American Option Pricing using Binomial Trees
Looped vs Vectorized

Usama Buttar

December 4, 2024

1 Introduction and Setup

1.1 What are American Options?

American options are financial derivatives that grant the holder the right, but not the obligation, to:

- **Buy** (Call option) or **Sell** (Put option) the underlying asset at a predetermined strike price (K) **at any time** before or at expiration.

This feature makes American options more flexible and potentially more valuable than their European counterparts, which can only be exercised at expiration.

Types of American Options:

1. **American Call Option:**
   - Grants the right to **buy** the underlying asset at (K).
   - Payoff at exercise:
   $$Payoff = max(S - K, 0)$$
   - Early exercise is rare unless there are dividends or carrying costs.
2. **American Put Option:**
   - Grants the right to **sell** the underlying asset at (K).
   - Payoff at exercise:
   $$Payoff = max(K - S, 0)$$
   - Early exercise is more common, especially if the underlying asset price falls significantly.

1.2 The Binomial Tree Method

The **Binomial Tree Method** models the price evolution of the underlying asset as a tree of possible outcomes:

1. At each time step (i), the asset price moves up (u) or down (d) with probabilities (q) and ($1 - q$), respectively.
2. The stock price at any node (i, j) is computed as:
$$S_{i,j} = S_0 u^j d^{i-j}$$

Where:

- (i): Current time step.
- (j): Number of upward movements.

1.3 Pricing American Options with Binomial Trees

For American options, the value at each node is the **maximum** of two components:

1. **Exercise Value:** The payoff if the option is exercised at that node.

   - For a Call Option:
   $$ExerciseValue = max(S_{i,j} - K, 0)$$
   - For a Put Option:
   $$ExerciseValue = max(K - S_{i,j}, 0)$$

2. **Continuation Value:** The discounted expected value of holding the option until the next step:
   $$ContinuationValue = e^{-r\Delta t}\left[qC_{i+1}^{j+1} + (1-q)C_{i+1}^{j}\right]$$

   Where:

   - (r): Risk-free interest rate.
   - $\Delta t = \frac{T}{N}$: Length of each time step.
   - $q = \frac{e^{r\Delta t}-d}{u-d}$: Risk-neutral probability of an upward movement.

At each node, the option value is:

$$C_{i,j} = max\left(ExerciseValue, ContinuationValue\right)$$

1.4 Early Exercise

The ability to exercise early introduces path dependency. The optimal exercise strategy depends on:

1. The relationship between the exercise value and continuation value.
2. The time remaining until expiration $(T - t_i)$.

1.5 Challenges of Pricing American Options

The early exercise feature adds complexity:

- **Backward Induction:** The tree must be traversed from maturity to the present, computing the option value at each node based on the above equations.
- **Path Dependency:** Every node requires a comparison between exercising and holding.

1.6 Goals of This Notebook

This notebook explores two implementations for pricing American options:

1. **Slow Implementation (Loop-based):** A straightforward method to compute values iteratively.
2. **Fast Implementation (Vectorized):** An optimized method using NumPy to enhance performance.

1.7 Implementation Overview

1. **Stock Prices at Maturity:**
   - Compute stock prices at each node at the final time step: $S_{N,j} = S_0 u^j d^{N-j}$
2. **Option Payoff at Maturity:**
   - For a Put Option: $C_{N,j} = \max(K - S_{N,j}, 0)$
   - For a Call Option: $C_{N,j} = \max(S_{N,j} - K, 0)$
3. **Backward Induction:**
   - For each prior step (i), compute the option value at node (i, j) using:

$$C_{i,j} = max\left(K - S_{i,j}, e^{-r\Delta t}\left[qC_{i+1}^{j+1} + (1-q)C_{i+1}^{j}\right]\right)$$

     - Replace $(K - S_{i,j})$ with $(S_{i,j} - K)$ for a Call Option.
4. **Final Value at Root:**
   - The option value at the root node $(C_{0,0})$ is the price of the American option.

This structured approach ensures accurate pricing while highlighting the computational demands of early exercise flexibility.

[1]:
```
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.
- r: Annual risk-free interest rate.
- N: Number of time steps in the binomial tree.
- u: Up factor, representing the proportionate price increase.
- d: Down factor, typically $(1/u)$ for recombining trees.
- *opttype*: Type of option ('P' for Put, 'C' for Call).

These parameters define the setup for the binomial tree and the characteristics of the American option.

[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 = 'P'   # Option type: 'P' for Put, 'C' for Call
```

3 American 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.
   - $e^{-r\,dt}$: Discount factor for each time step.
2. **Initialize Asset Prices at Maturity:**
   - Use the formula: $S_{i,j} = S_0 u^j d^{i-j}$
3. **Compute Option Payoff at Maturity:**
   - For a Put Option: $C_N^j = \max(K - S_N^j, 0)$
4. **Backward Recursion Through the Tree:**
   - For each node:
     - Compute the continuation value: $C_i^j = e^{-r\,dt}\left[qC_{i+1}^{j+1} + (1-q)C_{i+1}^j\right]$
     - Update the value to the maximum of the exercise and continuation values: $C_i^j = max(\mathrm{C}_i^j, K - S_i^j)$

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

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

    # Backward recursion through the tree
    for i in range(N - 1, -1, -1):
        for j in range(i + 1):
            S = S0 * u**j * d**(i - j)  # Price at node (i, j)
```

```
            C[j] = disc * (q * C[j + 1] + (1 - q) * C[j])  # Continuation value
            if opttype == 'P':  # Put option
                C[j] = max(C[j], K - S)  # Max of continuation or exercise
            else:                 # Call option
                C[j] = max(C[j], S - K)  # Max of continuation or exercise

    return C[0]

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

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

[3]: 4.654588754602527

4 American Tree: Fast Implementation

The fast implementation optimizes performance by vectorizing operations with NumPy.

4.1 Key Differences:

1. **Vectorized Initialization:**
   - Compute asset prices at maturity using array operations.
2. **Barrier Condition Application:**
   - Directly compute the maximum of continuation and exercise values at each step.
3. **Backward Recursion:**
   - Use array slicing to propagate values efficiently through the tree.

This implementation significantly reduces computational overhead, especially for larger tree sizes.

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

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

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

    # Backward recursion through the tree
    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])  # Continuation value
        if opttype == 'P':  # Put option
            C = np.maximum(C[:i + 1], K - S)  # Max of continuation or exercise
        else:               # Call option
            C = np.maximum(C[:i + 1], S - K)  # Max of continuation or exercise
    return C[0]

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

```
func:american_fast_tree args:[(100, 1, 100, 0.06, 3, 1.1, 0.9090909090909091),
{'opttype': 'P'}] took: 0.0003 sec
```

[4]: 4.654588754602527

5 Comparison: Slow vs. Fast Implementation

5.1 Objective:

Compare the runtime performance of the loop-based (slow) and vectorized (fast) implementations for various tree sizes.

5.2 Test Setup:

1. Use different numbers of time steps (N) to analyze how execution time scales with complexity.
2. Measure and compare runtimes for `american_slow_tree` and `american_fast_tree`.

5.3 Results:

- The vectorized implementation is consistently faster.
- The performance gap widens as the number of time steps increases.

This highlights the efficiency of vectorized operations for large-scale problems.

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

```
Time Steps: 3
func:american_slow_tree args:[(100, 1, 100, 0.06, 3, 1.1, 0.9090909090909091),
{'opttype': 'P'}] took: 0.0001 sec
func:american_fast_tree args:[(100, 1, 100, 0.06, 3, 1.1, 0.9090909090909091),
{'opttype': 'P'}] took: 0.0002 sec

Time Steps: 50
func:american_slow_tree args:[(100, 1, 100, 0.06, 50, 1.1, 0.9090909090909091),
{'opttype': 'P'}] took: 0.0049 sec
func:american_fast_tree args:[(100, 1, 100, 0.06, 50, 1.1, 0.9090909090909091),
{'opttype': 'P'}] took: 0.0033 sec

Time Steps: 100
func:american_slow_tree args:[(100, 1, 100, 0.06, 100, 1.1, 0.9090909090909091),
{'opttype': 'P'}] took: 0.0165 sec
func:american_fast_tree args:[(100, 1, 100, 0.06, 100, 1.1, 0.9090909090909091),
{'opttype': 'P'}] took: 0.0054 sec

Time Steps: 1000
func:american_slow_tree args:[(100, 1, 100, 0.06, 1000, 1.1,
0.9090909090909091), {'opttype': 'P'}] took: 1.0286 sec
func:american_fast_tree args:[(100, 1, 100, 0.06, 1000, 1.1,
0.9090909090909091), {'opttype': 'P'}] took: 0.0614 sec
```

```
Time Steps: 5000
func:american_slow_tree args:[(100, 1, 100, 0.06, 5000, 1.1,
0.9090909090909091), {'opttype': 'P'}] took: 19.9946 sec
func:american_fast_tree args:[(100, 1, 100, 0.06, 5000, 1.1,
0.9090909090909091), {'opttype': 'P'}] took: 0.8815 sec
```

6 Conclusion

6.1 Key Takeaways:

1. **American Options and Binomial Trees:**
    - The binomial tree method provides a straightforward and accurate approach to price American options, which require evaluation of both exercise and continuation values at each step.
2. **Performance Comparison:**
    - Both implementations yield the same results for the option price.
    - The vectorized method (`american_fast_tree`) is significantly faster, especially for large tree sizes.
3. **Scalability:**
    - The vectorized implementation is better suited for real-world scenarios where high computational efficiency is critical.

This project demonstrates the importance of optimization in numerical methods, paving the way for more advanced pricing models.