Pricing derivatives in a Black-Scholes setting through Montecarlo simulations

Gergo Berta
MFE
College of Management of Technology
EPFL
Email: gergo.berta@epfl.ch

Matteo Ferrazzi
MFE
College of Management of Technology
EPFL
Email: matteo.ferrazzi@epfl.ch

Matteo Sansonetti
MFE
College of Management of Technology
EPFL
Email: matteo.sansonetti@epfl.ch

***Abstract*—We are interested in developing an algorithm that allows the user to calculate the price of commonly traded options, to have the most common pool we use the stocks of SP500. We want to focus on widely traded options since they represent the majority of traded options and more sophisticated derivatives are based on these standard and well-known models. We decide to price the derivatives with the Black-Scholes-Merton model, even if it simplifies the real-world options assuming, for example, constant volatility, it is optimal to deeply understand how the pricing process works. We implement a Montecarlo simulation to estimate the prices of the derivatives, it is among the most popular tool to predict outcomes that involve random variables. Montecarlo method gives us the possibility, if the number of simulations is high enough, to estimate with a great degree of precision the prices. This type of approach leaves also room for possible parallelization in case the number of simulations exceeds a certain threshold. To increase the speed of execution we make the user choose if he wants to use a C++ compiler, we also give him the possibility to set some parameters such as the number of steps he wants to have in a year, the stock price, the date from which estimate. All these features can be chosen by the User through a visual interface. The following paper considers standard derivatives and leaves room for deepening the subject.**

1. Introduction

For our analysis, we want to use the data from SP500, The download of the data is always critical, if the initial dataset is misspecified the analysis will lead to wrong results anyways.
First we use the ticker provided by Wikipedia to get the names of the stocks needed for our analysis, however, some of them are misspecified, hence a manual correction is needed. To set up the Python environment we imported in the *adv_prog_proj* file the needed libraries to execute the code and then we imported it in the interface file as a library, in particular, to correctly setup the Python environment, the following libraries are needed:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
import datetime
from datetime import timedelta
from sklearn.preprocessing import PolynomialFeatur
from sklearn.linear_model import LinearRegression
poly_reg = PolynomialFeatures(degree=5)
import requests
from xml.etree import ElementTree
import nasdaqdatalink as quandl
import datetime
from scipy import interpolate

import bs4 as bs
```

These libraries are essential for running the code successfully. to correctly implement the GUI, the following impoertation are also needed:

```
import tkinter as tk
import adv_prog_proj as deriv
import numpy as np
from tkinter import ttk
```

Notice that we import the code of the function as a module in the script of the GUI.

We also set up the C++ and compile.sh file to allow the function in *adv_prog_proj* to take advantage of the speed of C++. We further explain how the code works in one of the following subsections.

2. Description of the research question and the relevant literature

In this project we focus on the Black-Scholes-Merton model. Under this model stocks prices are geometric brownian motions that evolve according to:

$$\frac{dS_t}{S_t} = (\mu - \delta)dt + \sigma dB_t \tag{1}$$

which has the solution:

$$S_t = e^{(\mu - \frac{\sigma^2}{2} - \delta)t + \sigma B_t} \tag{2}$$

and therefore stock log-return are normally distributed over any horizon according to:

$$log(\frac{S_{t+\Delta}}{S_t}) = \sigma(B_{t+\Delta} - B_t) + (\mu - \delta - \frac{\sigma^2}{2})\Delta \quad (3)$$

In these equations μ is the mean of the stock return, σ is the standard deviation of the stock return, δ is the dividend yield and B_t is a standard brownian motion. These values are assumed to be constants and therefore the model is widely used thanks to its ease. In fact it's possible to obtain a closed form solution for the price of put and call options. The closed formulas for the prices of put and call european options are respectively:

$$P = Ke^{-rT}\Phi(-d_2) - S_0\Phi(-d_1) \quad (4)$$

$$C = S_0\Phi(d_1) - Ke^{-rT}\Phi(d_2) \quad (5)$$

where:

S_0 : Current price of the underlying asset
K : Strike price of the option
r : Risk-free interest rate
T : Time to expiration of the option
Φ : Cumulative standard normal distribution function

$$d_1 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(r + \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}}$$

$$d_2 = d_1 - \sigma\sqrt{T}$$

On the other hand, this ease has a cost and the model present some issues:

- Constant μ and especially constant σ are not realistic. In the real world, volatility is time-varying and persistent due to volatility clustering.
- Normally distributed returns are not realistic too. It is possible to observe negative skewness and fat tails that deviate from this assumption. It would be better to have a jump-diffusion term to the stock to try to fix this problem.
- Continuous trading that is implicitly required by the model since options are priced by a replicating strategy that implies continuous trading. This assumption is becoming more and more realistic with the passage of time thanks to new technologies.
- Absence of transaction costs. As for continuous trading, this is implicitly assumed for the replicating strategy's construction.
- It is not possible to directly price path-dependent options through the Black-Scholes-Merton model
- Finally, it is also impossible to take into account the possibility of American style contract that can be exercised at any date before the maturity

Therefore, our project is based on this model but we wanted to try to avoid some of these restrictions for example using Montecarlo simulations that can help us to price both American style and path-dependent options.

This method consists in simulate a lot of different paths of the stock and the approximate an expected value through a summation:

$$E[\phi(S(T))] \approx \frac{1}{N}\sum_{i=1}^{N}\phi(S_N^{(i)}) \quad (6)$$

where:

N : Number of simulations
ϕ : A function (in our case the payoff function) that we apply to terminal value of the stock
$S(T)$: Value of the stock at time T
$S_N^{(i)}$: Value of the stock at the terminal date in path i

There are two two different types of errors in performing Montecarlo simulations for SDEs. The first one is the discretization error due to the numerical method used to approximate the SDE. In our case we assumed to know the exact solution of SDE of stock prices that is a geometric brownian motion under Black-Scholes assumptions. Therefore, our results will not be affected by that error. The second type of error is the statistical error that arises from the approximation of the expected value through a summation.

In our analysis, we decide not to consider the dividends component since they are pretty tedious to estimate.

The different types of derivatives we decide to price through Montecarlo simulations are:

- European call
- European put
- Asian call
- Asian put
- American call
- American put

We decided to focus on put and call options since they can be seen as the building blocks for all the other type of derivatives contract. In fact, for european option we can approximate any piecewise linear payoff on the underlying asset with a combination of the asset and put or call options written on that asset. If the payoff function is continuous, we can also interpolate it we can approximate it through a finite number of call and put option as well as the underlying asset.

3. The methodology/algorithm applied to address the research question and potentially its complexity

3.1. Interest rates

For discounting the payoffs we had to determine the appropriate interest rate. For this, we use a yield curve interpolation technique commonly used in the finance literature. Our methodology closely follows that proposed by [1]. We use the most recent yields on US Treasury bonds and fit a third-order polynomial that best captures the shape

of the yield curve. From the best-fitting polynomial, we then interpolate the implied forward rates which are given by the relationship (assuming one uses continuous discounting)

$$f(0;t_1,t_2) = \frac{r_2 t_2 - r_1 t_1}{t_2 - t_1}$$

In our algorithm the gap between each interpolated interest rate r_2 and r_1 is fixed to be the equal-sized time step we split a year into. That is if T denotes the number of years, N is the number of periods each year is split into hence $dt = \frac{T}{T \cdot N}$ is the time step between any two points of interpolation. With this, we can simplify the above expression to get

$$f(0;t_i,t_{i+1}) = r_{i+1} \cdot (i+1) - r_i \cdot i$$

Which is exactly the formula we use to get the implied forward rates.
To discount European and Asian options, we simply use the expression:

$$\delta = \exp\left[-\sum_{i=1}^{T \cdot N}\left(f_{i,i+1} \cdot \frac{1}{T \cdot N}\right)\right]$$

Which is consistent with continuous discounting. For the American options, the discount rate used for each period to discount the intrinsic value becomes

$$\delta_{i,i+1} = \exp\left(-dt \cdot f_{i,i+1}\right)$$

3.2. European options

The Montecarlo algorithm to price European options, both normal ones and Asian ones, consists in:

- Choose the number of simulations n that we want to perform.
- Create n realizations of the geometric Brownian motion path evaluated in a grid of points with the same time distance within each other between the starting date and the maturity date.
  Since we want to price also Asian options we can not only generate the terminal value of the stock price but we need all the history of the path.
  For this purpose, we need first to create a matrix of standard normal realizations and then perform the cumulative summation over the rows to obtain the matrix Z. Finally apply this formula:

$$S = S_0 \exp\left(\mu t - \frac{\sigma^2 t}{2} + \sigma\sqrt{dt}Z\right) \qquad (7)$$

  where:

$$dt : \text{Step size}$$

  we obtain the matrix of prices.
  Another way to obtain the matrix of prices would be to perform a double nested loop and populate the matrix according to:

$$S(i,j) = S(i,j-1)\exp((\mu - \frac{\sigma^2}{2})dt + \sigma\sqrt{dt}Z) \quad (8)$$

  where:

$$dt : \text{Step size}$$
$$Z : \text{Standard normal random variable}$$

- Evaluate the payoff function of the option at the terminal date for each path of the Brownian motion (for the European call $(S_T - K)^+$, for the European put $(K - S_T)^+$, for the Asian call $(K - mean(S_t))^+$, for the Asian put $(mean(S_t) - K)^+$, where with $mean(S_t)$ we refer to the mean over the time period considered).
- computes the mean of the terminal values over the n simulations and discount it to the starting time at the risk-free rate.

3.3. American options

The Montecarlo algorithm to price American options is more sophisticated and it is based on the the Longstaff-Schwartz model. In continuous time it is possible to exercise an American option at any time. Of course, it is not possible to perfectly model this but what the model takes into account is the possibility to exercise in a large, but still finite, number of times. Therefore, it is something similar to a Bermudan option but with a number of possible exercise dates that approach infinity. The main idea is to start from the terminal date and go backward in time. At each time we check whether the exercise value is greater than the continuation value, that is the value of holding the stock until the next possible exercising date instead of exercising it. In detail the algorithm consists in:

- Choose the number of simulations n that we want to perform.
- Create n realizations of the geometric Brownian motion path evaluated in a grid of points with the same time distance within each other in the same way as for the European options.
- Evaluate the payoff function of the option, which is the same as European call and put respectively, at the terminal time.
- Go backward and for each time calculate the exercise value and the continuation value. the continuation value is calculated as the fitted value of polynomial regression. At first, we run the following polynomial regression with degree=5 of the value at the next time period ($\Pi_{t+1,i}$) over a 5^{th} order polynomial of the stock price at the current period ($S_{t,i}$):

$$\Pi_{t+1,i} = \beta_0 + \sum_{j=1}^{5}\beta_j S_{t,i}^j \qquad (9)$$

  The fitted value of this polynomial regression discounted at the risk-free rate by one period will be the continuation value.
- The value of the stock will be the maximum between the exercise value and the continuation value.

- Going backward in this way it is possible to calculate the value of the option at the starting time for each realization of the path of the geometric Brownian motion.
- The value of the option at the starting date will be the average of the values of the option at period 0 over the n simulations. Doing this we take into account the possibility of exercising the option already at the starting time. If we want to avoid this we can calculate the value of the option as the mean of the values of the option in period 1 discounted by one period at the risk-free rate.

This algorithm follows the main steps explained in [2].

4. A discussion of the implementation of the algorithm (code) and, if possible, its parallel implementation and performance

4.1. Data download and estimation of parameters

To implement the algorithms to price these types of options in Python at first we used a function that returns a list containing all the tickers of the stocks in the S&P500 excluding the one for which is not possible to retrieve data from yahoo finance.

For the interest rate interpolation, we used two functions. The first function downloaded the most recent Overnight Bank Funding Rate from the New York Federal Reserve's website as a proxy for the one-day interest rate. The second function does multiple operations. Firstly, it downloads the most recent Par yield data for the other available maturities using the Nasdaq datalink package. After that, it combines the two data frames on the greatest common data and uses Scipy's interpolate package to fit the cubic polynomial to the interest rates of varying maturities. From the yield curve, we can then get the forward rates for the desired frequencies using the methodology defined in Section 3.1.

Then we created a function named inputs that takes the ticker of the stock chosen by the user and another parameter that represents how many years the user wants to go back in time to estimate the mean and standard deviation of the stock return. The function downloads from yahoo finance the prices of the stock over the period considered and calculates the return of the stock. It finally returns the annualized mean and volatility and the close price of the stock of the day when the interface is being used.

4.2. European options

The function for pricing European options takes as inputs:

- T: maturity date (years)
- N: number of time steps within each year
- n: number of simulations that is set to 100000
- sigma: the estimated volatility of the stock return (annualized)
- mu: the average return of the stock (annualized)
- S0: the current price of the stock
- K: the strike price
- type: a string with the type of the derivative contract
- mode: a string to decide if the matrix of stock prices will be generated on python or C++

Then if the mode is Python the function will create the matrix of stock prices using a (n,NT+1) matrix of standard normal and the np.cumsum method. While, if the mode is C++ the code will use the function BM of the BMmodule that we created using pybind11. This function creates the matrix of stock prices through a double nested for loop that in each iteration generates a standard random normal using the Box-Muller algorithm. If the user wants to compute the price of the option using mode='C++' he, firstly, needs to run the compile.sh file to create the BMmodule that will then be imported in Python.

Then the function will evaluate the payoff function at the terminal date and return the mean over the n simulations discounted at the risk-free rate.

4.3. American options

The function for American options takes the same inputs as the one for European options. It also computes the matrix of stock prices and the terminal value of the contract in the same way. The polynomial regression, used to compute the continuation value at each period, is run with the sklearn library. and the function returns the mean of the values of the option price at time 0 over all the n simulations.

4.4. Complexity

In Python we tried to avoid for loops, since they are quite inefficient. For instance, in order to achieve this to populate the matrix we directly created the matrix with all Gaussian distributed elements and then use element-wise operations to create the matrix of stock prices.

The complexity of the function for European derivatives is O(N*T*n) due to the double nested loops to populate the matrix of stock prices. Since the operations that we execute in the first loop, that is the loop over the n simulations, are independent that part of the code could be highly parallelized. On the other hand, it is not possible to parallelize the second loop since to perform every iteration we need to know the value of the previous iteration.

The algorithm for European option is quite fast also with an increasing number of simulations. On the other hand, the function to calculate the price of American options becomes quite slow if we increase the number of simulations too much. This is due to the polynomial regression that needs to be performed over n values and it becomes tedious. Therefore, we decided to set the maximum number of simulations to 100000 which seems a reasonable number to us. In this way, the user can get the results in a short time but we also provide a quite accurate

output.

All these functions are in the file `adv_prog_proj`. Then we import this in the file named `deriv_app` where we implemented the GUI.

4.5. C++ function

The C++ code it is up to create the module "BMmodule". The main goal of the code is to simulate a Geometric Brownian Motion (GBM) and return a NumPy array with the waited results. In the following a description how the code works:

- The code begins by including necessary C++ libraries and headers:
  - <algorithm>
  - <cmath>
  - <iostream>
  - <pybind11/pybind11.h>
  - <pybind11/stl.h>
  - <pybind11/complex.h>
  - <pybind11/numpy.h>
  - <array>
  - <complex>
- As seen in class we use a The namespace py = pybind11 to create a namespace alias for convenience.
- Gaussian Box-Muller Transformation: We want to return a standard normal random number taking advantage of the Box_muller transform. We do it with a function that generates two independent random variables and uses a while loop to calculate their Euclidean distance until it is less than one and then pass it to the Box_muller transform.
- GBM Simulation Function: It is similar to the function in Python: It takes as input various parameters, and in order to calculate the price, a random number is needed, hence in the loop, there is gaussian_box_muller() function that generates it. Lastly, based on the GBM formula the results are stored in a py.
- The PYBIND11_MODULE() macro creates a function that will be called when an import statement is issued from within Python

Once this C++ code is compiled it is possible to import it in Python as the "BMmodule" module. The user can choose to use the C++ function to perform simulations of GBM processes, in this case, the C++ code will pass the results as NumPy array.

4.6. PyBind11

PyBind11 allows us to access the C++ function *BMmodule*, giving Python the possibility to call C++ code and vice versa. We use PyBind11 rather than other libraries to join C++ code and Python because it is easy to implement and pretty spread and known methods. Here there some advantages we exploit:

- Pybind11 simply links C++ with Python and allows to match the higher performance of the first language with the ease of usage of the second one.
- It is automatic, once the compiled.sh runs, the conversion of C++ types to Python is immediate and without the need for special human intervention
- Easy to import in the environment. It can be included as a library in both Python and C++.
- Especially used for NumPy arrays: PyBind11 performs well when (like our case) we run operations on Numpy arrays., a popular numerical computing library in Python. It allows efficient exchange of data between C++ and NumPy arrays, enabling high-performance computations in C++ with the convenience of Python.

In general, PyBind11 make programmers' life easier, allowing them to enable C++ functions in Python code.

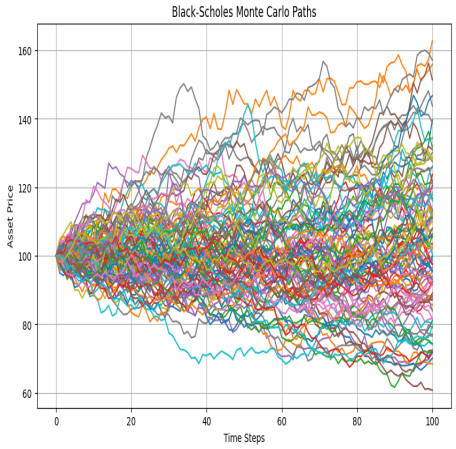
4.7. Montecarlo

For our project, the Montecarlo method played a main role. First, we want to give a definition of it: It is a computational technique that solves problems without a deterministic solution numerically. Going further into details, in our case we have used Montecarlo with the following steps:

- Firstly we simulate asset prices: Once the user has specified the needed parameters in the request, via Montecarlo the program generates, according to Black-Scholes SDE, a large number of random samples of the stock's price. (Look at the previous section to understand how the model works).
- Once we have the price's path, we can calculate the option payoff according to the user input.
- Discount the calculated payoffs with the risk-free rate calculated as described in the previous section back to the present value using an appropriate risk-free interest rate.
- We replicate this operation for a certain number of times and then we take the average, getting the option price.
- we can finally estimate summary statistics about the values we have found.

When we talk about numerical methods we must always remember that the approximation has an error associated. The aim of making a very high number of samples is one to have the smallest error possible. Typically the error estimated is around 10^{-5} but it can range depending on the parameters.
Looking at image 4.7 we can see 100 patterns of Montecarlo simulation with the following parameters:

- Initial asset price = 100
- Risk-free = 0.05
- Volatility = 0.2



- Time to maturity = 1
- Number of time steps =100

Remember that drift must be = 0 otherwise, the pattern diverges shortly.

4.8. GUI

Our group decided to use Python's built-in Tkinter package to provide a simple GUI for our application. The GUI can be called in the terminal by the command "python deriv_app.py" in the current working directory. This will prompt a pop-up window for the user as shown by Figure 1. To design the interface we used the Object-Oriented programming standard to make clean and interpretable code that makes it easy to expand. In the interface, the user specifies the inputs that will be used for the estimation of the option price.

Firstly through an option menu, the user can choose which programming language she would like to use to estimate the Geometric Brownian Motion price matrix that will be used for the Monte Carlo estimation. If the user specifies the "C++" option, then the estimation will be done through the "BMmodule" a library created by Pybind.
Secondly, the user can choose the ticker of the specific stock it wants to calculate the price of an option on from a second drop-down menu. The tickers include all the constituent stocks of the S&P500 index.
In the next three rows, the user inputs the number of years for which the option contract is active: T (float), the number of periods to split a year to: N (integer) and the number of simulated paths: n (integer) over which the price of the option will be calculated.
The user can then define the number of years over which the Geometric Brownian Motion parameters will be estimated over, by default this is set to 2 years.
In the last two dropdown menus, the user can choose between the three types of options, namely European, American and Asian and specify whether she would like to price a call or a put contract.
Finally, the strike price can be specified as a numeric value.

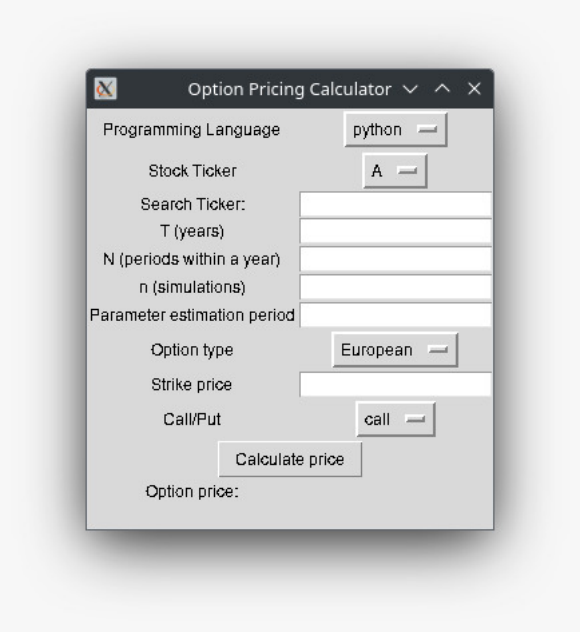


Figure 1. Tkinter GUI for the application

5. A description of how to maintain and update the codebase

One issue we address is the correction of the tickers of the stocks in SP500, the code should be periodically updated in order to avoid mistakes, moreover SP500 changes quite often, given that it represents the pool of the 500 largest publicly traded companies in the United States, so merge, acquisition or simply change in the value of a stock could change the composition of SP500.
Another issue worth to be considered is the possible update of python and its libraries if used on another version of the kernel or with a device with a different operating system there could be some incompatibility issues.

6. Results

In the results we want to present the graphical interface the user gets and some possible outputs, in particular, we want to show the different features the client can select and the error the program throws in case of a wrong parameter choice. Secondly, we want to do a comparison between what our program predicts as the option price and the option price found in yahoo finance, since the user can input many different parameters we do not expect the exact same price, however, we should have a relatively close value.

In the following section, we want to show some of the possible results, given a certain input:

Figure 2 shows the output of an American call on Apple. The code uses Python to calculate an option price of 3.23.

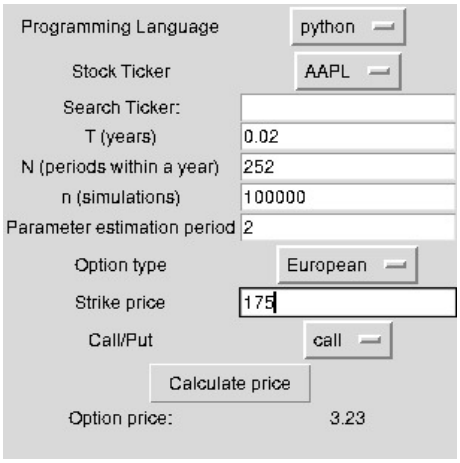


Figure 2. GUI Showcase: Option pricing with Python

Looking at figure 3, we can notice that the value computed by the program with C++ is the same that we found when the user chooses Python. We expect this result since the choice of language should have an impact only on the speed of execution.

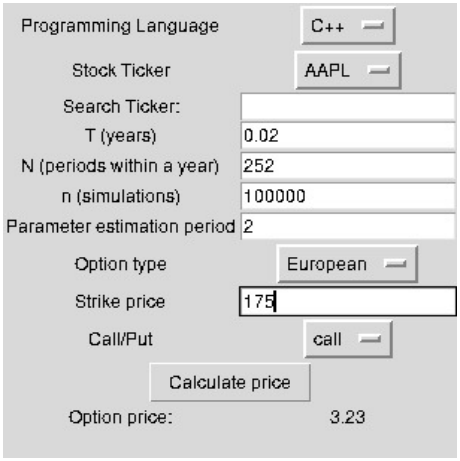


Figure 3. GUI Showcase: Option pricing with C++

Figure 4 shows the output of a European call on Apple. The code uses C++ to calculate an option price, however, the GUI displays the error *Number of simulation must be less than* 10^{-5} because the user is trying to run the simulation with more than 100000 iterations. In order to avoid the program to crash because of an excessive number of iterations the possible number is bounded.

Figure 5 shows the output of an American call on Applied Materials, Inc. The code uses Python to calculate an option price of 16.17.

Figure 6 shows the output of an Asian put on American Electric Power Company Inc. The code uses Python to calculate an option price of 9.53.

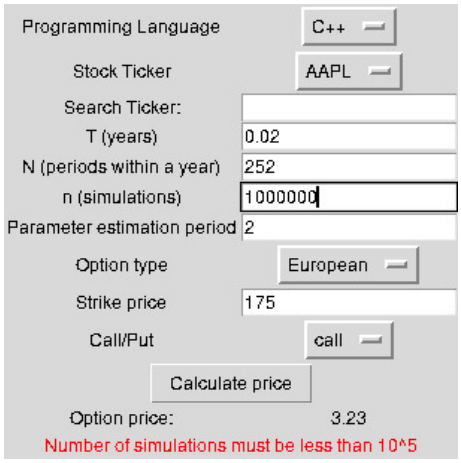


Figure 4. GUI Showcase: Error handling

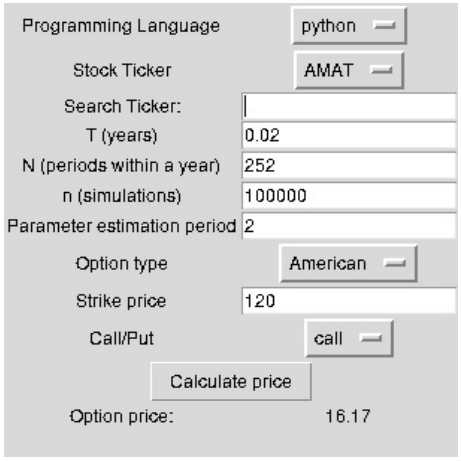


Figure 5. GUI Showcase: ARMAT American call

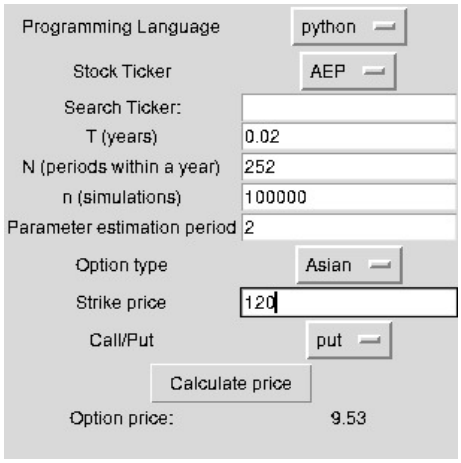


Figure 6. GUI showcase: Asian put

Figure 9 shows the market price of an American call option on AAPL with a strike price of 175 which expires on June 2nd. We can see that our estimate for the price of such a contract from Figure 3 before is quite close, despite the relative simplicity of the estimation method we use. Our algorithm overestimates the price of the call option which is potentially due to our formulation of the model which ignores dividend payouts. The Monte Carlo estimation method is also sensitive to the specified estimation period, hence choosing a larger or smaller estimation period for the Brownian parameters might reflect more accurately what the actual market uses.
For example re-doing the results with a parameter estimation period of just half a year moves us very close to the actual market value. Nevertheless fine-tuning the precise period estimation is one of the profound problems in derivative pricing and hence it is out of the scope of this project.

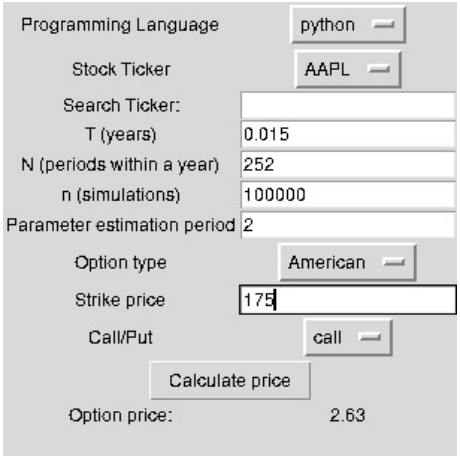


Figure 7. AAPL call option estimation (2 years)

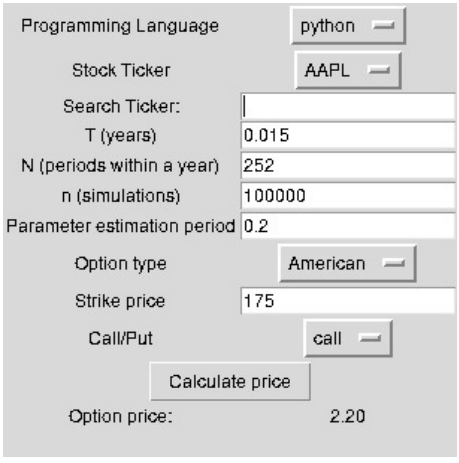


Figure 8. AAPL call option estimation (0.2 years)

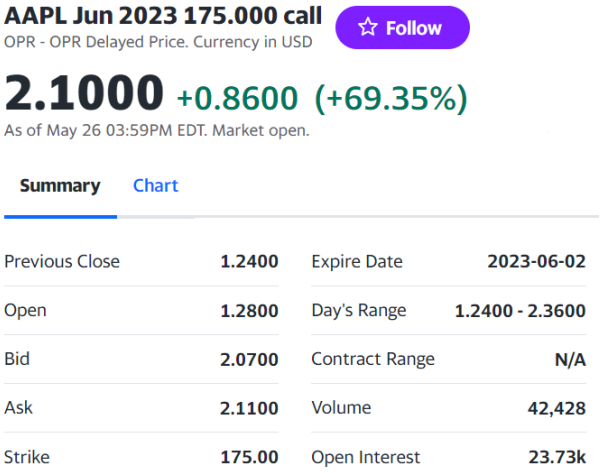


Figure 9. AAPL call option price

7. Conclusion and final remarks

In conclusion, our aim was the development of an algorithm for calculating the prices of commonly traded options, having as a pool of stocks the ones included in SP500. Despite some of the limitations of the Black-Scholes-Merton model, we choose it as it is the most commonly used benchmark for pricing the options. In the future, we are looking to enhance this model with a number of extensions.

In order to price derivative contracts effectively one needs to have an appropriate model for price movements which incorporates dividends and share repurchases which could distort the Brownian parameters.
Furthermore, for the interest rate, we used a simple interpolation method to calculate the implied forward rates for discounting. As we discussed getting the forward rates aligned with market expectations is especially important for American options, for which the forward rates are used explicitly in the recursive valuation. These limitations of our model may be addressed by future research.

To make the numerical estimation of the results, the Monte Carlo method is implemented to get the prices of derivatives. We also took advantage of the parallelization to improve the efficiency of the code. To improve user choices, there is a visual interface that allows the user to set the needed parameters such as the number of steps per year, the stock price, and the starting date for price estimation. Even if the code could offer even more flexibility, according to us, the possible choices offered to the user are already a good starting point. Overall, the paper contributes to the understanding of derivative pricing using the Black-Scholes-Merton model and Monte Carlo simulation. It combines theoretical concepts with practical implementation, allowing users to estimate option prices effectively and facilitating further investigation in the field of derivatives.

References

[1] P. Hagan and G. West, “Methods for constructing a yield curve,” 01 2008.

[2] C. Pirrong, “Using monte carlo to value derivatives with early exercise,” April 2020.