Pricing Derivatives Using Black-Scholes-Merton Model

Rafiq Islam

2023-11-12

Table of contents

## Introduction

In this blog, we will explore how to price simple equity derivatives using the Black-Scholes-Merton (BSM) model. We will derive the mathematical formula and then provide Python code to implement it.

### Background and Preliminaries

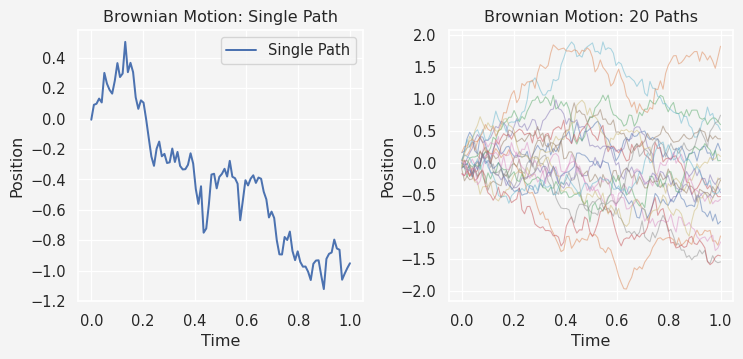
Before proceeding to the deep of the discussion, we need to know some definition and terminology

**Brownian Motion:** Brownian motion is a concept with definitions and applications across various disciplines, named after the botanist Robert Brown, is the random, erratic movement of particles suspended in a fluid (liquid or gas) due to their collisions with the fast-moving molecules of the fluid.

*Brownian motion is a stochastic process defined as a continuous-time process with the following properties:*

* almost surely.
* has independent increments.
* For , (normally distributed with mean 0 and variance ).
* has continuous paths almost surely.

from mywebstyle import plot\_style  
plot\_style('#f4f4f4')  
import numpy as np  
import matplotlib.pyplot as plt  
  
# Parameters  
n\_steps = 100 # Number of steps  
n\_paths = 20 # Number of paths  
time\_horizon = 1 # Total time  
dt = time\_horizon / n\_steps # Time step  
t = np.linspace(0, time\_horizon, n\_steps) # Time array  
  
# Generate Brownian motion  
def generate\_brownian\_paths(n\_paths, n\_steps, dt):  
 # Standard normal increments scaled by sqrt(dt)  
 increments = np.random.normal(0, np.sqrt(dt), (n\_paths, n\_steps))  
 # Cumulative sum to generate paths  
 return np.cumsum(increments, axis=1)  
  
# Generate one path and multiple paths  
single\_path = generate\_brownian\_paths(1, n\_steps, dt)[0]  
multiple\_paths = generate\_brownian\_paths(n\_paths, n\_steps, dt)  
  
# Plotting  
fig, axes = plt.subplots(1, 2, figsize=(7.9, 3.9))  
  
# Single path  
axes[0].plot(t, single\_path, label="Single Path")  
axes[0].set\_title("Brownian Motion: Single Path")  
axes[0].set\_xlabel("Time")  
axes[0].set\_ylabel("Position")  
axes[0].legend()  
  
# Multiple paths  
for path in multiple\_paths:  
 axes[1].plot(t, path, alpha=0.5, linewidth=0.8)  
axes[1].set\_title(f"Brownian Motion: {n\_paths} Paths")  
axes[1].set\_xlabel("Time")  
axes[1].set\_ylabel("Position")  
  
plt.tight\_layout()  
plt.show()



**Geometric Brownian Motion (GBM)**  
A stochastic process is said to follow a geometric Brownian motion if it satisfies the following equation:

Which can be written as

To solve the GBM, we apply Ito’s formula to the function and then by Taylor’s expansion, we have

By definition we have

The term is negligible compared to the term and it is also assume that the product is negligible. Furthermore, the quadratic variation of i.e., . With these values, we obtain

with . Now we have the following

## Black-Scholes-Merton Formula

Now we are ready to derive the BSM PDE. The payoff of an *option* at maturity is is known. To find the value at an earlier stage, we need to know how V behaves as a function of and . By Ito’s lemma we have

Now let’s consider a portfolio consisting of a short one option and long shares at time . The value of this portfolio is

over the time , the total profit or loss from the changes in the values of the portfolio is

Now by the discretization we have,

At this point, if is the risk-free interest rate then we will have following relationship

The rationale of this relation is that no-aribtrage assumption. Thus, we have

This is the famous Black-Scholes-Merton PDF, formally written with the boundary conditions as follows

This Black-Scholes-Merton PDE can be reduced to the heat equation using the substitutions , , and . Let’s derive the solution step by step in full mathematical detail and show how this leads to the normal CDF.

#### Step 1: Substitutions

We aim to reduce the BSM PDE:

to the heat equation. Using the substitutions:

* , where , and maps ,
* , so ,
* , where is the transformed function.

#### Step 2: Derivative Transformations

For , we compute derivatives.

1. The first derivative of with respect to :

* where implies . Thus:

1. The second derivative of with respect to :

* Using the product rule:

1. The time derivative:

#### Step 3: Transforming the PDE

Substituting the above derivatives into the BSM PDE, we rewrite each term.

1. For :
2. For :
3. For :

Substituting all these into the BSM PDE:

Divide through by :

To simplify, let , where and are constants. Substituting and choosing and , the equation reduces to:

#### Step 4: Solving the Heat Equation

The heat equation has a well-known solution using Fourier methods:

where is the initial condition.

For the BSM problem, the initial condition is the payoff:

Performing the integration leads to the final solution involving the cumulative normal distribution function:

where:

Transforming back to the original variables gives the Black-Scholes formula:

where:

Similarly, we can derive the price of a European put option:

Where:

### Asymptotic Behavior of the BSM formula for call and put options

What if ? In that case,

1. , causing and
2. The cdf and
3. The second term as

In this case, the price of a call option and the price of a put option

## Greeks: Delta and Gamma

**Delta** () is the sensitivity of the option price to changes in the underlying asset price:

This is the **central difference approximation**, which provides a more accurate estimate of delta compared to the forward or backward difference methods.

* : Calculate the option price with the spot price increased by .
* : Calculate the option price with the spot price decreased by .

**Gamma** () measures the rate of change of delta with respect to the underlying asset price:

Gamma () measures the rate of change of delta () with respect to the underlying spot price ().

* : Option price with the spot price increased by .
* : Option price at the current spot price.
* : Option price with the spot price decreased by .

**Relationship Between Delta and Gamma:**

* Gamma represents how much delta changes for a small change in .
* If gamma is high, delta is more sensitive to changes in , which is important for hedging strategies.

## Implementation

### Notation

* : Spot price of the stock.
* : Strike price of the option.
* : Time to maturity (in years).
* : Risk-free rate (continuously compounded).
* : Dividend yield (continuously compounded).
* : Volatility of the stock.
* : Cumulative distribution function of the standard normal distribution.

from dataclasses import dataclass  
import numpy as np  
from scipy.stats import norm  
  
@dataclass  
class Equity:  
 spot: float  
 dividend\_yield: float  
 volatility: float  
  
@dataclass  
class EquityOption:  
 strike: float  
 time\_to\_maturity: float  
 put\_call: str  
  
@dataclass  
class EquityForward:  
 strike: float  
 time\_to\_maturity: float  
  
def bsm(underlying: Equity, option: EquityOption, rate: float) -> float:  
 S = underlying.spot  
 K = option.strike  
 T = option.time\_to\_maturity  
 r = rate  
 q = underlying.dividend\_yield  
 sigma = underlying.volatility  
  
 # Handle edge case where strike is effectively zero  
 if K < 1e-8:  
 if option.put\_call.lower() == "call":  
 return S   
 else:  
 return 0.0  
  
 d1 = (np.log(S / K) + (r - q + 0.5 \* sigma\*\*2) \* T) / (sigma \* np.sqrt(T))  
 d2 = d1 - sigma \* np.sqrt(T)  
  
 if option.put\_call.lower() == "call":  
 price = S \* np.exp(-q \* T) \* norm.cdf(d1) \  
 - K \* np.exp(-r \* T) \* norm.cdf(d2)  
 elif option.put\_call.lower() == "put":  
 price = K \* np.exp(-r \* T) \* norm.cdf(-d2) \  
 - S \* np.exp(-q \* T) \* norm.cdf(-d1)  
 else:  
 raise ValueError("Invalid option type. Must be 'call' or 'put'.")  
  
 return price  
  
def delta(underlying: Equity, option: EquityOption, rate: float) -> float:  
 bump = 0.01 \* underlying.spot  
 bumped\_up = Equity(spot=underlying.spot + bump,   
 dividend\_yield=underlying.dividend\_yield,   
 volatility=underlying.volatility)  
 bumped\_down = Equity(spot=underlying.spot - bump,   
 dividend\_yield=underlying.dividend\_yield,   
 volatility=underlying.volatility)  
 price\_up = bsm(bumped\_up, option, rate)  
 price\_down = bsm(bumped\_down, option, rate)  
 return (price\_up - price\_down) / (2 \* bump)  
  
def gamma(underlying: Equity, option: EquityOption, rate: float) -> float:  
 bump = 0.01 \* underlying.spot  
 bumped\_up = Equity(spot=underlying.spot + bump,   
 dividend\_yield=underlying.dividend\_yield,   
 volatility=underlying.volatility)  
 bumped\_down = Equity(spot=underlying.spot - bump,   
 dividend\_yield=underlying.dividend\_yield,   
 volatility=underlying.volatility)  
 original\_price = bsm(underlying, option, rate)  
 price\_up = bsm(bumped\_up, option, rate)  
 price\_down = bsm(bumped\_down, option, rate)  
 return (price\_up - 2 \* original\_price + price\_down) / (bump\*\*2)  
  
def fwd(underlying: Equity, forward: EquityForward, rate: float) -> float:  
 S = underlying.spot  
 K = forward.strike  
 T = forward.time\_to\_maturity  
 r = rate  
 q = underlying.dividend\_yield  
 forward\_price = S \* np.exp((r - q) \* T) - K  
  
 return forward\_price  
  
def check\_put\_call\_parity(  
 underlying: Equity,   
 call\_option: EquityOption,   
 put\_option: EquityOption,   
 rate: float  
 ) -> bool:  
  
 call\_price = bsm(underlying, call\_option, rate)  
 put\_price = bsm(underlying, put\_option, rate)  
 S = underlying.spot  
 K = call\_option.strike  
 T = call\_option.time\_to\_maturity  
 r = rate  
 q = underlying.dividend\_yield  
  
 parity\_lhs = call\_price - put\_price  
 parity\_rhs = S \* np.exp(-q \* T) - K \* np.exp(-r \* T)  
  
 return np.isclose(parity\_lhs, parity\_rhs, atol=1e-4)

### Example Usage

Say, we want to price a call option on an equity with spot price with dividend yield , and volatility . The strike price of the call is , with time to maturity in years and the risk free rate . Next, we want to see the asymptotic behavior of the call option if the strike price with interest rate 0. Next, we want to price a put option on the same equity but strike price , time to maturity in years and interest rate is 0. Finally, we want to check if the put-call parity relationship is hold. In each case, we consider a bump or small change in the stock price.

if \_\_name\_\_ == "\_\_main\_\_":  
 eq = Equity(450, 0.014, 0.14)  
 option\_call = EquityOption(470, 0.23, "call")  
 option\_put = EquityOption(500, 0.26, "put")  
   
 print(bsm(eq, option\_call, 0.05))   
 print(bsm(eq, EquityOption(1e-15, 0.26, "call"), 0.0))   
 print(bsm(Equity(450, 0.0, 1e-9), option\_put, 0.0))   
  
 # Check put-call parity  
 eq = Equity(450, 0.015, 0.15)  
 option\_call = EquityOption(470, 0.26, "call")  
 option\_put = EquityOption(470, 0.26, "put")  
 print(check\_put\_call\_parity(eq, option\_call, option\_put, 0.05))

5.834035584709966  
450  
50.0  
True

## References

* Karatzas, I., & Shreve, S. E. (1991). *Brownian Motion and Stochastic Calculus*.
* Options, Futures, and Other Derivatives by John C. Hull
* Arbitrage Theory in Continuous Time Book by Tomas Björk

**Share on**

**You may also like**