Barrier Option Pricing using Binomial Trees
Looped vs Vectorized

Usama Buttar

December 2, 2024

1 Introduction and Setup

Barrier options are a type of path-dependent options, where the payoff depends not only on the final price of the underlying asset but also on whether the asset price crosses a specified barrier during the life of the option.

1.1 Types of Barrier Options:

1. **Knock-In Options:** Become active only if the underlying asset's price breaches the barrier.
2. **Knock-Out Options:** Expire worthless if the underlying price breaches the barrier.

In this project, we focus on **Knock-Out Options**, specifically the **Up-and-Out Call Option**, which:

- Becomes worthless if the asset price rises above a certain barrier level (`H`) during the option's life.
- Pays ($\max(S - K, 0)$) at maturity, provided the barrier has not been breached.

1.2 Pricing Approach:

We use the **Binomial Tree Method**, which:

1. Represents price movements over time as a tree structure.
2. Models upward and downward movements using fixed factors (`u` and `d`).
3. Incorporates the barrier condition:
    - If the price crosses the barrier, the option value at that node is set to zero.
4. Uses backward induction to calculate option prices at earlier nodes based on future values.

1.3 Binomial Tree Representation

The binomial tree represents the evolution of stock prices using nodes (i, j), where:

- (i) is the time step.
- (j) is the ordered price outcome (from lowest to highest).

The stock price at each node is computed as:

$$S_{i,j} = S_0 u^j d^{i-j}$$

The option value at each node (i, j) is denoted as $(C_{i,j})$. At maturity $(i = N)$, the final payoff function is defined, and the tree is traversed backward to compute prices at earlier nodes.

1.4 Barrier Option Characteristics

For an **Up-and-Out Barrier Put Option**:

1. At maturity $(T = t_N)$, the terminal payoff is:

$$C_N^j = \max(K - S_N^j, 0) \cdot \nVdash(S_N^j < H)$$

   - Here, H is the barrier, and $\nVdash(S_N^j < H)$ is the indicator function, which is 1 if $S_N^j < H$ and 0 otherwise.

2. For nodes earlier in the tree $(i < N)$:

   - If the price breaches the barrier $(S_i^j \geq H)$:

$$C_i^j = 0$$

   - Otherwise, the option value is computed using the risk-neutral valuation formula:

$$C_i^j = e^{-r\Delta T} \left[qC_{i+1}^{j+1} + (1 - q)C_{i+1}^j \right]$$

     where q is the risk-neutral probability of an upward movement.

This mathematical foundation ensures accurate pricing of barrier options while incorporating their path-dependent nature.

1.5 Implementation Goals:

- Compare two methods for pricing:
  1. **Loop-based approach:** Iterates over nodes explicitly.
  2. **Vectorized approach:** Uses NumPy for efficient computation.
- Demonstrate the significant performance advantage of vectorized operations.

[1]:
```python
import numpy as np  # For numerical operations
from functools import wraps  # To preserve metadata of decorated functions
from time import time  # For performance measurement

# A decorator to measure the execution time of functions
def timing(f):
    @wraps(f)
    def wrap(*args, **kw):
        ts = time()
        result = f(*args, **kw)
        te = time()
        print(f'func:{f.__name__} args:[{args}, {kw}] took: {te - ts:.4f} sec')
        return result
    return wrap
```

2 Model Parameters

2.1 Parameters Explained:

- S_0: Initial stock price.
- K: Strike price of the option.
- T: Time to maturity in years.
- H: Barrier price (e.g., for an up-and-out option).
- r: Annual risk-free rate.
- N: Number of time steps in the binomial tree.
- u: "Up" factor representing the proportionate increase in price.
- d: "Down" factor, typically `1/u` for a recombining tree.
- *opttype*: Type of option ('C' for Call, 'P' for Put).

These parameters define the barrier option and are used across both implementations.

[2]:
```
# Initialize model parameters
S0 = 100   # Initial stock price
K = 100    # Strike price
T = 1      # Time to maturity (in years)
H = 125    # Barrier price (e.g., for up-and-out)
r = 0.06   # Annual risk-free rate
N = 3      # Number of time steps
u = 1.1    # Up factor
d = 1/u    # Down factor, ensuring a recombining tree
opttype = 'C'  # Option type: 'C' for Call, 'P' for Put
```

3 Barrier Tree: Slow Implementation

The slow implementation uses nested loops to compute option prices.

3.1 Steps:

1. **Precompute Constants:**
   - `dt`: Duration of each time step.
   - `q`: Risk-neutral probability of an upward price movement.
   - `disc`: Discount factor for time step.
2. **Initialize Asset Prices at Maturity:**
   - Use the formula $S_{i,j} = S_0 \cdot u^j \cdot d^{i-j}$.
3. **Compute Option Payoff:**
   - Apply the barrier condition (set payoff to zero if the barrier is breached).
   - Calculate the payoff for call or put options.
4. **Backward Recursion:**
   - Traverse the tree backward, updating option values at each node.

[3]:
```python
@timing
def barrier_tree_slow(K, T, S0, H, r, N, u, d, opttype='C'):
    # Precompute constants
    dt = T / N
    q = (np.exp(r * dt) - d) / (u - d)
    disc = np.exp(-r * dt)

    # Initialize asset prices at maturity
    S = np.zeros(N + 1)
    for j in range(N + 1):
        S[j] = S0 * u**j * d**(N - j)

    # Compute option payoff
    C = np.zeros(N + 1)
    for j in range(N + 1):
        if opttype == 'C':
            C[j] = max(0, S[j] - K)  # Call option
        else:
            C[j] = max(0, K - S[j])  # Put option

    # Apply barrier condition at maturity
    for j in range(N + 1):
        if S[j] >= H:  # Up-and-out condition
            C[j] = 0

    # Backward recursion
    for i in range(N - 1, -1, -1):
        for j in range(i + 1):
            S[j] = S0 * u**j * d**(i - j)  # Price at node (i, j)
            if S[j] >= H:
                C[j] = 0  # Barrier condition
            else:
                C[j] = disc * (q * C[j + 1] + (1 - q) * C[j])
    return C[0]

# Example usage
barrier_tree_slow(K, T, S0, H, r, N, u, d, opttype='C')
```

```
func:barrier_tree_slow args:[(100, 1, 100, 125, 0.06, 3, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.0002 sec
```

[3]: 4.00026736854323

4 Barrier Tree: Fast Implementation

The fast implementation improves performance by vectorizing operations with NumPy.

4.1 Key Differences from the Slow Implementation:

1. **Vectorized Initialization:**
   - Calculate asset prices at maturity using array operations instead of loops.
2. **Barrier Check:**
   - Use array indexing to efficiently apply the barrier condition.
3. **Backward Recursion:**
   - Replace nested loops with vectorized calculations to propagate values through the tree.

This approach is significantly faster, particularly for larger tree sizes.

[4]:
```python
@timing
def barrier_tree_fast(K, T, S0, H, r, N, u, d, opttype='C'):
    # Precompute constants
    dt = T / N
    q = (np.exp(r * dt) - d) / (u - d)
    disc = np.exp(-r * dt)

    # Initialize asset prices at maturity (vectorized)
    S = S0 * d ** np.arange(N, -1, -1) * u ** np.arange(0, N + 1)

    # Compute option payoff (vectorized)
    if opttype == 'C':
        C = np.maximum(S - K, 0)  # Call option
    else:
        C = np.maximum(K - S, 0)  # Put option

    # Apply barrier condition at maturity
    C[S >= H] = 0  # Up-and-out condition

    # Backward recursion
    for i in range(N - 1, -1, -1):
        S = S0 * d ** np.arange(i, -1, -1) * u ** np.arange(0, i + 1)
        C[:i + 1] = disc * (q * C[1:i + 2] + (1 - q) * C[0:i + 1])
        C = C[:-1]  # Trim array for the next iteration
        C[S >= H] = 0  # Apply barrier condition
    return C[0]

# Example usage
barrier_tree_fast(K, T, S0, H, r, N, u, d, opttype='C')
```

```
func:barrier_tree_fast args:[(100, 1, 100, 125, 0.06, 3, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.0009 sec
```

[4]: 4.00026736854323

5 Comparison: Slow vs. Fast Implementation

5.1 Objective:

Evaluate the runtime performance of the slow and fast implementations across different tree sizes.

5.2 Test Setup:

- Use various numbers of time steps (`N`) to observe how execution time scales with complexity.
- Compare the runtimes of `barrier_tree_slow` and `barrier_tree_fast`.

5.3 Results:

- The vectorized implementation (`barrier_tree_fast`) consistently outperforms the loop-based version.
- As the number of time steps increases, the performance gap widens significantly.

The results highlight the importance of vectorization for large-scale computations.

[5]:
```
# Test and compare runtime for different tree sizes
for N in [3, 50, 100, 1000, 5000]:
    print(f"\nTime Steps: {N}")
    barrier_tree_slow(K, T, S0, H, r, N, u, d, opttype='C')
    barrier_tree_fast(K, T, S0, H, r, N, u, d, opttype='C')
```

```
Time Steps: 3
func:barrier_tree_slow args:[(100, 1, 100, 125, 0.06, 3, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.0003 sec
func:barrier_tree_fast args:[(100, 1, 100, 125, 0.06, 3, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.0026 sec

Time Steps: 50
func:barrier_tree_slow args:[(100, 1, 100, 125, 0.06, 50, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.0116 sec
func:barrier_tree_fast args:[(100, 1, 100, 125, 0.06, 50, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.0107 sec

Time Steps: 100
func:barrier_tree_slow args:[(100, 1, 100, 125, 0.06, 100, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.0148 sec
func:barrier_tree_fast args:[(100, 1, 100, 125, 0.06, 100, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.0057 sec

Time Steps: 1000
func:barrier_tree_slow args:[(100, 1, 100, 125, 0.06, 1000, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.6978 sec
func:barrier_tree_fast args:[(100, 1, 100, 125, 0.06, 1000, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.0619 sec
```

```
Time Steps: 5000
func:barrier_tree_slow args:[(100, 1, 100, 125, 0.06, 5000, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 18.3857 sec
func:barrier_tree_fast args:[(100, 1, 100, 125, 0.06, 5000, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 1.0130 sec
```

6 Conclusion

Barrier options add complexity to the pricing process due to their path-dependent nature.

6.1 Key Takeaways:

1. **Accuracy:**
    - Both the slow and fast implementations yield the same option prices.
2. **Performance:**
    - The vectorized implementation (`barrier_tree_fast`) is significantly faster and more efficient.
    - It becomes increasingly advantageous as the tree size grows.
3. **Scalability:**
    - The fast implementation scales well for larger problems, making it more suitable for real-world applications.

This project demonstrates the efficiency of vectorized numerical methods in handling advanced financial computations.