PART B FINAL PROJECT

LM Advanced Mathematical Finance Module

Part B C++ Spring Term 2021

PROJECT	SPRING TERM, PROGRAMMING IN C++
Lecturer	Dr Simon Hartley s.hartley@bham.ac.uk
Of	School of Mathematics, College of Engineering and Physical Sciences, University of Birmingham
Due	Friday 4th June 2021, 23:59
Weighting	50% of Part B

INSTRUCTIONS FOR THE SUBMISSION

Submit your code on Canvas as zip:

Submit as a single zip file containing all code for the problem.

The source files should compile without errors or warnings on Visual Studio 2019 as installed on the clusters, and the resulting executables should run without problems. **Using debug and x64 build options**. You do not have to submit the executable. You should comment you code and describe what each function does.

If you use another compiler other than Visual Studio, you should take your final code and compile it on Visual Studio on the remote clusters.

Unfortunately, different compilers follow slightly different rules, and even a small difference may cause the code to fail on Visual Studio. The source code should also be well formatted. This means that opening and closing brackets should be aligned in a readable way, and we will discuss such styles briefly in the lectures.

PLAGIARISM

Obviously, you are not allowed to plagiarise. This means that you are not allowed to directly copy the code from any online sources or any of your fellow students. Direct group work is also not allowed. While you are allowed (and even encouraged) to discuss the contents of this module (including problem sheets) with your fellow students, every one of you needs to write your own code.

NUMERICAL OPTION PRICING PROGRAM

Write a program that can price European Options and American Options with different methods: Analytically, with a Binomial Tree, and with Monte Carlo.

When you finish this problem, you should have:

- Created a Series of Classes in their own namespace to form a Mathematical Library for Numerical Option Pricing using STL Library Algorithms, Functions and Containers where possible.
- Implement puts and calls for each case.
- Tested your classes with Example data.

Description

In the provided source code(see link) which we have been working on in the Week 10 Labs which uses the c++ class heirachy:(See Diagram)

Source

Monte Carlo European Option

In the labs we created a C++ interitance hierarchy and implemented an algorithm for the Monte Carlo European Option. Using the example code add a new function to the program for Eurpean Options to match (adding steps to a stockpath):

```
Inputs:
  steps{20}, paths{ 1000 }, expiry{ 1 }, spot{ 1.0 }, rate{ 0.05 }, volatility{ 0.3 },
  dividend{ 0.0 }, strike{ 1.05 };

  double sum = 0.;
  for (int n = 0; n < paths; n++){
    // now create a path
    double dt = expiry / steps;
    stock_Path[steps];
    stock_Path[0] = spot;
    for (int i = 1; i <= steps; i++){
      double phi = random_generator(rng);
      stock_Path[i] = stock_Path[i - 1] * exp((myInterestRate - 0.5 * volatility * volatility) *
    }
    // and calculate S
    double S = 0.;
    for (int i = 1; i <= K; i++){
      S = S + fabs(stock_Path[i] - stock_Path[i - 1]);
    }
    // add in the payoff to the sum
    sum = sum + std::max(S - strike, 0.);
  }

  return sum / paths * exp(-rate * expiry);
```

Part 1: Monte Carlo European Option Stratified

Add to the implemented soluion to have a Function to use a Simplified Monte Carlo **stratified** sampling method. Stratified sampling method is a way to reduce the number of samples needed to converge on a price, reducing calculation time. This method forces our sample to be taken from different percentile buckets. For example, if we were to take samples using five different percentile buckets, we would take a uniform random sample between 0.0 and 0.2 and between 0.2 and 0.4 and so on until 1. We then take these stratified uniform samples and convert them to a normally distributed samples using the cumulative distribution function. Sampling in this way will help our sampling appear normally distributed much faster. This method is simple to implement for European Options because only the end point is needed for calculations.

Part 2: European Options Analytical Functions Black-Scholes model

Implement the Analytical Solution to the EuropeanPut and EuropeanCall classes. Complete the Analytical Functions for the European Options put and call using the Black Scholes Method from Week 1Slides02.pdf

Hints

There are assumptions for deriving the Black-Scholes model:

1. It is possible to borrow and lend cash at a known constant risk-free interest rate r.
2. The price of the underlying follows a geometric Brownian motion with constant drift and volatility.
3. There are no transaction costs.
4. The stock pays no dividend.
5. All securities are perfectly divisible (i.e. it is possible to buy any fraction of a share).
6. There are no restrictions on short selling.
7. There are no riskless arbitrage opportunities.
8. Security trading is continuous.

The Black-Scholes Equation can be written as:

$$C_{(S,t)} = S_t\Phi_{d_1} - exp(-r(T-t))K\Phi_{d_2}$$

In the case of the options we are considering those conditions are:

- $C_{(S,\ T)}$ = max(S − K, 0) // Call
- $C_{(S,\ T)}$ = max(K − S, 0) // Put
- $C_{(0,\ t)}$ = 0 for all t
- $C_{(S,\ t)}$ → S as S → ∞.

where d_1 is given by:

$$d_2 = d_1 - \sigma\sqrt{(T-t)}$$

and d_2 is given by:

$$d_1 = \frac{log(\frac{St}{K}) + (r + \frac{\sigma^2}{2})(T-t))}{\sigma\sqrt{(T-t)}}$$

and

$\Phi_{(.)} \text{is } the\ C\text{DF of } the\ st\text{and}ard\ \text{normal } dist\text{ribution}$

You can use the cumulative distribution function provided. You need to create a function for the call option, so we need to think about what arguments need to be supplied and what are the local variables to the function. Supplied data is the asset price, current time both of which may vary, and the parameters for the function are the strike price, interest rate, volatility and maturity, and we want to return a real number as the answer. Inside the function,we will need to calculate the value of d1 and d2.

Now you need to test the function under different cases. The function will need to account for each of the following cases:

- S = 0
- sigma = 0
- t = T - check if we are at(or very close to) maturity
- t > T - Option Expired

Since all will result in undefined mathematical values given the way that d1 and d2 are calculated. However, returning a value of plus or minus infinity for d1 and d2 does not result in an undetermined value for the cumulative normal distribution. As the function returns definite values from infinite limits. Using knowledge of the function we need go through each case and decide what are the appropriate responses.

Case S = 0

Here we must make use of the boundary condition of the problem, which is that the option value is worthless if the asset value is zero. The only slight caveat here is that you should check whether S is smaller than a very small number relative to the strike price rather than comparing it with zero. If the comparison is true the function will stop and return the value for Zero without executing any more code.

Case Sigma = 0

This case is very similar to when t=T. Depending on the sign of the numerator in the calculation for d1 and d2 the function will either return zero or the asset minus the discounted strike price.

Case t = T

Finally if t and T are almost equal then we are at maturity and we return the payoff. On adding each of these parts to the code you should test and validate each part using a range of parameters.

Case t > T

There is no sensible value that can be returned in this case so if you add in a check for it you would be looking to at least print a warning to the screen.

Part 3: European Options Functions Using the Binomial Method

Binomial options pricing model provides a generalizable numerical method for the valuation of options. Was introduced in Week 2 Slides09.pdf.

The model uses a “discrete-time” (lattice based) model of the varying price over time of the underlying financial instrument. Our problem, is to compute the value of the option at the root of the tree. We do so by starting at the leaves and working backward toward the root.

Hints

You can use the Binomial tree implementation in the week 11 labs.

For advanced Users may want to consider the Code Samples:

- Options Pricing on the GPU

Part 4: Monte Carlo European Option

Create a version of your classes which can use float types instead of doubles and compare the time taken.

Hints

The std::chrono provides us the high_resolution_clock which is the most accurate and hence it is used to measure execution time.

Timing alogritm:

- Step 1: Get the timepoint before the function is called

```
#include <chrono>
using namespace std::chrono;

// get the timepoint at this instant use function now()
auto start = high_resolution_clock::now();
```

- Step 2: Get the timepoint after the function is called

```
// After function call
auto stop = high_resolution_clock::now();
```

- Step 3: Get the difference in timepoints and cast it to required units

```
// Subtract stop and start timepoints and
// cast it to required unit. Predefined units
// are nanoseconds, microseconds, milliseconds,
// seconds, minutes, hours. Use duration_cast()
// function.
auto duration = duration_cast<microseconds>(stop - start);

// To get the value of duration use the count()
// member function on the duration object
cout << duration.count() << endl;
```

Notes for Interest

- Advanced Users may want to look at std::future and std::async from the week 11 lecture 5 and compare the times.

Part 5: Binomial American Option

For the American option another for loop is needed so to compare the immediate payoff to the discounted option price of its weighted branches.

```
Inputs:
  T {1.0}           // Time to maturity of option
  N {10e5}          // Height of the binomial tree/Number of timesteps
  v {0.2}           // Volatility of underlying asset
  rate {0.05}       // risk-free interest rate
  S {1.0}           // initial asset price
  K {1.0}           // Strike Price
  q {0.0}           // dividend yield

Calculations:
  dt = T / N;
  Up = exp(v * sqrt(dt));
  v0 = (Up*exp(-q * dt) - exp(-r * dt)) / (Up^2 - 1);
  v1 = exp(-r * dt) - v0;
  // initialize option value array
  for i = 0 to N
      v[i] = K - S * Up^(2*i - N);
      if v[i] < 0 then v[i] = 0;

  // for each time step loop backwards in time
  for j = N-1 down to 0
    for i = 0 to j
        // binomial value
        v[i] = v0 * v[i+1] + v1 * p[i]
        // exercise value
        exercise = K - S * up^(2*i - j)
        if v[i] < exercise then v[i] = exercise
  result = v[0]
```

Optional: Monte Carlo American Option

Add to the American Option to use Longstaff–Schwartz least-squares Monte Carlo method.

Hints

```julia
function lsmc_am_put(S, K, r, σ, t, N, P)
    Δt = t / N
    R = exp(r * Δt)
    T = typeof(S * exp(-σ^2 * Δt / 2 + σ * √Δt * 0.1) / R)
    X = Array{T}(N+1, P)

    for p = 1:P
        X[1, p] = x = S
        for n = 1:N
            x *= R * exp(-σ^2 * Δt / 2 + σ * √Δt * randn())
            X[n+1, p] = x
        end
    end

    V = [max(K - x, 0) / R for x in X[N+1, :]]

    for n = N-1:-1:1
        I = V .!= 0
        A = [x^d for d = 0:3, x in X[n+1, :]]
        β = A[:, I]' \ V[I]
        cV = A' * β
        for p = 1:P
            ev = max(K - X[n+1, p], 0)
            if I[p] && cV[p] < ev
                V[p] = ev / R
            else
                V[p] /= R
            end
        end
    end

    return max(mean(V), K - S)
end
```

For advanced Users may want to consider the Code Samples:

- American Option Pricing with Monte Carlo Simulation

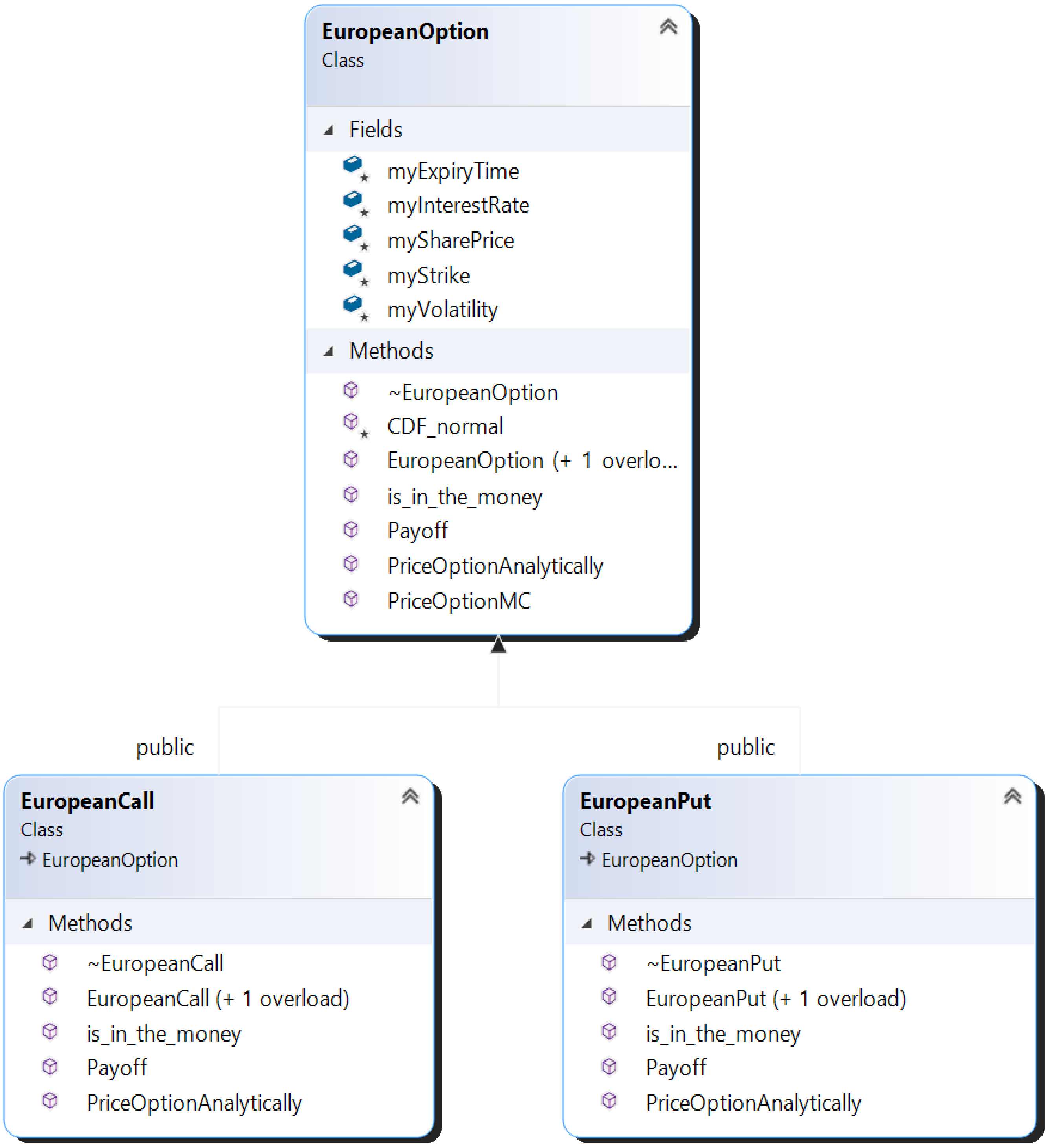
Notes

Here is example data to test and check your algorithms results:

```
// The data below are from
//   "Option pricing formulas", E.G. Haug, McGraw-Hill 1998
//   page 2-8, 24, 27
testvalues={
  //                       type, strike,  spot,    q rate,    r rate,    t,  vol,   value, tol
  { "EuropeanOptionData::Call",  65.00,  60.00, 0.00, 0.08, 0.25, 0.30,  2.1334, 1.0e-4},
  { "EuropeanOptionData::Put",   95.00, 100.00, 0.05, 0.10, 0.50, 0.20,  2.4648, 1.0e-4},
  { "EuropeanOptionData::Put",   19.00,  19.00, 0.10, 0.10, 0.75, 0.28,  1.7011, 1.0e-4},
  { "EuropeanOptionData::Call",  19.00,  19.00, 0.10, 0.10, 0.75, 0.28,  1.7011, 1.0e-4},
  { "EuropeanOptionData::Call",   1.60,   1.56, 0.08, 0.06, 0.50, 0.12,  0.0291, 1.0e-4},
  { "EuropeanOptionData::Put",   70.00,  75.00, 0.05, 0.10, 0.50, 0.35,  4.0870, 1.0e-4},
  { "EuropeanOptionData::Call", 100.00,  90.00, 0.10, 0.10, 0.10, 0.15,  0.0205, 1.0e-4},
  { "AmericanOptionData::Call", 100.00, 100.00, 0.10, 0.10, 0.50, 0.15,  4.0842, 1.0e-16 },
  { "AmericanOptionData::Call", 100.00, 110.00, 0.10, 0.10, 0.50, 0.15, 10.8087, 1.0e-16 },
  { "AmericanOptionData::Call", 100.00, 100.00, 0.10, 0.10, 0.50, 0.35,  9.5106, 1.0e-16 },
  { "AmericanOptionData::Call", 100.00, 110.00, 0.10, 0.10, 0.50, 0.35, 15.5689, 1.0e-16 },
  { "AmericanOptionData::Put",  100.00,  90.00, 0.10, 0.10, 0.10, 0.15, 10.0000, 1.0e-16 },
  { "AmericanOptionData::Put",  100.00, 100.00, 0.10, 0.10, 0.10, 0.15,  1.8770, 1.0e-16 },
  { "AmericanOptionData::Put",  100.00, 100.00, 0.10, 0.10, 0.50, 0.35,  9.5104, 1.0e-16 },
  { "AmericanOptionData::Put",  100.00, 110.00, 0.10, 0.10, 0.50, 0.35,   5.882, 1.0e-16 },
  { "AmericanOptionData::Put",  100.00, 100.00, 0.00, 0.00, 0.50, 0.15,  4.2294, 1.0e-16 }
}
```

Class Diagram

A simple diagram showing the inheritance relations, classes, member functions and attributes (member variables) for our Current **EuropeanOption** Classes Code.



Note

These can be simply generated via MS Visual Studio 2019:

- Goto Class View (left pane next to Solution explorer)
- Right click solution name
- Select view from popup menu
- View class diagram