CUDA PROGRAMMING ON GPU FOR OPTIONS PRICING

Project for the course of Computational Physics

Valerio Firmano, 918239
Enrico Fornasa, 962031
Matteo Tajana, 960633

July, 2020

UNIVERSITÀ DEGLI STUDI DI MILANO

We declare that this material, which we now submit for assessment, is entirely our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of our work. We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should we engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by us or any other person for assessment on this or any other course of study.

Valerio Firmano, Enrico Fornasa, Matteo Tajana
July, 2020

Contents

List of Figures

1 Introduction

The goal of this project is to show the computational power of GPUs in performing Monte Carlo simulations for pricing stocks' derivatives. Since the 2000s, GPGPUs (General Purpose Graphics Processing Units) have been massively used in computational simulations in various fields of computer science, due to their incredible power in managing and performing simple, parallel computations.

The environment in which our code has been developed is that of financial market simulations (computational finance), with the underlying idea of the quantum mechanical path integral technique. In particular, a lognormal stock evolution was simulated, and on this we created a library for options pricing—i.e. contracts which give the buyer the right, to buy or sell an underlying asset or instrument at a specified strike price prior to or on a specified date, depending on the form of the option.

All the simulations were made remotely in Unimi's LCM cluster using the `CUDA` programming language (`C`/`C++` extended form for GPUs) on a Tesla P100 graphical processing unit. Afterwards, a comparison between the analytical and computational results on the GPU and on the CPU showed the extraordinary performance of the first compared to the latter, as well as the coherence of the results obtained. As shown in Fig. 10, the simulations on the P100 were more than 30× times faster than the same calculations performed on the CPU, and led to the same results.

2 Mathematical and Financial Background

Firstly we introduce the mathematical concepts that ground quantitative finance and econophysics. One of the base assumptions usually made in finance and Monte Carlo simulations is that the evolution of the stock price follows a Markovian process. The price of the stock (as a function of time) depends *only* on its previous value—intuitive for $t \in \mathbb{N}$, non-trivial formulation for $t \in \mathbb{R}$. In other words, the future stock price depends solely on its current price

$$\cdots S(t_{i-1}) \xrightarrow{!} S(t_i) \xrightarrow{!} S(t_{t+1}) \cdots .$$

Considering that the variations of the stock prices are modeled as aleatory, one should exploit the techniques of stochastic calculus to analyze deeply this "random" evolution. The basic assumption made is that the equity returns follow a lognormal distribution, and was firstly introduced and analyzed by the French mathematician Louis Bachelier in his doctoral thesis in 1900 (later denied by Mandelbrot in 1963. However this model is still used in quantitative finance).[1] Let $S(t)$ be the stock price at time t. The price variations are described by the Itô process of stochastic calculus

$$\frac{\mathrm{d}S}{S} = r\,\mathrm{d}t + \sigma\sqrt{\mathrm{d}t}\,\omega, \tag{1}$$

where r is the risk free rate, σ is the volatility of the stock market and ω is a random number from a gaussian distribution with mean 0 and variance 1. Exploiting Itô's lemma one obtains

$$\mathrm{d}\ln S = \left(r - \frac{\sigma^2}{2}\right)\mathrm{d}t + \sigma\sqrt{\mathrm{d}t}\,\omega,$$

i.e. the prices logarithm is characterized by a generalized Wiener process (definition of lognormal evolution). Itô's lemma is simply obtained by expanding the differential $\mathrm{d}S$ at the first order in $\mathrm{d}t$ and at the second order in $\mathrm{d}W \equiv \sqrt{\mathrm{d}t}$, since $\mathrm{d}W^2 = \mathrm{d}t$. From this equation one finds the solution of the first one:

$$S(t_{i+1}) = S(t_i)\,\mathrm{e}^{\left(r-\frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}\,\omega}, \tag{2}$$

where $\Delta t = t_{i+1} - t_i$ holds $\forall t_i$, and ω is a stochastic variables pulled out from a gaussian distribution of null mean and unitary variance.

Alternatively one can integrate Eq. 1 with the finite differences (Euler) method:

$$S(t_{i+1}) = S(t_i)\left(1 + r\Delta t + \sigma\sqrt{\Delta t}\,\omega\right), \tag{3}$$

[1] "The Variation of Certain Speculative Prices", *The Journal of Business*, Vol. 36, 1963, pp. 394-419

where $\Delta t = t_{i+1} - t_i$. One can prove that the mean value $\langle S(t) \rangle = S(0)\mathrm{e}^{rt}$, since these gaussian numbers have null mean value.

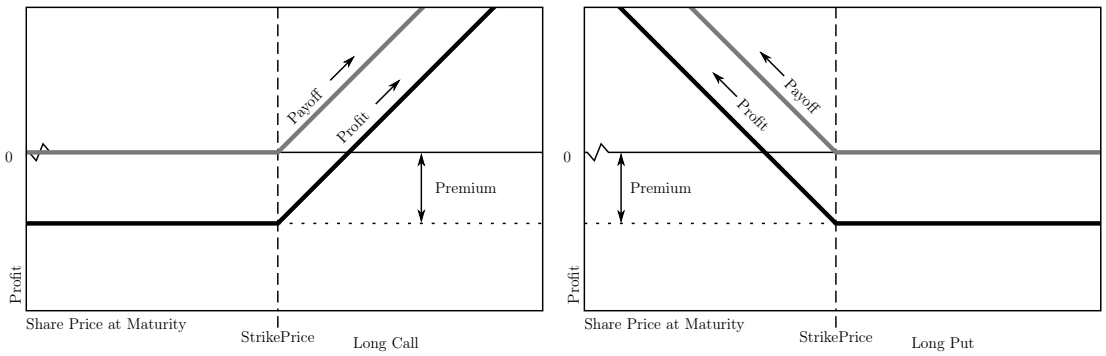

Figure 1: Payoffs for a call and a put options.

The basic plain vanilla option are divided in two categories:

- Call Options: the payoff is given by $\text{Payoff}_{\text{call}} = \max[S(T) - E, 0]$;
- Put Options: the payoff is given by $\text{Payoff}_{\text{put}} = \max[0, E - S(T)]$.

Alternatively, the price of these plain vanilla options can be calculated with the Black-Scholes Equation, which is discussed in Appendix A.

The most important characteristic of an option is the *leverage effect*, i.e. the amplification of the possible profit at the time to maturity. This is due to the fact that the right to buy a certain amount of stocks guarantees the amplification this profit compared to the one of a forward contract (buying the certain number of options). Since "there are no free lunches", this leverage effect is always coupled with a greater risk, compared to the pure stocks trading.

The main problem with the BS Equation is that in general it is a hard task to find an analytical solution for a PDE. Hence the necessity of appealing to Monte Carlo simulations. In order to make these simulations more accurate, one can exploit the physical technique of *path integral* on the stocks' stochastic paths, rearranged to fit the standards of finance in spite of those of quantum electrodynamics. We shall now consider an European option. Let $\mathcal{C}_{0,T}$ a stochastic path an a risk-neutral world—this represents one possible evolution of the underlying asset from $t = 0$ to T (time to maturity). Let $p(\mathcal{C}_{0,T})$ be the probability density of this path, and $\mathcal{P}(\mathcal{C}_{0,T})$ the payoff on that path. Then the option price will be given by

$$\text{Price} = \int_{\mathcal{C}_{0,T}} \mathrm{d}T\, p(\mathcal{C}_{0,T})\mathcal{P}(\mathcal{C}_{0,T})\mathrm{e}^{-rT}. \tag{4}$$

Thus the question of the European option pricing reduces to that of a path integral on all the possible stochastic paths that begin in $S(0)$ at $t = 0$.

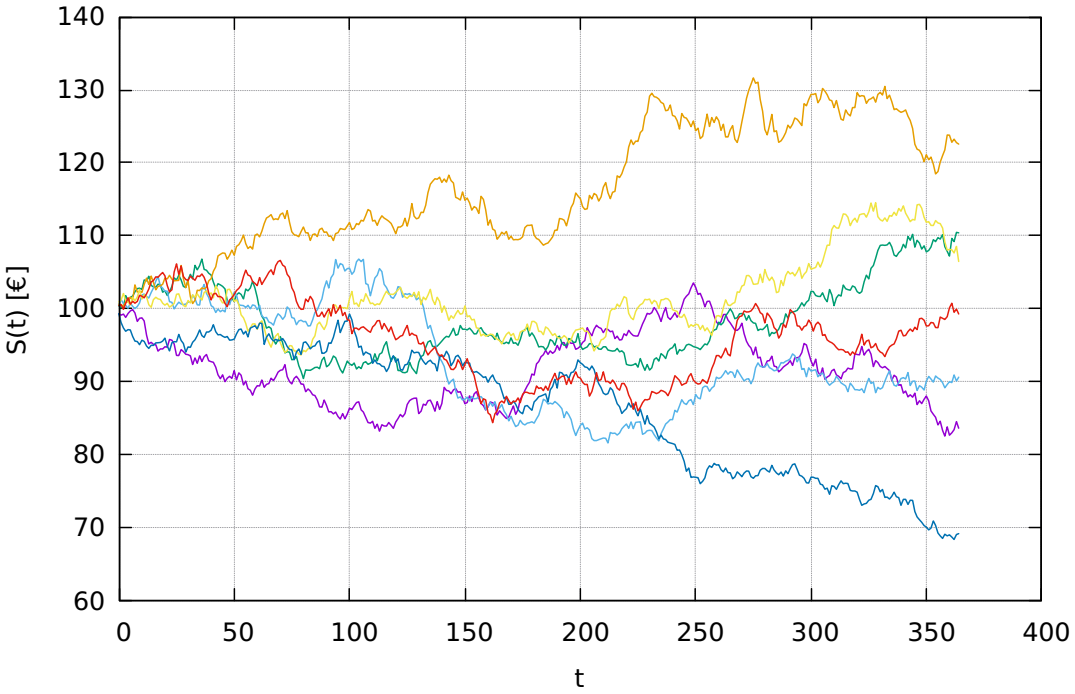

Figure 2: Possible evolutions of a stock with $S(0) = 100€$, $r = 0.15\%$ and $\sigma = 15\%$ over $T = 1\text{yr}$ divided in $m = 365$ steps.

3 Simulations

In this section we are presenting the simulations performed on the GPU Tesla P100. The simulations have been conducted on a series of stock derivatives: forward contract and options, and were performed with the exact scheme of integration and the finite differences Euler method, too. Each of these will be treated in a different subsection, in which we basically analyze its price as a function of the simulation parameters and the market data.

The "forward contract" is the simplest contract that returns a payoff equal to the value of the stock at the time to maturity $S(T)$—see Fig. 2 and the implementation in the Supplementary Material.

3.1 Plain vanilla options

All the plain vanilla simulations were made on using both schemes of integration Eqq. 2 and 3 on the GPU, while the CPU integrated only using the exact formulation.

Initially we checked the implementation of the plain vanilla options' payoff. The simulations confirmed the theoretical graphs pictured in Fig. 1.

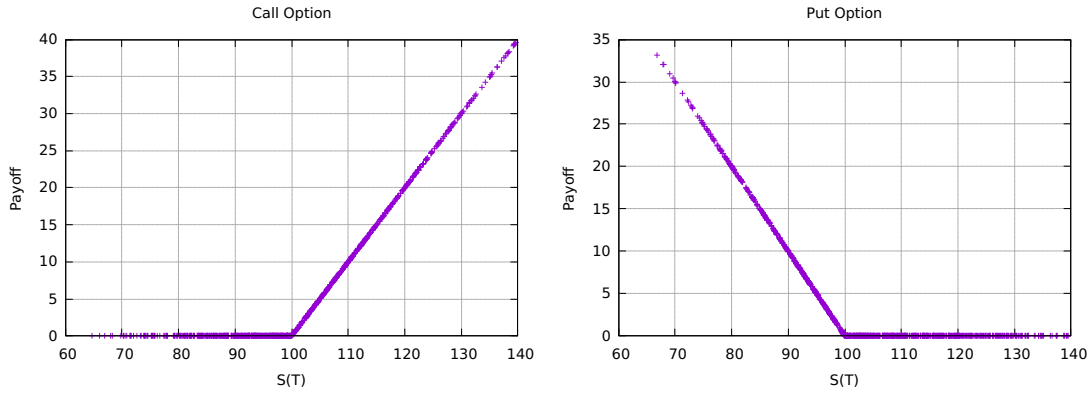

Figure 3: Simulations for two plain vanilla options on $N = 1$ stock with strike price $E = 100€$ on a stock market with $r = 0.15\%$, $\sigma = 15\%$. The simulations confirm Fig. 1.

Notice how the payoff is denser around the initial stock price $S(0) = 100€$. This is a sign of the genuinity of the pseudorandom numbers used. The Central Limit Theorem assures that a sum of independent random numbers is distributed according to a Gaussian. Thus the tails of the distribution are flatter than its central part.

Afterwards, we analyzed the call plain vanilla price. Our goal was to price these options starting from the market data available. For this option we performed $N = 5000$ Monte Carlo simulations for each of the $256 \times 20 = 5120$ threads instanciated on the GPU. In particular, once fixed the total number of threads, for $m \in \mathbb{N}$ we averaged firstly on the simulations on the single thread, and then on all the threads.

For each of the call and put options we performed different simulations with same interest rate r and different σ to highlight the effect of the volatility on the options' prices.

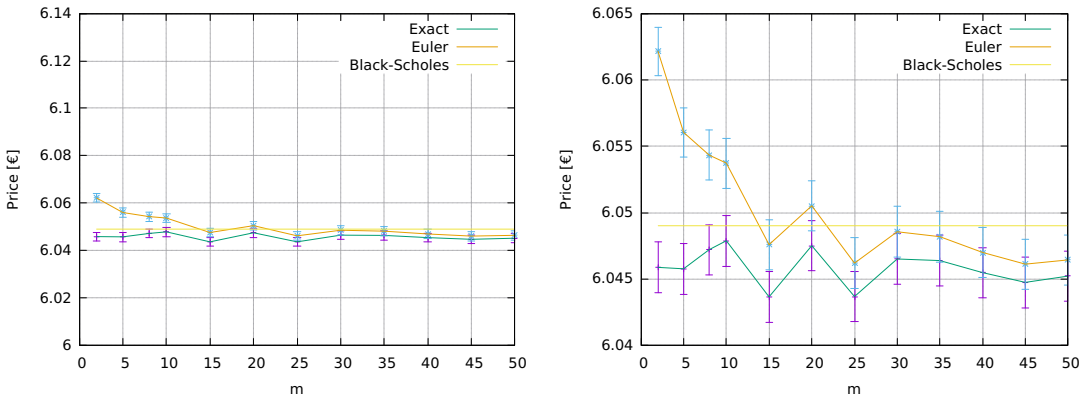

Figure 4: European call payoff as a function of the number of steps and magnification at large m. The magnification is made by zooming the y–axis in to amplify the view of the statistical oscillations. The market data are $r = 0.15\%$, $\sigma = 15\%$, $S(0) = 100€$ and $T = 1\text{yr}$.

The errorbars represented in the graphs are 1 standard deviation long, such that errorbars $\simeq 0.002€$. As m increases, the Euler's method price flattens and approaches the theoretical value obtained with the solution of the BS equation, while the exact solution is compatible whitin two times standard deviations with BS for each m.

We remark how the volatility affects the Euler integration, making it less precise as σ increases.

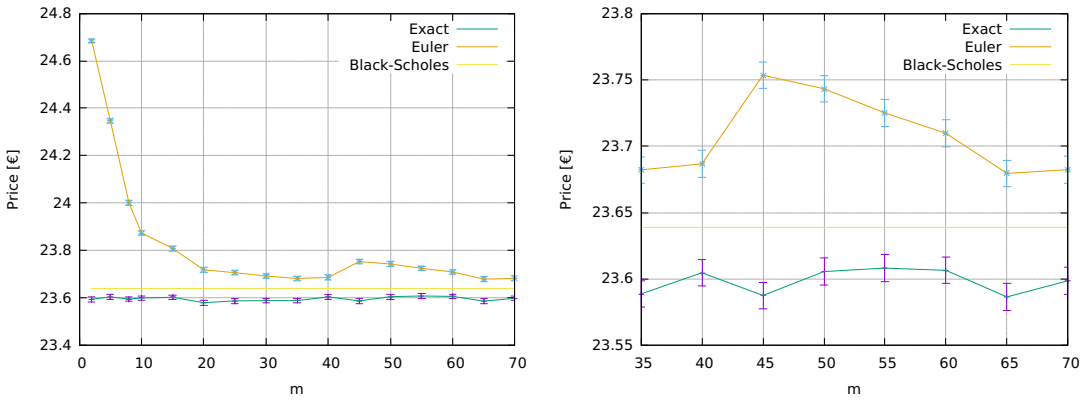

Figure 5: European call payoff as a function of the number of steps and magnification at large m. The magnification of the same graph is done for $m \geq 35$. The market data are $r = 0.15\%$, $\sigma = 60\%$, $S(0) = 100€$ and $T = 1\text{yr}$.

Here the errorbars represented in the graphs are 1 standard deviation long, such that errorbars $\simeq 0.01€$.

It is really important to consider the goodness of our results, with an accurate error analysis.

We remark that for each thread a number N of simulations were executed, hence the statistical mean of the payoffs and its relative standard deviation were comuputed. Therefore the behavior (for each thread) of the error is $1/\sqrt{N}$. Then, after these values were copied back to CPU, the weighted mean on all threads' average payoffs was evaluated. Since the standard deviations are of the same order of magnitude for each thread, then the weighted mean's error is proportional to the inverse root of the total number of simulations executed. Hence error $\sim 1/\sqrt{N \cdot N_{\text{threads}}}$

The final error is independent of the number of steps m, thus one shoud not worry about the incompatibility with BS of Euler's payoff for small ms. On the contrary it is appreciably affected by the market data, in particular the volatility. For $\sigma = 15\%$ the error is about $\approx 0.002€$, while for an higher one ($\sigma = 60\%$) the error becomes $\approx 0.01€$.

Similar simulations were made for the put options, and also in this case the interest rate r was fixed at 0.15%, while the volatility was changed from 15% to 60%. The results are presented in the following graphs.

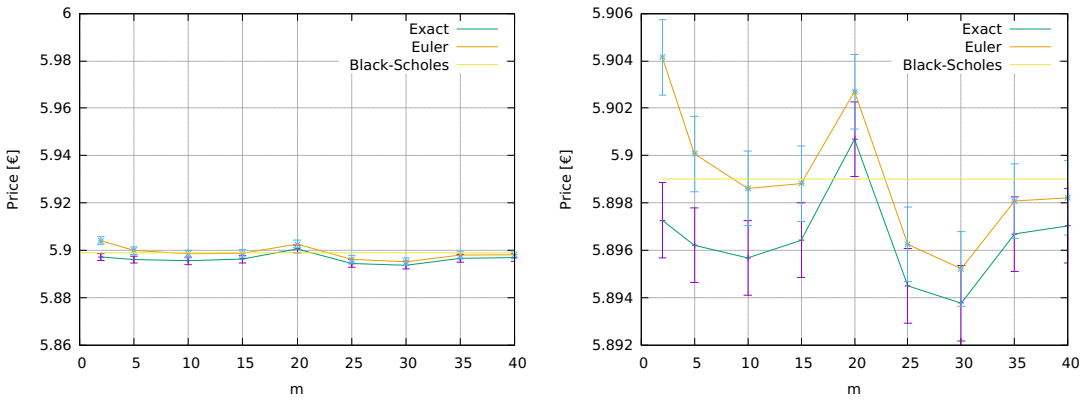

Figure 6: European put payoff as a function of the number of steps and magnification on the prices to highlight the fluctuations. The market data are $r = 0.15\%$, $\sigma = 15\%$, $S(0) = 100€$ and $T = 1\text{yr}$.

The errorbars represented in the graphs are 1 standard deviation long, such that errorbars $\simeq 0.002€$. Also in the case of a put option, the fluctuations are damped as the number of steps m increases. Increasing the volatility to 60% we can see again how this affects the Euler integration at low steps.

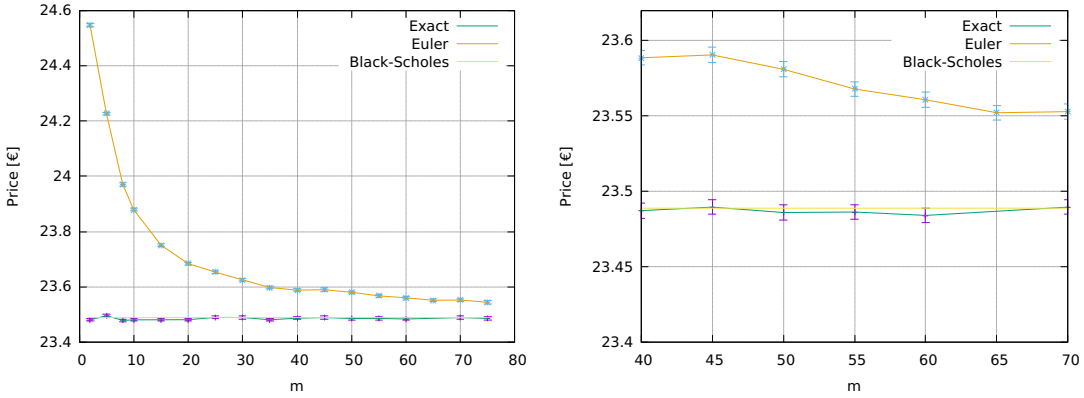

Figure 7: European put payoff as a function of the number of steps and magnification at large m. The market data are $r = 0.15\%$, $\sigma = 60\%$, $S(0) = 100€$ and $T = 1\text{yr}$.

Here the errorbars represented in the graphs are 1 standard deviation long, such that errorbars $\simeq 0.005€$.

One can see how also in this case the Euler scheme of integration approaches the exact one as m increases. However, with these market data, we can not say that the exact scheme is effectively approximated, since the results are more than three times standard deviations distant. The considerations discussed before about the statistical errors for the call options hold similarly for the put ones.

3.2 Exotic positive performance corridor option

Once completed the simulations for the plain vanilla options, we priced a more complicated one, the positive performance corridor exotic option. This exotic option shows a peculiar payoff, given by

$$\text{Payoff} = \left[\left(\frac{1}{m} \sum_{i=0}^{m-1} P_i \right) - K \right]^+ , \tag{5}$$

where m is the length (in steps) of the path, K may be interpreted as a sort of strike price (in percentage), $[\cdots]^+$ is the positive part of the argument and

$$P_i = \begin{cases} 1 & \text{if} \quad 0 < \frac{1}{\sqrt{\Delta t}} \ln(S_{i+1}/S_i) < B\sigma \\ 0 & \text{otherwise.} \end{cases}$$

The simulations for the exotic option were made calculating the payoff as a function of the parameter B, which is a sort of "unit of measure" of the width of the barrier. All the simulations were performed with the exact scheme of integration (Eq. 2), and several market data were considered. In particular, the interest rate r was set to -0.15%, 0% and 0.15% to explore all the possible scenarios in which this option might be. Moreover, for each of these we considered different volatities of the stock market, choosing $\sigma = 15\%$ and $\sigma = 60\%$. The results follow:

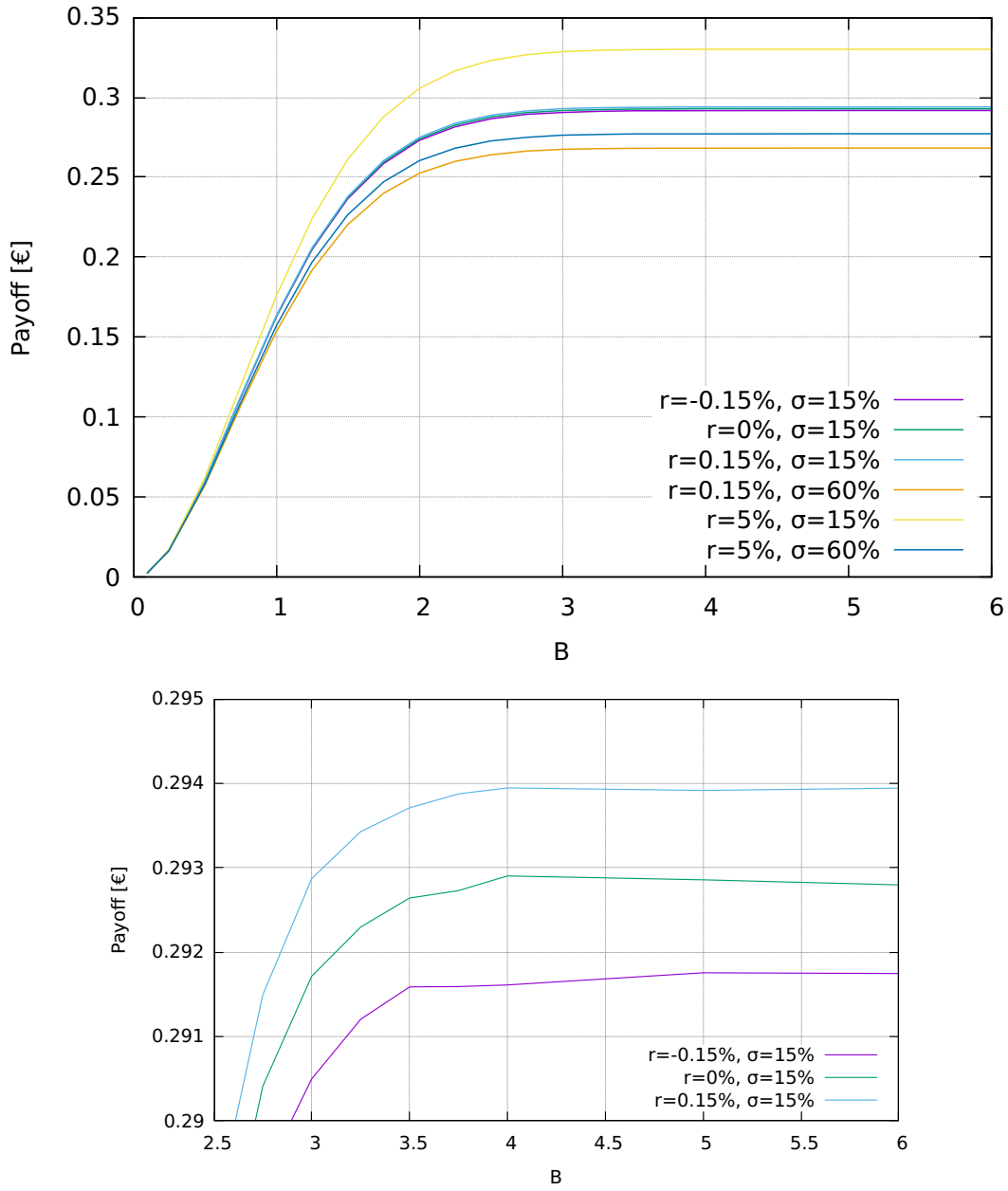

Figure 8: The positive performance corridor option payoff as a function of the barrier parameter B and a magnification on the most similar payoffs with $r = \pm 0.15\%$, $r = 0\%$ and $\sigma = 15\%$. The "strike price" was set to $K = 0.15$.

We shall now give a more intelligible interpretation of Eq. 5. The payoff may be interpreted as the percentage of times for which the logarithmic return of the underlying stock lies inside the barrier $(0, B\sigma)$. The "strike price" K is the investor's guess about the frequency of this event. Basically, if $P > K$

the payoff is $P - K$, and zero otherwise. As a consequence, the greater is B, the higher is the probability that a positive logarithmic return falls inside the barrier. Moreover, σ is deeply connected with the stochastic kicks the underlying stock experiences (Eq. 1). Thus, as σ increases also the probability of greater logarithmic returns grows. The bigger is this return, the higher the probability of overcoming the barrier is—although the barrier's width increases linearly with σ—and, as a consequence, $P_i = 0$ more often.

An analysis of Fig. 8 corroborates our intepretation. At $B \approx 3$ and forth the barrier is so wide that almost every positive logarithmic return lies inside its extrema. This means that the probability of having $P_i = 1$ becomes almost equal to that of having a positive logarithmic return.

It's interesting to see the different roles of σ and r in the evaluation of payoff. As we can see in Fig. 8 (in wich we have $m = 20$), for the same interest rate r, options with greater σ have a smaller payoff. Moreover a PPC option with $r = 0\%$ or $r = -0.15\%$ and $\sigma = 15\%$ is more profitable than one with $r = 5\%$ and $\sigma = 60\%$.

We can also consider the extreme cases $B = 0$, $K = 0$ and $K = 1$. The $B = 0$ case is trivial: the barrier has no width (it is a line) and the payoff is always zero. If $K = 0$ the payoff is given by the percentage of times the logarithmic payoff falls inside the barrier. Lastly, if $K = 1$ the payoff is null too. Indeed, $P \leq 1 \ \ \forall B \in \mathbb{R}$, and the positive part of the payoff sets it to zero.

3.3 Performance comparison between CPU and GPU

We shall now analyze in depth the advantages of GPU programming. Graphical processing units allow parallelization of serial codes. Their highly parallel structure makes them more efficient than general-purpose central processing units (CPUs) for algorithms that process large blocks of data in parallel. In this particular case we want to perform as many path integrals (Eq. 4) as possible simultaneously.

The hierarchical structure of GPU divides it in grids of threads—which are nothing but tiny processors—that work simultaneously and independently on each other. Each grid is divided in multiple blocks of a certain number of threads each, at most typically 256 or 512 depending on the architecture of the device. The time employed for the program to run will have a "stair" trend, as shown in Fig. 9.

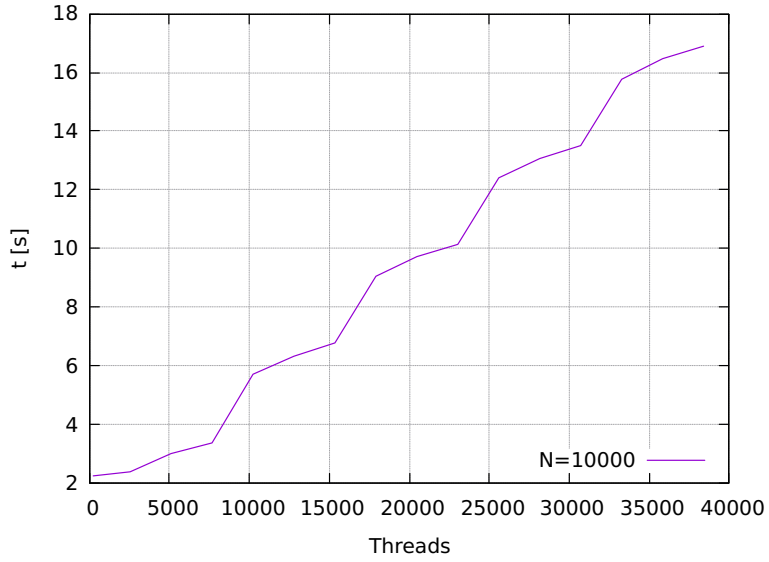

Figure 9: Time needed for calculating the price of a European call option on multiple threads. The 10000 Monte Carlo simulations integrated the lognormal process on a path of $m = 30$ steps.

This trend is due to the ability of GPU to compute the same operation simultaneously on several threads. However the stairs are not completely flat because in our code each thread has to allocate objects on the global memory at its instance.

Nevertheless, the same calculations performed on the CPU were incredibly less performing, as shown in Fig. 10.

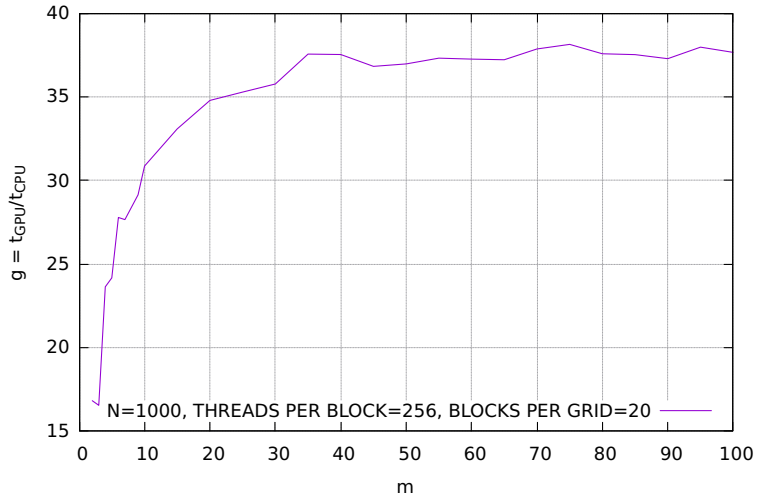

Figure 10: Plot of the gain factor g as a function of the steps m for different number of Monte Carlo simulations performed.

The time gain factor $g = t_{\text{GPU}}/t_{\text{CPU}}$ grows ($m < 30$) from a low value to approach a stationary one ≈ 37.5. This is due to the fact that GPU works better than CPU when it has to make several calculations. Moreover, as the number of steps m increases, the time gain saturates. This is due to the fact that each thread has to allocate a greater amount of memory, slowing the process down.

3.4 Unit Test

Lastly we implemented a unit test (see Supplementary Material for code) to check the exactness of the GPU results. This test lets an option evolve both on GPU and CPU independently and cross checks between the devices the price of an option. In particular, we implemented a check for a forward contract and a European Call option on 5 stochastic processes, but other choises are possible. Here are the test results:

OEC GPU [€]	OEC CPU [€]	FC GPU [€]	FC CPU [€]
39.5355802861	39.5355802861	139.5355802861	139.5355802861
58.8089015441	58.8089015441	158.8089015441	158.8089015441
8.8584289951	8.8584289951	108.8584289951	108.8584289951
2.2724059387	2.2724059387	102.2724059387	102.2724059387
18.6201200624	18.6201200624	118.6201200624	118.6201200624

Table 1: Unit Test results for a European Call Option (OEC) with $E = 100$€ and a forward contract (FC). The market data are $r = 20\%$, $\sigma = 15\%$, while the simulation were made for $m = 30$ steps.

One can observe how all the values coincide precisely up to the last digit. The results confirm that the simulations performed on the GPU lead to the correct pricing of the options and the evaluation of the underlying stock evolution.

4 Implementation of the Library

In this section we are going to show the characteristics of the library written for options' pricing. The scheme of this library is here pictured:

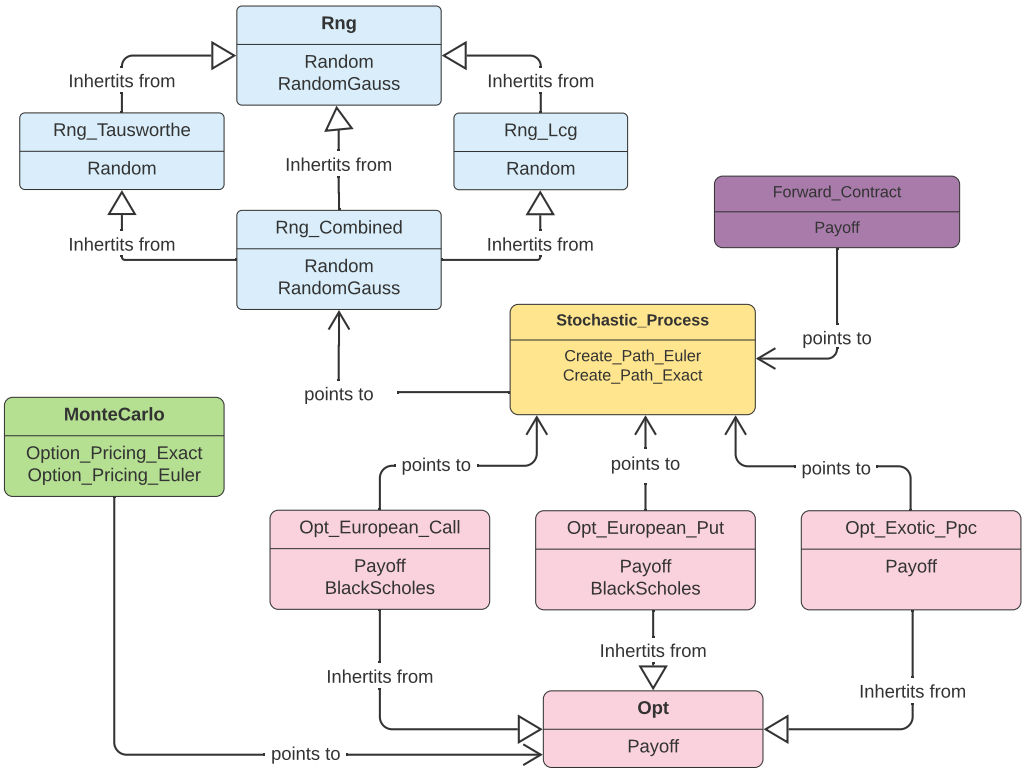

Figure 11: Scheme of the classes used in the library and their inheritances and references.

The library has been implemented as union of macro pieces (classes) that interact with each other to price the options.

4.1 The `main()` in a nutshell

The main algorithm—independently on the architecture on which it is run—of the library follows:

Algorithm 1: Option pricing pseudo `main()` code

Data: Stock market data ($S(0)$, r, σ), option type data (strike price E, time to maturity T), simulation parameters (number of simulations N, number of steps per simulation m).

Result: The option's price (payoff) F.

begin

- Read the market data from input file
- **for** $i = 0$ **to** N **do**
  - **for** $j = 0$ **to** $m - 1$ **do**
    - Get random gaussian number w
    - Let the stock evolve of a $\Delta t = t_{j+1} - t_j$ according to Eq. 2
  - Create an option on the ith path $\mathcal{C}_{0,T}$
  - Compute the option's payoff
- Calculate the mean value of the N payoffs following Eq. 4
- Write output data

This algorithm has a few facets worth to analyze. Firstly it can be applied to *any* kind of option: both plain vanilla and exotic—one should just tune the option data properly. Moreover, the use of `C++` classes allows us to keep the backbone of the code intact, just changing the option implementation properly. Secondly it can be performed on a GPU. In fact, the enormous number of threads (processors) available in this device make the parallelization of this code possible.

In particular, one can repeat this algorithm in each thread. Since all the threads in a GPU work simultaneously, the first `for` loop can be performed fastly and effectively multiple times—as many as the number of threads instantiated. Therefore, it is possible to carry multiple path integrations (Eq. 4) out at the same time.

4.2 Structure of the Library

Every passage of the previous algorithm has been implemented in `CUDA` programming language exploiting the `C++` classes. Before presenting the different single parts of the code—as well as the kernels used on the GPU—we show the pseudo code for the "extended" version of Algorithm 1. The new implementation should exploit the computational power of threads parallelization on the GPU.

Algorithm 2: Option pricing pseudo `main()` code with GPU integration

Data: Stock market data ($S(0)$, r, σ), option type data (strike price E, time to maturity T), simulation parameters (number of simulations N, number of steps per simulation m), GPU parameters (`THREADS_PER_BLOCK`, `BLOCKS_PER_GRID`).

Result: The option's price (payoff) F.

begin

- Read the market data from input file
- Allocate memory on the device for the calculation
- **HtD** Copy data to the device
- **if** *Data was copied correctly on the device* **then**
  - **do** *N Monte Carlo simulations simultaneously on each thread:*
  - **for** $i = 0$ **to** N **do**
    - **for** $j = 0$ **to** $m - 1$ **do**
      - Get random gaussian number w
      - Let the stock evolve of a $\Delta t = t_{j+1} - t_j$ according to Eq. 2
    - Create an option on the ith path $\mathcal{C}_{0,T}$
    - Compute the option's payoff
  - **DtH** Copy back to the host the payoffs obtained
  - Calculate the mean value of the N payoffs following Eq. 4 on each thread
- Calculate the average payoff among the threads
- Write output data

It's worth remarking the advantage of performing multiple simulations (in our case `THREADS_PER_BLOCK`×`BLOCKS_PER_GRID`) simultaneously in stead of running the program the same number of times consecutively. However the time gain is slowed down by the phases of copying back/forth the data from the host to the device and vice versa, in lines HtD and DtH. All these aspects have been already discussed more deeply in Sec. 3.3. Lastly, we underline how Algorithm 2 is nothing but Algorithm 1 run on every thread.

4.3 Classes used in the Library

A more compact and "abstract" version of the library scheme is displayed in Fig. 12. The basic idea is that one wants to perform many Monte Carlo simulations on a certain option and then average the results in order to price it. For doing so, the option must be constructed on a stochastic path (very important in case of path-dependent options) generated by a series of random (gaussian)

numbers, i.e. the "kicks" the underlying stock price experiences. We shall start from the last element of this flow chart to illustrate its implementation and then will go backwards to the `MonteCarlo` class. All the codes are reported in the Supplementary Material.

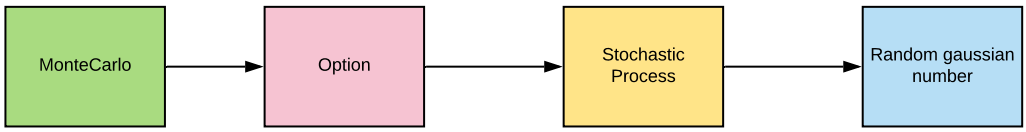

Figure 12: Basic flow of the program for pricing options.

4.3.1 `Rng`

The goal of this class is to generate the random gaussian number ω. This is accomplished by combining four different number generators: three Tausworthe generators (each with different parameters) and a linear congruential generator outputs through the bitwise `XOR` operator of `C++`. The linear congruential generator's parameters were taken from Numerical Recipes. The gaussianness is obtained with the Box-Muller transform that yields a pair of gaussian random numbers. We managed not to waste the second number of these by storing it in an auxiliary variable and implementing the `Get_Gauss()` function properly.

4.3.2 `Stochastic_Process`

The `Stochastic_Process` class generates the path[2] on which the options are built. The class has a pointer to a `Rng` object, and this pointer allows us to generate iteratively random gaussian numbers. These random gaussian numbers are used to let the stock price evolve like in Fig. 2 and its path is stored in an array of `double`.

We implemented two different types of paths. The first one is the "exact" one, obtained integrating analytically the lognormal process through Eq. 2. The other is obtained by integrating the same process with Euler's method Eq. 3.

4.3.3 `Opt`

All the options we priced are inherited from the class `Opt`, which presents the virtual method `Payoff()`. Different type of options were implemented in the library.

[2]I.e. the time evolution of the underlying option price $S(t_i)\ \forall i$ of the simulation, following Eq. 1.

The call and put options were implemented, and an exotic Positive Performance Corridor option as well. Each option has a pointer to a `Stochastic_Process` object through which it is possible to get the stock price at any time t_i. Lastly, the plain vanilla options have been provided with the method `BlackScholes()`, which returns the option's price as analytical solution of Eq. 6—the mathematical details are presented in the Appendix A.

4.3.4 `MonteCarlo`

The `MonteCarlo` class does nothing but the pricing of the option averaging its payoff over the simulations performed. This class presents a pointer to an option, so that it is possible to calculate the payoff both with the exact and the Euler's integration. This pricing is obtained averaging on all the payoffs calculated on each of the N simulations this class performs. The computation yields both the mean value and the standard deviation of the payoff.

4.4 Kernels used in the Library

The kernels are the landmarks of GPU computation. Their task is to split the same code out over the hundreds or thousands threads instantiated. Independently on the kind of option one wants to price, the base recipe is the same. The kernel has to launch on each thread the same simulation with different paths—i.e. the code between the lines labeled as HtD and DtH in Algorithm 2.

The main issue is that the memory that one can allocate in each thread is very limited. Hence the necessity to pass as many input data as possible as parameters of the function, so that the number of variables to instance on the threads is minimized.

In the library many kernels were implemented: both the plain vanilla options call and put, the exotic positive performance corridor and the forward contract. Obviously, both the exact and the Euler stochastic process evolution were implemented. This has been possible thank to the typical properties of classes—especially inheritance, abstraction and encapsulation—that made the work modular and easy-to-manage.

5 Conclusions

Nowadays time optimization and computer memory are becoming more and more important in every scientific field, since modern problems require a larger computational capacity than before. GPUs became since the 2000s an essential tool to achieve this goal, as well as an effective tool for fastening the codes execution. This kind of programming is used in many fields of computer simulations, especially in computational/quantitative finance, computational physics (bioinformatics, statistical physics, quantum mechanical physics, ...), data mining and automatic parallelization. Usually, in computational biophysics there is an expected speed-up of the simulations that goes from 6× up to 100× with respect to the same calculations performed on the CPU. The calculations we made in this work showed all of this: although in a complete different field with different calculations, the time gain factor g is in the same range, as shown in Fig. 10. The GPU weight was lightened as much as possible in order to make the computations feasable and fast. Though this is in general a hard task, the results obtained are satisfactory. We focused on the following points:

- The comparation between GPU and CPU performances confirmed our expectations, showing a gain in terms of time optimization. Moreover, we checked successfully through a unit test that, given the same initial conditions, GPU and CPU lead to identical results (with a relative confidence range of 10^{-8}), as summed up in Table 1.

- The two integration methods (exact Eq. 2 and Euler Eq. 3) were compared, in order to understand their limits and similarities. The result is that the more one increases the number of integration steps m, the more Euler integration approaches the exact one.

- The Monte Carlo simulations yielded the plain vanilla options prices compatible within two standard deviations from the analytical results given by the BS equation. This is true for all the plain vanilla options but the one with $\sigma = 60\%$, which is anyways within three standard deviations.

- For the exotic PPC options we showed the payoff behavior in function of the width of the barrier B. Moreover, one can observe how this value depends on the fact that the logarithmic return must be positive and within the barrier as this increases.

Last but not least, we cared to the easiness of an hypothetical external user to make use of it, too. The use of classes made this work modular and easily adaptable to become a base for further projects.

A The Black-Scholes Equation

As aforementioned, an option is a contract that gives the buyer the right to buy or sell an underlying asset at a specified strike price. In 1973, the economists F. Black and M. Scholes published a paper in which an analytical formula for the pricing of a plain vanilla option was derived.[3] The derivation is based on a series of assumptions, among which the most important are:

- the underlying asset follows a generalized Wiener process, expressed by Eq. 1;
- the volatility σ is constant;
- the rate curve is constant;
- the market follows perfectly the non-arbitrage hypothesis.[4]

The derivation of the Black-Scholes equation starts with Eq. 1 and Itô's lemma for a derivate of this option—whose price is denoted with $F = F(r, S)$:

$$\begin{cases} \mathrm{d}S = rS\mathrm{d}t + \sigma S\omega\sqrt{\mathrm{d}t} \\ \mathrm{d}F = \left(rS\frac{\partial F}{\partial S} + \frac{\partial F}{\partial r} + \frac{1}{2}S^2\sigma^2\frac{\partial^2 F}{\partial S^2}\right)\mathrm{d}t + \sigma S\frac{\partial F}{\partial S}\omega\sqrt{\mathrm{d}t} \end{cases}$$

The crucial point is that the stochastic process $\sqrt{\mathrm{d}t}$ is the same for both the equations. After some algebra one comes to the BS equation

$$\frac{\partial F}{\partial t} + rS\frac{\partial F}{\partial S} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 F}{\partial S^2} = rF. \tag{6}$$

This is general for *any* derived whose price depends only on S and t. The boundary condition for this PDE is given by option's final payoff (that depends on the price of the underlying asset!), thus it is

$$F(s, t = T) = \text{Payoff}(S),$$

where T is the standard notation for the time to maturity of the contract.

In particular, for a call option the solution is given by

$$F_{\text{call}} = SN(d_1) - EN(d_2)\mathrm{e}^{-rT}, \tag{7}$$

[3] "The Pricing of Commodity Contracts", *The Journal of Financial Economics*, Vol. 3, 1973, pp. 167-179

[4] It is impossible to realize an unmistakable profit greater than that of an investment with risk-free interest rate. The more operators are considered, the more true this is.

where

$$d_1 = \frac{\ln \frac{S}{E} + \left(r + \frac{\sigma^2}{2}\right) T}{\sigma\sqrt{T}} \quad \text{and} \quad d_2 = \frac{\ln \frac{S}{E} + \left(r - \frac{\sigma^2}{2}\right) T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}.$$

Similarly, the solution for a put option is given by

$$F_{\text{put}} = EN(-d_2)\mathrm{e}^{-rT} - SN(-d_1). \tag{8}$$

In both the equations, $N(d)$ is the normal cumulative distribution

$$N(d) = \int_{-\infty}^{d} \frac{\mathrm{d}x}{\sqrt{2\pi}} \mathrm{e}^{-\frac{x^2}{2}}.$$

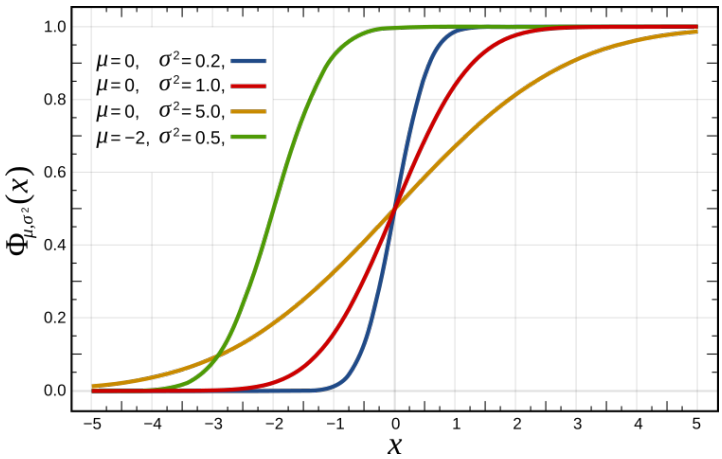

Figure 13: Cumulative distribution function for the normal distribution.