Group A&B: Exact Pricing Methods

Before answering the questions, it is important to explain some features of the code design in this part:

Namespaces

I used namespaces to create subsystems that can define an entity in case the code will be needed later. In this case, the unique namespace is GERALD::OPTIONS since it is the only entity that is defined. Besides, it will avoid name clashes in global scope considering that I have defined some Global Functions

Global Functions

I used Global Functions to answer some questions from these sections. This is because, some of them were repeated under different Classes. Given this fact, some Global Functions are Template Global Functions to use different Classes as argument.

Abstract Base Class

For this part, first I design an Abstract Base Class (ABC) to define all elements of an Option, such as the interest rate, the volatility, the strike price, the expiry date, cost of carry, the option type (call or put) and the name of the underlying asset. I also include the assign operator and 2 other member functions that will define variables used in either European or American Option Class (such as d1, d2, y1 and y2). Finally, I include a PVMF called Price() to be implemented in bothe Classes. The main reason for using this design is that both European and American Options have these settings in common, so it is important to consider a base Class whose functions will be overridden by the derived Class.

Derived Classes

In Derived Classes such as EuropeanOption and PerpetualAmericanOption I implemented some functionalities from Abstract Base in addition to other useful member functions to answer the questions.

Typedef of Tuple: Batch

In the ABC header I defined a type of Tuple called Batch. Since in some questions we need an array of different data types, I used the std container called Tuple that allows me to define a collection of double data types and string “data type” (the last one being a std class that collect a series of char data types):

typedef tuple<double, double, double, double, double, string, string, double> batch;

This is also used as input for Parameter Constructor in both Derived Class.

A. Exact Solutions of One-Factor Plain Options

For this part the main source file that answer all questions is testExactSolutions.cpp

a) The program structure follows an Inheritance design where EuropeanOption and PerpetualAmericanOption are Derived Classes that came from an Abstract Base Class called Option. Also, the header files and source files for each Class are separated such that constructors, destructors, selectors, getters and other member functions are defined.

One important thing to notice here is that, since Option Class is an Abstract Base Class (ABC), I decided to use the Derived Classes as initializer of data member of the ABC. This is because, we cannot create objects from ABC. So, given that I am interested to defined important features of options that are shared by Derived Classes in one ABC, I stablish this design.

In testExactSolutions.cpp, I calculate the price for both Put and Call options using batches from 1 to 4. The result is the following:

```
---------------------------------- Question a) ----------------------------------------
Batch1: the price of call is 2.13337 and the price of put is 5.84628
Batch2: the price of call is 7.96557 and the price of put is 7.96557
Batch2: the price of call is 0.204058 and the price of put is 4.07326
Batch2: the price of call is 92.1757 and the price of put is 1.2475
```

So, it matches with the result given in the instructions.

b) For this question I design 2 member functions for EuropeanOption Class that allows us to deploy both functionalities: calculate the price of Put/Call using Put-Call parity and prove this relationship. They are called PutCallParity_calculate() and PutCallParity_check().

```
// Put-Call parity
double EuropeanOption::PutCallParity_calculate(double S) const
{
    if (optionType() == "Call")
    {
        return Pricer(S) - S + K() * exp(-r() * T());
    }
    if (optionType() == "Put")
    {
        return Pricer(S) + S - K() * exp(-r() * T());
    }
}

void EuropeanOption::PutCallParity_check(double S, const EuropeanOption& Option) const
{
    cout << "Does the Put-Option parity holds?: ";
    if (optionType() == "Call")
    {
        cout << std::boolalpha << (Option.Pricer(S) - Pricer(S) + S == K() * exp(-r() * T())) << endl;
    }
    if (optionType() == "Put")
    {
        cout << std::boolalpha << (Pricer(S) - Option.Pricer(S) + S == K() * exp(-r() * T())) << endl;
    }
}
```

The results are the following:

```
---------------------------------- Question b) ----------------------------------------
First approach: For batch 1: The put price using put - call parity is :5.84628
which is the same as the call by using exact formula: 5.84628
This mean that we can apply the same function to all batches such that we obtain the puts showed in previous question:
(5.84628,7.96557,4.07326,1.2475)

Second approach: Now we can check if the put-call parity holds for a given call and put prices by using the same overloa
ded member function. Does the Put-Option parity holds?: true
```

It is important to stand out that to obtain that vector of prices I used a Global Function called Print() which will be overloaded for questions below

```cpp
void Print(const std::vector<double>& vectorParameter)
{
    std::cout << "(";
    for (std::size_t i = 0; i < vectorParameter.size(); i++)
    {
        if (i != vectorParameter.size() - 1)
        {
            std::cout << vectorParameter[i] << ",";
        }
        else {
            std::cout << vectorParameter[i];
        }
    }
    std::cout << ")" << endl;
}
```

c) To answer this questions I decided to create a Template Global Function called vectorPricer().

```cpp
template<typename T, typename B>
std::vector<double> vectorPricer(const std::vector<double>& vectorParameter, string parameter,
    double (T::* PricerFunction)(double) const, const B& batch)
{
    std::vector<double> vectorPrices;
    T Option(batch);
    for (std::size_t i = 0; i < vectorParameter.size(); i++)
    {
        if (parameter == "r") { Option.r(vectorParameter[i]); vectorPrices.push_back((Option.*PricerFunction)(get<7>(batch))); }
        if (parameter == "sigma") { Option.sigma(vectorParameter[i]);vectorPrices.push_back((Option.*PricerFunction)(get<7>(batch))); }
        if (parameter == "K") { Option.K(vectorParameter[i]); vectorPrices.push_back((Option.*PricerFunction)(get<7>(batch))); }
        if (parameter == "T") { Option.T(vectorParameter[i]); vectorPrices.push_back((Option.*PricerFunction)(get<7>(batch))); }
        if (parameter == "b") { Option.b(vectorParameter[i]); vectorPrices.push_back((Option.*PricerFunction)(get<7>(batch))); }
        if (parameter == "S") { vectorPrices.push_back((Option.*PricerFunction)(vectorParameter[i])); }
    }
    return vectorPrices;
}
```

This function takes as input the following:

- A vector of Parameter (such as vector of Spot prices). We can use any other vector of parameter in case we want to analyze how the price change when the value of that parameter change (could be interest rate, spot price, maturity date, etc)
- A string that defined which parameter are you interested in evaluating (could be interest rate, spot price, maturity date, etc)
- A pointer to a T Class member function. Given that the goal is to apply this function to other Derived class such as PerpetualAmericanOption Class, then I used a template argument T that defined the Class where belongs the member function at which is the pointer pointing to.
- A B template Parameter. This is used to pass a template parameter of any data type. However, in this case I use a defined type of Tuple introduced earlier in this documentation (Batch tuple) to save all data types required to construct an option object.

Also I used the Global Function called sequence() to defined a sequence from a to b with n intervals of size h, such that $h = \frac{b-a}{n}$:

```cpp
std::vector<double> Sequence(double a, double b, double h) // vector of sequence between number a and by that goes by h
{
    double n_elements = ((b - a) / h) + 1;
    std::vector<double> vector(n_elements);
    a -= h;
    for (size_t i = 0; i < vector.size(); ++i) {
        a += h;
        vector[i] = a;
    }
    return vector;
}
```

The result for this question:

```
------------------------------ Question c) ----------------------------------------
[(30,3.11926e-07),
(35,5.19359e-05),
(40,0.00199417),
(45,0.0278174),
(50,0.189181),
(55,0.76652),
(60,2.13337),
(65,4.5252)]
```

Where the output of the function is printed by using the following overloaded Global Function called Print() such that the price is related to the corresponding parameter value (in this case the spot price):

```
void Print(const std::vector<double>& vectorParameter, const std::vector<double>& vectorPrices)
{
    std::cout << "[";
    for (std::size_t i = 0; i < vectorParameter.size(); i++)
    {
        if (i != vectorParameter.size() - 1)
        {
            std::cout << "(" << vectorParameter[i] << "," << vectorPrices[i] << ")" << "," << endl;
        }
        else {
            std::cout << "(" << vectorParameter[i] << "," << vectorPrices[i] << ")";
        }
    }
    std::cout << "]" << endl;
}
```

d) This question can also be responded by using the previous vectorPricer() Global Function. However, I decided to create a matrixPricer() Global Function as well, that follows a similar reasoning of the first function:

```
template<typename T, typename B>
std::vector<double> matrixPricer(const std::vector<B>& matrixParameter,
    double (T::* PricerFunction)(double) const)
{
    std::vector<double> vectorPrices;
    for (int i = 0; i < matrixParameter.size(); i++)
    {
        T Option(matrixParameter[i]);
        vectorPrices.push_back((Option.*PricerFunction)(std::get<7>(matrixParameter[i])));
    }
    return vectorPrices;
}
```

This function takes as input the following:

- A matrix of Parameter. In this case, the matrix of parameter is defined as a vector of a template parameter. In this case, the template parameter will be the type of tuple called Batch. I used this definition, as explained earlier, to save all relevant data types for creating option objects.
- A pointer to a member function of T class. Again, I used a template parameter that defined the class of the member function at which the pointer is pointing to.

The result is printed using overloaded Print() function. The result is the following:

```
------------------------------ Question d) ----------------------------------------
This question can be responded by using the defined vectorPrice where we only use a different parameter vector:
[(0,0),
(0.05,0.268253),
(0.1,0.761015),
(0.15,1.24415),
(0.2,1.70096),
(0.25,2.13337),
(0.3,2.54494),
(0.35,2.93885),
(0.4,3.31764),
(0.45,3.68335),
(0.5,4.03758),
(0.55,4.38166),
(0.6,4.71666),
(0.65,5.04346),
(0.7,5.36283),
(0.75,5.6754),
(0.8,5.98171),
(0.85,6.28224),
(0.9,6.5774),
(0.95,6.86754),
(1,7.15299)]
However, I also designed a matrixPricer function that take as input a matrix of parameters and the result is a vector of
 option prices
(2.13337,7.96557,0.204058,92.1757)
```

Option Sensitivities, aka the Greeks

For this part the main source file that answer all questions is testGreeks.cpp. I implemented some member functions of the class EuropeanOption to find the greeks for each batch.

```cpp
// Greeks
double EuropeanOption::Delta(double S) const
{
    double d1 = d(S)[0]; // Get d1 for option pricing
    double d2 = d(S)[1]; // Get d2 for option pricing
    if (optionType() == "Call")
    {
        return exp((b() - r()) * T()) * N(d1);
    }
    if (optionType() == "Put")
    {
        return  -exp((b() - r()) * T()) * N(-d1);
    }
}

double EuropeanOption::Gamma(double S) const
{
    double d1 = d(S)[0]; // Get d1 for option pricing
    return n(d1) * exp((b() - r()) * T()) / (S * sigma() * pow(T(), 0.5));
}

double EuropeanOption::Vega(double S) const
{
    double d1 = d(S)[0]; // Get d1 for option pricing
    return S * pow(T(), 0.5) * exp((b() - r()) * T()) * n(d1);
}

double EuropeanOption::Theta(double S) const
{
    double d1 = d(S)[0]; // Get d1 for option pricing
    double d2 = d(S)[1]; // Get d2 for option pricing
    if (optionType() == "Call")
    {
        return  (-S * sigma() * exp((b() - r()) * T()) * n(d1) / (2 * pow(T(), 0.5))) -
            ((b() - r()) * S * exp((b() - r()) * T()) * N(d1)) -
            (r() * K() * exp(-r() * T()) * N(d2));
    }
    if (optionType() == "Put")
    {
        return  (-S * sigma() * exp((b() - r()) * T()) * n(d1) / (2 * pow(T(), 0.5))) -
            ((b() - r()) * S * exp((b() - r()) * T()) * N(-d1)) -
            (r() * K() * exp(-r() * T()) * N(-d2));
    }
}
```

a) The results are the following:

```
---------------------------------- Question a) ----------------------------------------
Call Delta: 0.594629
Put Delta: -0.356601
```

b) For this question I used the same sequence() and vectorPricer() function as before, and I obtained the following result:

```
---------------------------------- Question b) ----------------------------------------
Vector of Call Delta for different Spot:
(0.215772,0.289765,0.368319,0.447475,0.523785,0.594629,0.658306)
```

c) For this question I used sequence(), vectorPricer() and matrixPricer() functions as before, and I obtained the following result:

```
-------------------------------- Question c) ----------------------------------------
This question can be responded by using the defined vectorPrice where we only use a different parameter vector:
Vector of Call Deltas for different Maturity:
(0.914253,0.720538,0.672785,0.648932,0.633918,0.623252,0.615074,0.608461,0.602902,0.598084,0.59381,0.589946,0.586398,0.5
83098,0.579998,0.577059,0.574253,0.571556,0.568953,0.566427)

Vector of Call Gammas for different Maturity:
(0.0410644,0.0358298,0.0282044,0.023865,0.0209954,0.018918,0.0173234,0.0160482,0.0149973,0.0141111,0.01335,0.0126866,0.0
121014,0.0115798,0.0111108,0.010686,0.0102987,0.00994344,0.00961605,0.00931294)

However, we can also apply the matrixPricer function that takes as input a matrix of parameters and the result is a vect
or of option Gammas or Deltas
(12.4328,13.2208,13.9201,14.5474)
```

d) For this question I design a member function in EuropeanOption Class that return an approximation for Delta and Gamma

```cpp
// Function that approximate to Abstract Functions Greeks
double EuropeanOption::DeltaApproximation(double S, double h) const
{
    return (Pricer(S + h) - Pricer(S - h)) / 2 * h;
}

double EuropeanOption::GammaApproximation(double S, double h) const
{
    return (Pricer(S + h) - 2 * Pricer(S) + Pricer(S - h)) / pow(h, 2.0);
}
```

I also design a matrixGreeksApproximation() global function that takes as input a vector of Spot prices, a vector of h, and a string that defines which Greek to approximate. The result is a matrix of any Greek:

```
---------------------------------- Question d) ------------------------------------------
Delta approximation is: 0.594629, which is similar to the exact solution: 0.594629
Gamma approximation is: 0.0134937, which is similar to the exact solution: 0.0134936

Now we can also apply the matrixGreekApproximation that takes as input two vector of Spot price and h values and the res
ult is a matrix of approximation of Gammas and Deltas:

[(0.215772,0.215772,0.215772,0.215772,0.215772,0.215772,0.215772,0.215772,0.215772,0.215772)
(0.289765,0.289765,0.289765,0.289765,0.289765,0.289765,0.289765,0.289765,0.289765,0.289765)
(0.368319,0.368319,0.368319,0.368319,0.368319,0.368319,0.368319,0.368319,0.368319,0.368319)
(0.447475,0.447475,0.447475,0.447475,0.447475,0.447475,0.447475,0.447475,0.447475,0.447475)
(0.523785,0.523785,0.523785,0.523785,0.523785,0.523785,0.523785,0.523785,0.523785,0.523785)
(0.594629,0.594629,0.594629,0.594629,0.594629,0.594629,0.594629,0.594629,0.594629,0.594629)
(0.658306,0.658306,0.658306,0.658306,0.658306,0.658306,0.658306,0.658306,0.658306,0.658306)


[(0.0140733,0.0140733,0.0140733,0.0140733,0.0140733,0.0140733,0.0140733,0.0140733,0.0140733,0.0140733)
(0.0153905,0.0153905,0.0153905,0.0153905,0.0153905,0.0153905,0.0153905,0.0153905,0.0153905,0.0153905)
(0.0158974,0.0158974,0.0158974,0.0158974,0.0158974,0.0158974,0.0158974,0.0158974,0.0158974,0.0158974)
(0.0156491,0.0156491,0.0156491,0.0156491,0.0156491,0.0156491,0.0156491,0.0156491,0.0156491,0.0156491)
(0.0147874,0.0147873,0.0147873,0.0147873,0.0147873,0.0147873,0.0147873,0.0147873,0.0147873,0.0147873)
(0.0134937,0.0134936,0.0134936,0.0134936,0.0134936,0.0134936,0.0134936,0.0134936,0.0134936,0.0134936)
(0.0119498,0.0119498,0.0119498,0.0119498,0.0119498,0.0119498,0.0119498,0.0119498,0.0119498,0.0119498)
```

B. Perpetual American Options

For this section I developed a PerpetualAmericanOption Class that display some functionalities related to this option. the main source file that answer all questions is testPerpetualAmericanOption.cpp

a) I created a Derived Class called PerpertualAmericanOption that contain all functionalities from Option ABC. It initializes the values that defined an option object. Also, the header files and source files for each Class are separated such that constructors, destructors, selectors, getters and other member functions are defined. Finally, it is important to stand out that I also used the Batch definition of Tuple as input parameter for parameter Constructor. This is because I will used the vectorPricer() and matrixPricer() Global Functions for questions c) and d)

b) The result after designing this class is

```
---------------------------------- Question b) ------------------------------------------
The result for the call is 18.5035
The result for the put is 3.03106
```

c) The result after using vectorPricer() Global Function:

```
---------------------------------- Question c) ------------------------------------------
I used the vectorPricer to find different prices for different Spot input parameter:
[(80,6.64256),
(85,8.07301),
(90,9.7027),
(95,11.546),
(100,13.6174),
(105,15.9316),
(110,18.5035),
(115,21.3481),
(120,24.4804)]
```

d) The result after using vectorPricer() and matrixPricer() Global Functions:

```
---------------------------------- Question d) ------------------------------------------
As before, I used first vectorPricer for a new set of sigma parameters:
[(0,-nan(ind)),
(0.1,18.5035),
(0.2,26.5984),
(0.3,34.6188),
(0.4,42.1321),
(0.5,48.9929),
(0.6,55.158),
(0.7,60.6398),
(0.8,65.4822),
(0.9,69.7443),
(1,73.4904)]
But I also used matrixPricer:
(18.5035,26.5984,34.6188,42.1321)
```

Group C&D: Monte Carlo Pricing Methods

C. Monte Carlo 101

a) When it comes to relating the code with the theory provided, first it is important to find a relationship between SDEdefinition namespace and explicit Euler-Maruyama scheme to approximate a solution to SDE. First, in this case, the drift and function are related to the second and third term of the right hand side of the following equation

$$X_{n+1} = X_n + aX_n\Delta t_n + bX_n\Delta W_n,$$

i.e. aX_n is the *drift* function and bX_n is the *diffusion* function:

```
double drift(double t, double X)
{ // Drift term

    return (data->r)*X; // r - D
}


double diffusion(double t, double X)
{ // Diffusion term

    double betaCEV = 1.0;
    return data->sig * pow(X, betaCEV);

}
```

Such that *a* = *r* and *sig* = b, and *betaCEV* take necessarily the value of 1. Finally, the definitions

$$\begin{cases} \Delta t_n = \Delta t = T/N, \quad 0 \le n \le N-1 \\ \Delta W_n = \sqrt{\Delta t} z_n, \text{ where } z_n \sim N(0,1). \end{cases}$$

are related to the code such that $\Delta t = k$ and $\Delta W_n = \sqrt{k}$ and $z_n = dw$

```
double k = myOption.T / double (N);
double sqrk = sqrt(k);
```

```
// NormalGenerator is a base class
NormalGenerator* myNormal = new BoostNormal();
```

```
// Create a random number
dW = myNormal->getNormal();
```

Where myNormal is a pointer to a NormalGenerator object on the heap.

So in the code a for loop is created to perform N simulation (*Nsim* is the variable in the code) of an approximation to SDE using T intervals whose N+1 mesh points (boundaries) were created using a member function called *mesh* in template Range class and saved in a vector of doubles called *x*. These boundaries are the $t_i = i, for\ i \in [0, N]$ and follow the condition:

$$0 = t_0 < t_1 < .. < t_n < t_{n+1} < .. < t_N = T.$$

In this case we define a set of *subintervals* (t_n, t_{n+1}) of size $\Delta t_n \equiv t_{n+1} - t_n, \quad 0 \le n \le N-1$.

```cpp
// Utility functions
template <class Type>
std::vector<Type> Range<Type>::mesh(long nSteps) const
{ // Create a discrete mesh

    Type h = (hi - lo) / Type (nSteps);

    std::vector<Type>result(nSteps + 1);

    Type val = lo;

    for (long i = 0; i < nSteps + 1; i++)
    {
        result[i] = val;
        val += h;
    }

    return result;
}
```

Member function

```cpp
// Create the basic SDE (Context class)
Range<double> range (0.0, myOption.T);
double VOld = S_0;
double VNew;

std::vector<double> x = range.mesh(N);
```

Vector that has all mesh points of the intervals

```cpp
for (unsigned long index = 1; index < x.size(); ++index)
{

    // Create a random number
    dW = myNormal->getNormal();

    // The FDM (in this case explicit Euler)
    VNew = VOld  + (k * drift(x[index-1], VOld))
                + (sqrk * diffusion(x[index-1], VOld) * dW);

    VOld = VNew;

    // Spurious values
    if (VNew <= 0.0) coun++;
}

double tmp = myOption.myPayOffFunction(VNew);
price += (tmp)/double(NSim);
```

For loop to simulate N times the approximation

Finally, when I run the code, I obtained:

```
1 factor MC with explicit Euler
Number of subintervals in time: 20
Number of simulations: 100000
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Price, after discounting: 5.86659,
Number of times origin is hit: 0
```

b) For this question, I have provided comparison of the results of running Monte Carlo Simulation for different values of intervals and number of simulations for batch 1 with exact solutions under the following expression:

$$|C - C(NM, NSIM)|$$

Where C is the price obtained using exact solution (BS equation) and $C(NM, NSIM)$ is the price of option obtained using Monte Carlo simulation with NM intervals and NSIM simulations. It is expressed in absolute value to understand how far the approximation with respect to the actual value.

Batch 1: T = 0.25, K = 65, sig = 0.30, r = 0.08, S = 60 (then C = 2.13337, P = 5.84628).

BATCH 1 - Put		N° Simulations				
		0.001M	0.01M	0.1M	1M	10M
N°Intervalos	100	0.00223	0.06179	0.02693	0.00497	0.00445
	200	0.04688	0.10753	0.00615	0.01188	0.00413
	300	0.14382	0.14068	0.00593	0.00741	0.00454
	400	0.31635	0.08713	0.0107	0.00027	0.00257
	500	0.1867	0.09126	0.0081	0.00503	0.00086

`

BATCH 1 - Call		N° Simulations				
		0.001M	0.01M	0.1M	1M	10M
N°Intervalos	100	0.05927	0.00443	0.00294	0.00066	0.00114
	200	0.01877	0.04103	0.01551	0.00019	0.00061
	300	0.09345	0.03034	0.03463	0.00133	0.00039
	400	0.01839	0.01375	0.0274	0.00279	0.00075
	500	0.11009	0.0063	0.01526	0.00266	0.00054

For batch 1, we can see that when the number of simulations increases, the approximation is closer to the exact solution. This is also the case for the number of intervals, since we mostly obtained better approximations when we keep simulations constant and increase number of intervals. Therefore, it seems that for this batch, it is important to increase both. For put option the best solution found is NT = 400 and NSIM = 1M and for call it is NT = 200 and NSIM = 1M.

Batch 2: T = 1.0, K = 100, sig = 0.2, r = 0.0, S = 100 (then C = 7.96557, P = 7.96557).

BATCH 2 - Put		N° Simulations				
		0.001M	0.01M	0.1M	1M	10M
N°Intervalos	100	0.08063	0.09779	0.04233	0.00882	0.00469
	200	0.13815	0.17012	0.01081	0.02024	0.00597
	300	0.22446	0.22571	0.02509	0.01898	0.00559
	400	0.40889	0.13184	0.01742	0.00097	0.00375
	500	0.32128	0.17185	0.00762	0.00894	0.00018

BATCH 2 - Call		N° Simulations				
		0.001M	0.01M	0.1M	1M	10M
N°Intervalos	100	0.23	0.0246	0.02195	0.00307	0.00267
	200	0.08265	0.15306	0.03503	0.00444	0.0017
	300	0.31643	0.14408	0.09374	0.00678	0.0035
	400	0.24379	0.08961	0.06999	0.00587	5E-05
	500	0.34085	0.03299	0.04699	0.00415	0.00309

For batch 2, the result is almost the same. When we increase both NT and NSIM we obtain better approximations. The best result for put is NT = 500 and NSIM = 10M, and the best result for Call is NT = 400 and NSIM = 10M.

c) Batch 4: T = 30.0, K = 100.0, sig = 0.30, r = 0.08, S = 100.0 (C = 92.17570, P = 1.24750).

BATCH 4 - Put		N° Simulations				
		0.001M	0.01M	0.1M	1M	10M
N°Intervalos	100	0.01893	0.07446	0.04854	0.04525	0.04273
	200	0.08459	0.05734	0.02492	0.02293	0.01907
	300	0.03847	0.04643	0.02454	0.01814	0.01237
	400	0.08283	0.03609	0.01316	0.01024	0.00914
	500	0.10321	0.05649	0.00626	0.00678	0.00844

BATCH 4 - Call		N° Simulations				
		0.001M	0.01M	0.1M	1M	10M
N°Intervalos	100	6.3996	4.1345	2.7509	2.6516	2.8767
	200	5.055	1.6947	0.0368	0.868	1.3517
	300	6.4011	2.1608	0.5681	0.7901	0.8624
	400	2.7088	0.8433	0.7211	0.6743	0.7664
	500	8.1473	3.2257	1.2304	0.3307	0.5699

For the put option we can see that we obtained an accuracy to two places behind the decimal point for NT = 500 and for NSIM >= 0.1M. When NT = 400 and NSIM = 10M we also obtained an accuracy of two places behind the decimal point. However we cannot reach an accuracy of two places behind decimal point for call option. The best approximation for this is NT = 500 and NSIM = 1M.

D. Advanced Monte Carlo

a) For this part I created a function called accuracy that returns a vector containing SD and SE results with vector input argument that contains price result from each simulation:

```
std::vector<double> accuracy(const std::vector<double>& vectorPrices, const double& r, const double& T) // Funct
{
    double priceOptionSum = 0.0; // Sum of each simulation option prices for SD calculation
    double priceOptionSumSquare = 0.0; // Sum of squares of each simulation option prices for SD calculation
    double NSim = vectorPrices.size(); // Number of simulations
    typename std::vector<double>::const_iterator i;
    for (i = vectorPrices.begin(); i != vectorPrices.end(); ++i)
    {
        priceOptionSum += *i; // sum option prices for Standard Deviation
        priceOptionSumSquare += pow(*i, 2.0); // sum square of option prices for Standard Deviation
    }
    // Stanrdard Deviation
    double SD = pow((priceOptionSumSquare - (pow(priceOptionSum, 2.0)/ NSim)) / (NSim - 1), 0.5) * exp(-r * T);
    double SE = SD / pow(NSim, 0.5); // Standard Error
    std::vector<double> vectorResults = { SD,SE }; // vector that save both SD and SE
    return vectorResults;
}
```

To avoid having to repeat the process for different combinations of NT and NSIM I decided to create loops inside the main source file such that each combination is store in a matrix. So, I had 3 matrixes for different combinations in order to perform a stress analysis of Price, SD and SE respectively. This code is in the main source file called TestMC2.cpp. Also, I save the UtilitiesDJD file in my boost file to avoid any problem when running in *release* mode.

```
std::vector<double> vectorNT = { 100.0, 200.0, 300.0, 400.0, 500.0};
std::vector<double> vectorNSIM = { 1000.0, 10000.0, 100000.0, 1000000.0, 10000000.0};
std::vector<vector<double>> matrixResultPrices(vectorNT.size());
std::vector<vector<double>> matrixResultAccuracySD(vectorNT.size());
std::vector<vector<double>> matrixResultAccuracySE(vectorNT.size());
```

The result is the following:

For both batches I only used the Put option. I obtained the SD and SE for both.

Batch 1: T = 0.25, K = 65, sig = 0.30, r = 0.08, S = 60 (then C = 2.13337, P = 5.84628).

BATCH 1 - SD Put		N° Simulations				
		0.001M	0.01M	0.1M	1M	10M
N°Intervalos	100	5.89612	6.0547	6.05775	6.04898	6.04901
	200	6.0702	6.08481	6.05047	6.05177	6.04681
	300	5.88842	6.07232	6.06286	6.05714	6.04754
	400	5.8996	6.02597	6.04473	6.04954	6.04706
	500	6.06848	6.09454	6.05203	6.04743	6.04884

`

BATCH1 - SE Put		N° Simulations				
		0.001M	0.01M	0.1M	1M	10M
N°Intervalos	100	0.18645	0.06055	0.01916	0.00605	0.0019129
	200	0.19196	0.06085	0.01913	0.00605	0.0019122
	300	0.18621	0.06072	0.01917	0.00606	0.0019124
	400	0.18656	0.06026	0.01912	0.00605	0.0019123
	500	0.1919	0.06095	0.01914	0.00605	0.0019128

Batch 2: T = 1.0, K = 100, sig = 0.2, r = 0.0, S = 100 (then C = 7.96557, P = 7.96557).

BATCH 2 - SD Put		N° Simulations				
		0.001M	0.01M	0.1M	1M	10M
N°Intervalos	100	10.1262	10.435	10.4359	10.4143	10.4125
	200	10.4157	10.5086	10.4152	10.4167	10.405
	300	10.0488	10.4754	10.4292	10.4229	10.4043
	400	10.1804	10.3668	10.3994	10.4097	10.4039
	500	10.4581	10.5022	10.4107	10.4052	10.4071

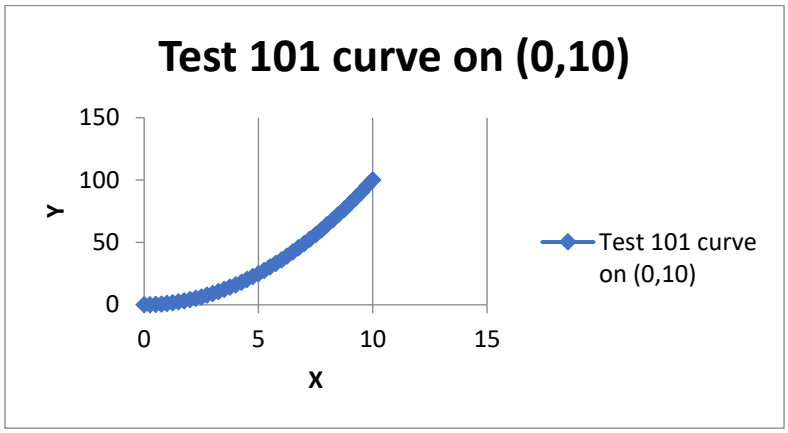
BATCH 2 – SE Put		N° Simulations				
		0.001M	0.01M	0.1M	1M	10M
N°Intervalos	100	0.320219	0.10435	0.033001	0.010414	0.003293
	200	0.329374	0.105086	0.032936	0.010417	0.00329
	300	0.31777	0.104754	0.03298	0.010423	0.00329
	400	0.321933	0.103668	0.032886	0.01041	0.00329
	500	0.330715	0.105022	0.032922	0.010405	0.003291

We can observe that when the number of simulations increase, the accuracy becomes mostly better. The same applies to the number of intervals. This is more explicit for SE, where the improvement in accuracy when both increases is clear and pronounced. It is important to stand out that SD is big, which lead us to deduce that the differences in prices between each simulation is not small, so, because of that, it is understandable to have more simulations to be closed to the exact solution.

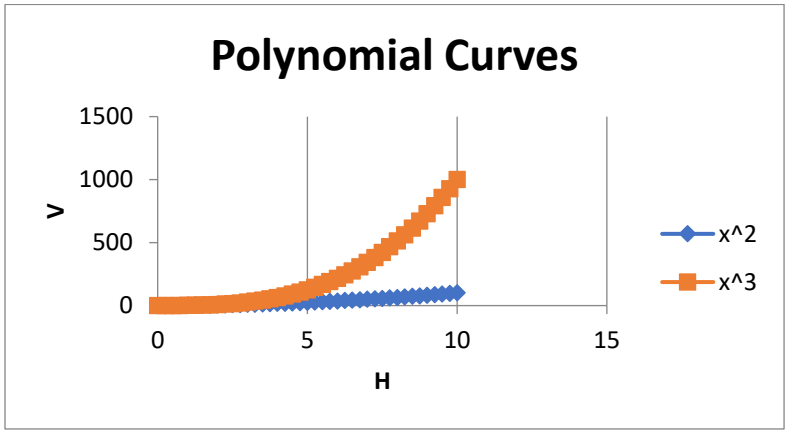
Group E: Excel Visualization

a) For this part I obtained the outputs:

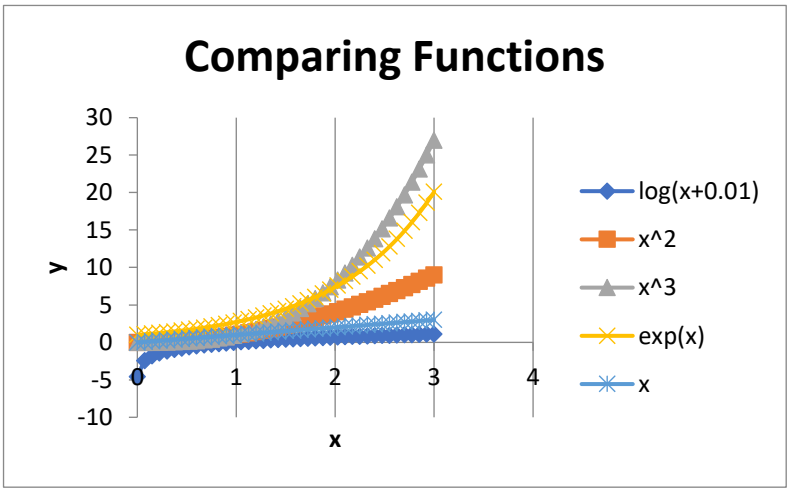
TestSingleCurve.cpp



TestTwoCurve.cpp

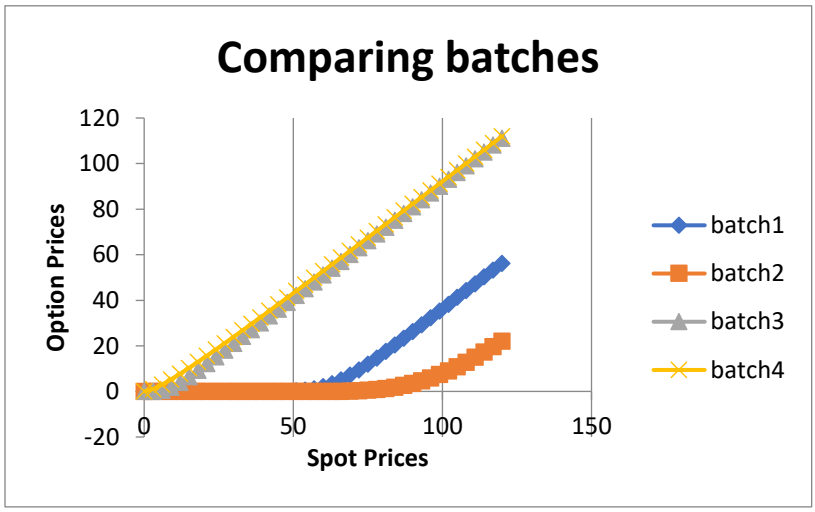


TestMultiCurve.cpp



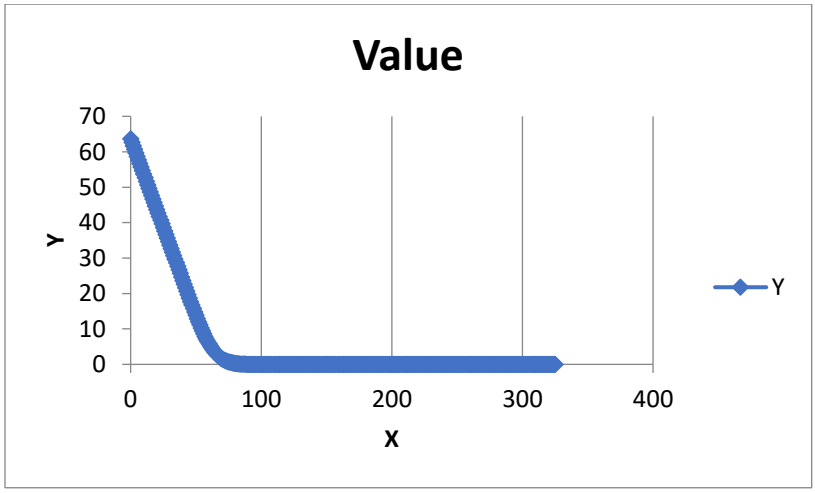
b) For this part, I test the previous program by using different batches presented in Group A under different values of Spot prices. I used the Global Functions created previously.

I created first the batches. Then I used the code given to obtain output of applying the Global Function vectorPrices again and evaluate this in the ExcelDriver code provided. All this code is in the main source file TestOptionCurve.cpp. The result is the following graph:



Group F: Finite Difference Methods

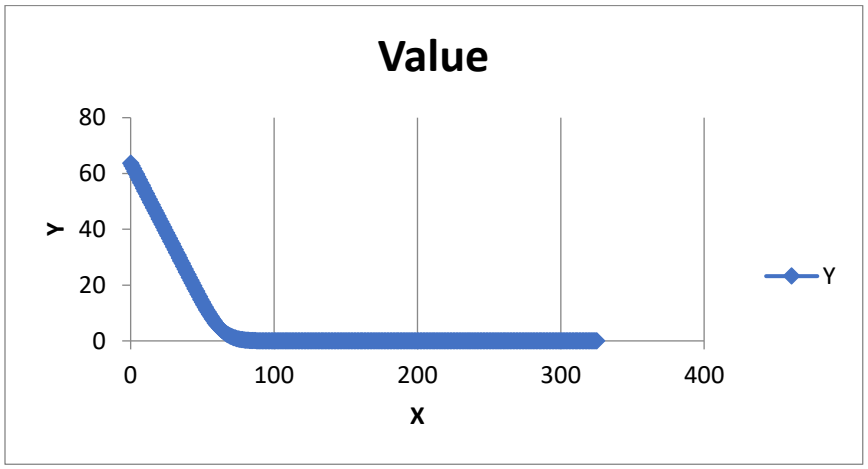
a) I obtained this result



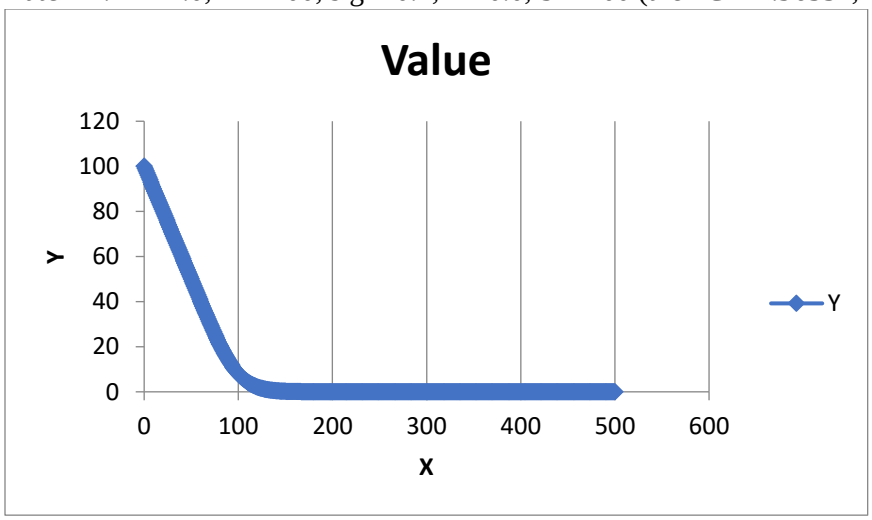
As we can see it is a Put option. This is created using the provided code in which fdir.xarr is the vector of Spot price (the x-axis) and fdirr.current() is the vector of FDM price (the y-axis). Then, we are using FDM to price an option.

b) Batch 1: T = 0.25, K = 65, sig = 0.30, r = 0.08, S = 60 (then C = 2.13337, P = 5.84628).

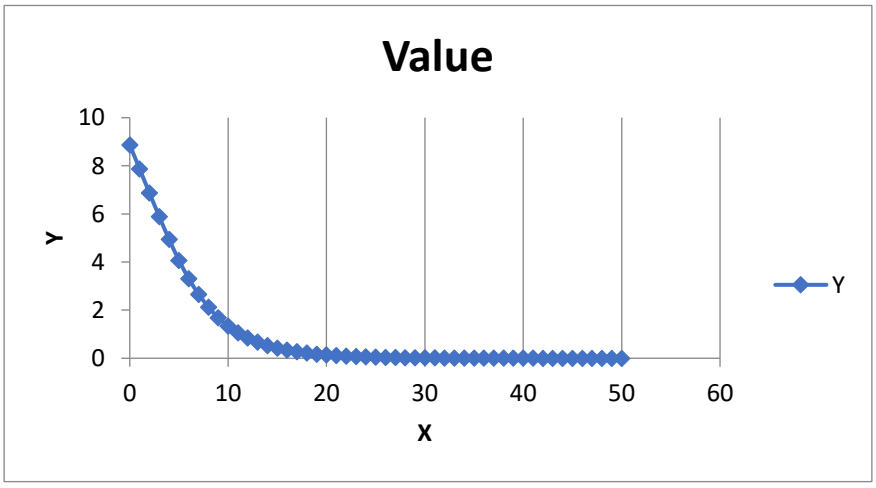
For this Batch I obtained the same result as a)



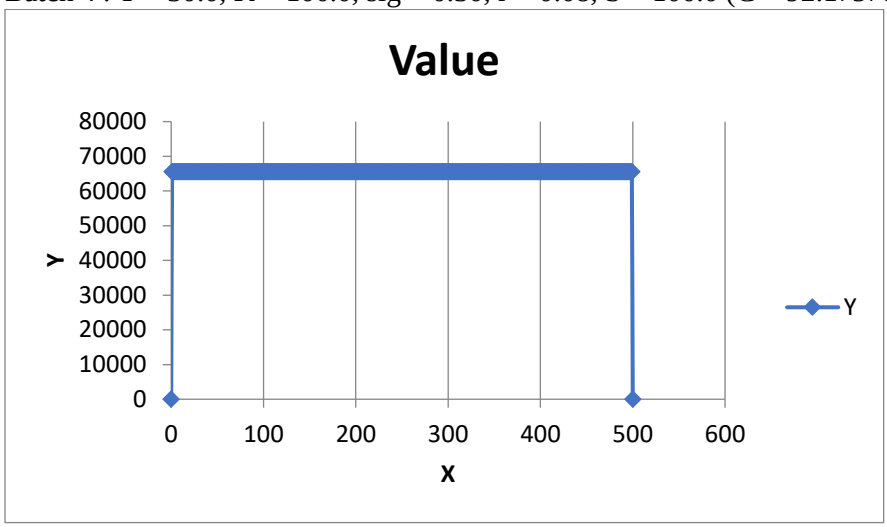
Batch 2 : T = 1.0, K = 100, sig = 0.2, r = 0.0, S = 100 (then C = 7.96557, P = 7.96557).

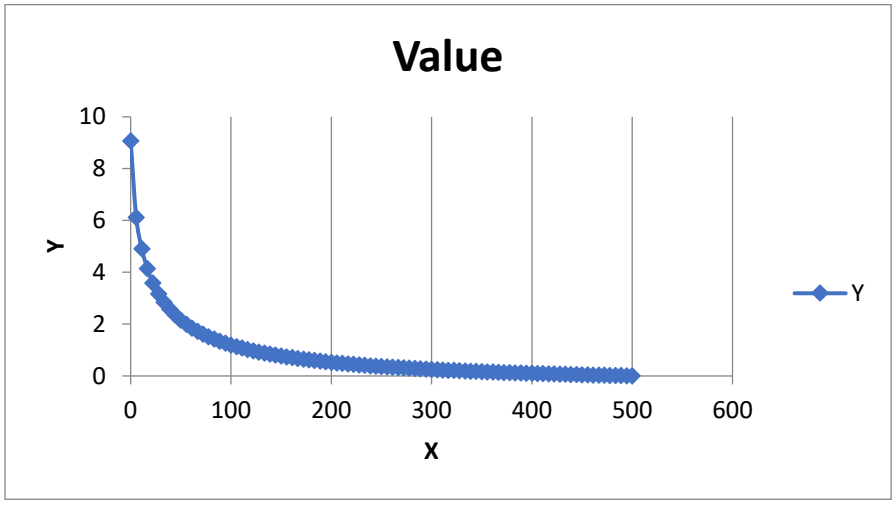


Batch 3 : T = 1.0, K = 10, sig = 0.50, r = 0.12, S = 5 (C = 0.204058, P = 4.07326).



Batch 4 : T = 30.0, K = 100.0, sig = 0.30, r = 0.08, S = 100.0 (C = 92.17570, P = 1.24750).





For Batch4 I obtained the first graph which is an unusual result. Because of that I tried different values of J and N, such that I found that the optimal is J = 90 and N = 60 000.

It is important to stand out that for this exercise I also create an Excel File (and store it on Excel File document) called “Comparing FDM vs ExactSolution” where I compare in different table the result of FDM price and BS price (exact solution). Those prices are similar, except for batch4. The FDM price is similar to exact solution for this Batch when I used the parameter J = 90 and N = 60 000.