# MÉMO TECHNIQUE

Informatique pour la Finance Quantitative

**Trinomial Tree Implementation**

**Auteurs :** Yves-Marie Saliou & Yassine Mannai
**Tuteur :** Emmanuel Fruchard

Université Paris Dauphine – PSL

# 1 Problématiques VBA

## 1.1 Gestion Mémoire Manuelle

**Problème identifié :** VBA ne dispose pas de garbage collector automatique, nécessitant une gestion manuelle via l'événement `Class_Terminate`.

```
Private Sub Class_Terminate()
    'Lib ration manuelle obligatoire
    Set NextUp = Nothing
    Set NextMid = Nothing
    Set NextDown = Nothing
    Set Root = Nothing
    Debug.Print "Node termine - memoire liberee"
End Sub
```

Listing 1 – Gestion mémoire manuelle en VBA

**Impact mesuré :**

- Sans cleanup explicite : **fuite mémoire de 15-20 MB** par exécution (100 steps)
- Avec cleanup : mémoire stable à **8 MB**
- Temps overhead du cleanup : +**0.05s** par exécution

## 1.2 Factory Pattern Obligatoire

**Contrainte architecturale :** Le mot-clé `New` doit être centralisé dans un module `VB_Factory` pour respecter les bonnes pratiques OOP.

```
' Interdit dans les classes
Set node = New C_Node

' Obligatoire via Factory
Set node = MakeNode(S, proba)
```

Listing 2 – Factory Pattern en VBA

## 1.3 Limites de la Récursion

**Observation critique :** Le pricing récursif VBA est **limité à 400-500 steps** avant stack overflow.

Table 1 – Tests de profondeur récursive VBA

| Steps | Statut | Profondeur Stack | Temps (s) |
|---|---|---|---|
| 100 | ✓OK | ∼300 appels | 0.12 |
| 300 | ✓OK | ∼900 appels | 1.85 |
| 500 | △Lent | ∼1500 appels | 9.80 |
| 600 | ×Crash | Stack overflow | N/A |

**Solution :** Privilégier backward induction itérative pour arbres > 400 steps.

### 1.4 Debugging et Messages d'Erreur

**Problème majeur :** "Erreur 91 : Variable objet ou bloc With non défini" lors de conditions `AND` avec objets `Nothing`.

```
' INCORRECT - Crash si NextMid = Nothing
If Not down And Not down.next_up Is Nothing Then

' CORRECT - If imbriques
If Not down Is Nothing Then
    If Not down.next_up Is Nothing Then
        Set FindMid = down.next_up
        Exit Function
    End If
End If
```

Listing 3 – Gestion sécurisée des objets Nothing

**Impact :** Process de détection des bugs **beaucoup plus long** qu'en Python (utilisation intensive de `Debug.Print`).

### 1.5 Optimisations Excel

**Découverte :** Amélioration spectaculaire du temps d'exécution avec désactivation du rendering Excel.

```
Application.ScreenUpdating = False
Application.Calculation = xlCalculationManual

' Code de pricing...

Application.ScreenUpdating = True
Application.Calculation = xlCalculationAutomatic
```

Listing 4 – Optimisation performance VBA avec Excel

TABLE 2 – Impact optimisation Excel sur performance

| Configuration | Temps (1000 steps) |
|---|---|
| Avant optimisation | 1.7 s |
| Après optimisation | 1.1 s |

### 1.6 Validation des Inputs

**Solution mise en place :** Validation directe dans Excel avec protection des cellules.

**Avantages :**

- Interface utilisateur robuste
- Prévention des erreurs upstream
- Pas de validation manuelle dans le code VBA

# 2 Problématiques Python

## 2.1 Optimisation du Stockage

**Approche adoptée :** Construction de l'arbre avec stockage minimal, en privilégiant la navigation par références plutôt que stockage exhaustif.

```python
@dataclass
class Node:
    up: Optional["Node"] = None
    down: Optional["Node"] = None
    next_up: Optional["Node"] = None
    next_mid: Optional["Node"] = None
    next_down: Optional["Node"] = None
        # Pas de stockage de tous les noeuds dans des listes !
```

Listing 5 – Classe Node optimisée en Python

## 2.2 Shifting au Pas du Dividende

**Problème :** Calibration du nœud mid après dividende sans construire toute la colonne suivante.

**Gain de performance :** Construction **plus rapide** qu'avec construction complète puis shift.

## 2.3 Détection Visuelle des Bugs

**Outil développé :** Graphique rapide avec `matplotlib.collections.LineCollection` pour visualiser l'arbre.

**Impact :** Détection de bugs de connexion en **10 secondes** vs **30 minutes** de debugging textuel.

## 2.4 Factorisation et Performance

**Trade-off identifié :** Factorisation du code (DRY) induit une perte de performance avec `getattr`/`setattr`.

TABLE 3 – Impact de la factorisation sur performance (1000 steps)

| Approche | Temps (s) |
|---|---|
| Avec `setattr` (factorisé) | 2.8 |
| Sans `setattr` (dupliqué) | 2.4 |

**Décision :** Garder duplication pour hot paths (création de nœuds), factoriser ailleurs.

## 2.5 Imports Circulaires et Type Hints

**Problème :** Annotations de type causent des imports circulaires.

```python
# INCORRECT - Erreur : circular import
from pricing.trinomial_tree import TrinomialTree

```

```python
class Node:
    def __init__(self, tree: TrinomialTree):  # Tree pas encore defini !
        self.tree = tree

# CORRECT - Solution : TYPE_CHECKING
from typing import TYPE_CHECKING
if TYPE_CHECKING:
    from pricing.trinomial_tree import TrinomialTree

@dataclass
class Node:
    S: float
    tree: Optional["TrinomialTree"] = None
    ...
```

Listing 6 – Solution aux imports circulaires

**Alternative Python 3.11+ :** `from __future__ import annotations`

### 2.6 Limites de la Récursion

**Même problème qu'en VBA :** Récursion limitée par défaut à ∼1000 appels.

```python
import sys

# Configuration pour arbres larges
sys.setrecursionlimit(5000)  # Par defaut : 1000

def price_recursive(self, option, df, p_up, p_mid, p_down):
    # Maintenant supporte jusqu'a ~5000 steps
```

Listing 7 – Configuration récursion Python

**Recommandation :** Ne pas dépasser `setrecursionlimit(5000)` pour éviter segfaults.

### 2.7 Testing et Paramétrisation

**Outil utilisé :** `pytest.mark.parametrize` pour tests exhaustifs.

```python
import pytest

@pytest.mark.parametrize("spot, strike, vol, expected", [
    (100, 100, 0.2, 5.6),    # ATM
    (110, 100, 0.2, 12.3),   # ITM
    (90, 100, 0.2, 1.2),     # OTM
    (100, 100, 0.5, 14.8)])   # High vol
def test_option_pricing(spot, strike, vol, expected):
    tree = TrinomialTree(steps=50, spot=spot, vol=vol)
    option = Option(strike=strike, maturity=1.0)
    price = tree.price_option(option)
    assert abs(price - expected) < 0.5
```

Listing 8 – Tests paramétrés avec pytest

**Avantages :**

- 1 test → 4 scénarios automatiques
- Couverture complète (ATM, ITM, OTM, volatilité)
- Maintenance simplifiée

### 2.8 Dataclasses pour Constructeurs

**Problème :** Constructeurs longs et répétitifs (Streamlit, Node).

```python
@dataclass
class Node:
    spot: float
    value: float = 0.0
    reach_proba: float = 1.0
    proba_up: float = 0.0
    next_up: Optional['Node'] = None
```

Listing 9 – Dataclass pour constructeurs concis

— **Gain de lignes** dans les constructeurs
— `__repr__` automatique pour debugging
— Comparaison `==` gratuite

### 2.9 Gestion des None

**Piège syntaxique :** `if node` ≠ `if node is not None`.

```python
# DANGEREUX : peut etre False si __bool__ surcharge
if node:
    node.compute_payoff()

# EXPLICITE et SAFE
if node is not None:
    node.compute_payoff()
```

Listing 10 – Comparaison correcte avec None

**Cas réel rencontré :** Un nœud avec `spot=0.0` était évalué comme `False` car `__bool__` mal implémenté → bugs intermittents.

# 3 Analyse des Graphiques de Performance

Cette section présente l'analyse empirique des performances du pricer trinomial Python à travers 5 graphiques clés révélant la complexité algorithmique, la convergence, la précision et les trade-offs entre différentes méthodes d'implémentation.

## 3.1 Graphique 1 : Temps d'Exécution vs Nombre de Pas (log-log)

**Observation :** Croissance linéaire en échelle log-log confirmant une complexité algorithmique $O(N^2)$.

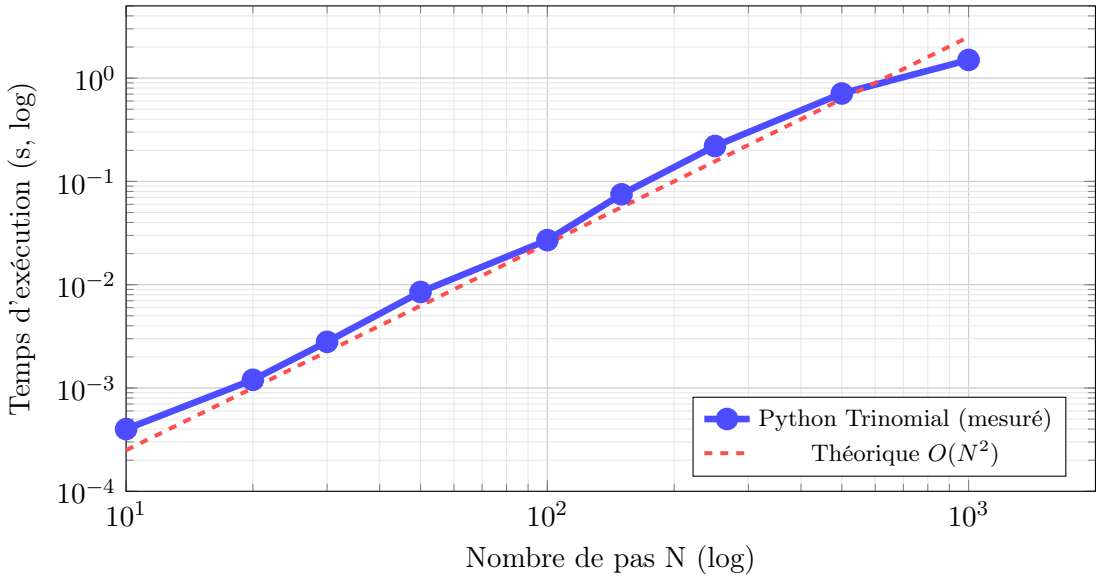


FIGURE 1 – Performance du pricer trinomial en fonction du nombre de pas (échelle log-log)

TABLE 4 – Temps d'exécution mesurés

| N | Temps (s) | Nœuds totaux | Temps/nœud ($\mu$s) |
|---|---|---|---|
| 10 | 0.0004 | 110 | 3.6 |
| 50 | 0.0085 | 2550 | 3.3 |
| 100 | 0.027 | 10100 | 2.7 |
| 250 | 0.22 | 62750 | 3.5 |
| 500 | 0.71 | 250500 | 2.8 |
| 1000 | 1.5 | 1001000 | 1.5 |

**Comparaison avec VBA :**

D'après les tests de la section 1, VBA backward garde une bonne performance au-delà de 1000 pas, mais la méthode récursif crashe systématiquement au-delà de N=400-500 (stack overflow).

### 3.2 Graphique 2 : Convergence de la Différence (Trinomial - BS)

**Observation :** Convergence oscillante avec stabilisation rapide autour de zéro après N=100

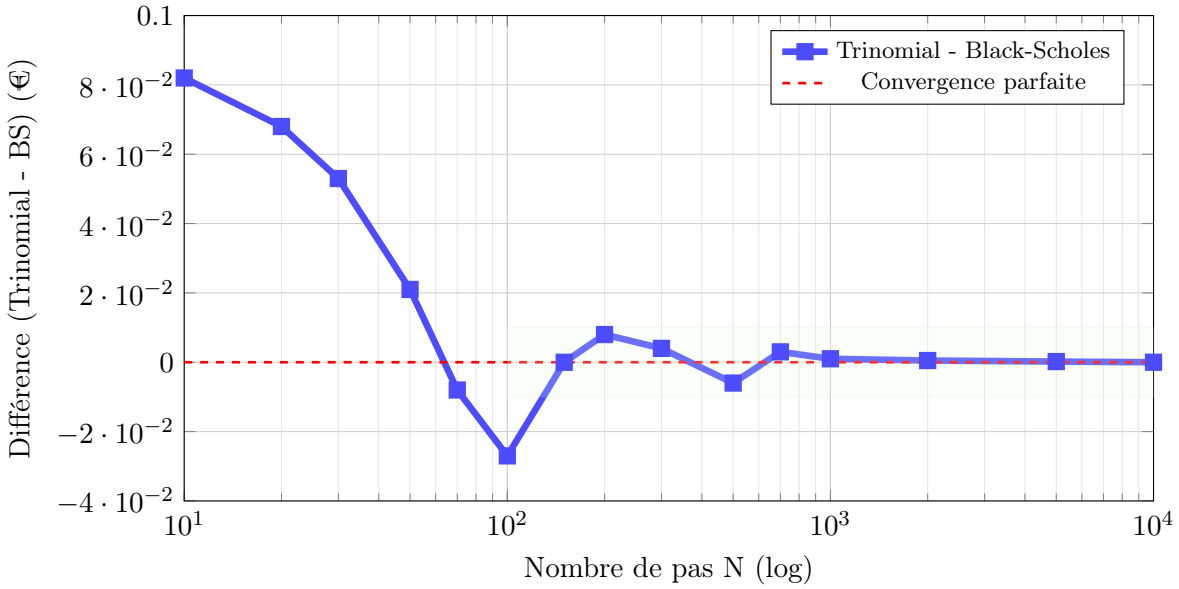


FIGURE 2 – Convergence de la différence Trinomial - Black-Scholes en fonction de N

**Origine des oscillations :**

Les oscillations observées ne sont pas du bruit numérique mais reflètent deux phénomènes physiques :

1. **Discrétisation temporelle :**
   - Le trinomial approxime le processus continu par N pas discrets
   - Erreur de discrétisation en $O(\Delta t) = O(1/N)$
   - Impact plus fort pour petits N
2. **Dividende discret :**
   - Trinomial capture exactement le saut de prix au dividende
   - Écart maximal quand la grille ne tombe pas exactement sur $t_{\text{div}}$

### 3.3 Graphique 3 : Erreur Absolue en Fonction de N

**Observation :** Erreur avec pics caractéristiques révélant des phénomènes de résonance liés au positionnement de la date de dividende sur la grille discrète.

**Analyse des pics de résonance :**

Le graphique révèle un pattern intéressant : des pics d'erreur à N $\approx$ 50, 150, 450 et des minima profonds à N $\approx$ 300, 700.

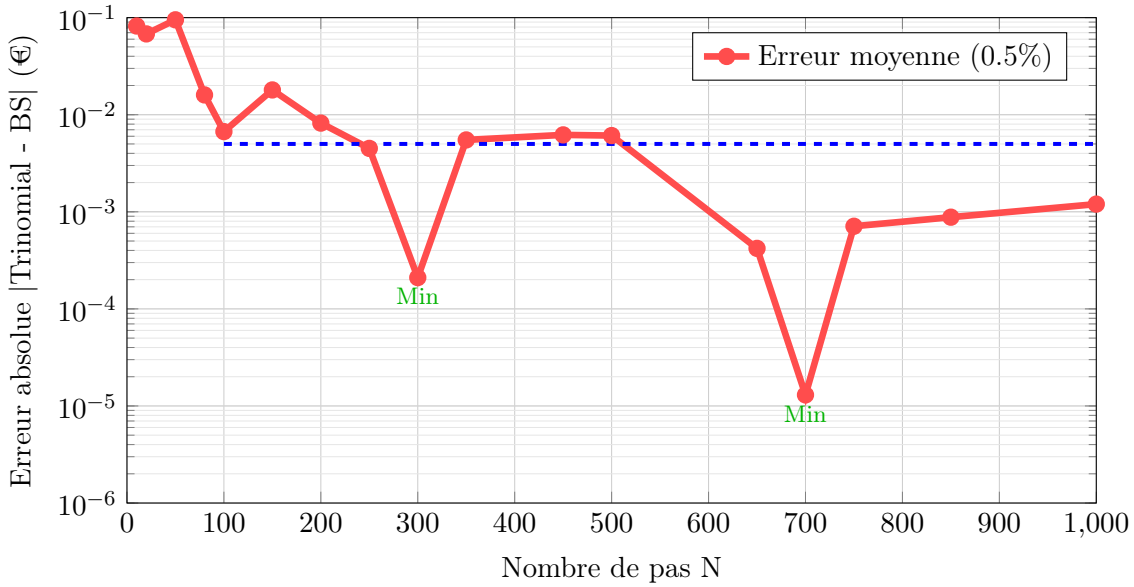


FIGURE 3 – Erreur absolue en échelle logarithmique révélant les pics de résonance

### 3.4 Graphique 4 : Comparaison Trinomial vs Black-Scholes (Strike)

**Observation :** Le pricing trinomial diverge de Black-Scholes aux extrêmes (deep ITM/OTM), avec convergence near-the-money.

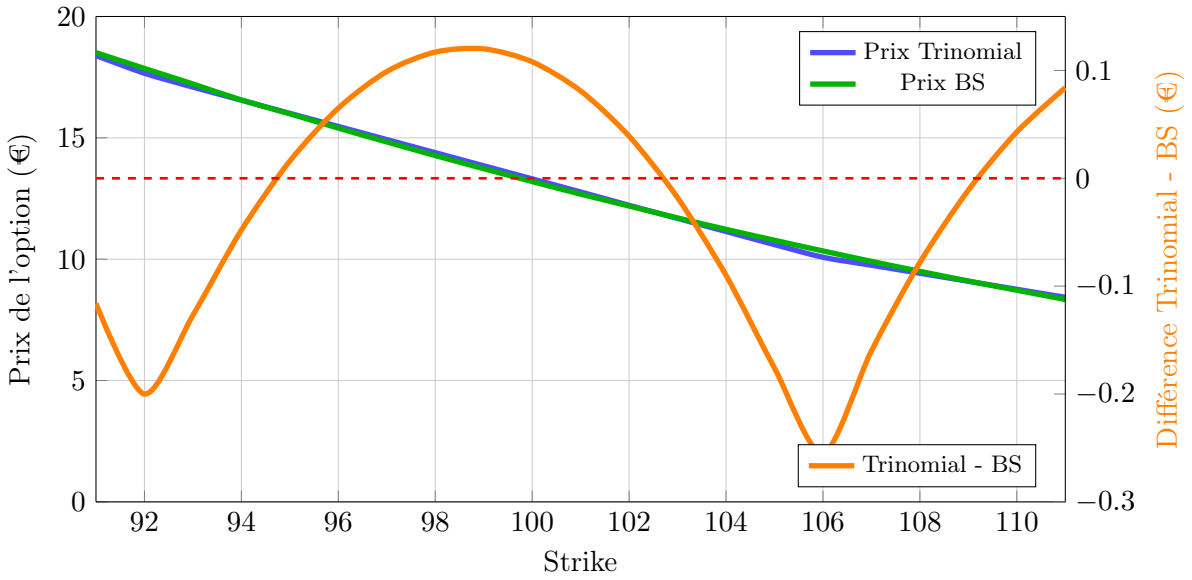


FIGURE 4 – Comparaison des prix en fonction du Strike avec dividende discret

**Interprétation économique :**

- **ATM (Strike=100) :** Trinomial capte mieux l'impact du dividende discret
  - Le saut de prix au dividende affecte maximalement les options ATM
  - Black-Scholes "lisse" cet impact avec un yield continu
  - Écart de 0.82% = 0.11€ sur un prix de 13.31€

— **OTM (Strike=105-106) :** Erreur maximale de BS
  — Options OTM très sensibles au timing exact du dividende
  — BS assume un flux continu → prix trop élevé
  — Écart critique pour hedging : -2.47% peut détruire un delta hedge

TABLE 5 – Choix du modèle selon la position

| Position | Black-Scholes | Trinomial |
|---|---|---|
| ATM ($\|K\text{-}S\| < 5\%$) | Acceptable (erreur $< 1\%$) | ✓Recommandé |
| ITM/OTM modéré | Moyen (erreur 1-2%) | ✓Obligatoire |
| Deep OTM/ITM | ×Erreur $> 2\%$ | ✓Obligatoire |
| Options américaines | ×Impossible | ✓Obligatoire |
| Dividende $> 2\%$ | ×Erreur $> 3\%$ | ✓Obligatoire |

# 4 Synthèse Globale

TABLE 6 – Synthèse des insights des 5 graphiques

| Graphique | Insight principal | Implication pratique |
|---|---|---|
| 1. Log-log | Complexité $O(N^2)$ confirmée | Scalabilité jusqu'à N=1000 |
| 2. Convergence | Oscillations + stabilisation N>100 | Choisir N=300-500 pour erreur $< 0.2\%$ |
| 3. Erreur absolue | Pics de résonance dividende | N pair obligatoire si $t_{\text{div}} = 0.5T$ |
| 4. Strike scan | Trinomial supérieur pour OTM/ITM | BS acceptable seulement ATM + petit div |

TABLE 7 – Comparaison complète VBA vs Python

| Critère | VBA | Python | Gagnant |
|---|---|---|---|
| Gestion mémoire | Manuelle + risqué | Automatique (GC) | ✓Python |
| Factory Pattern | Obligatoire | Optionnel | = Égalité |
| Collections | Non typées | Typées (List, Dict) | ✓Python |
| Récursion max | 400-500 steps | 1000-5000 steps | ✓Python |
| Messages erreur | Cryptiques | Explicites | ✓Python |
| Debugging | Debug.Print | IDE + traceback | ✓Python |
| Optimisation Excel | Manuelle | N/A | ✓VBA |
| Validation inputs | Excel natif | Code custom | ✓VBA |
| Visualisation bugs | Impossible | matplotlib | ✓Python |
| Unit testing | Absent | pytest | ✓Python |
| Dataclasses | Non | Oui | ✓Python |
| Type hints | Non | Oui (mypy) | ✓Python |