Reflections on Option Pricing Models

Tom Beaugé

January 27, 2025

Abstract

This document reflects on my self-directed exploration of financial models for option pricing, beginning with the single-period binomial tree. I discuss the theoretical foundations, implementation challenges, and insights gained from constructing and analyzing the model. This work serves as a foundation for future investigations into more complex pricing frameworks, such as multi-period trees, stochastic calculus, and market microstructure.

1 Introduction

The pricing of financial derivatives represents a cornerstone of modern quantitative finance. My project begins with the simplest discrete-time model—the binomial tree—to build intuition about no-arbitrage pricing, risk-neutral valuation, and dynamic hedging. Though elementary, this model encodes key principles that generalize to continuous-time frameworks like the Black-Scholes-Merton model. My goal is to iteratively implement and analyze increasingly sophisticated models, using each step to deepen my understanding of stochastic processes, measure theory, and computational finance.

2 The Binomial Tree Model

With its very straightforward concept and basic mathematics, the binomial model seemed like a logical introduction to options pricing. However, this overtly simplistic binary idea that the price of an option can either go up or down is something to be noted. In the words of Paul Wilmott: "the model is for demonstration purposes only, it is not the real thing. As a model of the financial world it is too simplistic, as a concept for pricing it lacks the elegance that makes other methods preferable, and as a numerical scheme it is prehistoric. Use once and then throw away." This word of caution was kept in mind while I sought to implement it based on Chapter 15 of his book *Paul Wilmott Introduces Quantitative Finance*.

2.1 A Simple, One-step Binomial Tree

In essence, the binomial model, introduced by Cox, Ross, and Rubinstein (1979), provides a discrete-time approximation of asset price dynamics, such that the underlying asset price S_t evolves in a probability space as:

$$S_{t+1} = \begin{cases} S_t \cdot u & \text{with probability } p, \\ S_t \cdot d & \text{with probability } 1-p. \end{cases}$$

where $u > 1 + r > d > 0$ ensures arbitrage-free dynamics, and r is the risk-free rate. Its simplicity belies its ability to replicate derivative payoffs through a self-financing portfolio of the underlying asset and risk-free bonds.

2.1.1 Implementation

From what was understood, the main idea of the binomial model is to create a risk-neutral valuation. This was achieved by constructing a portfolio composed of the underlying asset and risk-free bonds to exactly match the option's payoff. To prevent any arbitrage, the option's price must thus correspond to the cost of replicating this portfolio. The methodology for constructing this risk-free portfolio was directly inspired by this YouTube video, with the slight addition of fixed interest rates to account for the time value of money.

The relationship governing the replicating portfolio is expressed as:

$$P_t = \Delta S_t - V_t$$

Here, P_t represents the value of the portfolio at time t, Δ is the hedge ratio indicating the number of units of the underlying asset held, S_t is the current price of the underlying asset, and V_t is the current price of the option. This construction ensures that the portfolio is risk-free by balancing the position in the underlying asset with the short position in the option.

For the portfolio P_t to be risk-free, its future value must be identical in both the up and down states. This condition leads to:

$$\Delta S_u - V_u = \Delta S_d - V_d$$

Solving for the hedging ratio Δ associated with the number of units of the underlying asset yields:

$$\Delta = \frac{V_u - V_d}{S_u - S_d}$$

This calculation ensures that the portfolio replicates the option's payoff in both possible future states, thereby enforcing the no-arbitrage condition.

2.1.2 Insights

By pricing the option in such a way, we are hedged against any possible fluctuation in price, as the arbitrary probability p that the option will increase in price appears at no point in our pricing calculations. Playing around with the expected value, I was also able to notice that for a specific probability p, the expected value matched the pricing of my option echoing the First Fundamental Theorem of Asset Pricing applies here.

First Fundamental Theorem of Asset Pricing

The **First Fundamental Theorem of Asset Pricing** states that a financial market is **arbitrage-free** if and only if there exists at least one **equivalent martingale measure** (also known as a risk-neutral measure).
In other words, the absence of arbitrage opportunities in a market is equivalent to the existence of a probability measure under which discounted asset prices follow a martingale process.

Deriving the risk neutral probability p:

Assume the expected return equals the risk-free rate of the underlying asset:

$$\mathbb{E}[S_{t+1}] = (1 + r)S_0$$

where:

$$\mathbb{E}[S_{t+1}] = pS_u + (1 - p)S_d$$

Thus, solving for the risk-neutral probability p yields:

$$p = \frac{1 + r - d}{u - d}$$

2.1.3 Limits

Its assumptions—constant u, d, r, and no transaction costs—are economically restrictive. Moreover, the model oversimplifies price fluctuations to just an up and down movement. It's also completly inadapted for more complex options, very poorly estimates those with longer expirations and does not account for volatility.

These limitations motivate the need for more sophisticated models.

2.1.4 Future Directions

This work lays the groundwork for several extensions:

- **Multi-Period Trees**: Generalizing to n-period trees to approximate continuous processes.
- **American Options**: Incorporating early exercise features via dynamic programming.
- **Volatility**: Experimenting with time-varying u and d to model volatility.

2.2 Going Multi-step

The next step in my exploration was to extend the single-step binomial tree to a multi-period framework. This extension allows for a improvements in the modelisation of our asset price, allowing us to have a rudimentary estimate of the option's price over a larger period of time. The fluctuations in the asset price are still binary, but this new improvement allow us to better represent the random walk of our asset.

2.2.1 Implementation

To simplify the calculations, I have decided to only carry the theory from our previous one-step model. Our simple one-period option was priced using the hedge ratio Δ. However, to make the computation more straightforward, I have chosen to reuse the risk-neutral probability p we found earlier to price the option at each node.

1. **Building the Stock Price Tree:** A two-dimensional array is used to store the possible stock prices at maturity. At each node corresponding to i upward movements (and $n - i$ downward movements) after n steps, the stock price is calculated as:

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

   where S_0 is the initial stock price.

2. **Option Payoff Calculation:** At maturity, the option payoff is computed at each node. For a call option, the payoff is:

$$\text{Payoff} = \max(S_{n,i} - K, 0),$$

   and for a put option:

$$\text{Payoff} = \max(K - S_{n,i}, 0),$$

   where K is the strike price.

3. **Backward Induction:** After setting the terminal payoffs, the tree is traversed backward to compute the option value at earlier nodes. At each node, the option value is obtained as:

$$V_{i,j} = \frac{p \, V_{\text{up}} + (1 - p) \, V_{\text{down}}}{1 + r},$$

   where:

   - V_{up} is the option value from the node corresponding to an upward movement,
   - V_{down} is the option value from the node corresponding to a downward movement,

- $1 + r$ is used to incorporate the risk-free rate per period.

This backward induction continues until the root node is reached, where the option price at time zero is obtained.

A short code excerpt is shown below:

```
public MultiStepBinomialTree(double initialPrice, double strikePrice,
        double probabilityUp, double upFactor, double downFactor,
        double interestRate, boolean isCall, int steps) {

    // Calculate risk-neutral probability: q = (1 + r - d) / (u - d)
    double q = calculateRiskNeutralProbability(upFactor, downFactor, interestRate);

    // Initialize arrays for stock prices and option values
    stockPriceMaturity = new double[steps + 1][steps + 1];
    optionValues = new double[steps + 1][steps + 1];

    // Backward induction to build the tree and calculate option values
    for (int step = steps; step >= 0; step--) {
        for (int i = 0; i <= step; i++) {
            // Calculate stock price at the node: S = initialPrice * u^i * d^(step-i)
            stockPriceMaturity[step][i] = initialPrice * Math.pow(upFactor, i) *
            ↪  Math.pow(downFactor, step - i);

            // Set option payoff at maturity or compute the option value by backward
            ↪  induction
            if (step == steps) {
                optionValues[step][i] =
                ↪  calculateOptionPayoff(stockPriceMaturity[step][i], strikePrice,
                ↪  isCall);
            } else {
                optionValues[step][i] = calculateOptionValue(optionValues[step +
                ↪  1][i], optionValues[step + 1][i + 1], interestRate, q);
            }
        }
    }
}
```

Note: The code snippet is a simplified version of the actual implementation, which includes additional error handling and input validation.

The final option price is available as the value of the root node of the tree, i.e., `optionValues[0][0]`. This represents the present value of the option under the given parameters and assumptions of the binomial model.

Example Case To illustrate the implementation, consider the following example:

- Initial stock price (S_0): \$100
- Strike price (K): \$105
- Up factor (u): 1.1
- Down factor (d): 0.9
- Risk-free rate (r): 5% per period
- Number of steps: 3
- Option type: Put

Figure 1: Option values for a 3-step tree

Figure 2: Stock prices

The rightmost nodes in Figure **??** show the payoffs of the put option at expiration, computed as:

$$P_T = \max(K - S_T, 0)$$

Some nodes have a value of \$0.00, indicating that the stock price was above the strike price at expiration, making the put option worthless.

The deeper the stock price falls below $K = 105$, the higher the put option value. The highest option price is observed in paths where the stock price significantly declines.

The root node of the tree gives an option price of \$17.49, which represents the fair value of the put option under these assumptions.

2.2.2 Insights

The following figure illustrates the evolution of computation time as the number of steps increases in the multi-step binomial tree model.

Figure 3: Computation time for Binomial Trees up to 1000 steps)

The observed quadratic trend in computation time suggests that the model's complexity scales poorly with large steps. Optimization strategies such as dynamic programming, parallel processing, and memory-efficient algorithms should be explored to improve performance. If we look at the code snippet above, the time complexity is dominated by the nested loops:

Outer loop: Iterates over steps from n to 0 (total $n + 1$ steps). Inner loop: For each step s, iterates i from 0 to s (total $s + 1$ iterations per step).

Per-node operations:
For each node (step, i), `calculateOptionValue` computes the option value using:

$$\text{Value} = \frac{q \cdot \text{Payoff}_{\text{Up}} + (1 - q) \cdot \text{Payoff}_{\text{Down}}}{1 + r}$$

Nonetheless, while most operations to calculate the payoff is constant time (*see code above*), *Math.pow* is of complexity $O(\log n)$.

This simplifies to a total time complexity of:

$$O(n^2 \log n)$$

note that the graphs rather represent a $O(n^3 \log n)$ *complexity as they plot binomials tree from 1 to n-steps*

As a general benchmark, the current implementation computes in 10 seconds binomial trees up to 1100 steps with the memory usage that maxes out around 225MB. The current challenge is to enhance computational speed while simultaneously optimizing memory efficiency.

2.2.3 Speeding up the process

Given the quadratic growth in computation time and the substantial memory footprint, several optimizations were implemented to enhance both speed and memory efficiency in the multi-step binomial tree model.

1. Precomputing Powers Repeated calls to `Math.pow` were identified as a significant performance bottleneck, given their logarithmic time complexity $O(\log n)$. To address this the increase and decrease factors are now precomputed using induction:

```
double[] upPowers = new double[steps + 1];
double[] downPowers = new double[steps + 1];
upPowers[0] = 1.0;
downPowers[0] = 1.0;
for (int j = 1; j <= steps; j++) {
    upPowers[j] = upPowers[j - 1] * upFactor;
    downPowers[j] = downPowers[j - 1] * downFactor;
}
```

This optimization reduces the total time complexity from:

$$O(n^2 \log n) \quad \text{to} \quad O(n^2)$$

This results in a major time-related improvement - 2800 steps are now computed in 10 seconds - but now significantly more memory is used, peaking around 1600MB.

2. Allocating jagged arrays Instead of assigning a square array for our stock price and option value through time, leading to unnecessary empty entries, each row was allocated with the exact size they needed (i.e. step number + 1)

This led to some improvements in the memory usage which now maxes around 1300MB. Its time efficiency also increased reaching 3100 steps in 10 seconds. This is hypothesized to be due to a reduction in garbage collection pressure as well as a better cache utilisation since no superfluous data is present.

Summary With the implementation of storing stock and option values for every node, the code has been optimised. Parallelising computations was deemed unnecessary, as most operations depend on previous inputs. While its implementation led to noticeable improvements, it was perhaps even counterproductive for smaller step sizes. It could nonetheless be used to speed up the process of computing all binomial trees for our graphs since they are calculated independently but as of right now we have enough data points. The focus should remain on establishing a valid and efficient model.

Further optimisations would likely require better hardware configurations (which are beyond the scope of this project) or the adoption of a language with more efficient memory management, such as Rust, or a more 'hands-on' language like C++.

Overall, the performance upgrade is significant. Previously, computing trees up to 1000 steps took over 6 seconds, whereas the optimised program now completes the same task in under half a second. This improvement enables data extraction after a large number of steps, better reflecting the constant fluctuations of real-world assets. However, the $O(n^2)$ time complexity still limits the model's long-term usability, highlighting the need for a more efficient model in this regard.

2.3 Towards continuity

We now aim to transition from our discrete model to a continuous framework, while retaining much of the underlying theory. Consequently, the option price can still be expressed using the familiar formula:

$$V = \frac{p\,V_{\text{up}} + (1-p)\,V_{\text{down}}}{1 + r\delta t},$$

Notice the subtle yet important modification: we've replaced $1+r$, the discrete discount factor, with $1 + r\delta t$,which approximates the continuous compounding factor $e^{1+r\delta t}$. This adjustment bridges the gap between discrete and continuous discounting, setting the stage for a smooth transition to continuous-time option pricing models.

We will now assume that our underlying asset follows a Geometric Brownian Motion (GBM). While this choice is primarily guided by the model used in the reference book, its advantages are evident.

It captures randomness in a mathematically straightforward manner, which will help us greatly in the calculations to come. Additionally, its logarithmic nature ensures that the asset price remains strictly positive, which aligns with real-world financial behavior

However, it's important to acknowledge a key limitation: while assuming constant volatility simplifies the model and offers analytical convenience, it does not accurately reflect the dynamic nature of volatility observed in real markets. This simplification, though useful for theoretical development, should be considered with caution when interpreting real-world applications.

2.3.1 Finding the up facor u and down factor d

Our initial assumption that $V_{\text{up}} = uV$ and $V_{\text{down}} = vV$ now proves itself incompatible with Brownian motion.

Instead, for an asset price S_t that follows GBM, its dynamics are given by the stochastic differential equation:

$$dS_t = \mu S_t \, dt + \sigma S_t \, dW_t,$$

where:

- $\mu$is the drift
- σ is the volatility
- W_t is the standard Brownian motion

Using a solution of this differential found on Wikipedia we find the following equation:

$$S_{t+\delta t} = S_t \exp\left\{ \underbrace{\left(\mu - \frac{1}{2}\sigma^2\right)\delta t}_{\text{Deterministic term}} + \underbrace{\sigma\sqrt{\delta t}\,\epsilon}_{\text{Stochastic term}} \right\}$$

where ϵ is the standard normal random variable

Since the deterministic term is constant

$$E\left[\frac{S_{t+\delta t}}{S_t}\right] = e^{\left(\mu - \frac{1}{2}\sigma^2\right)\delta t} E\left[e^{\sigma\sqrt{\delta t}\,\epsilon}\right]$$

An interesting property found in **probIntroRoss** can be used here:

Our deterministic term is constant and our stochastic term is composed of the random variable ϵ scaled by a constant. $\epsilon \sim N(0,1)$ thus $\text{Var}[\epsilon] = 1$

Using properties of the variance:

$$\text{Var}\left[\ln\left(\frac{S_{t+\delta t}}{S_t}\right)\right] = \sigma^2 \, \delta t$$

We may also use the following approximation

The expected value of the ratio $\frac{S_{t+\delta t}}{S_t}$ is then

$$E\left[\frac{S_{t+\delta t}}{S_t}\right] = E\left[\exp\left\{\left(\mu - \frac{1}{2}\sigma^2\right)\delta t + \sigma\sqrt{\delta t}\,\epsilon\right\}\right]$$

Stochastic Calculus

A defining feature of stochastic calculus is its ability to handle processes whose paths are nowhere differentiable, such as Brownian motion. Unlike smooth functions in standard calculus, the trajectory of a Brownian motion is highly irregular, exhibiting infinite fluctuations over any time interval, no matter how small.