Binomial Option Pricing Looped vs Vectorized

Usama Buttar

December 2, 2024

1 Introduction and Setup

The Binomial Option Pricing Model is a method to evaluate options by modeling the possible price paths of the underlying asset over time. It uses a discrete-time framework with "up" and "down" movements defined at each time step.

1.1 Key Components:

- **Binomial Tree Representation:** Each node represents a potential price.
- **Timing Wrapper:** A utility to measure the execution time of different implementations.

We'll start by importing necessary libraries and defining a timing utility function.

[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:

- $S0$: Initial stock price.
- K: Strike price of the option.
- T: Time to maturity in years.
- 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 $1/u$ to ensure a recombining tree.

- *opttype*: Type of option (C for Call, P for Put).

These parameters will be shared across both implementations.

[2]:
```
# Initialize model parameters
S0 = 100   # Initial stock price
K = 100    # Strike price
T = 1      # Time to maturity (in years)
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 Binomial Tree: Slow Implementation

The slow implementation iterates over nodes explicitly using loops.

3.1 Steps:

1. **Precompute Constants:**
   - `dt`: Duration of a single time step.
   - `q`: Risk-neutral probability of an upward price movement.
   - `disc`: Discount factor for time step.
2. **Initialize Stock Prices at Maturity:**
   - Use the formula $S_{i,j} = S_0 \cdot u^j \cdot d^{i-j}$. \$.
3. **Compute Option Values at Maturity:**
   - For a European call, $C_{N,j} = \max(S_{N,j} - K, 0)$.
4. **Step Backward Through the Tree:**
   - Use the risk-neutral valuation formula.
   - Compute option prices at earlier nodes from terminal values.

[3]:
```
@timing
def binomial_tree_slow(K, T, S0, r, N, u, d, opttype='C'):
    # Precompute constants
    dt = T / N  # Time step duration
    q = (np.exp(r * dt) - d) / (u - d)  # Risk-neutral probability
    disc = np.exp(-r * dt)  # Discount factor

    # Initialize stock prices at maturity (time step N)
    S = np.zeros(N + 1)
    S[0] = S0 * d**N  # Lowest price at maturity
    for j in range(1, N + 1):
        S[j] = S[j - 1] * u / d  # Increment upward in the tree

    # Initialize option values at maturity
    C = np.zeros(N + 1)
    for j in range(N + 1):
```

```
        C[j] = max(0, S[j] - K)  # Payoff for European call option

    # Step backward through the tree
    for i in range(N, 0, -1):  # Start from last step
        for j in range(i):
            C[j] = disc * (q * C[j + 1] + (1 - q) * C[j])  # Risk-neutral␣
↪valuation

    return C[0]  # Option price at the root node

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

```
func:binomial_tree_slow args:[(100, 1, 100, 0.06, 3, 1.1, 0.9090909090909091),
{'opttype': 'C'}] took: 0.0001 sec
```

[3]: 10.145735799928817

4 Binomial Tree: Fast Implementation

The fast implementation optimizes calculations by vectorizing operations using NumPy.

4.1 Key Differences from the Slow Implementation:

1. **Vectorized Initialization:**
   - Directly calculate stock prices at maturity using array operations.
   - Use NumPy's `arange` to generate indices for *up* and *down* movements.
2. **Backward Propagation:**
   - Replace inner loops with array slicing for efficient calculations.

This approach significantly reduces computational overhead, especially for larger time steps.

[4]:
```
@timing
def binomial_tree_fast(K, T, S0, r, N, u, d, opttype='C'):
    # Precompute constants
    dt = T / N  # Time step duration
    q = (np.exp(r * dt) - d) / (u - d)  # Risk-neutral probability
    disc = np.exp(-r * dt)  # Discount factor

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

    # Initialize option values at maturity
    C = np.maximum(S - K, 0)  # Payoff for European call option

    # Step backward through the tree
    for i in range(N, 0, -1):
        C = disc * (q * C[1:i + 1] + (1 - q) * C[0:i])  # Vectorized valuation
```

```
    return C[0]  # Option price at the root node

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

```
func:binomial_tree_fast args:[(100, 1, 100, 0.06, 3, 1.1, 0.9090909090909091),
{'opttype': 'C'}] took: 0.0007 sec
```

[4]: 10.145735799928826

5 Comparison: Slow vs. Fast Implementation

5.1 Objective:

Evaluate the runtime performance of both implementations across various tree sizes.

5.2 Test Setup:

- Use different numbers of time steps (N) to observe how execution time scales.
- Compare the runtimes of `binomial_tree_slow` and `binomial_tree_fast`.

5.3 Results:

The vectorized implementation is consistently faster, particularly as N increases. This demonstrates the efficiency of NumPy in handling 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}")
    binomial_tree_slow(K, T, S0, r, N, u, d, opttype='C')
    binomial_tree_fast(K, T, S0, r, N, u, d, opttype='C')
```

```
Time Steps: 3
func:binomial_tree_slow args:[(100, 1, 100, 0.06, 3, 1.1, 0.9090909090909091),
{'opttype': 'C'}] took: 0.0001 sec
func:binomial_tree_fast args:[(100, 1, 100, 0.06, 3, 1.1, 0.9090909090909091),
{'opttype': 'C'}] took: 0.0001 sec

Time Steps: 50
func:binomial_tree_slow args:[(100, 1, 100, 0.06, 50, 1.1, 0.9090909090909091),
{'opttype': 'C'}] took: 0.0013 sec
func:binomial_tree_fast args:[(100, 1, 100, 0.06, 50, 1.1, 0.9090909090909091),
{'opttype': 'C'}] took: 0.0006 sec

Time Steps: 100
func:binomial_tree_slow args:[(100, 1, 100, 0.06, 100, 1.1, 0.9090909090909091),
{'opttype': 'C'}] took: 0.0069 sec
```

```
func:binomial_tree_fast args:[(100, 1, 100, 0.06, 100, 1.1, 0.9090909090909091),
{'opttype': 'C'}] took: 0.0062 sec

Time Steps: 1000
func:binomial_tree_slow args:[(100, 1, 100, 0.06, 1000, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.6002 sec
func:binomial_tree_fast args:[(100, 1, 100, 0.06, 1000, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.0082 sec

Time Steps: 5000
func:binomial_tree_slow args:[(100, 1, 100, 0.06, 5000, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 11.1670 sec
func:binomial_tree_fast args:[(100, 1, 100, 0.06, 5000, 1.1,
0.9090909090909091), {'opttype': 'C'}] took: 0.0541 sec
```

6 Conclusion

The Binomial Option Pricing Model provides a versatile framework for pricing options through a discrete-time approximation.

6.1 Key Takeaways:

1. **Accuracy:**
   - Both implementations yield identical results for the option price.
2. **Performance:**
   - The vectorized approach (`binomial_tree_fast`) is significantly faster, especially as the number of time steps increases.
3. **Scalability:**
   - For large-scale problems, leveraging NumPy or similar libraries is essential to achieve efficient computation.

This project highlights the importance of optimization in numerical methods and serves as a foundational step toward more advanced option pricing models.