Project: Option Pricing using a Trinomial Tree Model

Chenjie Li - Double Degree - Quant Financial Engineering
ESILV & Université Paris Dauphine–PSL 272

October 28, 2025

Abstract

This paper develops and compares trinomial tree methods for option pricing, emphasizing rigorous construction under the risk-neutral measure and practical treatment of dividends. We present the mathematical framework (up/mid/down branching with calibrated probabilities p_u, p_m, p_d and multiplier α), backward induction for European and American options, and finite-difference estimates of Delta and Gamma. We analyze convergence toward Black–Scholes, explaining error spikes as strike K moves away from S_0 due to grid misalignment and payoff non-smoothness, and note simple smoothing fixes (terminal interpolation, $N/N{+}1$ averaging). To improve efficiency, we introduce pruning by reach-probability thresholds and volatility bands, showing large speedups for recursive solvers but limited gains for iterative backward passes. Implementations in VBA and Python (recursive with memoization + pruning, and iterative/vectorized) are benchmarked; runtime scales roughly as $\mathcal{O}(N^2)$. Small numerical discrepancies between languages stem from floating-point accumulation, `expm1`/renormalization choices, and date/dividend alignment but remain $\sim 10^{-12}$ and immaterial. We conclude with limitations (e.g., VBA stack depth, discrete dividend jumps) and extensions (American/exotic payoffs, adaptive trees, Monte Carlo/PDE). This work also draws on material from the *VBA & Python – Quant Info* course by Emmanuel Fruchard at Université Paris Dauphine–PSL.

1. Introduction

In this project, we focus on a fundamental aspect of quantitative finance: how a discrete model such as the trinomial tree can accurately approximate the continuous valuation provided by the Black–Scholes model.

The main objective is to design and compare different implementations of the trinomial model for pricing European and American options. This model relies on discretizing the evolution of the underlying asset into three possible movements at each step (up, stable, down). It converges toward the continuous Black–Scholes model as the number of steps increases.

First, we study the convergence of the trinomial model toward the theoretical Black–Scholes price. This analysis highlights specific behaviors, particularly the appearance of *convergence spikes* when the strike moves away from the spot price. These spikes occur because the tree's grid becomes misaligned with the payoff threshold, and because the payoff $(S-K)^+$ is not smooth in K.

In a second stage, we compare implementations in VBA and Python, focusing on performance and numerical differences. The analysis includes:

- execution speed and algorithmic complexity depending on tree size;
- pricing differences due to rounding and decimal representation in memory, which may vary between programming languages;
- the impact of optimization techniques such as pruning on recursive and iterative methods.

Finally, we provide a comparative synthesis to identify best practices depending on the language and approach used.

2. Trinomial Tree Model: Mathematical Framework and Dividend Handling

2.1 Discrete-time evolution of the underlying

The trinomial tree is a discrete-time model for the evolution of the underlying asset price S_t. Over a small time step Δt, starting from node (n, j) with underlying level $S_{n,j}$, the model assumes that at the next time level $n+1$, the asset can move to three possible states:

$$\begin{aligned} S_{n+1,j+1} &= S_{n,j} \cdot \alpha \quad \text{(up branch)}, \\ S_{n+1,j} &= S_{n,j} \cdot m \quad \text{(middle branch)}, \\ S_{n+1,j-1} &= S_{n,j} \cdot \frac{1}{\alpha} \quad \text{(down branch)}, \end{aligned}$$

where $\alpha > 1$ is the up/down jump multiplier and m is a drift-adjustment factor. A common construction is to align the middle branch with the risk-neutral forward ("no-arbitrage" drift).

We attach transition probabilities

$$p_u, \quad p_m, \quad p_d$$

for Up / Middle / Down. These satisfy

$$p_u + p_m + p_d = 1, \qquad p_u \geq 0, \quad p_m \geq 0, \quad p_d \geq 0.$$

Under the risk-neutral measure, we impose two calibration conditions at each node:

- The **expected value** of the stock after one step matches the risk-neutral drift:

$$\mathbb{E}[S_{n+1} \mid S_{n,j}] = S_{n,j} \cdot e^{(r-q)\Delta t},$$

  where r is the risk-free rate and q represents the dividend yield / carry term.

- The **variance** of the one-step move matches the target volatility σ:

$$\mathrm{Var}[S_{n+1} \mid S_{n,j}] \approx S_{n,j}^2 \cdot \sigma^2 \Delta t.$$

Solving these moments gives (p_u, p_m, p_d) as functions of $(\alpha, \Delta t, r, q, \sigma)$. In practice, we pick

$$\alpha \approx \exp(\kappa\, \sigma \sqrt{\Delta t}), \quad \kappa \simeq 3,$$

to spread the grid enough so that probabilities remain positive and numerically stable.

2.2 Backward valuation under the risk-neutral measure

Let N be the total number of time steps until maturity $T = N\Delta t$. Let $V_{n,j}$ denote the value of the option at node (n, j).

Terminal condition (payoff at maturity). For a European call with strike K:

$$V_{N,j} = \max(S_{N,j} - K, 0),$$

and for a European put:

$$V_{N,j} = \max(K - S_{N,j}, 0).$$

Backward induction (European option). For $n = N-1, N-2, \ldots, 0$:

$$V_{n,j} = e^{-r\Delta t}\Big(p_u\, V_{n+1,j+1} + p_m\, V_{n+1,j} + p_d\, V_{n+1,j-1}\Big).$$

Early exercise (American option). For an American claim, at each interior node:

$$V_{n,j} = \max\left(\text{Intrinsic}_{n,j},\ e^{-r\Delta t}\big(p_u\, V_{n+1,j+1} + p_m\, V_{n+1,j} + p_d\, V_{n+1,j-1}\big)\right),$$

where

$$\text{Intrinsic}_{n,j} = \begin{cases} \max(S_{n,j} - K, 0) & \text{(call)}, \\ \max(K - S_{n,j}, 0) & \text{(put)}. \end{cases}$$

The backward induction returns $V_{0,0}$, which is the option's fair value at $t = 0$.

2.3 Continuous vs. discrete dividends

Dividends are a major source of numerical subtlety.

Continuous yield. In the textbook setup with a continuous dividend yield q, the expected drift under the risk-neutral measure becomes $(r - q)$, which simply appears in

$$\mathbb{E}[S_{n+1} \mid S_{n,j}] = S_{n,j} \cdot e^{(r-q)\Delta t}.$$

In that case the tree remains "clean and centered" around the forward.

Discrete cash dividends. In equity markets, dividends are typically paid as *fixed cash amounts* on specific ex-dividend dates, not as a continuous yield. At a dividend date t_d with dividend D, the stock price drops by D:

$$S^+ = S^- - D.$$

This means every node that survives across t_d must undergo a downward jump.

This causes two issues:

- The "middle branch" is no longer just a scaled copy of the previous middle branch: after the cash drop, the grid realigns on a shifted level.
- The dates in the tree ($n\Delta t$) rarely match the real calendar dividend date exactly. We therefore detect "is this step at the dividend date?" using a tolerance, and then apply the cash drop to all reachable nodes at that step.

As a result, two implementations that apply the dividend at slightly different sub-steps or with slightly different date rounding can generate slightly different node sets and slightly different prices. This effect is especially visible for deep out-of-the-money options, where the price is driven by rare tail paths.

2.4 Greeks (Delta and Gamma) from the tree

We estimate sensitivity directly from the local structure of the tree near the root.

Delta. Approximate the derivative of option value with respect to the underlying using a one-step finite difference:

$$\Delta \approx \frac{V^{(\mathrm{up})} - V^{(\mathrm{down})}}{S^{(\mathrm{up})} - S^{(\mathrm{down})}},$$

where $V^{(\mathrm{up})}$ is the option value after an "up" move of the stock, etc.

Gamma. Gamma is estimated using a second difference:

$$\Gamma \approx \frac{\frac{V^{(\mathrm{up})} - V^{(\mathrm{mid})}}{S^{(\mathrm{up})} - S^{(\mathrm{mid})}} - \frac{V^{(\mathrm{mid})} - V^{(\mathrm{down})}}{S^{(\mathrm{mid})} - S^{(\mathrm{down})}}}{\frac{1}{2}\left(S^{(\mathrm{up})} - S^{(\mathrm{down})}\right)}.$$

In practice, we extend the first time step to generate synthetic "up / mid / down" values and reuse the same pricing logic.

3. Why do we observe convergence spikes (Trinomial → BS) when K moves away from S_0?

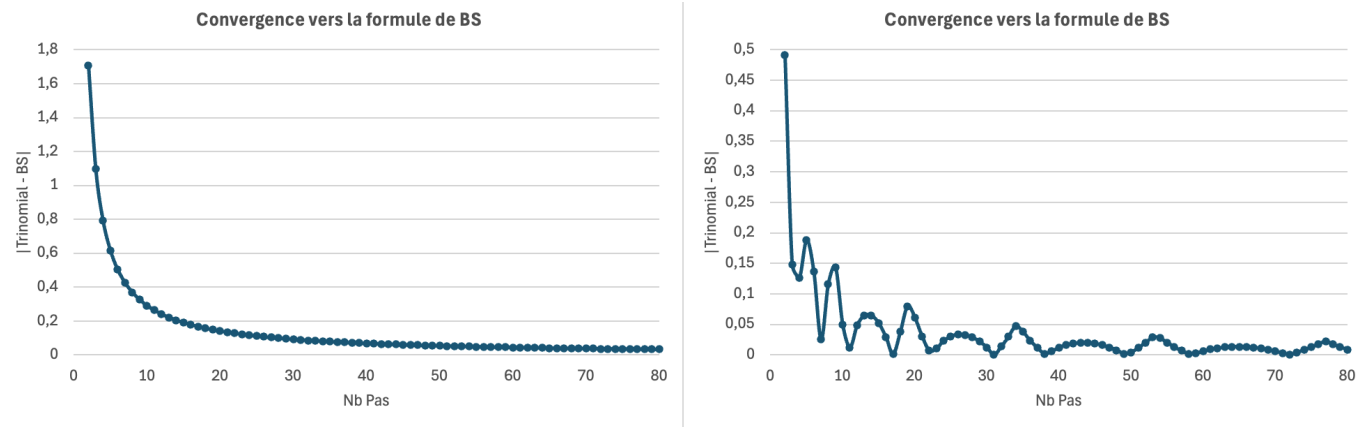

Figure 1: European Call Option – Left: S0@100/Strike@105; Right: S0@100/Strike@150 (other parameters identical)

Why spikes?

- **Threshold switch.** At maturity, the first in-the-money node $j^* = \min\{j : S_{N,j} > K\}$ can shift when N changes. The entry or exit of a node (and its probability) makes the price *jump*.
- **Tail weight.** When K is far from S_0, the price depends on rare events: a small variation in grid/probability can cause a large relative variation ⇒ spikes.
- **Non-smooth payoff.** $(S-K)^+$ has a kink at K. If K lies between two terminal nodes, grid misalignment with N produces oscillations.

Why smoother near ATM? Many nodes surround K; their contributions partially offset each other, resulting in nearly monotonic convergence.

How to smooth it?

- **Interpolation at strike** (linear) at maturity if $S_{N,j} \le K \le S_{N,j+1}$.
- **Average over N and N+1:** use $\frac{1}{2}(V_N + V_{N+1})$.
- **Align the grid** (when possible) to bring K closer to a terminal node.
- **Increase N / use a more regular scheme** (choice of α, Leisen–Reimer-style adjustments).

4. Error vs Strike: reading $|\text{Trinomial} - \text{BS}|$ as a function of strike K

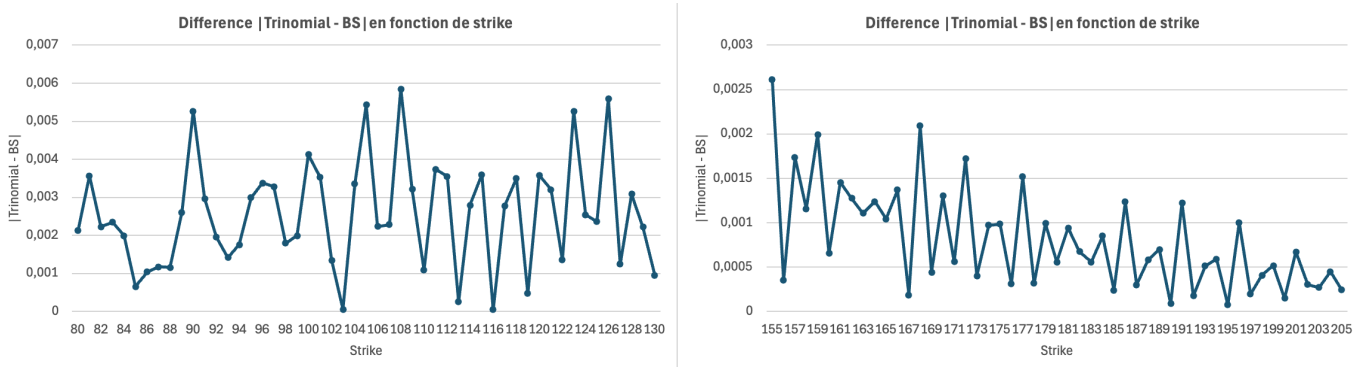

Figure 2: Convergence toward Black–Scholes as a function of Strike K – European Call, $S_0 = 100$

Left panel: $K \in [80, 130]$

- This region includes the *at-the-money* zone ($K \simeq S_0$).
- Near ATM, the curvature (gamma) of the option price is maximal: tiny discretization errors (ΔS, Δt) cause a **larger pricing error**.
- The trinomial model values the payoff on a **discrete grid** $S_{N,j}$. When K lies near a terminal node, the tree price (a piecewise function) "snaps" to the grid, while Black–Scholes is smooth $\Rightarrow$ **spikes/oscillations**.
- Consequence: **higher and irregular errors** in this strike range.

Right panel: $K \in [155, 205]$

- Here, the call is *deep out-of-the-money* ($K \gg S_0$): both price and gamma are small $\Rightarrow$ **lower sensitivity** to grid errors.
- The strike–grid misalignment still exists, but its effect is **significantly weaker**.
- Result: **lower average error** and **reduced spikes**.

Summary

- Two factors drive the difference between panels:
  1. **Moneyness:** high gamma near ATM $\Rightarrow$ larger errors.
  2. **Strike–grid misalignment** at maturity: piecewise tree pricing $\Rightarrow$ small "teeth" when K crosses discrete nodes.
- To **smooth/reduce** oscillations: increase N, apply payoff smoothing / interpolation at the final step, or use a **Richardson extrapolation**-type averaging.

5. Pruning and its role

5.1 Motivation

A plain trinomial tree with N time steps has on the order of $\mathcal{O}(N^2)$ nodes.
Recursive pricing explores a very large number of branches, including far OTM states that contribute almost nothing to the final price.
Pruning is an optimization that:

- cuts branches that are numerically irrelevant (extremely low probability of being reached),
- cuts branches that are economically irrelevant (states so far out-of-the-money that they will never come back in a meaningful way),
- reduces recursion depth, runtime, and memory pressure.

5.2 Probability-based pruning

We associate to each node (n, j) a reach probability $\pi_{n,j}$ under the risk-neutral measure. We then define a threshold ε (e.g. 10^{-7}). If

$$\pi_{n,j} < \varepsilon,$$

we discard that node and all of its descendants.

Practically:

- This avoids building huge subtrees in very low-probability tails.
- When we cut an "extreme" branch, we typically redirect its tiny probability mass back into a surviving branch so that $p_u + p_m + p_d = 1$ remains true locally.
- Result: the backward induction (or recursion) works almost unchanged, but on a much smaller effective tree.

5.3 Volatility-band pruning (standard deviation capping)

Another pruning rule is to refuse to explore nodes outside a certain number of standard deviations from the central (forward) path. After n steps, most of the mass lies within roughly $\pm k\sigma\sqrt{n\Delta t}$ in log-space. We keep only nodes in that band.
Interpretation:

- Nodes beyond that band have exponentially tiny probability.
- We track for each time layer n the minimum and maximum index j we allow.
- Everything outside that window is skipped.

5.4 Impact on recursive vs iterative pricing

Recursive pricing. Pruning is extremely effective. We simply never descend into irrelevant subtrees, which:

- prevents unnecessary function calls,
- keeps memoization tables small,
- reduces stack growth and lowers the risk of "Out of stack space" in VBA.

Iterative backward pricing. In an iterative backward pass, we already loop level-by-level through arrays of node values. Masking out some nodes does not drastically change the cost of scanning the whole layer. In fact:

- pruning can even add overhead (extra conditional checks),
- performance gains are modest to none.

Conclusion: pruning is mainly a performance booster for recursive implementations; it is less impactful for iterative backward solvers.

6. Runtime performance and practical ranking

6.1 What the plots show

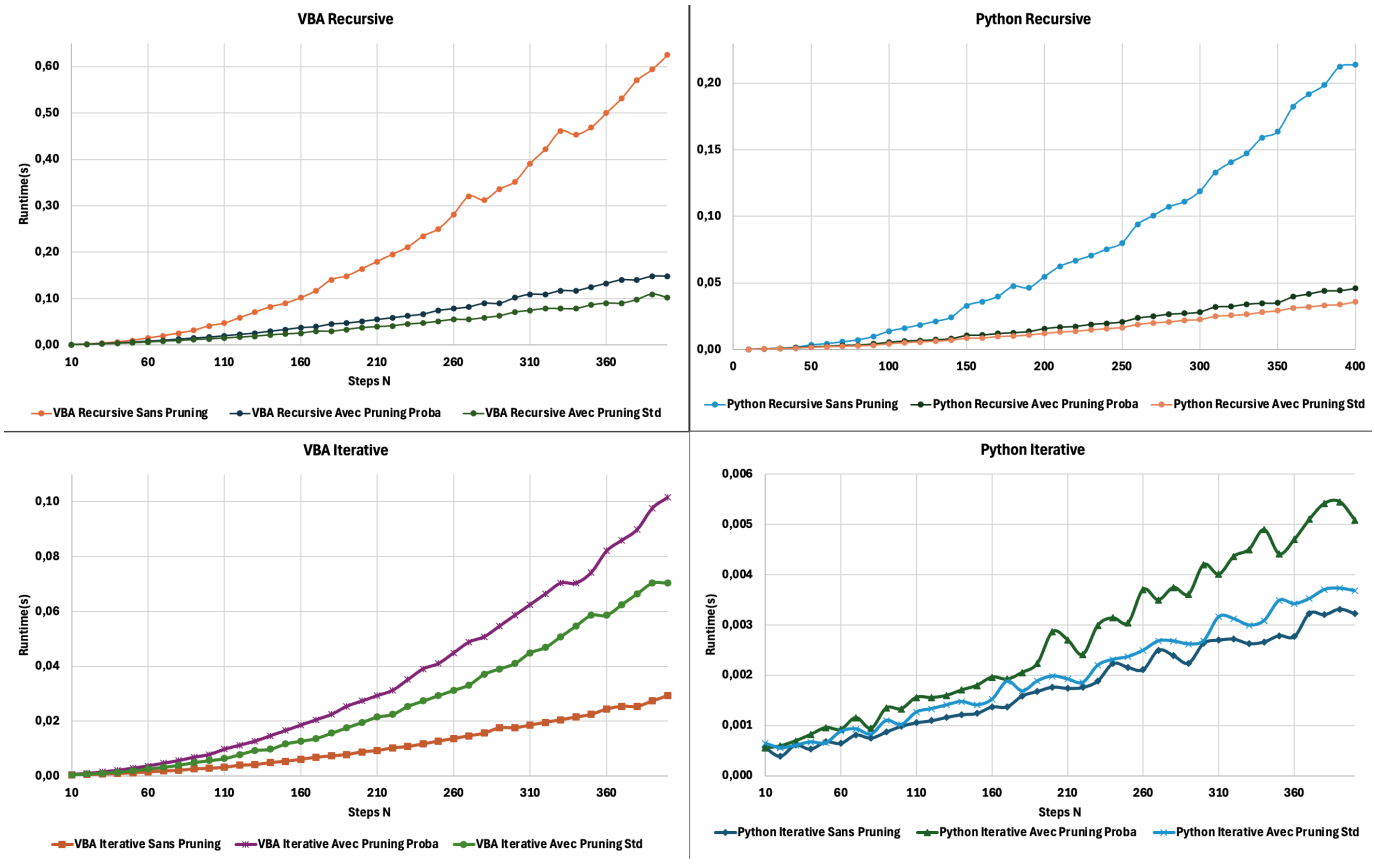

Figure 3: Pruning Effect on Runtime

From empirical timing:

- The slope in log–log runtime vs. N is almost linear, consistent with overall $\mathcal{O}(N^2)$ complexity for a recombining trinomial tree.
- **Python > VBA** for equivalent algorithms, mainly due to lower function call overhead and vectorization (NumPy-style memory access).
- **Iterative (backward array)** is generally cheaper than **recursive** plus memoization, unless recursion is aggressively pruned.

6.2 Ranking (fastest to slowest)

1. **Python – Iterative (no pruning):** fastest and most stable.
2. **Python – Recursive with pruning:** strong speed-up compared to naive recursion; can approach iterative speed if many branches are cut.
3. **VBA – Iterative (no pruning):** best choice on the VBA side.
4. **VBA – Recursive with pruning:** better than raw recursion, but still suffers from VBA call overhead and limited stack depth.
5. **Iterative with pruning** (Python/VBA): often not worth it, because additional checks outweigh the reduced state space.
6. **VBA – Recursive (no pruning):** slowest and most fragile (risk of "Out of stack space").

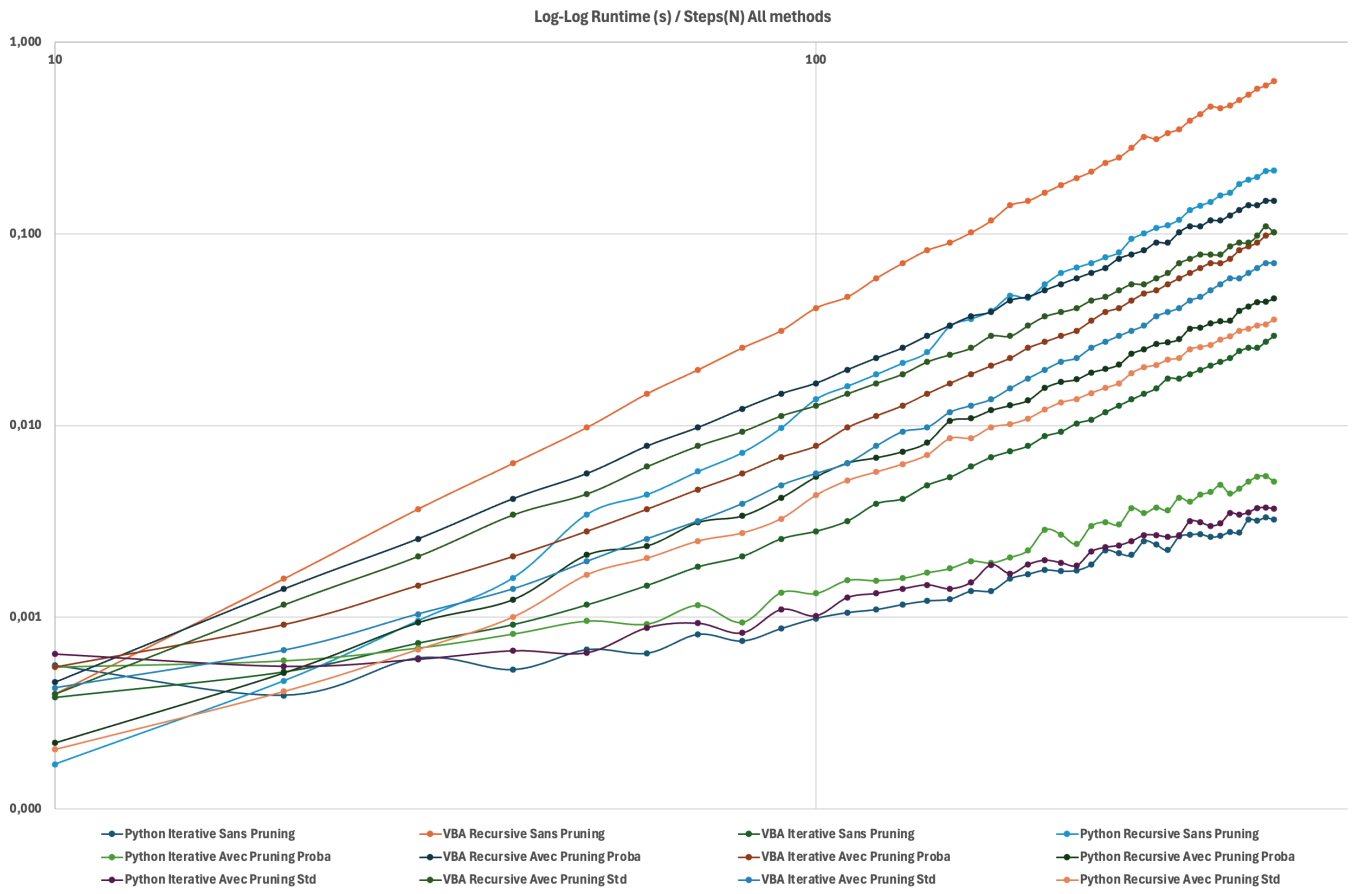

Figure 4: Runtime Comparison of All Methods

6.3 Practical conclusions

- **For production / single-price usage:** prefer **Python iterative without pruning**.
- **If recursion is imposed:** enable **probability/std pruning**, especially for large N, to avoid stack explosion.
- **In VBA:** favor **iterative backward induction**; only use recursion with pruning and for modest tree depths.
- **Pruning:** invaluable for recursive traversal; of limited or negative benefit for iterative backward arrays.

7. Implementation architecture: VBA vs. Python, iterative vs. recursive

7.1 Common goals

Both implementations (VBA and Python) share the same core tasks:

1. Build or implicitly represent the trinomial lattice of stock prices over N steps, including discrete dividends when they occur.
2. Compute the terminal payoff (European call/put, possibly American).
3. Roll back the option value under the risk-neutral probabilities, applying early exercise for American options.
4. Extract price, delta, gamma, runtime, and convergence diagnostics.

7.2 VBA implementation

We implemented two distinct approaches in VBA:

(a) Recursive VBA pricer.

- Each node is priced by a function that (i) gets the node's stock level, (ii) recursively calls its children (up/mid/down), (iii) discounts the expected continuation value, and (iv) applies early exercise logic if American.
- Memoization (caching) avoids recomputing the same node multiple times.
- Pruning is critical: we cut branches with negligible reach probability, or beyond a volatility band, to drastically reduce recursion.

Pros:

- Very close to the mathematical definition of backward induction.
- Natural place to insert early exercise checks.

Cons:

- VBA has limited call stack depth, so deep recursion can trigger "Out of stack space".
- High call overhead makes naive recursion slow.
- Requires pruning to be practical for large N.

(b) Iterative (backward) VBA pricer.

- We explicitly allocate arrays of node values at maturity.
- We then iterate backward in time: for each node index j at time n, compute the discounted expectation of its three children at $n+1$, and for American options take the max with intrinsic value.
- After finishing, the array at time 0 gives $V_{0,0}$, i.e. today's price.

Pros:

- No deep call stack, so it is numerically safer in VBA.
- Faster than recursion in practice.

Cons:

- Code becomes more mechanical (array indexing, bounds, etc.).
- Pruning brings little or no speed improvement, because we already traverse entire layers.

Summary for VBA: The iterative backward solver is the "production" approach in Excel: it is robust, predictable, and avoids stack overflows.

7.3 Python implementation

We mirror both approaches in Python:

(a) Recursive Python pricer.

- The structure is conceptually similar to the VBA recursive version: recursion + memoization + early exercise check.
- Pruning again reduces the number of visited branches.

Pros:

- Python handles recursion + dictionary memoization with less overhead than VBA.
- With pruning, runtime can fall dramatically vs. naive recursion.

Cons:

- Python still has a recursion depth limit for extremely large N.
- Handling discrete dividends and calendar date alignment still adds logic complexity.

(b) Iterative / vectorized Python pricer.

- This approach mirrors the iterative VBA algorithm but uses Python data structures (lists or NumPy arrays).
- We build/payoff at maturity, then roll backward layer by layer using vectorized operations when possible.

Pros:

- Extremely fast and stable.
- Clean memory model: just arrays per time layer.
- Natural place to compute Greeks by bumping the first layer.

Cons:

- Requires more careful array bookkeeping to keep the recombining structure consistent after discrete dividends.

Summary for Python: The iterative (vectorized) approach is the fastest and most scalable solution. Recursive + pruning is still interesting for clarity and for demonstrating how pruning accelerates a tree walk.

8. Why do Python and VBA not give exactly the same price?

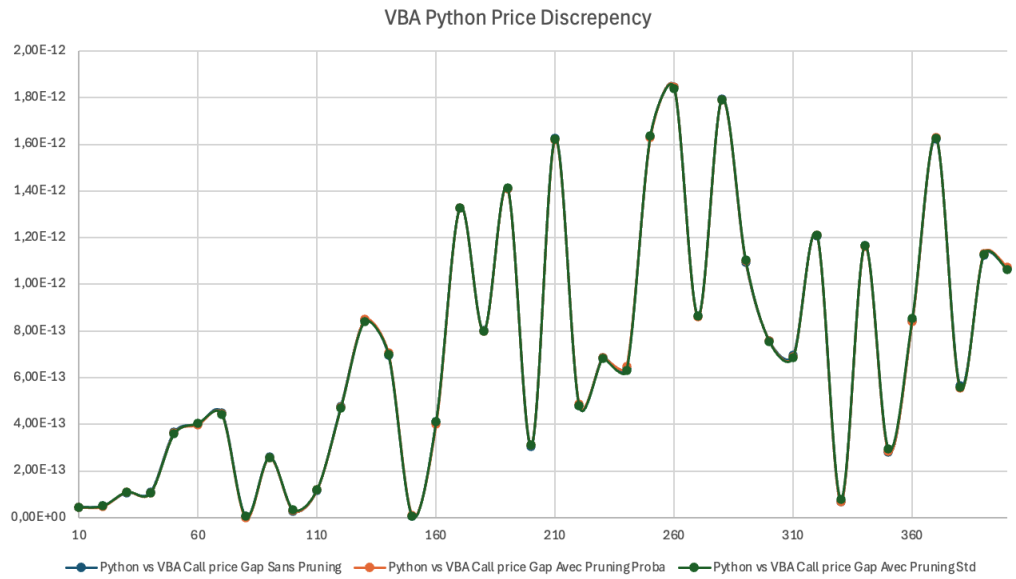

Figure 5: VBA–Python Price Discrepancy

#1 64-bit floats & operation order. Double precision arithmetic is not associative:

$$(a + b) + c \neq a + (b + c).$$

Python (vectorized sums, different loop order) and VBA (explicit `For` loops) do not accumulate rounding error in the same order, leading to typical discrepancies around 10^{-13}–10^{-12}.

#2 `exp` vs `expm1` and renormalization. Computing $e^x - 1$ as `expm1(x)` vs. $\exp(x) - 1$, and then forcing $p_d + p_m + p_u = 1$ to eliminate drift in probabilities, can introduce tiny differences for small x.

#3 Noise amplification across steps. Terms like $\alpha^{\pm j}$, repeated discounting, and division by small step sizes propagate machine noise as N grows (e.g. 100–400 steps).

#4 Decision rules and clamps. Conditions like $>$ vs. $\geq$, clamping $S \geq 10^{-12}$ to avoid negative prices after dividends, etc., can introduce microscopic discontinuities.

#5 Input conversion. Dates (string $\rightarrow$ date), or slightly different rounding of r, σ, cash dividends, etc., can differ by 10^{-15} in one language vs. the other.

Differences on the order of 10^{-12} are normal floating-point noise and have no practical pricing impact.

9. Conclusion

By reproducing the behavior of the Black–Scholes model from a discrete approach, we observed both the convergence mechanisms and the numerical limitations of a trinomial tree model.

Our analysis shows that convergence toward the Black–Scholes price depends strongly on the strike's position relative to the spot price. When the strike moves far from the initial level of the underlying, **convergence spikes** appear. These spikes come from misalignment between the strike and the terminal nodes of the tree, and from the kink (non-smooth derivative) of the payoff $(S-K)^+$ at the exercise boundary. Near the money (at-the-money), convergence is generally smoother, although the option's gamma is high and local discretization errors can temporarily dominate.

We also compared implementations in VBA and Python. Python proved faster, more stable, and more scalable thanks to vectorized iterative backward valuation. VBA is fully usable for moderate tree sizes, especially in iterative mode, but naive recursion in VBA is slow and can hit "Out of stack space". Introducing pruning is especially effective in recursive implementations, because pruning cuts entire subtrees with negligible probability mass and dramatically reduces runtime.

Finally, we quantified that Python and VBA can disagree by around 10^{-12} purely due to floating-point accumulation order, handling of exponentials and renormalization, and slightly different date/dividend alignment. These differences are negligible for practical pricing purposes.

This project suggests several natural extensions:

- pricing American and exotic options (early exercise, path dependence);
- adaptive or non-uniform trees to improve convergence without exploding node count;
- numerical alternatives such as Monte Carlo engines or finite-difference PDE solvers for more complex payoffs.