

C++ PROJECT: Implementing Exotic Option Valuation:

BarrierOption Up_and_Out and BarrierOption Down_and_Out using Monte Carlo Simulation in C++

Barrier Options

Barrier options are a Category of exotic option where the payoff depends not only on the underlying asset price at expiration but also on whether the asset price breaches a predefined barrier level during the life of the option.

Knock-Out Options

- A **knock-out option** ceases to exist (becomes worthless) if the underlying asset price crosses the barrier level during its lifetime.
 - **Up-and-Out Call**: A call option where the option becomes void if the underlying price rises **above** a certain barrier.
 - **Down-and-Out Call**: A call option where the option becomes void if the underlying price falls **below** a certain barrier.

Knock-In Options

- A **knock-in option** only comes into existence if the underlying asset price crosses the barrier level during its lifetime.
 - **Up-and-In Call**: A call option where the option is activated if the underlying price rises **above** a certain barrier.
 - **Down-and-In Call**: A call option where the option is activated if the underlying price falls **below** a certain barrier.

Monte Carlo Approach

Monte Carlo methods simulate the price paths of the underlying asset to estimate the value of these options.

Monte Carlo Up-and-Out Call Option

- The option starts as valid.
- If at any time during the simulation the underlying price **exceeds** the barrier:
 - The option is immediately **knocked out** (worth zero).
- If the barrier is never breached:
 - The payoff at expiration is: $\max(S_T - K, 0)$

where S_T is the terminal price of the underlying asset, and K is the strike price.

Monte Carlo Down-and-Out Call Option

- The option starts as valid.
- If at any time during the simulation the underlying price **falls below** the barrier:
 - The option is immediately **knocked out** (worth zero).
- If the barrier is never breached:
 - The payoff at expiration is: $\max(S_T - K, 0)$

Monte Carlo Up-and-In Call Option

- The option starts as **inactive**.
- If at any time during the simulation the underlying price **exceeds** the barrier:
 - The option is **activated** and behaves like a standard call option.
- If the barrier is never breached:
 - The option remains worthless.

Monte Carlo Down-and-In Call Option

- The option starts as **inactive**.
- If at any time during the simulation the underlying price **falls below** the barrier:
 - The option is **activated** and behaves like a standard call option.
- If the barrier is never breached:
 - The option remains worthless.

Key Factors in Monte Carlo Simulations

1. Barrier Monitoring Frequency:

- Continuous Monitoring: The barrier is checked at every time step of the simulation (most realistic).
- Discrete Monitoring: The barrier is checked at specific intervals (e.g., daily, weekly).

2. Path Dependency:

- Barrier options are path-dependent, meaning the entire price path of the underlying asset matters, not just the terminal price.

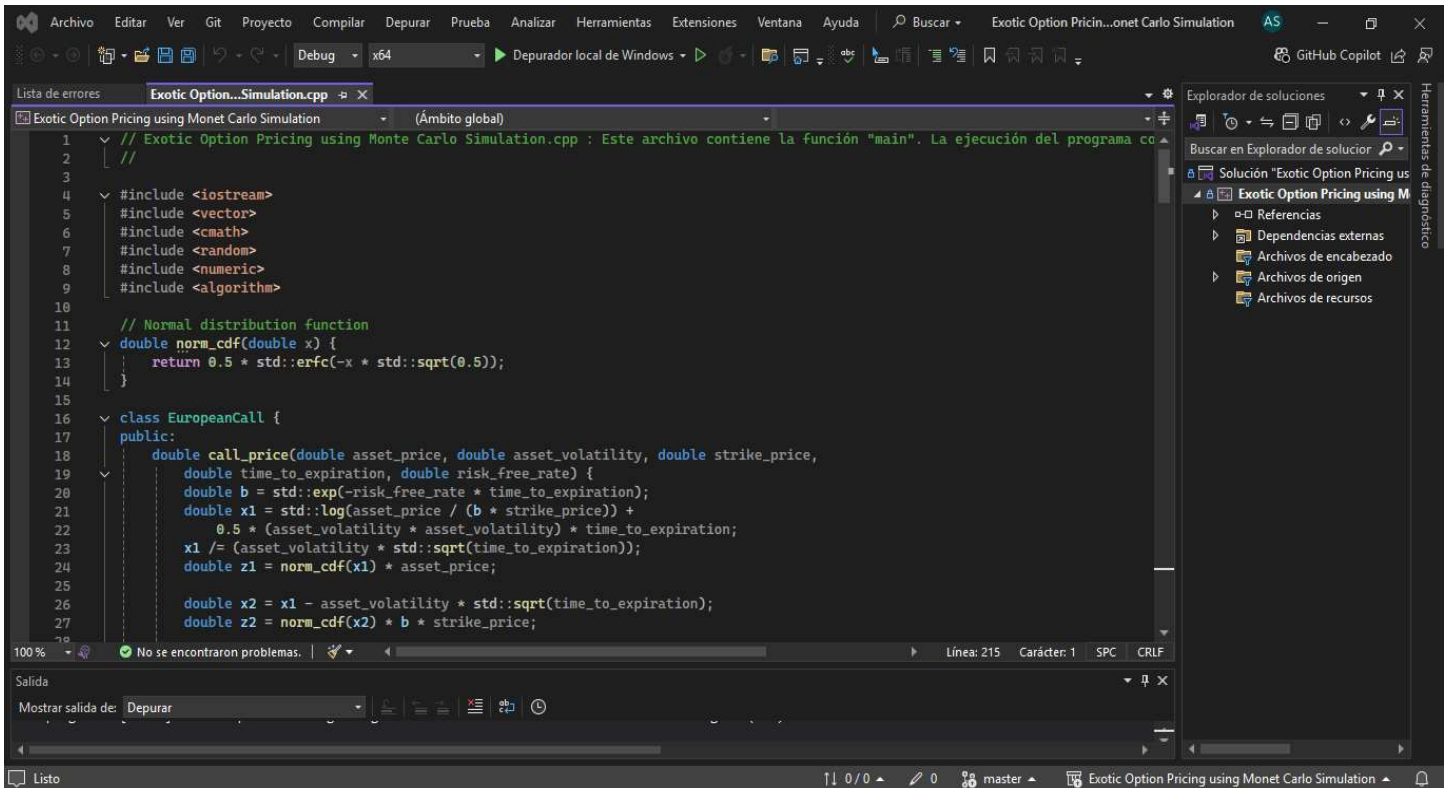
3. Numerical Convergence:

- A large number of simulated paths are required for convergence to an accurate estimate.

4. Discounting:

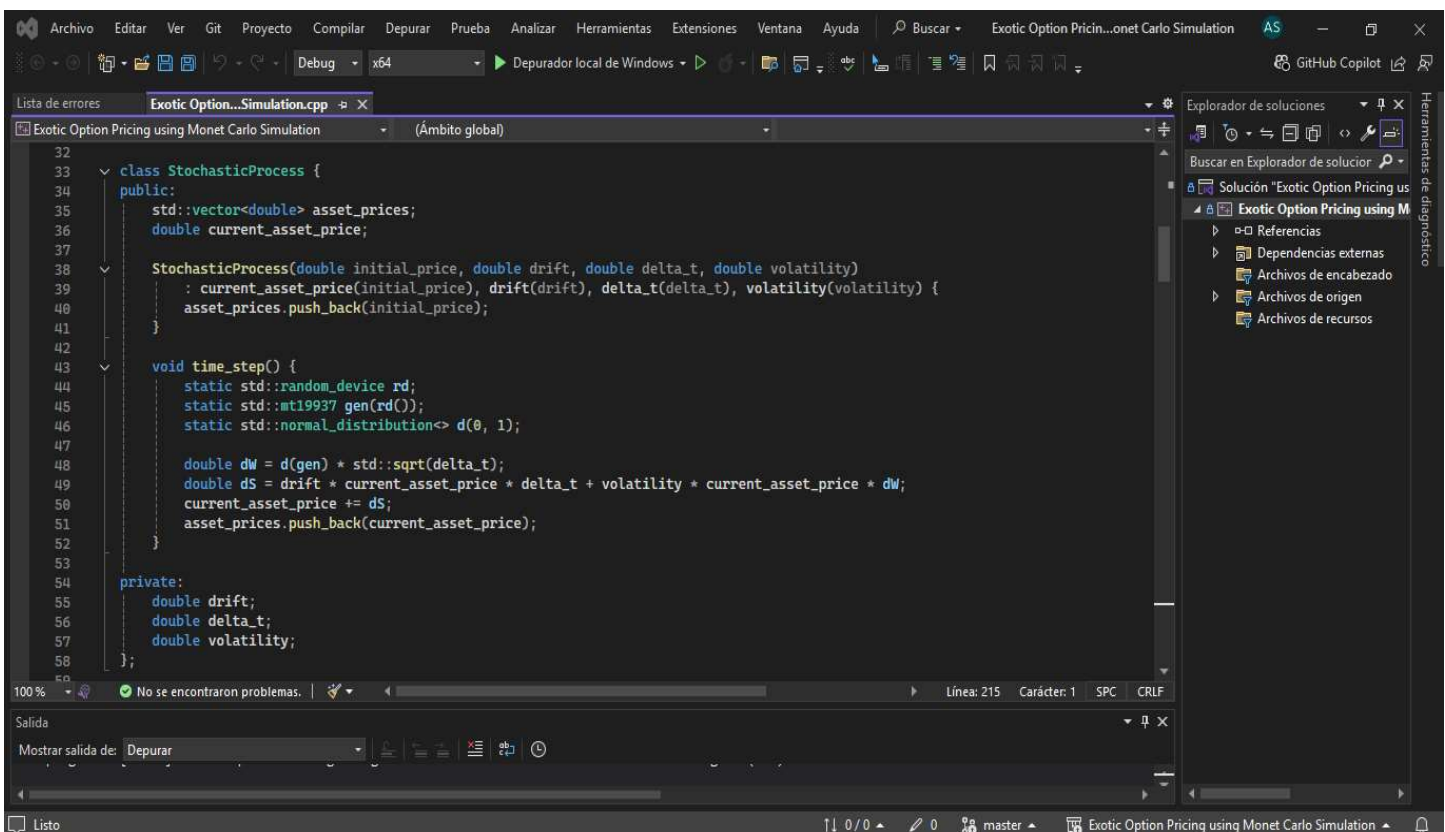
- The expected payoff is discounted to the present value using the risk-free rate:
Option Price = e^{-rT} * Expected Payoff

IMPLEMENTATION in C++ Using Visual Studio Code Editor



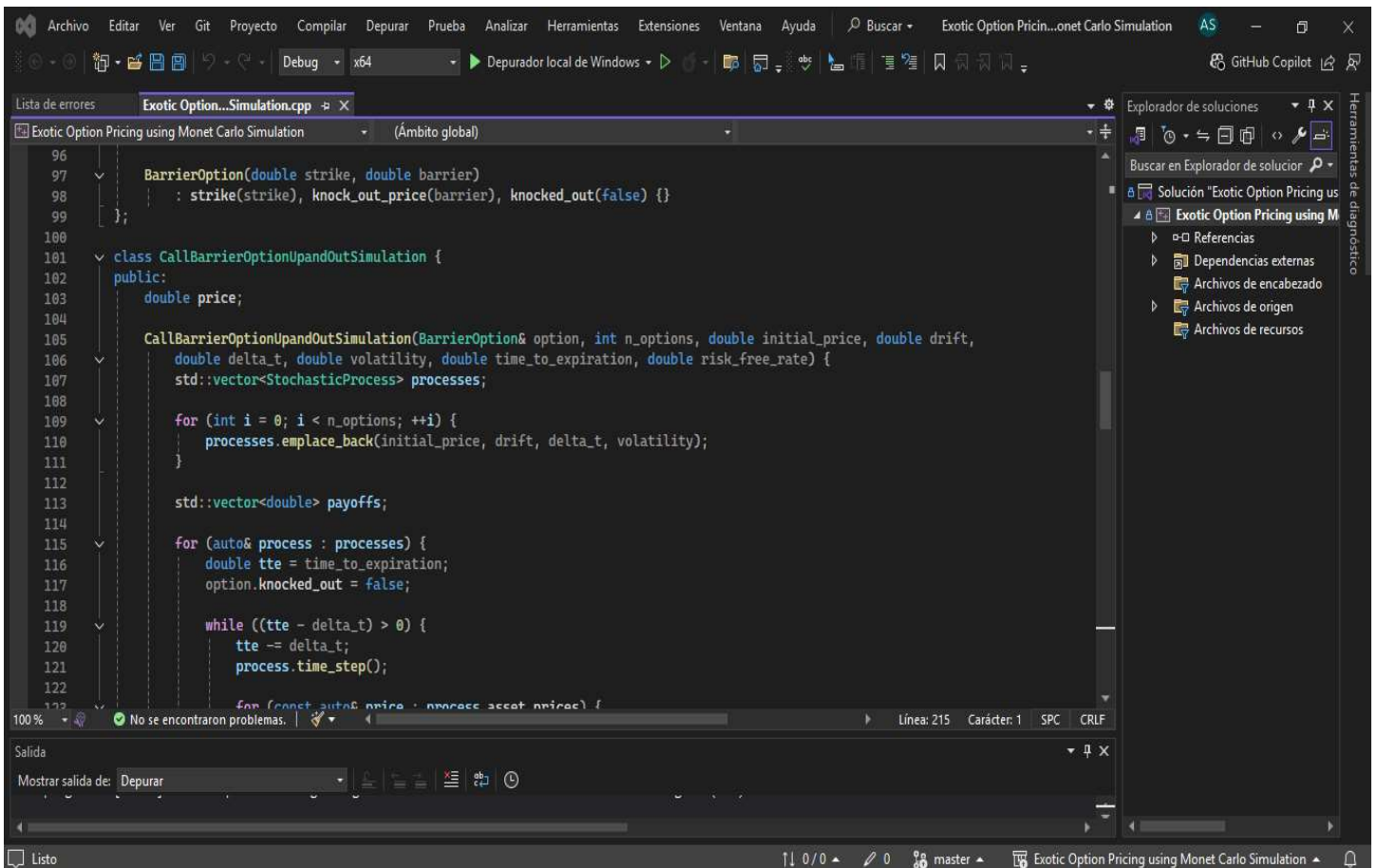
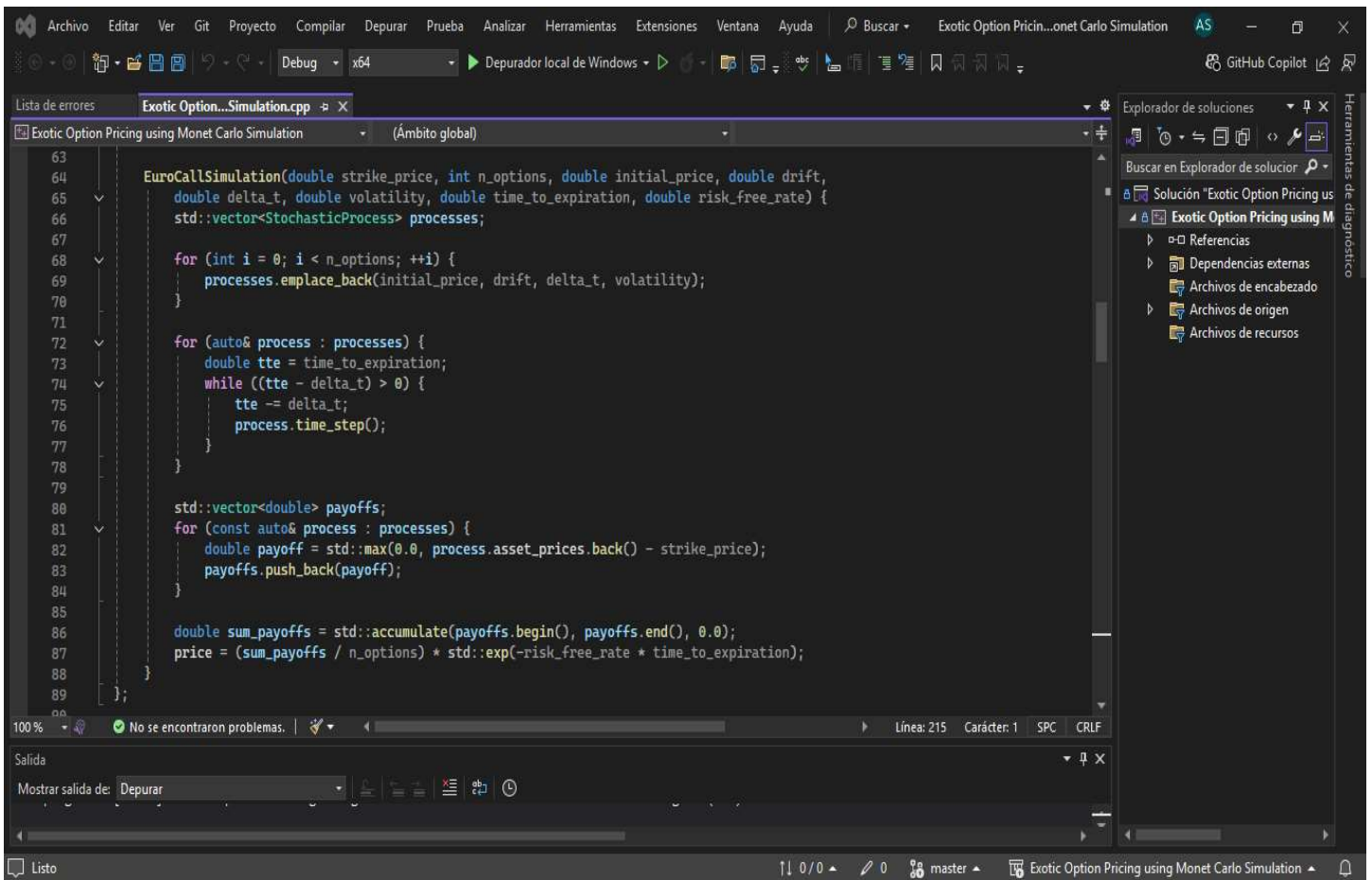
The screenshot shows the Visual Studio Code editor with the file `Exotic Option Pricing using Monet Carlo Simulation.cpp` open. The code implements a normal distribution function and a European Call option pricing function. The `norm_cdf` function uses the error function to calculate the cumulative distribution function. The `EuropeanCall` class has a public method `call_price` that calculates the option price based on asset price, volatility, strike price, time to expiration, and risk-free rate.

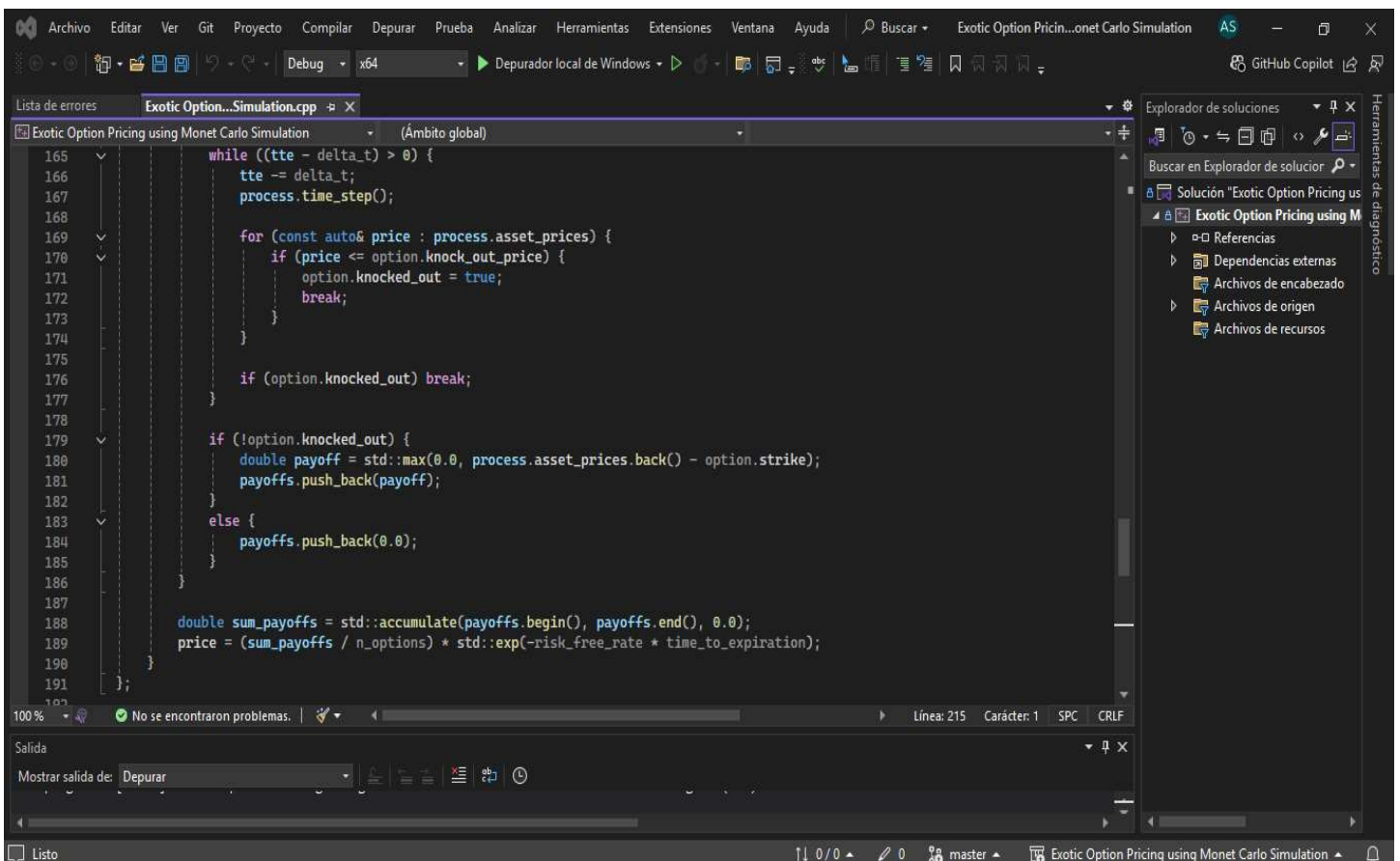
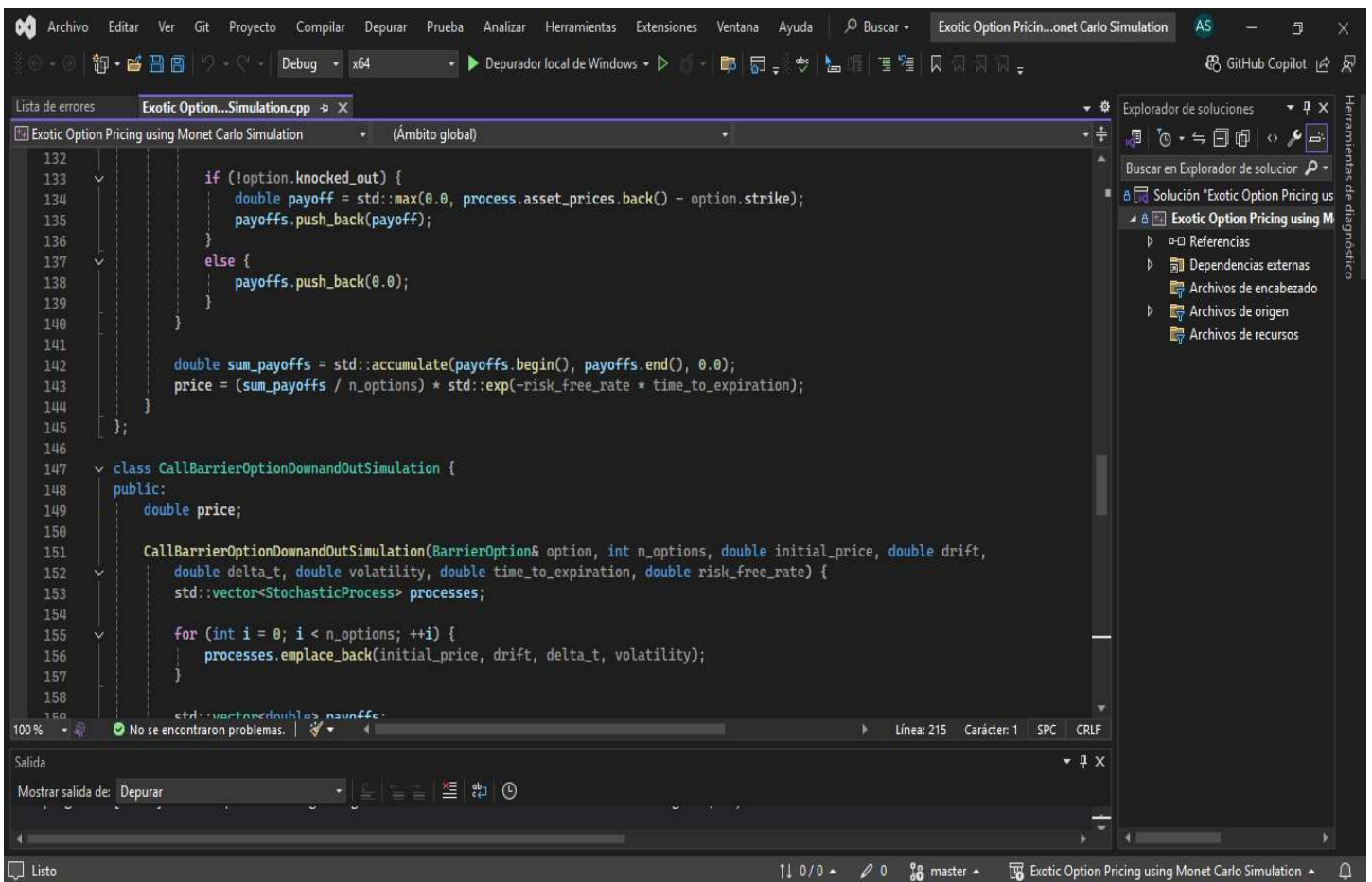
```
1 // Exotic Option Pricing using Monte Carlo Simulation.cpp : Este archivo contiene la función "main". La ejecución del programa comienza y termina en este archivo.
2 //
3
4 #include <iostream>
5 #include <vector>
6 #include <cmath>
7 #include <random>
8 #include <numeric>
9 #include <algorithm>
10
11 // Normal distribution function
12 double norm_cdf(double x) {
13     return 0.5 * std::erfc(-x * std::sqrt(0.5));
14 }
15
16 class EuropeanCall {
17 public:
18     double call_price(double asset_price, double asset_volatility, double strike_price,
19         double time_to_expiration, double risk_free_rate) {
20         double b = std::exp(-risk_free_rate * time_to_expiration);
21         double x1 = std::log(asset_price / (b * strike_price)) +
22             0.5 * (asset_volatility * asset_volatility) * time_to_expiration;
23         x1 /= (asset_volatility * std::sqrt(time_to_expiration));
24         double z1 = norm_cdf(x1) * asset_price;
25
26         double x2 = x1 - asset_volatility * std::sqrt(time_to_expiration);
27         double z2 = norm_cdf(x2) * b * strike_price;
```



The screenshot shows the Visual Studio Code editor with the file `Exotic Option Pricing using Monet Carlo Simulation.cpp` open. The code implements a `StochasticProcess` class that simulates the evolution of an asset price over time using a random walk. The class has a public method `time_step` that updates the asset price based on drift, volatility, and a random number generated from a standard normal distribution.

```
32
33 class StochasticProcess {
34 public:
35     std::vector<double> asset_prices;
36     double current_asset_price;
37
38     StochasticProcess(double initial_price, double drift, double delta_t, double volatility)
39         : current_asset_price(initial_price), drift(drift), delta_t(delta_t), volatility(volatility) {
40         asset_prices.push_back(initial_price);
41     }
42
43     void time_step() {
44         static std::random_device rd;
45         static std::mt19937 gen(rd());
46         static std::normal_distribution<> d(0, 1);
47
48         double dW = d(gen) * std::sqrt(delta_t);
49         double dS = drift * current_asset_price * delta_t + volatility * current_asset_price * dW;
50         current_asset_price += dS;
51         asset_prices.push_back(current_asset_price);
52     }
53
54 private:
55     double drift;
56     double delta_t;
57     double volatility;
58 };
59
```





```
192
193 int main() {
194     // Black-Scholes European Call price
195     EuropeanCall call_option;
196     double call_price = call_option.call_price(296, 0.234, 295, 1.0 / 12.0, 0.08);
197     std::cout << "Black-Scholes European Call price: " << call_price << std::endl;
198
199     // Monte Carlo Euro Call price
200     EuroCallSimulation euro_sim(296, 10000, 295, 0, 1.0 / 365.0, 0.2435, 39.0 / 365.0, 0.0017);
201     std::cout << "Monte Carlo Euro Call price: " << euro_sim.price << std::endl;
202
203     // Monte Carlo Up-and-Out Call price
204     BarrierOption up_and_out(296, 330);
205     CallBarrierOptionUpandOutSimulation up_and_out_sim(up_and_out, 10000, 295, 0, 1.0 / 365.0, 0.2435, 39.0 / 365.0, 0.0017);
206     std::cout << "Monte Carlo Up and Out Call: " << up_and_out_sim.price << std::endl;
207
208     // Monte Carlo Down-and-Out Call price
209     BarrierOption down_and_out(296, 29);
210     CallBarrierOptionDownandOutSimulation down_and_out_sim(down_and_out, 10000, 295, 0, 1.0 / 365.0, 0.2435, 39.0 / 365.0, 0.0017);
211     std::cout << "Monte Carlo Down and Out Call: " << down_and_out_sim.price << std::endl;
212
213     return 0;
214 }
215
```

RESULTS Below

```
> functions
> navbar
> images
> pages
JS AboutUs.js
JS Developers.js
JS Footer.js
JS Header.js
JS Join.js
JS Partners.js
JS Properties.js
JS Subscribe.js
# all.min.css
S App.js
S index.js
# style.css
webfonts
package-lock.json
package.json
README.md

18
19 //La fonction norm_cdf est utile pour des applications statistiques et probabilist
20 // ** Les modèles financiers (comme la valorisation d'options avec le modèle Black
21 // ** Les calculs probabilistes pour des phénomènes suivant une loi normale.
22 // ** Les tests statistiques utilisant des scores Z.
23
24 // Class for European Call and Put Option Pricing
25 class EuropeanOption {
26 public:
27     // Function to calculate the price of a European call option
28     double call_price(double asset_price, double asset_volatility, double strike_p
29         double time_to_expiration, double risk_free_rate) {
30         double b = std::exp(-risk_free_rate * time_to_expiration);
31         double x1 = std::log(asset_price / strike_price) +
32             0.5 * (asset_volatility * asset_volatility) * time_to_expirati

PROBLEMS 52 OUTPUT DEBUG CONSOLE TERMINAL PORTS JUPYTER COMMENTS
Monte Carlo Down and Out Call: 10.5444
PS C:\Users\kamel\node\renting_platform1> ^C
PS C:\Users\kamel\node\renting_platform1>
PS C:\Users\kamel\node\renting_platform1> & 'c:\Users\kamel\.vscode\extensions\ms-vscode.cpptool
-rce5act2.syz' '--stdout=Microsoft-MIEngine-Out-v2121r11.4jp' '--stderr=Microsoft-MIEngine-Error-
exe' '--interpreter=mi'
Black-Scholes European Call price: 9.50308
Monte Carlo Euro Call price: 10.4186
Monte Carlo Up and Out Call: 4.88681
Monte Carlo Down and Out Call: 10.2046
PS C:\Users\kamel\node\renting_platform1> ^C
PS C:\Users\kamel\node\renting_platform1>
PS C:\Users\kamel\node\renting_platform1> & 'c:\Users\kamel\.vscode\extensions\ms-vscode.cpptool
-or5n3fqe.oqi' '--stdout=Microsoft-MIEngine-Out-3fdageye.lyf' '--stderr=Microsoft-MIEngine-Error-
exe' '--interpreter=mi'
Black-Scholes European Call price: 9.50308
Monte Carlo Euro Call price: 10.5771
Monte Carlo Up and Out Call: 4.77273
Monte Carlo Down and Out Call: 10.1583
PS C:\Users\kamel\node\renting_platform1>
```