Imperial College
London

Python for Black–Scholes: Implementing Option Pricing

Session 3

From Theory to Code

```
$ python black_scholes.py
```

Imperial College
London

The Black–Scholes Formula in Python

Mathematical Formula:

$$C(t, S) = S \times \Phi(d_1) - K \times e^{-r(T-t)} \times \Phi(d_2)$$

where:

$$d_1 = [\ln(S/K) + (r + 0.5\sigma^2)(T-t)] / (\sigma\sqrt{(T-t)})$$

$$d_2 = d_1 - \sigma\sqrt{(T-t)}$$

Python Implementation Strategy:

- Import libraries: `numpy` for math, `scipy.stats` for Φ
- Calculate d_1 and d_2 step by step
- Use `norm.cdf()` for cumulative normal distribution Φ
- Combine components to get the call price

Key Python Functions:

- `np.log()` for natural logarithm
- `np.exp()` for exponential
- `np.sqrt()` for square root
- `norm.cdf()` for cumulative normal distribution Φ

Imperial College
London

Implementing Black–Scholes Call Price

Complete Python Function:

```
import numpy as np
from scipy.stats import norm
def black_scholes_call(S, K, T, r, sigma):
    """
    Calculate European call option price using Black-Scholes.
    Parameters:
    S: current stock price
    K: strike price
    T: time to maturity (in years)
    r: risk-free interest rate
    sigma: volatility
    """
    # Calculate d1
    d1 = (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    # Calculate d2
    d2 = d1 - sigma * np.sqrt(T)
    # Calculate call price
    call_price = S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
    return call_price
```

Note: Pay attention to indentation - function body uses 4 spaces.

Imperial College
London

Testing the Black–Scholes Function

Example Calculation:

```
# Option parameters
S0 = 100 # Current stock price: £100
K = 105 # Strike price: £105
T = 0.5 # Time to maturity: 6 months
r = 0.05 # Risk-free rate: 5%
sigma = 0.25 # Volatility: 25%
# Calculate call price
c all_price = black_scholes_call(S0, K, T, r, sigma)
print(f"Black-Scholes Call Price: £{call_price:.2f}")
```

Expected Output:

```
Black-Scholes Call Price: £6.04
```

Interpretation:

- The fair value of this call option is £6.04
- This is the price that prevents arbitrage opportunities
- We can verify this using Monte Carlo simulation

Imperial College
London

Put–Call Parity in Python

Mathematical Relationship:

$$C_t - P_t = S_t - K \times e^{-r(T-t)}$$

Rearranging to Find Put Price:

$$P_t = C_t - S_t + K \times e^{-r(T-t)}$$

Python Implementation:

```python
def put_from_call_parity(call_price, S, K, T, r):
    """
    Calculate put price using put-call parity.
    Parameters:
    call_price: price of the call option
    S: current stock price
    K: strike price
    T: time to maturity
    r: risk-free rate
    """
    put_price = call_price - S + K * np.exp(-r * T)
    return put_price
```

Intuition: A call + discounted strike = put + stock

Imperial College
London

Black–Scholes Put Formula

Mathematical Formula:

$$P(t, S) = K \times e^{-r(T-t)} \times \Phi(-d_2) - S \times \Phi(-d_1)$$

Python Implementation:

```python
def black_scholes_put(S, K, T, r, sigma):
    """
    Calculate European put option price using Black-Scholes.
    """
    # Calculate d1 and d2 (same as for call)
    d1 = (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    # Calculate put price
    put_price = K * np.exp(-r * T) * norm.cdf(-d2) - S * norm.cdf(-d1)
    return put_price
```

Key difference: The put formula uses $-d_1$ and $-d_2$ in norm.cdf().

Imperial College
London

Risk–Neutral Pricing: A Simpler Approach

The Problem with Real-World Pricing:

- In Session 2, we used GBM with drift μ (expected return)
- But μ is hard to estimate and depends on investor preferences
- The PDE/replication derivation is mathematically complex

The Risk-Neutral Solution:

$$V_0 = e^{-rT} \times E^{\mathbb{Q}}[\text{Payoff at } T]$$

Key Insight: Replace μ with r (risk-free rate) in stock dynamics. Simply simulate, compute payoffs, average, and discount. Same price as Black–Scholes, but much more flexible!

Why This Works:

- No-arbitrage ensures the price is unique
- Risk-neutral measure $\mathbb{Q}$ is a mathematical tool, not reality
- Extremely convenient for Monte Carlo methods

Imperial College
London

Simulating Under Risk–Neutral Measure

Real-World vs Risk-Neutral Dynamics:

Real-world (Session 2):

```
S_T = S0 * np.exp((mu - 0.5*sigma**2)*T + sigma*np.sqrt(T)*Z)
```

Risk-neutral (Session 3):

```
S_T = S0 * np.exp((r - 0.5*sigma**2)*T + sigma*np.sqrt(T)*Z)
```

The Only Change: Replace μ with r

Complete Example:

```python
import numpy as np
# Parameters
S0 = 100
r = 0.05 # Use risk-free rate instead of mu
sigma = 0.25
T = 0.5
# Generate random normal
Z = np.random.randn()
# Simulate final stock price under risk-neutral measure
S_T = S0 * np.exp((r - 0.5*sigma**2)*T + sigma*np.sqrt(T)*Z)
print(f"Simulated stock price: £{S_T:.2f}")
```

Imperial College
London

Risk–Neutral Monte Carlo Implementation

Complete Python Function:

```
import numpy as np
def monte_carlo_call_risk_neutral(S0, K, T, r, sigma, N_sims):
    # Set seed for reproducibility
    np.random.seed(42)
    # Initialize payoffs list
    payoffs = []
    # Run Monte Carlo simulation
    for i in range(N_sims):
        # Generate random normal
        Z = np.random.randn()
        # Simulate stock price under risk-neutral measure
        S_T = S0 * np.exp((r - 0.5*sigma**2)*T + sigma*np.sqrt(T)*Z)
        # Calculate call option payoff
        payoff = max(S_T - K, 0)
        payoffs.append(payoff)
    # Calculate expected payoff
    expected_payoff = np.mean(payoffs)
    #Discount to present value
    option_price = np.exp(-r * T) * expected_payoff
    return option_price
```

Critical Details: Use r (not mu), discount by e^{-rT}, proper indentation (4 spaces for function, 8 for loop).

Imperial College
London

Comparing Black–Scholes vs Monte Carlo

Running Both Methods:

```
# Parameters
S0 = 100
K = 105
T = 0.5
r = 0.05
sigma = 0.25
N_sims = 50000
# Method 1: Black-Scholes (analytical)
bs_price = black_scholes_call(S0, K, T, r, sigma)
# Method 2: Risk-Neutral Monte Carlo (numerical)
mc_price = monte_carlo_call_risk_neutral(S0, K, T, r, sigma, N_sims)
# Compare results
print(f"Black-Scholes Price: £{bs_price:.2f}")
print(f"Monte Carlo Price: £{mc_price:.2f}")
print(f"Difference: £{abs(bs_price - mc_price):.2f}")
```

Expected Output:

```
Black-Scholes Price: £6.04
Monte Carlo Price: £6.02
Difference: £0.02
```

Key Observations:

- Both methods give very similar results
- Monte Carlo converges to Black–Scholes as N_sims increases
- Monte Carlo is more flexible for complex payoffs (Asian, barrier options)

Imperial College
London

Summary: Black–Scholes in Python

Key Takeaways:

1. Black-Scholes Formula Implementation

Used `scipy.stats.norm.cdf()` for cumulative normal distribution. Broke complex formula into steps (d_1, d_2, then price). Created reusable functions for call and put pricing.

2. Put-Call Parity

Relationship: $C - P = S - K \times e^{-rT}$. Can derive put price from call price (or vice versa). Implemented in Python to verify consistency.

3. Risk-Neutral Pricing

Simplified Monte Carlo: use r instead of μ. Formula: $V_0 = e^{-rT} \times E^{\mathbb{Q}}[\text{Payoff}]$. More flexible than Black-Scholes for exotic options.

Next Steps:

- Experiment with different parameters (S, K, T, σ)
- Try pricing put options using both methods
- Explore exotic options where Monte Carlo shines (Asian, barrier, lookback)