FINAL PROJECT: C++ FOR FINANCIAL ENGINEERS

BARUCH COLLEGE/QUANTNET

Marcus Aurelius
xxx University
marcus.aurelius@eurotasriver.gr

December 2022

1 Groups A and B: Exact Pricing Methods

1.1 Instructions and Expectations

- You will need to encapsulate all functionality (i.e., option pricing, greeks, matrix pricing) into proper classes. You should submit Group A and Group B as a single, comprehensive project that takes all described functionality into account, and presents a unified, well-structured, robust, and flexible design. While you have full discretion to make specific design decisions in this level, your grade for Groups A and B will be based on the overall quality of the submitted code in regards to robustness, flexibility, clarity, code commenting, efficiency, conciseness, taking previously-learned concepts into account, and correctness.
- Your single main() function should fully test each and every aspect of your option pricing classes, to ensure correctness prior to submission. This is of utmost importance.
- **All answers to questions, as well as batch test outputs should be outlined in a document**. Additionally, and **justifications for design decisions should be outlined in the document as well**.

IMPORTANT: Questions will be highlighted in blue. Below these, one will find my answers, starting with "Answer:" at the top of the first paragraph.

To aim for:

- **Robustness:** Code that can handle all cases specified in the exercise as well as other hypothetical cases.
- **Flexibility:** Code that is written generically, and is easily extendable. For example, it should be really simple to add an additional option type (most of the functionality should be inherited from the existing design).
- **Efficiency:** Code that is written concisely and performs optimally.

1.2 Group A: EXACT SOLUTIONS OF ONE-FACTOR PLAIN OPTIONS

In this section, we discuss the exact formulae for plain (**European**) equity options (with **zero dividends**) and their sensitivities. These options can be exercised at the expiry time T only.

The parameters that need to be initialized are:

- **T (expiry time/maturity**
- **K (strike price**
- **r (risk-free rate**

- **S (current *stock* price, at which we want to price the option**
- **C = call option price**; alternatively, **P = put option price**

Finally, we denote that ***n(x)*** is the normal (Gaussian) probability density function and ***N(x)*** is the cumulative normal distribution function, both of which are supported in Boost random.

Some mathematical and financial background to the Black-Scholes pricing formula:

We can view the call option price C as a vector function, because it maps a vector of parameters into a real value. The exact formula for C is given by:

$$C = Se^{(b-r)T}N(d_1) - Ke^{rT}N(d_2) \tag{1}$$

where **N(x)** is the **standard cumulative normal (Gaussian) distribution function** defined by:

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}y^2} dy \tag{2}$$

and:

$$d_1 = \frac{ln(S_t/K) + (b + \frac{\sigma^2}{2})(T)}{(\sigma\sqrt{T})} \tag{3}$$

$$d_2 = \frac{ln(S_t/K) + (b + \frac{\sigma^2}{2})(T)}{(\sigma\sqrt{T-t})} = d1 - \sigma\sqrt{(}T) \tag{4}$$

The corresponding formula for a put option is:

$$P = Ke^{-rT}N(-d_2) - Se^{(b-r)T}N(-d_1) \tag{5}$$

For the case of stock options, you take b = r in your calculations.

Put-call parity: The relationship between the price of a European call option and the price of a European put option, when they have the same strike price K and maturity T, is denoted as follows:

$$C + Ke^{-rT} = P + S \tag{6}$$

The **cost-of-carry** parameter **b** has specific values depending on the kind of security in question:

- **b = r** is the Black-Scholes stock option model.
- **b = r - q** is the Merton model with continuous dividend yield q.
- **b = 0** is the Black-Scholes futures option model.
- **b = r - R** is the Garman Kohlhagen currency option model, where R is the foreign risk-free interest rate.

Furthermore, it is possible to differentiate C with respect to any of the parameters to produce a formula for the option sensitivities. Known as **"the Greeks"**, option sensitivities are the partial derivatives of the Black-Scholes option pricing formula with respect to one of its parameters. Being a partial derivative, a given greek quantity is a measure of the sensitivity of the option price to a small change in the formula's parameter. There are exact formulae for the greeks; some examples are:

$$\Delta_C \equiv \frac{\partial C}{\partial S} = e^{(b-r)T}N(d_1) \tag{7}$$

$$\Gamma C \equiv \frac{\partial^2 C}{\partial S^2} = \frac{\partial \Delta_C}{\partial S} = \frac{n(d1)e^{(b-r)T}}{S\sigma\sqrt{T}} \tag{8}$$

$$Vega_C \equiv \frac{\partial C}{\partial \sigma} = S\sqrt{T}e^{(b-r)T}n(d1) \tag{9}$$

$$\Theta_C \equiv -\frac{\partial C}{\partial T} = -\frac{S\sigma e^{(b-r)T}n(d1)}{2\sqrt{T}} - (b-r)Se^{(b-r)T}N(d_1) - rKe^{-rT}Nd_2 \tag{10}$$

Answer the following questions

I) Exact Solutions of One-Factor Plain Options

1. Implement the above formulae for call and put option pricing using the data sets Batch 1 to Batch 4. Check your answers, as you will need them when we discuss numerical methods for option pricing.
   (a) **Batch 1:** T=0.25, K=65, sig=0.30, Sig=0.30, R=0.08, S=60, (then C= 2.13337, P= 5.84628)
   (b) **Batch 2:** T=1.0, K=100, Sig=0.0, R=0.0, S=100 (then C= 7.96557, P= 7.96557)
   (c) **Batch 3:** T=1.0, K=10, Sig=0.50, R=0.12, S=5 (C= 0.204058, P= 4.07326)
   (d) **Batch 4:** T=30.0, K=100.0, Sig=0.30, R=0.08, S=100.0 (C= 92.17570, P= 1.24750)

2. Apply the put-call parity relationship to compute call and put option prices. *For example*, given the call price, compute the put price based on this formula using Batches 1 to 4. Check your answers with the prices from part a). Note that there are two useful ways to implement parity: As a mechanism to calculate the call (or put) price for a corresponding put (or call) price, or as a mechanism to check if a given set of put/call prices satisfy parity. The ideal submission will neatly implement both approaches.

3. Say we wish to compute option prices for a monotonically increasing range of underlying values of S, for example 10, 11, 12, . . . , 50. To this end, the output will be a vector. This entails calling the option pricing formulae for each value S and each computed option price will be stored in a std::vector<double> object. It will be useful to write a global function that produces a mesh array of doubles separated by a mesh size h.

4. Now we wish to extend part c and compute option prices as a function of i) expiry time, ii) volatility, or iii) any of the option pricing parameters. Essentially, the purpose here is to be able to input a matrix (vector of vectors) of option parameters and receive a matrix of option prices as the result. Encapsulate this functionality in the most flexible/robust way you can think of.

Answers:

1.

Batch 1	Price	Delta	Gamma	Theta	Vega
Call	2.13337	0.372483	0.0420428	-8.42817	11.3515
Put	5.84628	-0.627517	0.0420428	-3.33114	11.3515

Batch 2	Price	Delta	Gamma	Theta	Vega
Call	7.96557	0.539828	0.0198476	-3.96953	39.6953
Put	7.96557	-0.460172	0.0198476	-3.96953	39.6953

Batch 3	Price	Delta	Gamma	Theta	Vega
Call	0.204058	0.185048	0.106789	-0.420257	1.33486
Put	4.07326	-0.814952	0.106789	0.644047	1.33486

Batch 4	Price	Delta	Gamma	Theta	Vega
Call	92.1757	0.988761	0.000179578	-0.616838	16.162
Put	1.2475	-0.0112394	0.000179578	0.108905	16.162

Figure 1: Black-Scholes Exact Pricing and Greeks for Call and Put, Batches 1 - 4

2. In this exercise, I specified three functions pertaining to put-call parity.

The **first** function, PutCallParity() calculates the price of the opposite option "type". As such, if the option at hand is a call, it will compute the put price under put-call parity conditions. The same applies with the reverse and having a put option first.

The **second** function, Parity_Difference() calculates the difference in value between the opposite option "type"'s price under put-call parity and that of the exact price derived from the Black-Scholes formula.

Finally, the **third** function Parity_checker() uses the output from Parity_Difference() and prints out a string as to whether there is put-call parity or not. While I could have simply integrated Parity_Difference() into Parity_checker(), I thought it would be interesting to have access to the differences themselves, as well as the final answer. This is, in my opinion, a matter of personal choice rather than of design.

Given recommendations of Avi Palley on 12/23/2021, a rounding of tolerance error of 1e-8 is sufficient in order to determine put-call parity. Without such a condition, none of the four batches respect precisely put-call parity conditions. If the rounding error condition applies, all four batches respect put-call parity conditions.

3. In the following two graphs, I output option prices for a monotonically increasing range of underlying values of S (from 10 to 50, with mesh size 1).

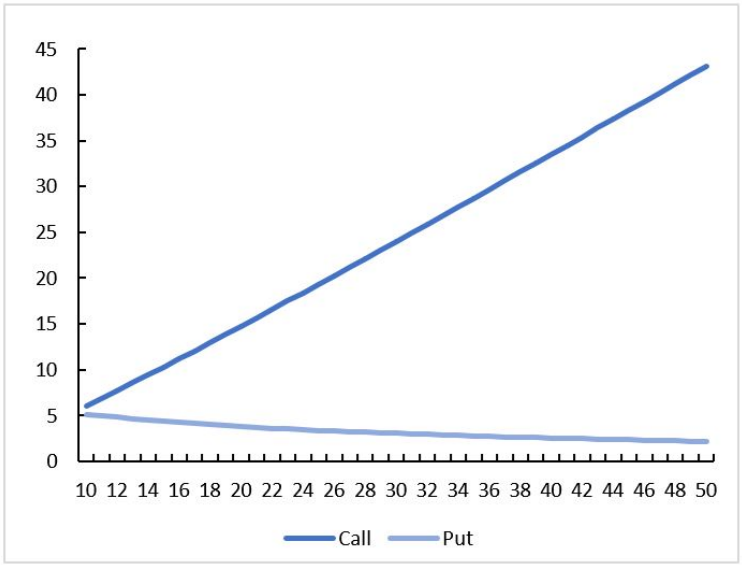


Figure 2: Option prices for a monotonically increasing range of values of **S**, Batch 1

The exact outputs for both of the call and put prices can be found by running my code.

4.

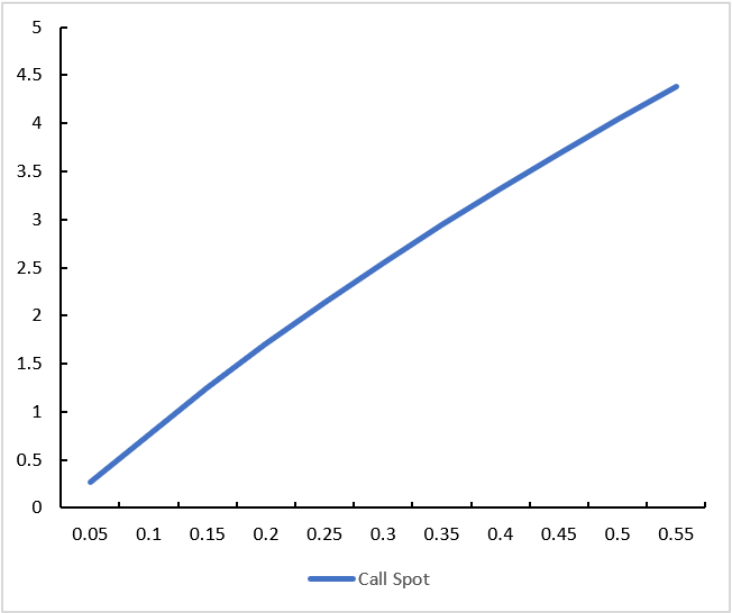


Figure 3: Call spot option prices for a monotonically increasing range of values of **T**, Batch 1

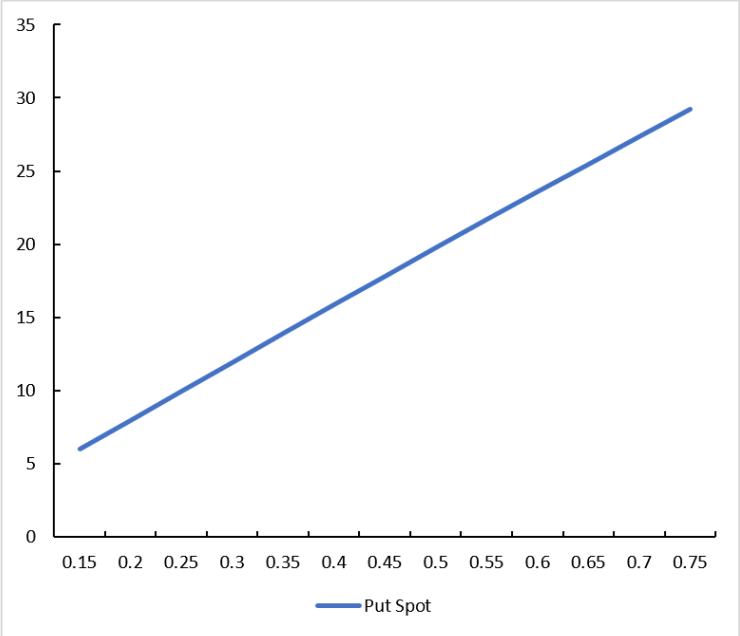


Figure 4: Put spot option prices for a monotonically increasing range of values of **Sig**, Batch 2

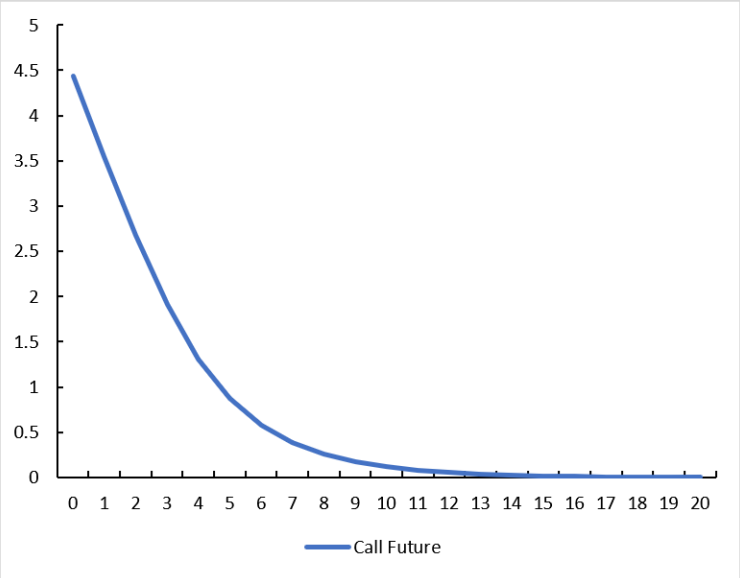


Figure 5: Call future option prices for a monotonically increasing range of values of **K**, Batch 3

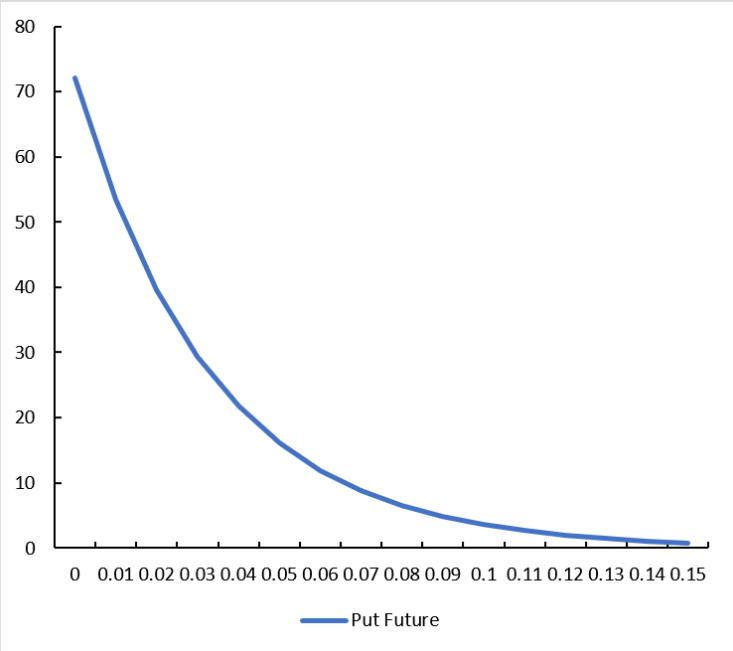


Figure 6: Put future option prices for a monotonically increasing range of values of **R**, Batch 4

II) Option Sensitivities, or "the Greeks:

1. Implement the above formulae, for gamma, for call and put future option pricing using the data set: **K=100.0, S=105, T=0.5, r=0.1, b=0.0, and sig=0.36 (exact delta call = 0.5946, delta put = -0.3566)**

2. We now use the code in part a to compute call delta price for a monotonically increasing range of underlying values of S, for example 10, 11, 12, . . . , 50. To this end, the output will be a vector and it entails calling the above formula for a call delta for each value S and each computed option price will be store in a **std::vector<double>** object. It will be useful to reuse the above global function that produces a mesh array of double separated by a mesh size h.

3. Incorporate this into your above **matrix pricer code**, so you can input a matrix of option parameters and receive a matrix of either Delta or Gamma as the result.

4. We now use divided differences to approximate option sensitivities. In some cases, an exact formula may not exist (or is difficult to find) and we resort to numerical methods. In general, we can approximate first and second-order derivatives in S by 3-point second order approximations, for example:

$$\Delta = \frac{V + (S + h) - V(S - h)}{2h} \tag{11}$$

$$\Gamma = \frac{V(S + h) - 2V(S) + V(S - h)}{h^2} \tag{12}$$

In this case the parameter h is ‘small’ in some sense. By Taylor’s expansion you can show that the above approximations are second order accurate in h to the corresponding derivatives.

The objective of this part is to perform the same calculations as in parts a and b, but now using divided differences. Compare the accuracy with various values of the parameter h (In general, smaller values of h produce better approximations but we need to avoid round-offer errors and subtraction of quantities that are very close to each other). Incorporate this into your well-designed class structure.

Answers:

Part 2:

1.

Batch A. 2	Price	Delta	Gamma	Theta	Vega
Call Future	12.4328	0.594629	0.0134936	-8.39684	26.7781
Put Future	7.6767	-0.356601	0.0134936	-8.87245	26.7781

S:	105.00	T:	0.5	Sig:	0.36
K:	100.00	R:	0.1	B:	0.00

Figure 7: Call and Put future option prices for batch data given in part 2

2. and 3.

In Group B's exercises b and c, I included in the vector part directly into my matrix functionalities. Indeed, storing the results from computations into a vector and printing these out for a monotonically increasing range of a parameter is what I have precisely defined in my matrix definitions. Below, I have thus computed call and put deltas for an increasing range of values of stock underlying price S (70 to 120, with one point increments), as well as for gammas (identical for calls and puts).

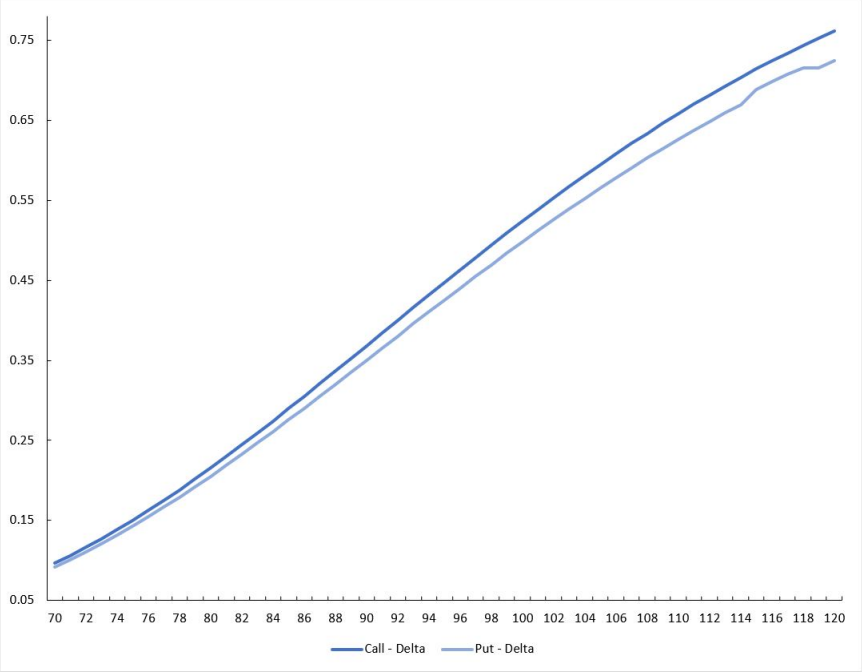


Figure 8: Call and Put future option deltas increasing monotonically from 70 to 120 with one point increments/mesh size

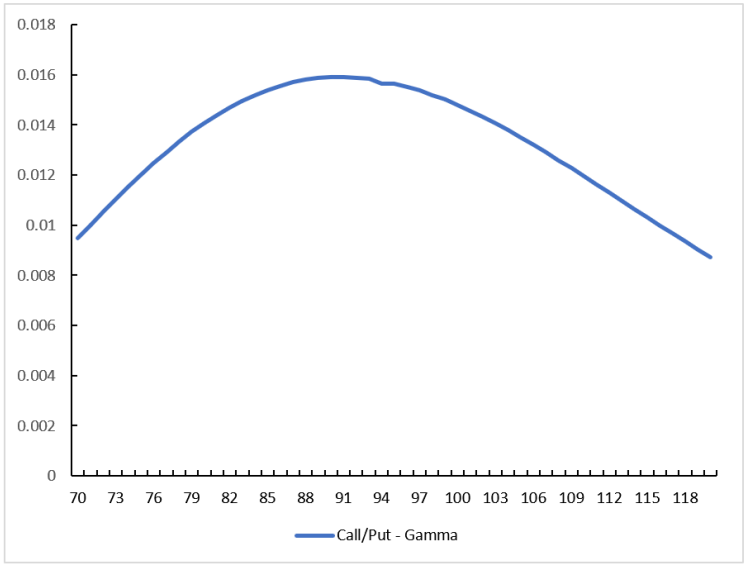


Figure 9: Call and Put future option gammas increasing monotonically from 70 to 120 with one point increments/mesh size

4.

In this subexercise, we reproduce the same calculations as in parts 1 and 2 , but now using divided differences.

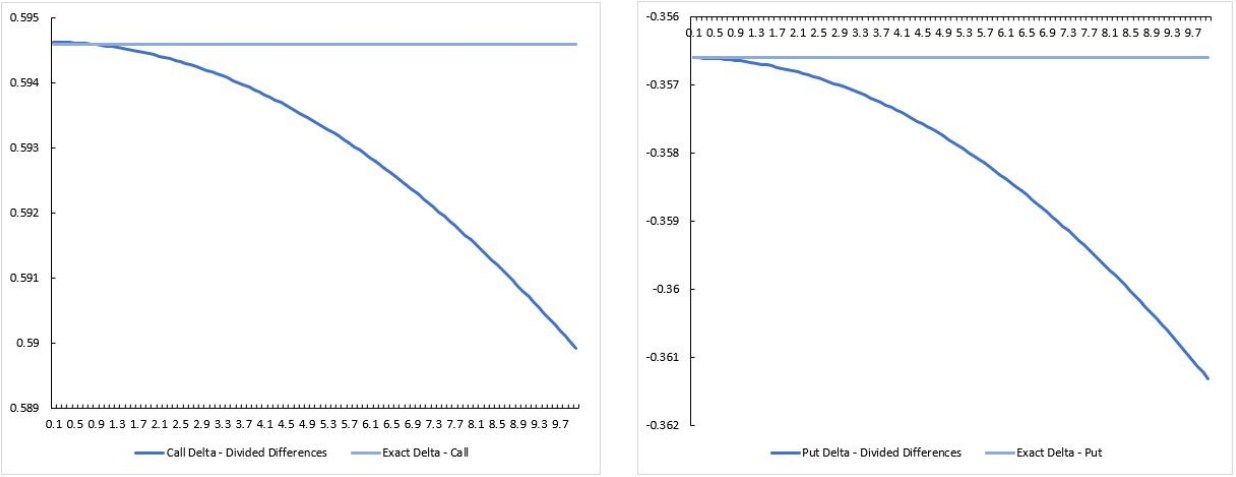


Figure 10: Call and Put future option deltas with increasing values of parameter h for divided differences approximation.

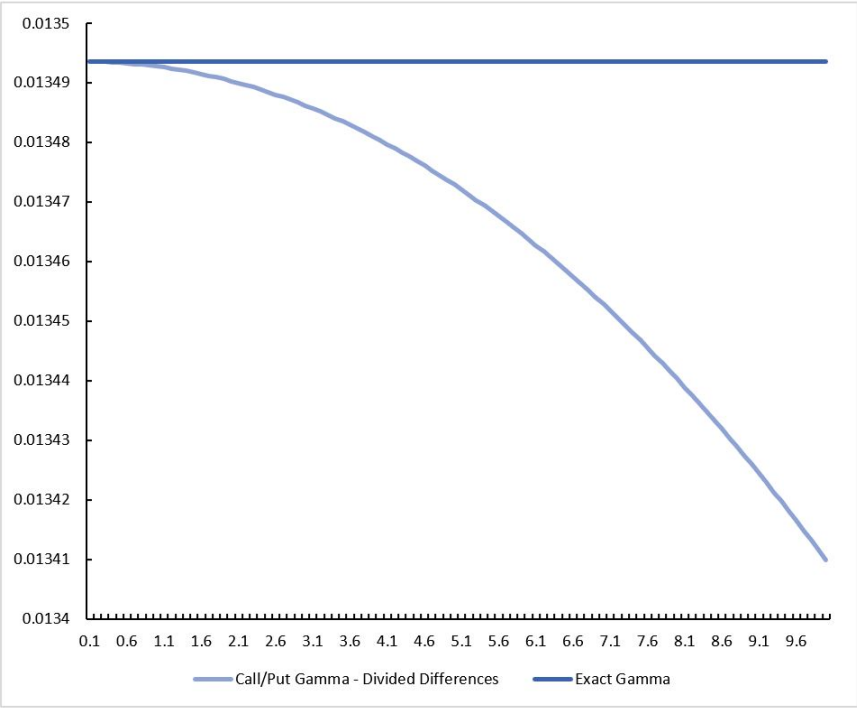


Figure 11: Call and Put future option gammas with increasing values of parameter h for divided differences approximation.

One can quite effortlessly see that the smaller the value of h, and as it tends towards 0, the smaller the difference between the finite difference approximation and its exact computation. This is because, as h approaches 0, the difference between the function and its finite difference approximations becomes arbitrarily small, resuling in a more accurate approximation of the derivatives. There is a tradeoff however, as more computations are needed to approximate the derivative: this is due notably to the denominators in both formulas, in which a decreasing value of h leads to either a linear (2h) or quadratic (h^2) 'complexification' of computations.

1.3 Group B: PERPETUAL AMERICAN OPTIONS

A European option can only be exercised at the expiry date T and an exact solution is known. An American option is a contract that can be exercised at any time prior to T. Most traded stock options are American style. In general, there is no known exact solution to price an American option but there is one exception, namely perpetual American options. The formulae are:

$$C = \frac{K}{y_1 - 1}\left(\frac{y_1 - 1}{y_1}\frac{S}{K}\right)^{y_1} \tag{13}$$

$$y_1 = \frac{1}{2} - \frac{b}{\sigma^2} + \sqrt{\left(\frac{b}{\sigma^2} - \frac{1}{2}\right)^2 + \frac{2r}{\sigma^2}} \tag{14}$$

for call options, and:

$$P = \frac{K}{1 - y_2}\left(\frac{y_2 - 1}{y_2}\frac{S}{K}\right)^{y_2} \tag{15}$$

$$y_1 = \frac{1}{2} - \frac{b}{\sigma^2} - \sqrt{\left(\frac{b}{\sigma^2} - \frac{1}{2}\right)^2 + \frac{2r}{\sigma^2}} \tag{16}$$

for put options.

In general, the perpetual price is the time-homogeneous price and is the same as the normal price when the expiry price T tends to infinity. In general, American options are worth more than European options.

Answer the following questions

I) Exact Solutions of One-Factor Plain Options

1. Program the above formulae, and incorporate into your well-designed options pricing classes.

2. Test the data with K=100, sig=0.1, r=0.1, b=0.02, S=110 (check C=18.5035, P=3.03106

3. We now use the code in part a) to compute call and put option price for a monotonically increasing range of underlying values of S, for example 10, 11, 12, . . . , 50. To this end, the output will be a vector and this exercise entails calling the option pricing formulae in part a) for each value S and each computed option price will be stored in a std::vector<double> object. It will be useful to reuse the above global function that produces a mesh array of double separated by a mesh size h.

4. Incorporate this into your above **matrix pricer** code, so you can input a matrix of option parameters and receive a matrix of Perpetual American option prices.

Answers:

1. Results can be found in my code output, but also in the graph below, which comprises instance where S=110.

2.

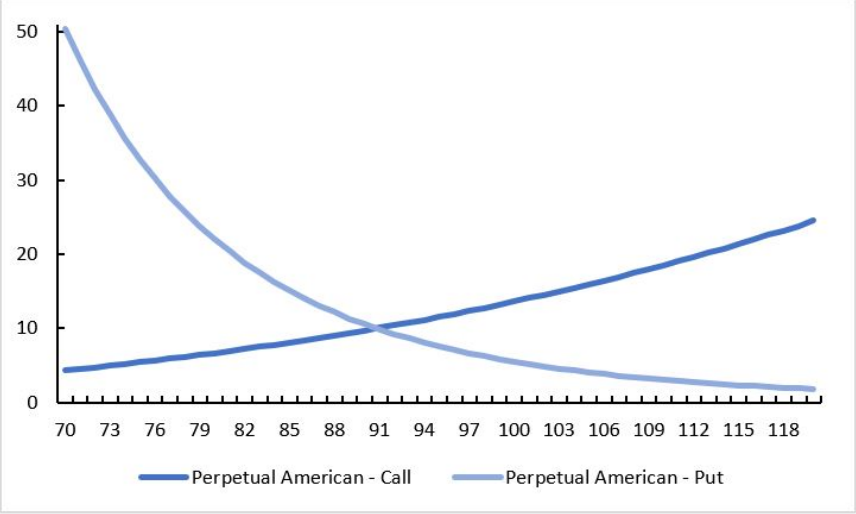


Figure 12: Call and Put Perpetual American option computations, with monotonically increasing values of S.

2 Groups C and D: Monte Carlo Pricing Methods

2.1 Group C: MONTE CARLO 101

The document should contain a detailed, complete analysis and will be graded on how well it demonstrates understanding of the accuracy and efficiency of Monte Carlo methods in the below context.

We focus on a linear, constant-coefficient, scalar (one-factor) problem. In particular, we examine the case of a one-factor European call option using the assumptions of the original Black Scholes equation. We give an overview of the process.

At the expiry date t = T the option price is known as a function of the current stock price and the strike price. The essence of the Monte Carlo method is that we carry out a simulation experiment by finding the solution of a stochastic differential equation (SDE) from time t = 0 to time t = T. This process allows us to compute the stock price at t = T and then the option price using the payoff function. We carry out M simulations or draws by finding the solution of the SDE and we calculate the option price at t = T . Finally, we calculate the discounted average of the simulated payoff and we are done.

Summarizing the process is:

1. Construct a simulated path of the underlying stock.
2. Calculate the stock price at t=T.
3. Calculate the call price at t=T (use the **payoff** function).

   Execute the steps 1-3M times.
4. Calculate the averaged call price at t=T.
5. Discount the price found in step 4 to t=0.

The first step is to replace continuous time by discrete time. To this end, we divide the interval [0, T] (where T is the expiry date) into a number of subintervals as shown in Figure 1. We define N + 1 mesh points as follows:

$$0 = t_0 < t_1 < ... < t_{n+1} < ... < t_N \equiv T \tag{17}$$

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

In general, we speak of a non-uniform mesh when the sizes of the subintervals are not necessarily the same. However, in this book we consider a class of finite difference schemes where the N subintervals have the same size (we then speak of a uniform mesh), namely ¢t = T=N.

Having defined how to subdivide [0,T] into subintervals, we are now ready to motivate our finite difference schemes; for the moment we examine the scalar linear SDE with constant coefficients:

$$dX = aXdt + bXdW, \quad a, b \ constant \tag{18}$$

Condition X(0) = A applies. Regarding notation, we do not show the dependence of variable X on t, and we wish to show this dependence we prefer to write X = X(t) instead of the form X_t.

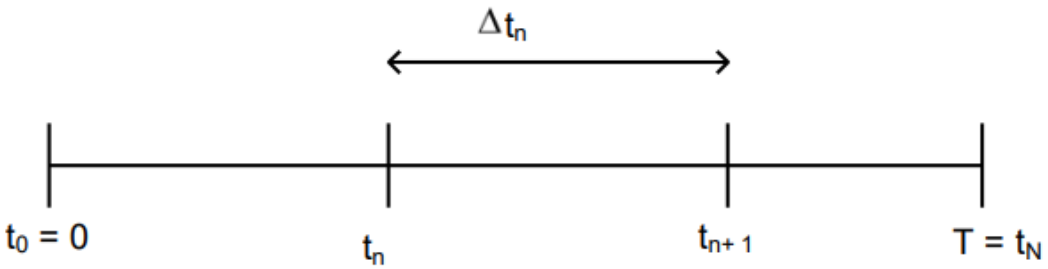


Figure 13: Mesh generation

We write equation (6) as a stochastic integral equation between the (arbitrary) times s and t as follows:

$$X(t) = X(s) + \int_s^t aX(y)dy + \int_s^t bX(y)dW(y),\ s < t \tag{19}$$

We evaluate equation (7) at two consecutive mesh points and using the fact that the factors a and b in equation (6) are constant we get the exact identity:

$$\begin{aligned} X(t_{n+1}) &= X(t_n) + \int_{t_n}^{t_{n+1}} aX(y)dy + \int_{t_n}^{t_{n+1}} bX(y)dW(y) \\ &= X(t_n) + a\int_{t_n}^{t_{n+1}} X(y)dy + b\int_{t_n}^{t_{n+1}} X(y)dW(y) \end{aligned} \tag{20}$$

We now approximate equation (8) and in this case we replace the solution X of equation (8) by a new discrete function Y (that is, one that is defined at mesh points) by assuming that it is constant on each subinterval; we then arrive at the discrete equation:

$$\begin{aligned} Y_{n+1} &= Y_n + aY_n\int_{t_n}^{t_{n+1}} dy + bY_n\int_{t_n}^{t_{n+1}} dW(y) \\ &= Y_n + aY_n\Delta t_n + bY_n\Delta W_n \\ where&: \\ \Delta W_n &= W(t_{n+1}) - W(t_n),\ 0 \le n \le N-1 \end{aligned} \tag{21}$$

This is called the (explicit) **Euler-Maruyama scheme** and it is a popular method when approximating the solution of SDEs. In some cases we write the solution in terms of a discrete function Xn if there is no confusion between it and the solution of equations (6) or (7). In other words, we can write the discrete equation (9) in the equivalent form:

$$\begin{cases} X_{n+1} &= X_n + aX_n\Delta t_n + bX_n\Delta W_n \\ X_0 &= A \end{cases} \tag{22}$$

In many examples the mesh size is constant and furthermore the Wiener increments are well-known computable quantities:

$$\begin{cases} \Delta t_n = \Delta t = \frac{T}{N}, & 0 \le n \le N-1 \\ \Delta W_n = \sqrt{\Delta t}z_n, & z_n \sim N(0,1) \end{cases} \tag{23}$$

Incidentally, we generate increments of the Wiener process by a random number generator for independent Gaussian pseudo-random numbers, for example **Box-Muller**, **Polar Marsaglia**, **Mersenne Twister** or **lagged Fibonacci** generator methods. In this book, we focus exclusively on the generators in Boost Random.

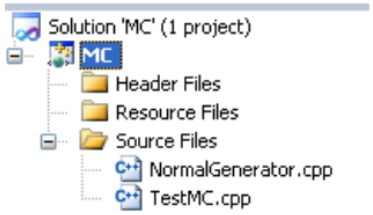
We deal almost exclusively with one-step marching schemes; in other words, the unknown solution at time level n + 1 is calculated in terms of known values at time level n.

Answer the following questions

I) Monte Carlo 101

1. Study the source code in the file TestMC.cpp and relate it to the theory that we have just discussed. The project should contain the following source files and you need to set project settings in VS to point to the

correct header files:



Compile and run the program **as is** and make sure there are no errors.

2. Run the MC program again with data from Batches 1 and 2. Experiment with different value of NT (time steps) and NSIM (simulations or draws). In particular, how many time steps and draws do you need in order to get the same accuracy as the exact solution? How is the accuracy affected by different values for NT/NSIM?

3. Now we do some stress-testing of the MC method. Take Batch 4. What values do we need to assign to NT and NSIM in order to get an accuracy to two places behind the decimal point? How is the accuracy affected by different values for NT/NSIM?

Answers:

1. The setting up of the code was different from that encountered in Visual Studio, and used by most people on QuantNet. When using Visual Studio Code, path inclusions are different, and I have thus contributed to the QuantNet forum by explaining how to set up the Boost library, include such path into tasks.json, and use the Boost libraries within our relevant source and header files.

With these modifications, the code is up and running.

In this exercise, we focus on a linear (scalar linear stochastic differential equation of factor one) and constant-coefficient (constant coefficients a and b) problem.

We examine the case of a one-factor European call option using the assumptions of the original Black-Scholes equation.

Starting on line 65, we create an instance of an option (OptionData myOption) using the OptionData header file. We provide necessary variables for strike (K), underlying asset price (S), expiry date (T), constant volatility parameter (Sig), and an option type (1 for "Call", -1 for "Put"). OptionData encapsulates all data in one place.

The first essence of our exercise here is to discretize our domain [0,T] , from t=0 to the time of expiry T of the security's option. We proceed by defining N+1 mesh points of uniform size (uniform mesh). This explains the equation: $\Delta t_n \equiv t_{n+1} - t_n,\ 0 \leq n \leq N-1$, where the difference (Delta) between two mesh points anywhere in our domain is of the same size. In effect, we can, therefore, discretize our domain uniformly as such: $\Delta t = \frac{T}{N}$.

In our code, this is done by calling the mesh() function (defined in Utilities/Geometry/Range.hpp), which by definition in this particular instance, returns a uniform mesh. Mesh size (h) is computed by taking the upper bound (hi) and lower bound (lo) of our domain, taking the size between these two values (hi-lo) and dividing by the number of steps (nSteps) we wish to create. A vector named 'result' is created with size (nSteps +1). This make sense.

Assume we have a domain from t= 0 to t=4. From this, 3 mesh points are created, with uniform separation, and time goes from t=0 to t=4. In this particular instance, and applicable in our computation too, we create a vector of size 4 (3 nSteps + 1), and iterate from t=0 to t=4 by adding to each space in the vector the position of the mesh point created.

Line 87, we are prompted to specify the number of simulations we wish to create. The whole point of Monte Carlo simulations is to construct simulated paths of the underlying asset, compute the call price at t= T by using the payoff() function, and discount back to time t=0. Such is the purpose of derivatives pricing, but more generally, of any time-discounting models and expected values. A mathematical process must be defined to determine how these

simulated paths are conducted.

A mathematical process must be defined to determine how these simulated paths are conducted.

For the simulation of a Wiener process, we use standardized normally distributed numbers Z $\sim N(0,1)$ in order to compute the increments $\Delta W_n = \sqrt{\Delta t} z_n$. Mathematically, a **Wiener process** (often called **Brownian motion**) on interval [0,T] is a random variable W(t) that depends continuously on t $\in$ [0,T] and satisfies the following:

$$\begin{aligned} &W(0) = 0, \\ &\text{For } 0 \leq s < t \leq T, \;\; W(t) - W(s) \sim \sqrt{t-s} N(0,1), \\ &\text{where N(0,1) is a normal distribution with zero meant and unit variance.} \end{aligned} \tag{24}$$

Different pseudo-random number generators exist: Box-Muller, Polar Marsaglia, Mersenne Twister, or lagged Fibonacci. Here, we use lagged Fibonacci as a number generator for independent Gaussian **pseudo-random** numbers to generate increments of the Wiener process.

The lagged Fibonacci numbers represent a generalization of the 'classical' Fibonacci numbers and the associated recursion formula.

Once we have determined the number of simulations we wish to have, we use a for-loop to calculate a path for each simulation, starting at line 108. Within this for-loop, we call another for-loop, which calls for a randomly generated number at each mesh point within this specific simulation. Taking a pointer to a dynamically allocated instance with default constructor BoostNormal() (NormalGenerator $*$ myNormal = new BoostNormal();), we in effect call a variate generator using boost libraries lagged_fibonacci607 and normal_distribution (in Utilities/RNG/NormalGenerator.hpp). We create a random number by calling getNormal() line 122 within the for-loop.

From our initial stochastic differential equation:

$$\begin{aligned} &dX = aXdt + bXdW, \;\; a, b \; constant \\ &X(0) = A \end{aligned} \tag{25}$$

we write the equation as a stochastic integral equation between arbitrary times s and t, as seen in equation (19). Ensues equation (20) and then derive equation (21) by approximation with the **Euler-Maruyama scheme**.

In the second for-loop (iterating from one mesh point to the next up until t=T) starting on line 118, we have the following variables distinctly found in equation (22): **VNew** is X(n+1), **VOld** is X(n), **k** is the size between two mesh points (or $\Delta t_n = \Delta t = \frac{T}{N} here$, **sqrk** is $\sqrt{\Delta t}$, **dW** is the Wiener increment using myNormal pointer to call getNormal() in order to generate a pseudo-random number, **diffusion(x[index-1])** is bX(n) and **drift(x[index-1])** is aX(n). dW has mean 0 and variance 1 in our particular case. In sum, **drift coefficient** 'a' plays an important role for the study of the first moment of X(t) whereas the diffusion coefficient will play an important role for the study of the second moment of X(t).Further intuition of the mathematics can be found in the literature, but the focus here is the computing.

When the for-loop is completed, we have in effect completed one simulation. VNew is equals to the simulated price of the underlying asset at t=T. From there, we calculate the payoff by using the myPayOffFunction() function which calculates the difference between the price of the simulated price of the underlying asset and the strike price and returns the positive difference, be it for a call or a put. From there, we add the positive difference to the variable price, and average out with respect to the number of simulations that have been conducted.

We then iterate through the NSim number of simulations we have determined, and complete the same process. By the end of the number of NSim simulations, we will have obtained an average price for a call of a put, and we then discount the average price found to t=0 by exponentiating by the factor (-RT).

A graphical representation of the Monte Carlo process can be found below.

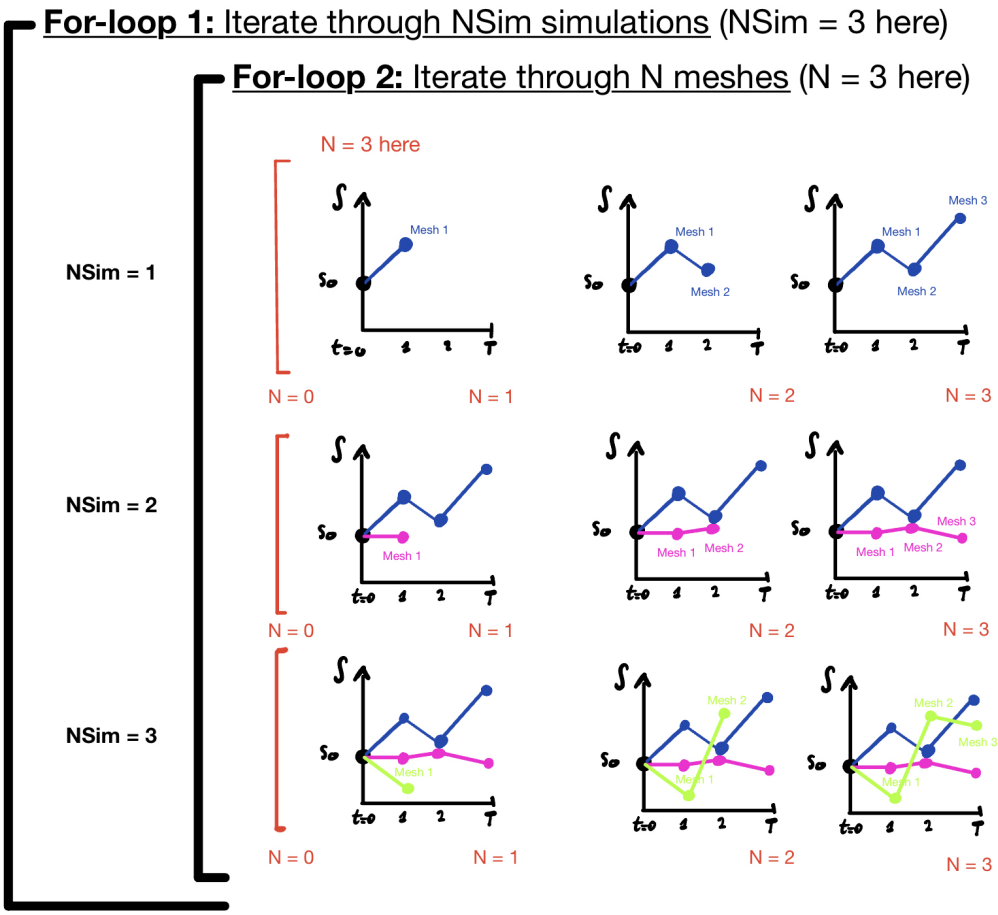


Figure 14: A simplified Monte Carlo simulation drawing

---

2. For recall, here are the data for batches 1 and 2:

Batch 1: T=0.25, K=65, Sig=0.30, R=0.08, S=60, (then C= 2.13337, P= 5.84628)
Batch 2: T=1.0, K=100, Sig=0.0, R=0.0, S=100 (then C= 7.96557, P= 7.96557)

The **exact solution** of Batch 1 is C = 2.13337 and P = 5.84628.
The **exact solution** of Batch 2 is C = 7.96557 and P = 7.96557.

Time Steps (NT)	NSIM - 10^2	NSIM - 10^3	NSIM - 10^4	NSIM - 10^5	NSIM - 1M	NSIM - 3M
10	C = 2.38525 P = 6.0245	C = 2.08001 P = 6.02809	C = 2.12297 P = 5.84177	C = 2.11117 P = 5.8456	C = 2.11674 P = 5.84074	C = 2.12031 P = 5.83088
100	C = 2.09492 P = 6.58781	C = 2.0741 P = 5.84851	C = 2.1378 P = 5.90807	C = 2.13043 P = 5.85125	C = 2.13271 P = 5.85125	C = 2.13181 P = 5.84931
250	C = 1.48732 P = 6.28052	C = 2.17636 P = 5.88371	C = 2.08923 P = 5.98463	C = 2.16636 P = 5.85382	C = 2.13136 P = 5.85559	C = 2.1329 P = 5.84031
500	C = 2.17558 P = 5.95189	C = 2.02328 P = 6.03298	C = 2.12707 P = 5.93754	C = 2.14863 P = 5.83818	C = 2.13071 P = 5.84125	C = 2.13232 P = 5.84109
1000	C = 2.36385 P = 6.30289	C = 2.10416 P = 6.07412	C = 2.12827 P = 5.87066	C = 2.13874 P = 5.87066	C = 2.13658 P = 5.83585	C = 2.13475 P = 5.83973

Figure 15: **Batch 1**: Call and Put prices

Time Steps (NT)	NSIM - 10^2	NSIM - 10^3	NSIM - 10^4	NSIM - 10^5	NSIM - 1M	NSIM - 3M
10	C = 8.4427 P = 8.10554	C = 7.86977 P = 8.36038	C = 7.98069 P = 7.98379	C = 7.9613 P = 8.0067	C = 7.96922 P = 7.99098	C = 7.98263 P = 7.97508
100	C = 7.77513 P = 9.44836	C = 7.73557 P = 7.88494	C = 7.94097 P = 8.06336	C = 7.94362 P = 8.0079	C = 7.9625 P = 7.97439	C = 7.96263 P = 7.97252
250	C = 6.32558 P = 8.82267	C = 8.08092 P = 8.05473	C = 7.79355 P = 8.19091	C = 8.0538 P = 7.99155	C = 7.95752 P = 7.98199	C = 7.96808 P = 7.95623
500	C = 7.98253 P = 8.10974	C = 7.62472 P = 8.28685	C = 7.93258 P = 8.13742	C = 8.01256 P = 7.95795	C = 7.96142 P = 7.95663	C = 7.96675 P = 7.95794
1000	C = 8.86798 P = 8.88433	C = 7.87164 P = 8.28691	C = 7.92226 P = 8.12311	C = 7.98184 P = 8.00674	C = 7.97529 P = 7.94982	C = 7.97317 P = 7.95572

Figure 16: **Batch 2**: Call and Put prices

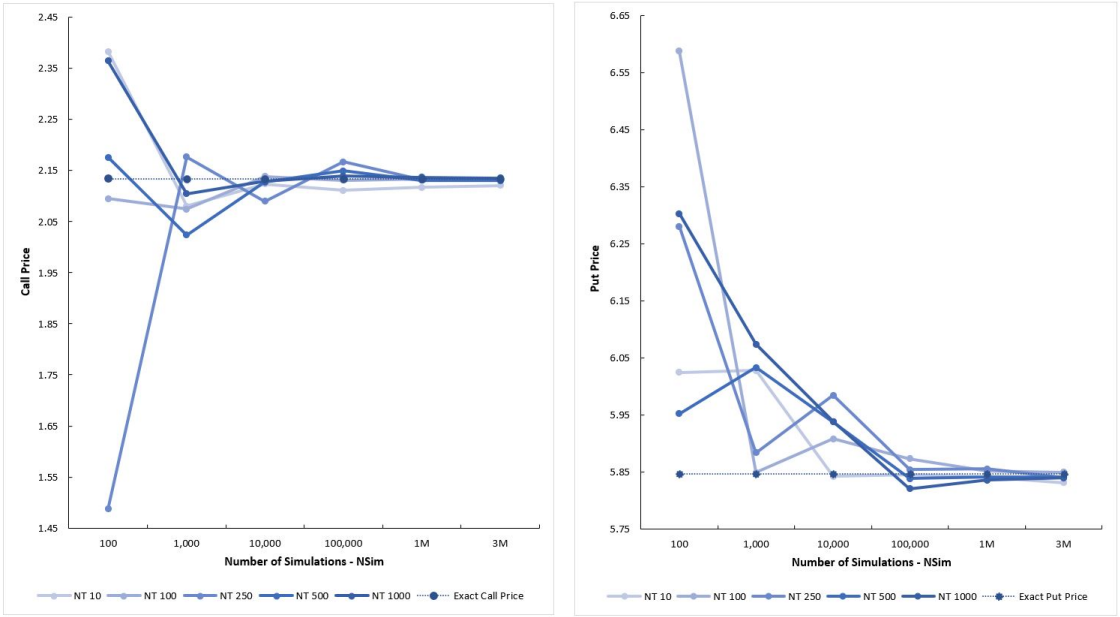


Figure 17: **Batch 1**: Call and Put prices w.r.t to the number of simulations

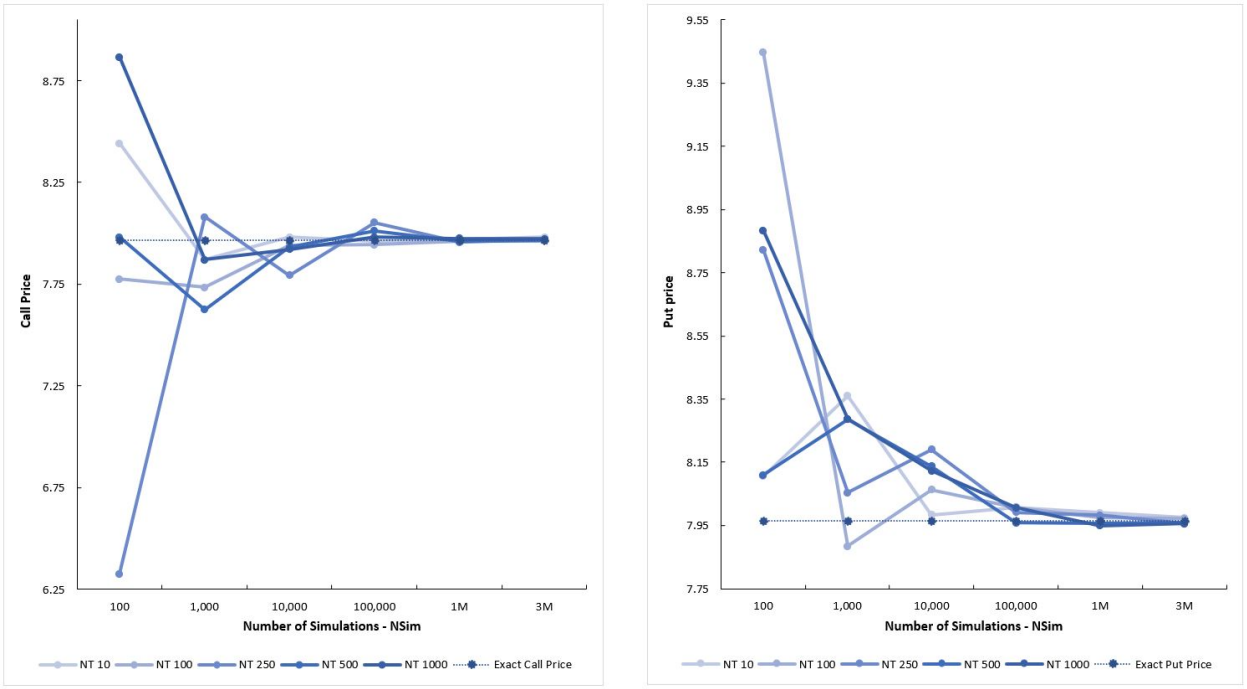


Figure 18: **Batch 2**: Call and Put prices w.r.t to the number of simulations

For the smallest number of number of simulations NSim (100), the call and put prices do not converge albeit the number of mesh points (Time steps NT) increasing from 10 up to 1000. This dynamic changes, however, when the number of simulations conducted increases ($10^3$3 up to 3M): both option prices asymptotically converge to their exact solution, both for batch 1 and batch 2. In simple words, and as can be seen in the outputted graphs for Batch 1 and 2, whilst increasing the number of simulations does have a significant impact on the convergence to the exact solution (NSim > 100,000), increasing the number of mesh points does not seem to have a significant impact on accuracy. For Batch 1 for example, for both call and put prices, having 250 mesh points and 100 simulations yields worst accuracy to that compared of 100 mesh points and 100 simulations.

The measure of accuracy is only mentioned in Group D. Thus, I cannot specify precisely when standard deviations and standard errors are minimized. I can, however, give an approximate range: conducting NSim >100,000 will give us good accuracy. There is not an undeniable linear relationship between NT/NSiM increases and decreases in error.

Comparing batches 1 and 2, we notice that although similar asymptotic properties are exhibited, time to expiry T seems to impact our analysis. I hypothesize that as T increases, the greater the number of simulations and mesh points need to be created.

3.

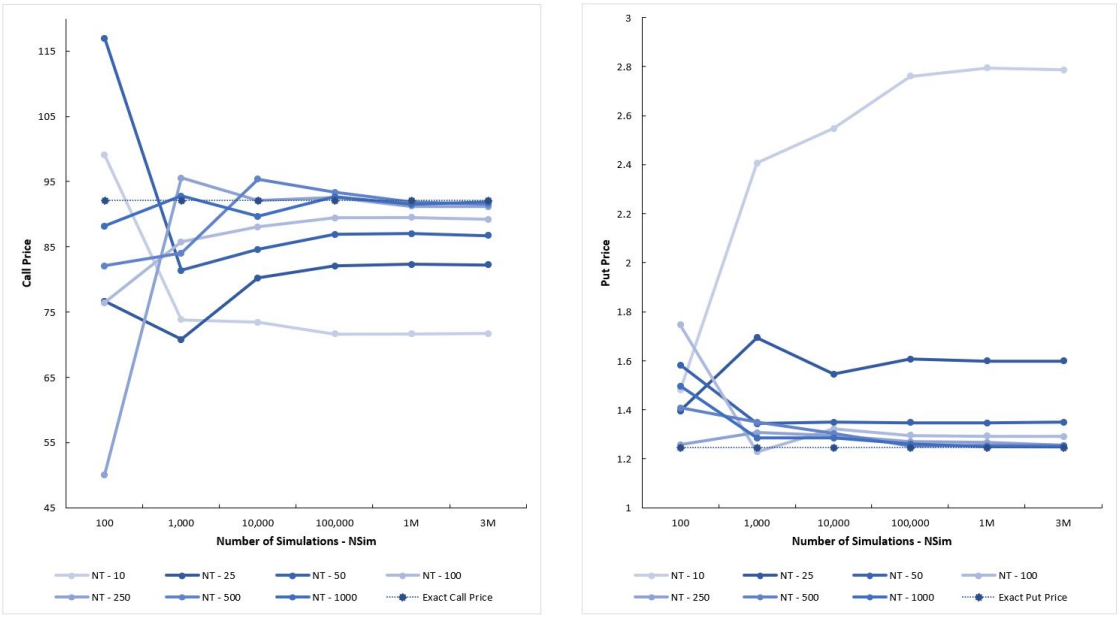


Figure 19: **Batch 4**: Call and Put prices w.r.t to the number of simulations

As suspected, the bigger the value of expiry date T, the harder it is to have a reasonable level of accuracy within 2 decimal points. In plain words, this can simply be explained by the fact that as the gap in time between t=0 and t=T increases, the larger the universe of events exist that can explain greater deviation in the underlying asset price. As such is the case, there is increasing difficulty forecasting for the true value of the call or put option. We can see this in the figure above, where we do not find the same convergence properties as with batches 1 and 2, where parameter T is much smaller.

While increasing the number of simulations to 10,000 seems to help, we have yet to find reasonable accuracy as long as we have a number of steps NT bigger than 50.

---

3 Group D: Advanced Monte Carlo

This section will build upon the provided Monte Carlo code from the previous section, by adding methods to track the accuracy of the MC simulation. You are expected to submit code in addition to a document with the answers to the below questions. The document should contain a detailed, complete analysis and will be graded on how well it presents understanding of the accuracy and efficiency of Monte Carlo methods in the below context. We wish to add functionality to the Monte Carlo pricer by providing estimates for the *standard deviation (SD)* and *standard error (SE)*, defined by:

$$SD = \sqrt{\frac{\Sigma C_{T,J}^2 - \frac{1}{M}(\Sigma C_{T,J})^2}{M-1}} * exp(-rT)$$
$$SE = \frac{SD}{\sqrt{M}} \tag{26}$$

where:

$C_{T,j}$ = call output price at t = T for the jth simulation, $i \leq j \leq M$,
M = number of simulations.

Implement this new functionality and test the software for a range of data for call and put options.

Answer the following questions

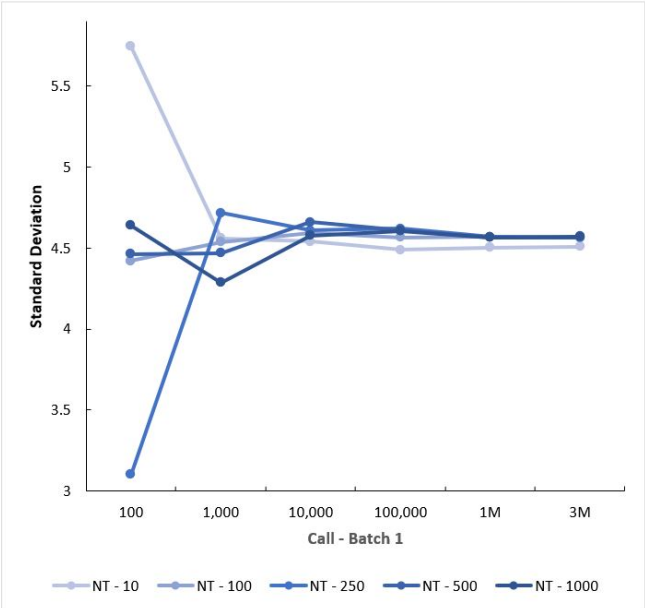
II) Advanced Monte Carlo

1. Create generic functions to compute the standard deviation and standard error based on the above formulae. The inputs are a vector of size M (M = NSIM), the interest-free rate and expiry time T. Integrate this new code into TestMC.cpp. Make sure that the code compiles.

2. Run the MC program again with data from Batches 1 and 2. Experiment with different values of NT (time steps) and NSIM (simulations or draws). How do SD and SE react for these different run parameters, and is there any pattern in regards to the accuracy of the MC (when compared to the exact method)?

Answers:

1.



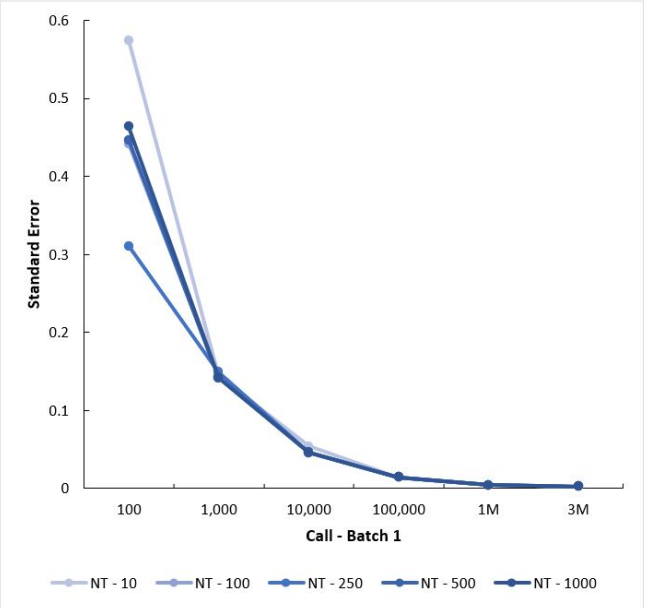


Figure 20: **Batch 1**: Call standard deviations and standard errors as a function of the number of simulations.

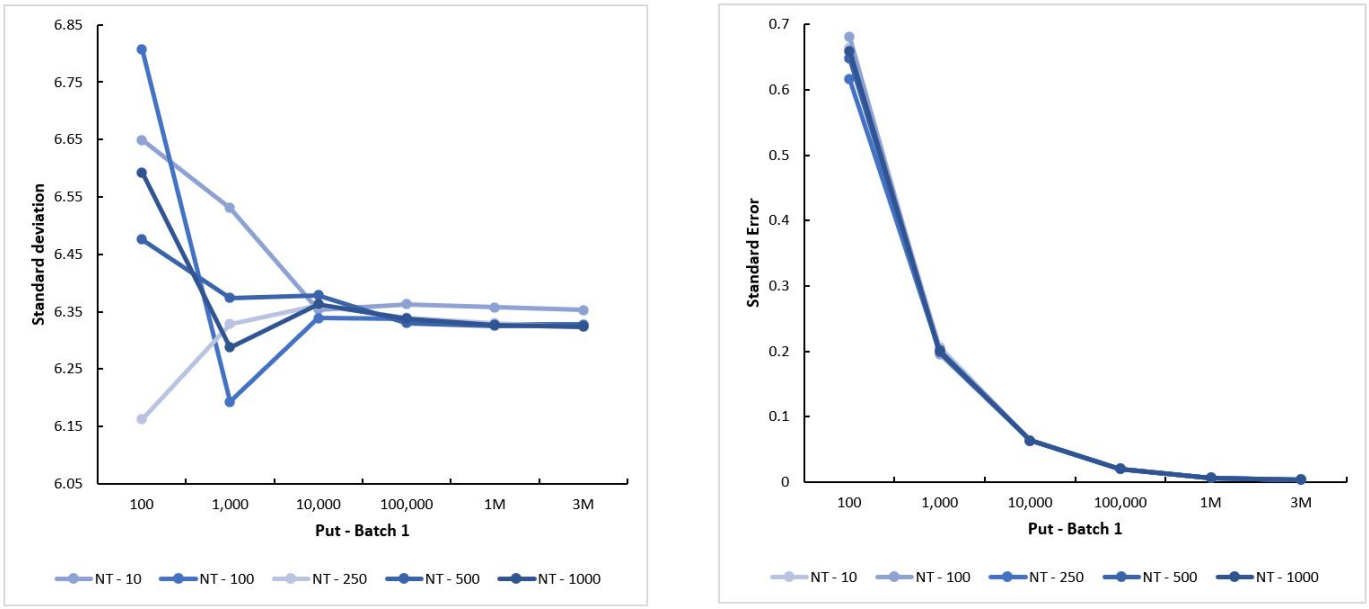


Figure 21: **Batch 1**: Put standard deviations and standard errors as a function of the number of simulations.

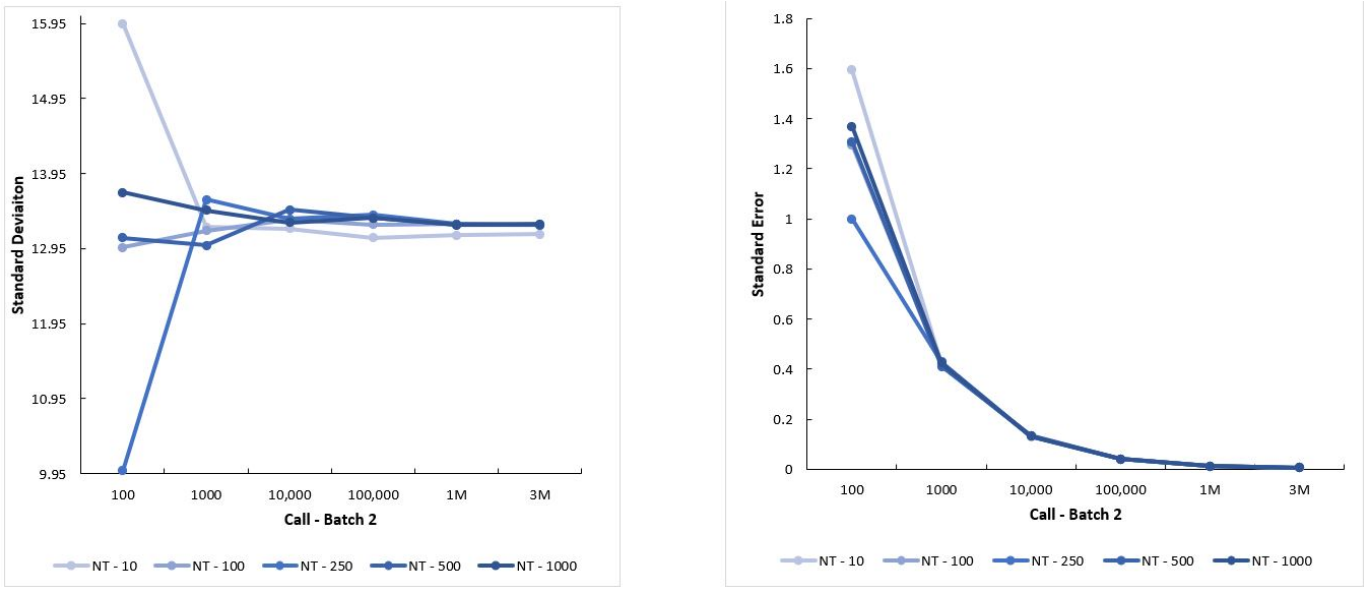
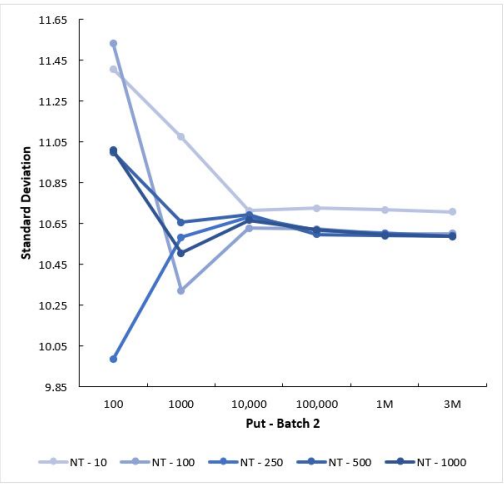


Figure 22: **Batch 2**: Call standard deviations and standard errors as a function of the number of simulations.



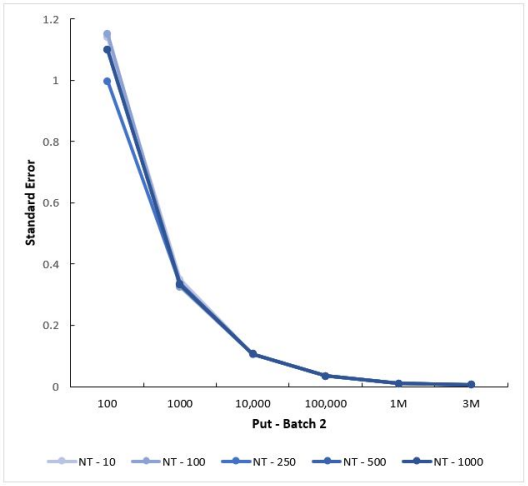


Figure 23: **Batch 2**: Put standard deviations and standard errors as a function of the number of simulations.

TimeSteps (NT)	NSIM - 10^2	NSIM - 10^3	NSIM - 10^4	NSIM - 10^5	NSIM - 1M	NSIM - 3M
10	Call_SD = 5.74235 Call_SE = 0.574235	Call_SD = 4.56004 Call_SE = 0.144201	Call_SD = 4.53966 Call_SE = 0.053966	Call_SD = 4.48792 Call_SE = 0.0141921	Call_SD = 4.50347 Call_SE = 0.00450347	Call_SD = 4.50733 Call_SE = 0.00260231
	Put_SD = 6.80743 Put_SE = 0.664961	Put_SD = 6.53114 Put_SE = 0.206533	Put_SD = 6.35409 Put_SE = 0.063540	Put_SD = 6.36351 Put_SE = 0.0201232	Put_SD = 6.35753 Put_SE = 0.00635753	Put_SD = 6.3531 Put_SE = 0.00366796
100	Call_SD = 4.42096 Call_SE = 0.442096	Call_SD = 4.53657 Call_SE = 0.143459	Call_SD = 4.59431 Call_SE = 0.0459431	Call_SD = 4.56541 Call_SE = 0.0146071	Call_SD = 4.56907 Call_SE = 0.00456907	Call_SD = 4.56341 Call_SE = 0.0026469
	Put_SD = 6.16271 Put_SE = 0.680743	Put_SD = 6.193 Put_SE = 0.19584	Put_SD = 6.35409 Put_SE = 0.0633922	Put_SD = 6.33739 Put_SE = 0.0200406	Put_SD = 6.32819 Put_SE = 0.006322819	Put_SD = 6.32818 Put_SE = 0.00365258
250	Call_SD = 3.10348 Call_SE = 0.310348	Call_SD = 4.71626 Call_SE = 0.149141	Call_SD = 4.60819 Call_SE = 0.0460819	Call_SD = 4.61918 Call_SE = 0.0146071	Call_SD = 4.56874 Call_SE = 0.00456874	Call_SD = 4.56622 Call_SE = 0.00263631
	Put_SD = 6.16271 Put_SE = 0.616271	Put_SD = 6.32851 Put_SE = 0.200125	Put_SD = 6.36157 Put_SE = 0.0633922	Put_SD =6.33983 Put_SE = 0.0200406	Put_SD = 6.33057 Put_SE = 0.00632819	Put_SD = 6.32403 Put_SE = 0.00365258
500	Call_SD = 4.46366 Call_SE = 0.446366	Call_SD = 4.46701 Call_SE = 0.141259	Call_SD = 4.65911 Call_SE = 0.0465911	Call_SD = 4.60659 Call_SE = 0.145673	Call_SD = 4.56594 Call_SE = 0.00456594	Call_SD = 4.56359 Call_SE = 0.00263479
	Put_SD = 6.47617 Put_SE = 0.647617	Put_SD = 6.37448 Put_SE = 0.201579	Put_SD = 6.37884 Put_SE = 0.0637884	Put_SD = 6.33016 Put_SE = 0.0200177	Put_SD = 6.32519 Put_SE = 0.00632519	Put_SD = 6.32606 Put_SE = 0.00365235
1000	Call_SD = 4.6389 Call_SE = 0.46389	Call_SD = 4.287 Call_SE = 0.14188	Call_SD = 4.57764 Call_SE = 0.0457764	Call_SD = 4.60492 Call_SE = 0.014562	Call_SD = 4.56492 Call_SE = 0.00456492	Call_SD = 4.57048 Call_SE = 0.00263877
	Put_SD = 6.59259 Put_SE =0.0659259	Put_SD = 6.28758 Put_SE = 0.198831	Put_SD = 6.36341 Put_SE = 0.0636341	Put_SD = 6.33814 Put_SE = 0.020043	Put_SD = 6.32676 Put_SE = 0.00632676	Put_SD = 6.32431 Put_SE = 0.00365134

Figure 24: **Batch 1**: Call and put standard deviations and standard errors as a function of the number of simulations. Numerical output.

TimeSteps (NT)	NSIM - 10^2	NSIM - 10^3	NSIM - 10^4	NSIM - 10^5	NSIM - 1M	NSIM - 3M
10	Call_SD = 15.9463 Call_SE = 1.59463	Call_SD = 13.2399 Call_SE = 0.418681	Call_SD = 13.2187 Call_SE = 0.132187	Call_SD = 13.0978 Call_SE = 0.0414189	Call_SD = 13.1331 Call_SE = 0.0131331	Call_SD = 13.1418 Call_SE = 0.00758742
	Put_SD = 11.4074 Put_SE = 1.14074	Put_SD = 11.0738 Put_SE = 0.350186	Put_SD = 10.7128 Put_SE = 0.107128	Put_SD = 10.7264 Put_SE = 0.03392	Put_SD = 10.7167 Put_SE = 0.0107167	Put_SD = 10.7069 Put_SE = 0.00618164
100	Call_SD = 12.9688 Call_SE = 1.29688	Call_SD = 13.192 Call_SE = 0.417169	Call_SD = 13.3356 Call_SE = 0.133356	Call_SD = 13.2733 Call_SE = 0.0419739	Call_SD = 13.2786 Call_SE = 0.0132786	Call_SD = 13.2652 Call_SE = 0.00765864
	Put_SD = 11.5338 Put_SE = 1.15338	Put_SD = 10.3224 Put_SE = 0.326424	Put_SD = 10.6268 Put_SE = 0.106268	Put_SD = 10.6247 Put_SE = 0.03392	Put_SD = 10.6018 Put_SE = 0.0106018	Put_SD = 10.6013 Put_SE = 0.00612066
250	Call_SD = 9.99293 Call_SE = 0.999293	Call_SD = 13.6073 Call_SE = 0.4303	Call_SD = 13.3493 Call_SE = 0.133493	Call_SD = 13.3991 Call_SE = 0.0423716	Call_SD = 13.2765 Call_SE = 0.0132765	Call_SD = 13.2704 Call_SE = 0.00766169
	Put_SD = 9.98476 Put_SE = 0.998476	Put_SD = 10.5831 Put_SE = 0.334666	Put_SD = 10.68 Put_SE = 0.1068	Put_SD = 10.6156 Put_SE = 0.03392	Put_SD = 10.6033 Put_SE = 0.0106033	Put_SD = 10.5893 Put_SE = 0.00611372
500	Call_SD = 13.093 Call_SE = 1.3093	Call_SD = 12.9957 Call_SE = 0.410959	Call_SD = 13.4667 Call_SE = 0.134667	Call_SD = 13.3622 Call_SE = 0.0422551	Call_SD = 13.2693 Call_SE = 0.0132693	Call_SD = 13.2646 Call_SE = 0.00765833
	Put_SD = 10.9997 Put_SE = 1.09997	Put_SD = 10.6574 Put_SE = 0.337015	Put_SD = 10.6934 Put_SE = 0.106934	Put_SD = 10.5966 Put_SE = 0.03392	Put_SD = 10.592 Put_SE = 0.010592	Put_SD = 10.5893 Put_SE = 0.00611372
1000	Call_SD = 13.6991 Call_SE = 1.36991	Call_SD = 13.4626 Call_SE = 0.425724	Call_SD = 13.2989 Call_SE = 0.132989	Call_SD = 13.3617 Call_SE = 0.0422533	Call_SD = 13.2672 Call_SE = 0.0132672	Call_SD = 13.2797 Call_SE = 0.00766702
	Put_SD = 11.0101 Put_SE = 1.10101	Put_SD = 10.5055 Put_SE = 0.332212	Put_SD = 10.6659 Put_SE = 0.106659	Put_SD = 10.6182 Put_SE = 0.03392	Put_SD = 10.5938 Put_SE = 0.0105938	Put_SD = 10.5872 Put_SE = 0.0061125

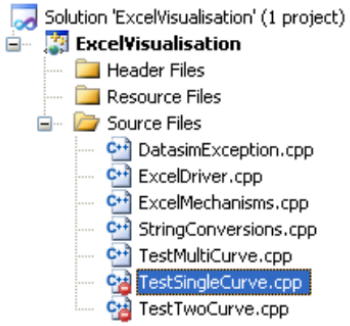
Figure 25: **Batch 2**: Call and put standard deviations and standard errors as a function of the number of simulations. Numerical output.

2. One can see that standard deviations converge for both batches 1 and 2, while standard errors converge to a steady-state value of 0 as the number of simulations grow. This ties back into the analysis given in Group C. However, and explained in Group C, I would expect for the standard errors not to converge toward 0, independent of the number of the number of simulations and the number of steps increasing.

---

4 Group E: Excel Visualization

Submission should consist of the working code and example Excel output files. We have developed a small package that allows us to display the output from the above schemes in Excel spreadsheets. We have provided ready-made test programs that you can customize to suit your needs. Make sure that you have the first FOUR files and ONE active test program in your project:



Answer the following questions I) Excel Visualization

1. Compile and run the sample programs TestSingleCurve, TestTwoCurve and TestMultipleCurve. Make sure that everything compiles and that you get Excel output.

2. We now wish to compute option price for a monotonically increasing range of underlying values of S, for example 10, 11, 12, …, 50. To this end, the output will be a vector and this exercise entails calling the exact option pricing formulae) for each value S and each computed option price will be stored in a std::vector<double> object. It will be useful to write a global function that produces a mesh array of double separated by a mesh size h. Print the output in Excel.

Answers:
1.

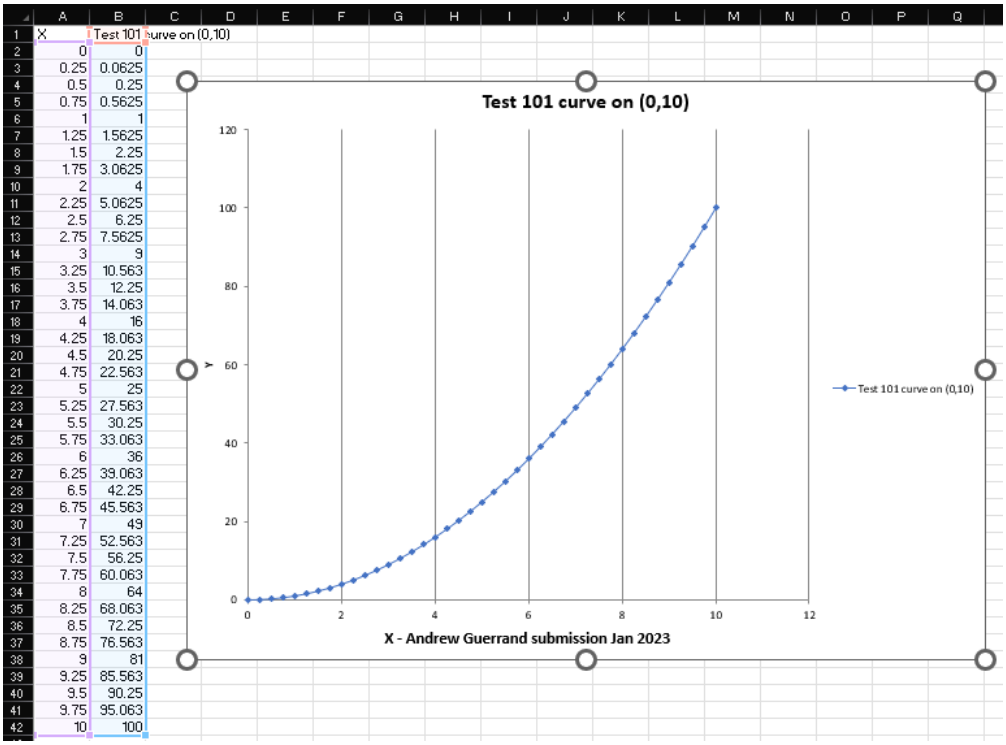
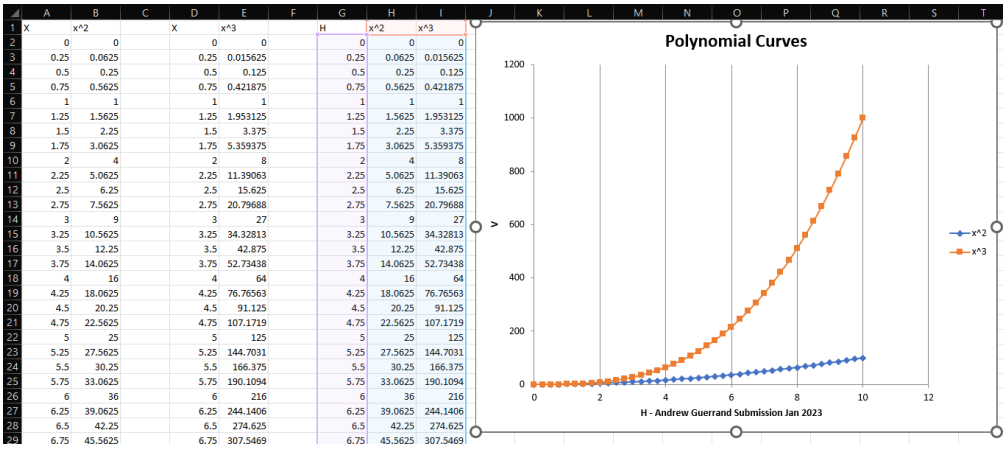
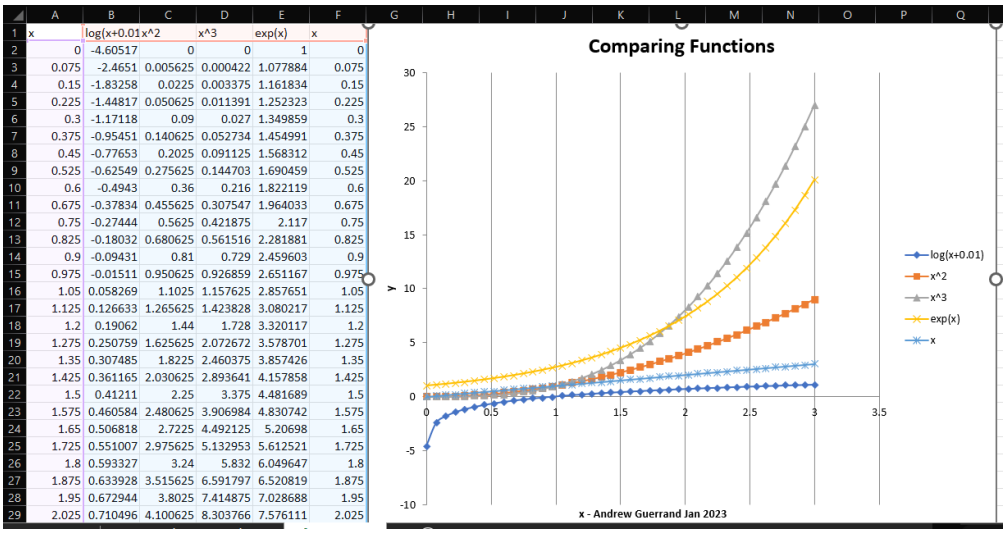


Figure 26: Output for TestSingleCurve.cpp



X	x^2		X	x^3		H	x^2	x^3
0	0		0	0		0	0	0
0.25	0.0625		0.25	0.015625		0.25	0.0625	0.015625
0.5	0.25		0.5	0.125		0.5	0.25	0.125
0.75	0.5625		0.75	0.421875		0.75	0.5625	0.421875
1	1		1	1		1	1	1
1.25	1.5625		1.25	1.953125		1.25	1.5625	1.953125
1.5	2.25		1.5	3.375		1.5	2.25	3.375
1.75	3.0625		1.75	5.359375		1.75	3.0625	5.359375
2	4		2	8		2	4	8
2.25	5.0625		2.25	11.39063		2.25	5.0625	11.39063
2.5	6.25		2.5	15.625		2.5	6.25	15.625
2.75	7.5625		2.75	20.79688		2.75	7.5625	20.79688
3	9		3	27		3	9	27
3.25	10.5625		3.25	34.32813		3.25	10.5625	34.32813
3.5	12.25		3.5	42.875		3.5	12.25	42.875
3.75	14.0625		3.75	52.73438		3.75	14.0625	52.73438
4	16		4	64		4	16	64
4.25	18.0625		4.25	76.76563		4.25	18.0625	76.76563
4.5	20.25		4.5	91.125		4.5	20.25	91.125
4.75	22.5625		4.75	107.1719		4.75	22.5625	107.1719
5	25		5	125		5	25	125
5.25	27.5625		5.25	144.7031		5.25	27.5625	144.7031
5.5	30.25		5.5	166.375		5.5	30.25	166.375
5.75	33.0625		5.75	190.1094		5.75	33.0625	190.1094
6	36		6	216		6	36	216
6.25	39.0625		6.25	244.1406		6.25	39.0625	244.1406
6.5	42.25		6.5	274.625		6.5	42.25	274.625
6.75	45.5625		6.75	307.5469		6.75	45.5625	307.5469

Figure 27: Output for TwoSingleCurve.cpp



x	log(x+0.01	x^2	x^3	exp(x)	x
0	-4.60517	0	0	1	0
0.075	-2.4651	0.005625	0.000422	1.077884	0.075
0.15	-1.83258	0.0225	0.003375	1.161834	0.15
0.225	-1.44817	0.050625	0.011391	1.252323	0.225
0.3	-1.17118	0.09	0.027	1.349859	0.3
0.375	-0.95451	0.140625	0.052734	1.454991	0.375
0.45	-0.77653	0.2025	0.091125	1.568312	0.45
0.525	-0.62549	0.275625	0.144703	1.690459	0.525
0.6	-0.4943	0.36	0.216	1.822119	0.6
0.675	-0.37834	0.455625	0.307547	1.964033	0.675
0.75	-0.27444	0.5625	0.421875	2.117	0.75
0.825	-0.18032	0.680625	0.561516	2.281881	0.825
0.9	-0.09431	0.81	0.729	2.459603	0.9
0.975	-0.01511	0.950625	0.926859	2.651167	0.975
1.05	0.058269	1.1025	1.157625	2.857651	1.05
1.125	0.126633	1.265625	1.423828	3.080217	1.125
1.2	0.19062	1.44	1.728	3.320117	1.2
1.275	0.250759	1.625625	2.072672	3.578701	1.275
1.35	0.307485	1.8225	2.460375	3.857426	1.35
1.425	0.361165	2.030625	2.893641	4.157858	1.425
1.5	0.41211	2.25	3.375	4.481689	1.5
1.575	0.460584	2.480625	3.906984	4.830742	1.575
1.65	0.506818	2.7225	4.492125	5.20698	1.65
1.725	0.551007	2.975625	5.132953	5.612521	1.725
1.8	0.593327	3.24	5.832	6.049647	1.8
1.875	0.633928	3.515625	6.591797	6.520819	1.875
1.95	0.672944	3.8025	7.414875	7.028688	1.95
2.025	0.710496	4.100625	8.303766	7.576111	2.025

Figure 28: Output for MultiSingleCurve.cpp

2.

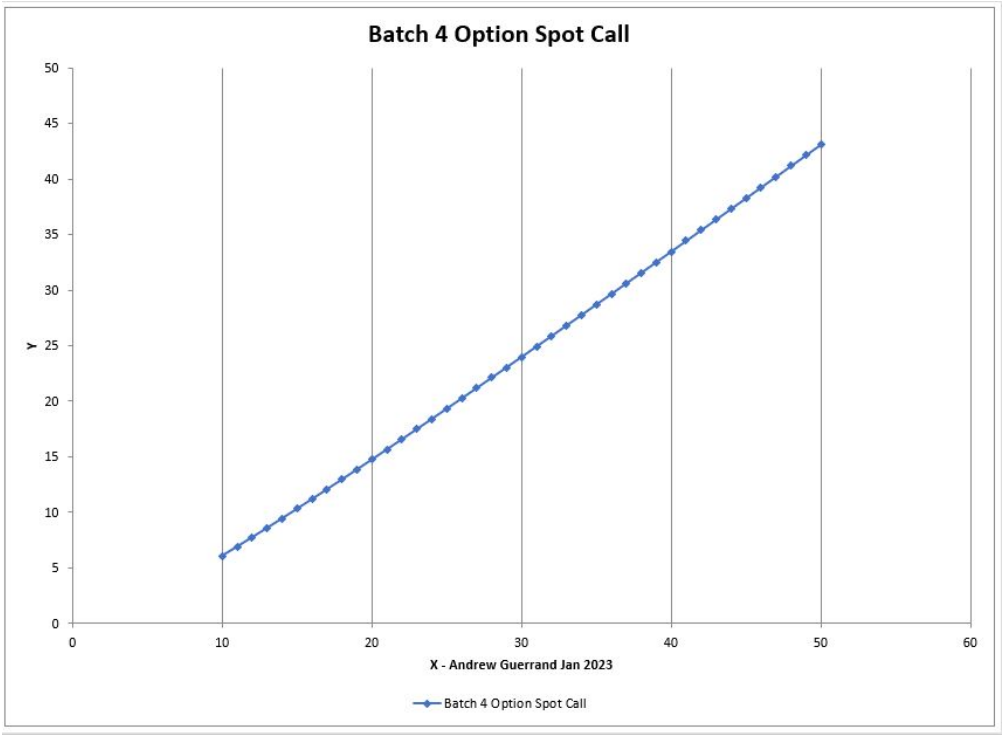


Figure 29: Batch 4, Call Spot Option with monotonically increasing values of S (10 to 50, with mesh size 1). Vector-method.

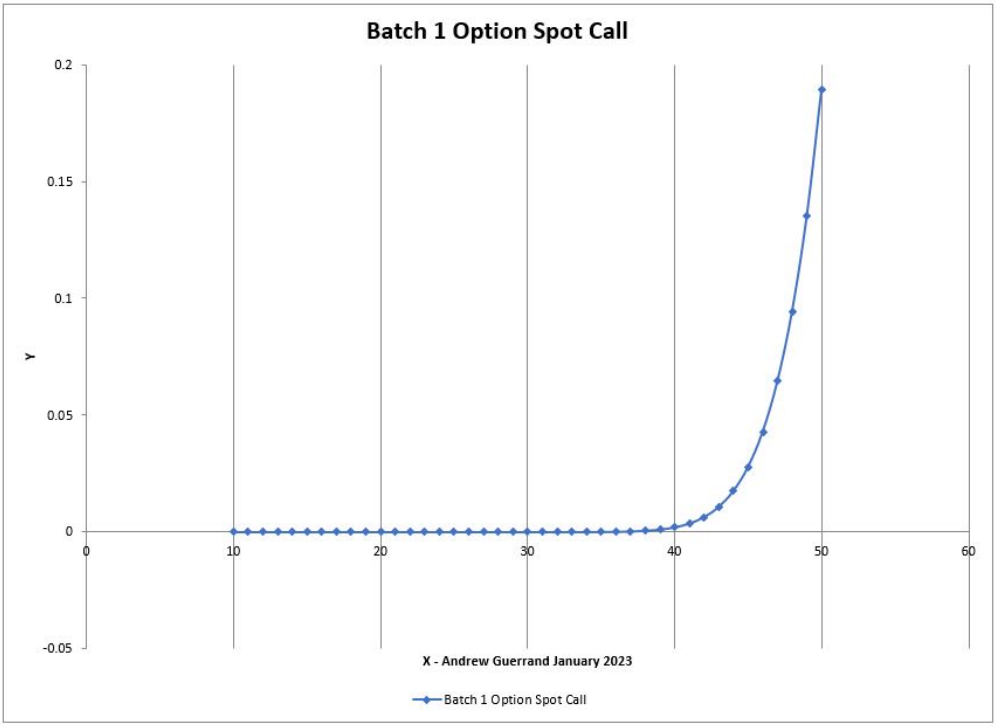


Figure 30: Batch 1, Call Spot Option with monotonically increasing values of S (10 to 50, with mesh size 1). Matrix-method.

5 Group F: Finite Difference Methods

Submission should consist of the working code, example Excel output files, and a document analyzing the accuracy of FDM versus the exact method. Alternatively, you may choose to modify the code to output to the console (instead of Excel).

As FDM is a very advanced topic, this section is meant to be a brief taste of how FDM may be implemented using the C++ techniques learned throughout this course. However, students are not expected to understand the intricacies of FDM for option pricing, at this juncture, as that is the topic of a number of advanced MFE-level courses.

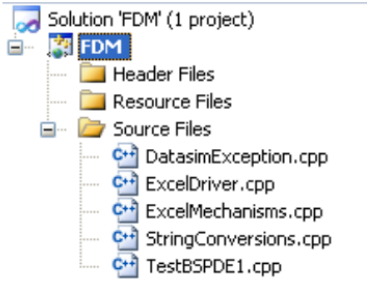
The objective is to run the code as is and is not intended as a course in the finite difference method.

The explicit Euler method is conditionally stable by which we mean that the mesh size k in time must be much less than the mesh size h in space (worst-case scenario is that k = O(h 2)). The objective of this exercise is to determine what the relationships are. Thus, for various values of h determine the value of k above which the finite difference approximation is no longer accurate.

Answer the following questions

I) Finite Difference Methods

1. Compile and run the project as in and make sure that you get Excel output. Examine the code and try to get an idea of what is going on. The following files should be in the project:



2. In this exercise we test the FD scheme. We run the programs using the data from Batches1 to 4. Compare your answers with those from the previous exercises. That's all.

   Remark: since we are using an explicit method, there is a relationship N = J*J where J is the number of mesh points in space and N is the number of mesh points in time. This is somewhat pessimistic and you can try with smaller values of N.

Answers:

1.

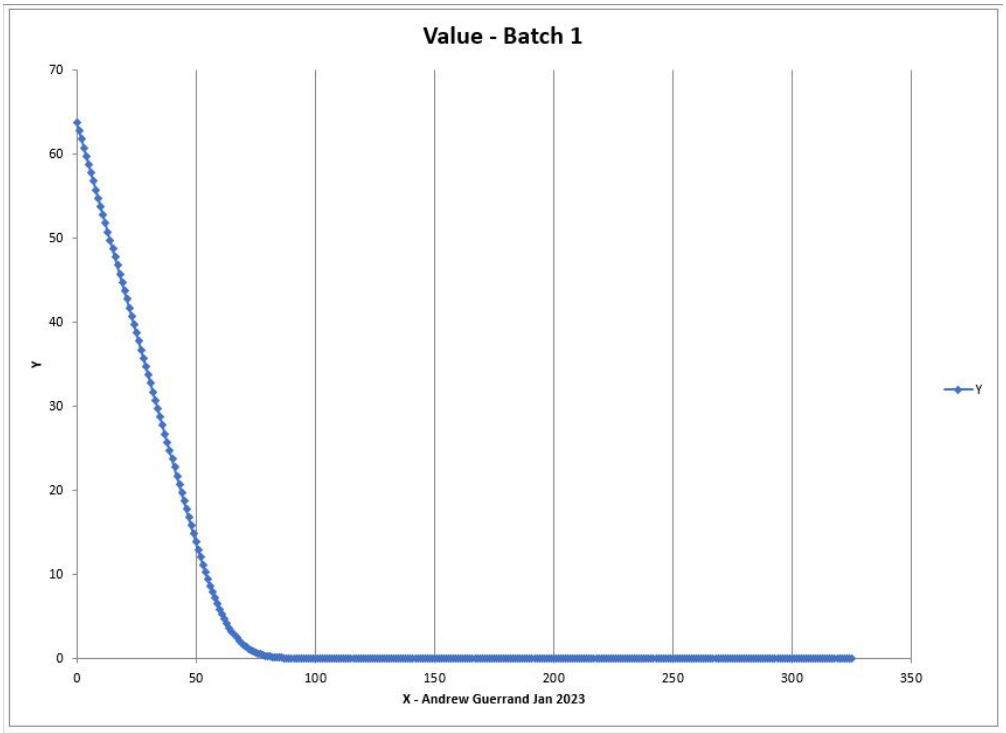


Figure 31: Batch 1 Output

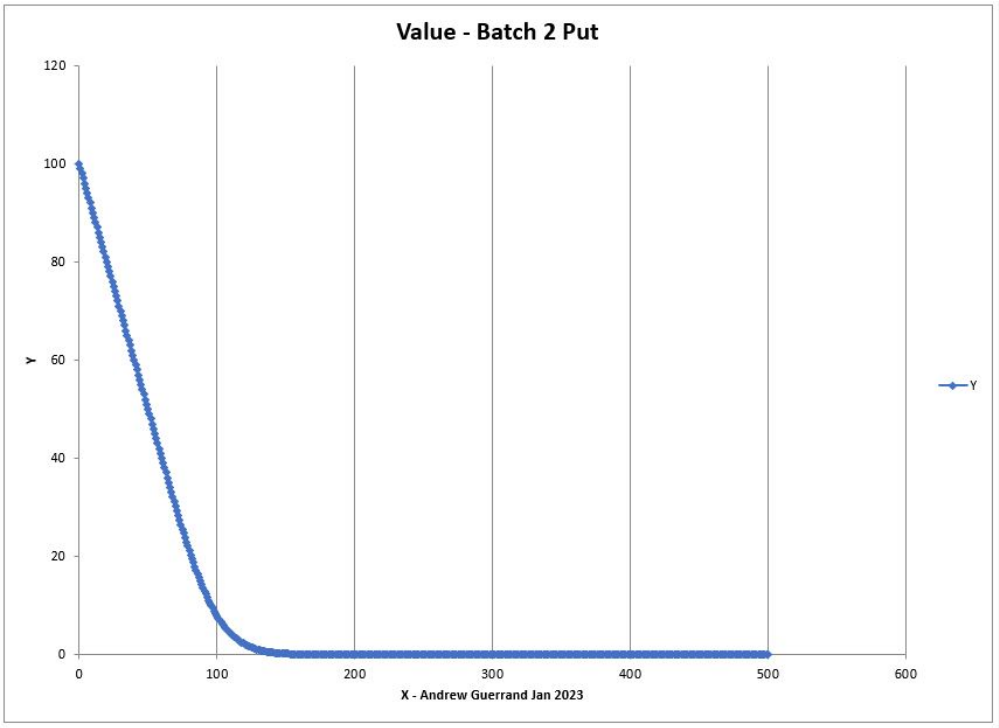


Figure 32: Batch 2 Output

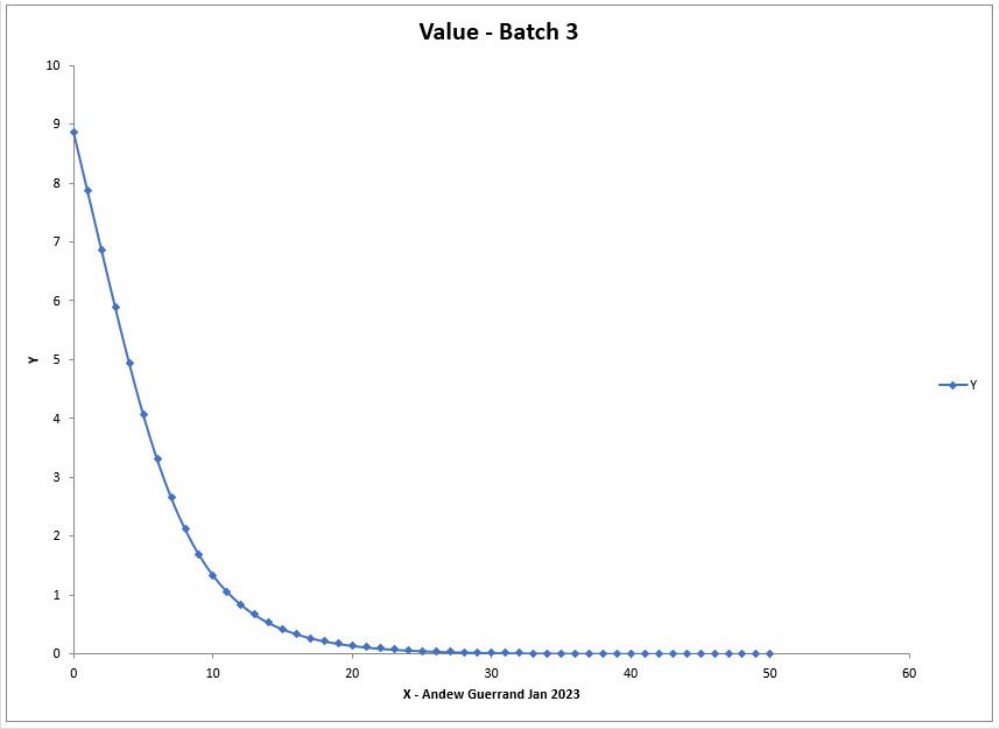


Figure 33: Batch 3 Output

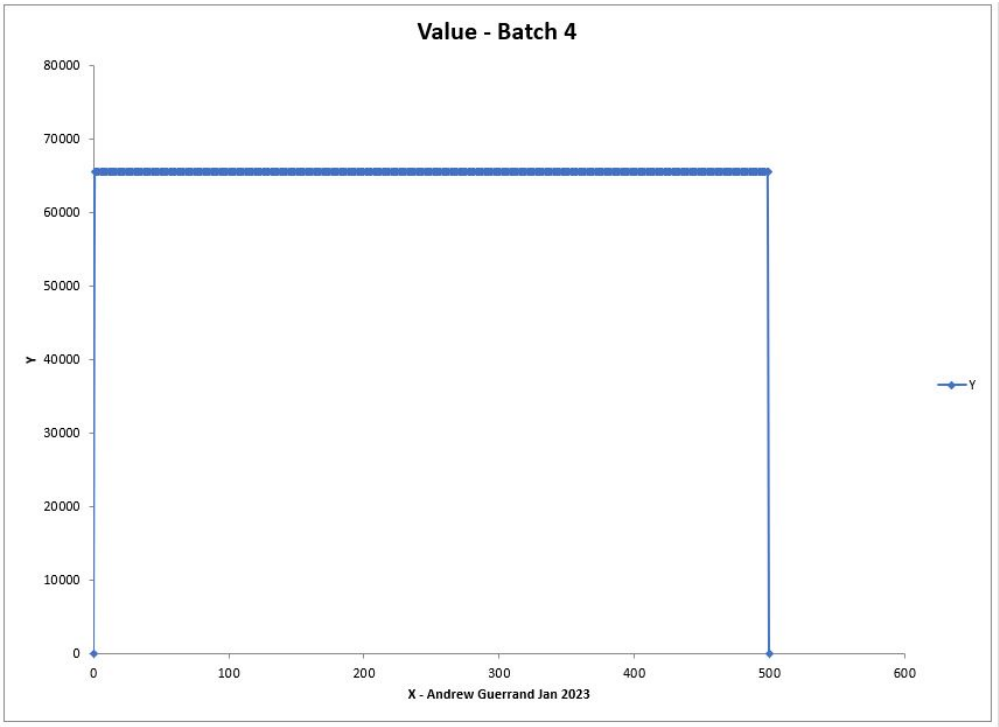


Figure 33: Batch 4 Output

Besides running the code and tweaking values of N, there is not much else to do. There is convergence for batches 1, 2, and 3, but not for batch 4.