesh array of spot prices as part b)
36 vector<double> mesh_deltas;
37 // vector to store computed deltas
38 vector<double> mesh_gammas;
39 // vector to store computed gammas
40
41 // calculate and store in respective vector
42 for (vector<double>::const_iterator itr = divided_difference_mesh
    .begin(); itr != divided_difference_mesh.end(); ++itr)
43 {
44     mesh_deltas.push_back(divided_difference_call.
        DividedDifferenceDelta(*itr, 1e-4));
45
46     mesh_gammas.push_back(divided_difference_call.
        DividedDifferenceGamma(*itr, 1e-4));
47 }
48
49 // output vectors
50 cout << "Spot\t\tDelta\tGamma" << endl;
51 for (size_t i = 0; i < divided_difference_mesh.size(); ++i)
52 {
53     cout << divided_difference_mesh[i] << "\t" << mesh_deltas[i]
        << "\t" << mesh_gammas[i] << endl;
54 }

```

Terminal Output

```
1 Computing approximate Delta and Gamma for European options using
  Divided Differences:
2 European Call Delta (h = 1.0): 0.59458
3 European Call Delta (h = 1e-4): 0.59463
4 European Call Gamma (h = 1.0): 0.01349
5 European Call Gamma (h = 1e-4): 0.01349
6 European Put Delta (h = 1.0): -0.35665
7 European Put Delta (h = 1e-4): -0.35660
8 European Put Gamma (h = 1.0): 0.01349
9 European Put Gamma (h = 1e-4): 0.01349
10
11 Computing approximate European Call Delta and Gamma for a
  monotonically increasing mesh of spot prices:
12 Spot          Delta    Gamma
13 105.00000      0.59463  0.01349
14 110.00000      0.65831  0.01195
15 115.00000      0.71397  0.01031
16 120.00000      0.76149  0.00871
17 125.00000      0.80120  0.00721
```

Compare with part a) and b) output

```
1 Original part a) output:
2 European Call Delta: 0.59463
3 European Call Gamma: 0.01349
4 European Put Delta: -0.35660
5 European Put Gamma: 0.01349
6
7 Original part b) output:
8 Spot          Delta    Gamma
9 105.00000      0.59463  0.01349
10 110.00000      0.65831  0.01195
11 115.00000      0.71397  0.01031
12 120.00000      0.76149  0.00870
13 125.00000      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 Group B, Section 1: Perpetual American Options

Question (a)

Implement 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

```
1 double PerpAmericanCall::Price(const double U) const
2 { // return the price of the perpetual american call option
3     if (U <= 0.0)
4     {
5         throw std::invalid_argument("Underlying spot price U must
6             be positive.");
7     }
8
9     double sigma_squared = m_data.Sig() * m_data.Sig();
10
11     double y1 = 0.5 - (m_data.B() / sigma_squared) + std::sqrt(
12         std::pow((m_data.B() / sigma_squared - 0.5), 2.0) + (2.0 *
13             m_data.R() / sigma_squared));
14
15     double call_rhs = ((y1 - 1.0) / y1) * (U / m_data.K());
16
17     // return call price
18     return (m_data.K() / (y1 - 1.0)) * std::pow(call_rhs, y1);
19 }
20
21 double PerpAmericanPut::Price(const double U) const
22 { // return the price of the perpetual american put option
23     if (U <= 0.0)
24     {
25         throw std::invalid_argument("Underlying spot price U must
26             be positive.");
27     }
28
29     double sigma_squared = m_data.Sig() * m_data.Sig();
30
31     double y2 = 0.5 - (m_data.B() / sigma_squared) - std::sqrt(
32         std::pow((m_data.B() / sigma_squared - 0.5), 2.0) + (2.0 *
33             m_data.R() / sigma_squared));
```

```

29     double put_rhs = ((y2 - 1.0) / y2) * (U / m_data.K());
30
31     // return put price
32     return (m_data.K() / (1.0 - y2)) * std::pow(put_rhs, y2);
33 }

```

Question (b)

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

Usage in main()

```

1  cout << "Computing Perpetual American Call and Put Prices:" <<
    endl;
2
3  OptionData perpetual_options_data(100.0, 0.1, 0.1, 0.0, 0.02);
4  // set T = 0
5  PerpAmericanCall perp_call(perpetual_options_data);
6  PerpAmericanPut perp_put(perpetual_options_data);
7
8  cout << "Perpetual American Call Price:" << perp_call.Price
    (110.0) << endl;
9  // spot price = 110.0
10 // Call price: 18.50350
11 cout << "Perpetual American Put Price:" << perp_put.Price(110.0)
    << endl;
12 // spot price = 110.0
13 // Put price: 3.03106

```

Terminal Output

```

1 Computing Perpetual American Call and Put Prices:
2 Perpetual American Call Price: 18.50350
3 Perpetual American Put Price: 3.03106

```

Question (c)

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()

```
1 cout << "Computing Perpetual American Call and Put Prices for a  
monotonically increasing mesh of spot prices:" << endl;  
2  
3 vector<double> perp_options_mesh = MeshArray(110.0, 5.0, 5.0);  
4 // create the mesh array  
5 vector<double> perp_call_prices;  
6 // vector to store call prices  
7 vector<double> perp_put_prices;  
8 // vector to store put prices  
9  
10 // calculate and store in respective vector, we will be using  
same option data parameters as used in Question a) and b)  
11 for (vector<double>::const_iterator itr = perp_options_mesh.begin  
(); itr != perp_options_mesh.end(); ++itr)  
12 {  
13     perp_call_prices.push_back(perp_call.Price(*itr));  
14     perp_put_prices.push_back(perp_put.Price(*itr));  
15 }  
16  
17 // output calculated vectors  
18 cout << "Spot\t\tCall\t\tPut" << endl;  
19 for (size_t i = 0; i < perp_options_mesh.size(); ++i)  
20 {  
21     cout << perp_options_mesh[i] << "\t" << perp_call_prices[i]  
        << "\t" << perp_put_prices[i] << endl;  
22 }
```

Terminal Output

```
1 Computing Perpetual American Call and Put Prices for a  
monotonically increasing mesh of spot prices:  
2 Spot          Call          Put  
3 110.00000     18.50350     3.03106  
4 115.00000     21.34806     2.29919  
5 120.00000     24.48045     1.76467  
6 125.00000     27.91596     1.36913  
7 130.00000     31.67005     1.07287
```

Question (d)

Incorporate this into your 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

```
1 void PricingMatrix::ComputeDeltaMatrix(const double h)
2 { // compute a delta matrix for a given matrix of option
   parameters
3     const std::vector<std::vector<double>>& strikes = m_params.
       GetStrikes();
4     const std::vector<std::vector<double>>& rates = m_params.
       GetRates();
5     const std::vector<std::vector<double>>& vols = m_params.
       GetVols();
6     const std::vector<std::vector<double>>& maturities = m_params.
       GetMaturities();
7     const std::vector<std::vector<double>>& carry = m_params.
       GetCarry();
8     const std::vector<std::vector<double>>& spots = m_params.
       GetSpots();
9
10    // verify outer matrix dimensions are equal
11    if (strikes.size() != rates.size() || strikes.size() != vols.
        size() || strikes.size() != maturities.size() || strikes.
        size() != carry.size() || strikes.size() != spots.size())
12    {
13        throw SizeMismatchException();
14    }
15
16    // verify inner matrix dimensions per row are equal
17    for (size_t i = 0; i < strikes.size(); ++i)
18    {
19        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())
20        {
```

```

21         throw SizeMismatchException();
22     }
23 }
24
25 // resize delta matrix
26 m_deltaMatrix.clear();
27 m_deltaMatrix.resize(strikes.size());
28
29 for (size_t i = 0; i < strikes.size(); ++i)
30 {
31     m_deltaMatrix[i].resize(strikes[i].size());
32
33     for (size_t j = 0; j < strikes[i].size(); ++j)
34     {
35         OptionData optionData(strikes[i][j], rates[i][j],
36                                vols[i][j], maturities[i][j], carry[i][j]);
37         double spot = spots[i][j];
38
39         // find string type and compute
40
41         if (m_type == "EuropeanCall")
42         {
43             EuropeanCall opt(optionData);
44             m_deltaMatrix[i][j] = opt.Delta(spot);
45         }
46         else if (m_type == "EuropeanPut")
47         {
48             EuropeanPut opt(optionData);
49             m_deltaMatrix[i][j] = opt.Delta(spot);
50         }
51         else if (m_type == "PerpAmericanCall")
52         {
53             PerpAmericanCall opt(optionData);
54             m_deltaMatrix[i][j] = opt.DividedDifferenceDelta(
55                 spot, h);
56         }
57         else if (m_type == "PerpAmericanPut")
58         {
59             PerpAmericanPut opt(optionData);
60             m_deltaMatrix[i][j] = opt.DividedDifferenceDelta(
61                 spot, h);
62         }
63         else
64         {
65             throw UnexpectedInputException();
66         }
67     }
68 }
69 }
70 }

```

Usage in main()

```
1 // create matrices of option parameters (5 x 5 matrices)
2 // b < r for the computation to work *Important
3 vector<vector<double>> perp_strikes =
4 {{50.0, 55.0, 60.0, 65.0, 70.0},{75.0, 80.0, 85.0, 90.0,
5   95.0},{100.0, 105.0, 110.0, 115.0, 120.0},{125.0, 130.0,
6   135.0, 140.0, 145.0},{150.0, 155.0, 160.0, 165.0, 170.0}};
7 vector<vector<double>> perp_rates =
8 {{0.10, 0.10, 0.10, 0.10, 0.10},{0.12, 0.12, 0.12, 0.12,
9   0.12},{0.14, 0.14, 0.14, 0.14, 0.14},{0.16, 0.16, 0.16, 0.16,
10  0.16},{0.18, 0.18, 0.18, 0.18, 0.18}};
11 vector<vector<double>> perp_vols =
12 {{0.10, 0.15, 0.20, 0.25, 0.30},{0.12, 0.17, 0.22, 0.27,
13   0.32},{0.14, 0.19, 0.24, 0.29, 0.34},{0.16, 0.21, 0.26, 0.31,
14   0.36},{0.18, 0.23, 0.28, 0.33, 0.38}};
15 // T = 0 for perpetual American options
16 vector<vector<double>> perp_maturities =
17 {{0.0, 0.0, 0.0, 0.0, 0.0},{0.0, 0.0, 0.0, 0.0, 0.0},{0.0, 0.0,
18   0.0, 0.0, 0.0},{0.0, 0.0, 0.0, 0.0, 0.0},{0.0, 0.0, 0.0, 0.0,
19   0.0}};
20 vector<vector<double>> perp_carry =
21 {{0.01, 0.02, 0.03, 0.04, 0.05},{0.01, 0.02, 0.03, 0.04,
22   0.05},{0.01, 0.02, 0.03, 0.04, 0.05},{0.01, 0.02, 0.03, 0.04,
23   0.05},{0.01, 0.02, 0.03, 0.04, 0.05}};
24 vector<vector<double>> perp_spots =
25 {{50.0, 55.0, 60.0, 65.0, 70.0},{75.0, 80.0, 85.0, 90.0,
26   95.0},{100.0, 105.0, 110.0, 115.0, 120.0},{125.0, 130.0,
27   135.0, 140.0, 145.0},{150.0, 155.0, 160.0, 165.0, 170.0}};
28
29 // use MatrixParameters matrix constructor
30 MatrixParameters perp_matrix_params(perp_strikes, perp_rates,
31   perp_vols, perp_maturities, perp_carry, perp_spots);
32
33 PricingMatrix perp_call_matrix(perp_matrix_params, "
34   PerpAmericanCall");
35 PricingMatrix perp_put_matrix(perp_matrix_params, "
36   PerpAmericanPut");
37
38 try
39 {
40     cout << "Computing Perpetual American Call Price Matrix:" <<
41       endl;
42     perp_call_matrix.ComputePriceMatrix();
43     perp_call_matrix.PrintPriceMatrix();
44
45     cout << "\nComputing Perpetual American Put Price Matrix:" <<
46       endl;
47     perp_put_matrix.ComputePriceMatrix();
48     perp_put_matrix.PrintPriceMatrix();
49 }
```



```

33     cout << "\nApproximating Perpetual American Call Deltas Using
        the Divided Difference Method (h=0.0001):" << endl;
34     perp_call_matrix.ComputeDeltaMatrix();
35     perp_call_matrix.PrintDeltaMatrix();
36
37     cout << "\nApproximating Perpetual American Call Gammas Using
        the Divided Difference Method (h=0.0001):" << endl;
38     perp_call_matrix.ComputeGammaMatrix();
39     perp_call_matrix.PrintGammaMatrix();
40
41     cout << "\nApproximating Perpetual American Put Deltas Using
        the Divided Difference Method (h=0.0001):" << endl;
42     perp_put_matrix.ComputeDeltaMatrix();
43     perp_put_matrix.PrintDeltaMatrix();
44
45     cout << "\nApproximating Perpetual American Put Gammas Using
        the Divided Difference Method (h=0.0001):" << endl;
46     perp_put_matrix.ComputeGammaMatrix();
47     perp_put_matrix.PrintGammaMatrix();
48
49 } catch (const ArrayException& e) {
50     cout << "Error:" << e.GetMessage() << endl;
51 } catch (...) {
52     cout << "Unexpected error." << endl;
53 }

```

Terminal Output

```

1 Computing Perpetual American Call Price Matrix:
2 PerpAmericanCall Price Matrix:
3     5.27344    9.64471   15.00000   21.41474   28.98287
4     8.16222   13.51932   19.74407   26.86789   34.93090
5    11.29627   17.45767   24.39484   32.11503   40.63138
6    14.67627   21.52479   29.07451   37.31877   46.25577
7    18.29429   25.74780   33.84046   42.55740   51.88807
8
9 Computing Perpetual American Put Price Matrix:
10 PerpAmericanPut Price Matrix:
11     3.34898    5.20830    7.39202    9.89313   12.70628
12     5.78085    8.24727   11.02933   14.11311   17.48878
13     8.59507   11.61025   14.93059   18.53791   22.41934
14    11.73922   15.25816   19.07065   23.15616   27.49980
15    15.17657   19.16301   23.43087   27.95844   32.72960
16
17 Approximating Perpetual American Call Deltas Using the Divided
    Difference Method (h = 1e-4):
18 PerpAmericanCall Delta Matrix:
19     0.42187    0.45905    0.50000    0.54503    0.59464
20     0.42364    0.45562    0.49016    0.52733    0.56724
21     0.42581    0.45415    0.48436    0.51641    0.55030

```

22	0.42815	0.45378	0.48084	0.50926	0.53902
23	0.43055	0.45407	0.47872	0.50442	0.53114
24					
25	Approximating Perpetual American Call Gammas Using the Divided Difference Method ($h = 1e-4$):				
26	PerpAmericanCall Gamma Matrix:				
27	0.02531	0.01350	0.00833	0.00549	0.00371
28	0.01634	0.00966	0.00640	0.00449	0.00324
29	0.01179	0.00749	0.00521	0.00381	0.00287
30	0.00907	0.00608	0.00439	0.00331	0.00257
31	0.00726	0.00508	0.00378	0.00292	0.00231
32					
33	Approximating Perpetual American Put Deltas Using the Divided Difference Method ($h = 1e-4$):				
34	PerpAmericanPut Delta Matrix:				
35	-0.33490	-0.32155	-0.30800	-0.29441	-0.28086
36	-0.33001	-0.31754	-0.30491	-0.29226	-0.27968
37	-0.32574	-0.31398	-0.30210	-0.29023	-0.27843
38	-0.32192	-0.31076	-0.29951	-0.28829	-0.27714
39	-0.31845	-0.30780	-0.29709	-0.28642	-0.27583
40					
41	Approximating Perpetual American Put Gammas Using the Divided Difference Method ($h = 1e-4$):				
42	PerpAmericanPut Gamma Matrix:				
43	0.04019	0.02570	0.01797	0.01329	0.01022
44	0.02324	0.01620	0.01202	0.00930	0.00742
45	0.01560	0.01148	0.00886	0.00707	0.00578
46	0.01140	0.00872	0.00692	0.00565	0.00470
47	0.00881	0.00693	0.00562	0.00467	0.00395