Présentation du cas : Arbre Trinomial et ses extensions.

Pour ce projet, nous avons développé un arbre trinomial afin de modéliser différentes trajectoires de l'actif sous-jacent et d'obtenir ainsi le prix d'une option à l'échéance. Notre approche a consisté à implémenter cet arbre sous forme orientée objet, à la fois en VBA et en Python. Nous avons d'abord choisi Python pour créer la structure de notre modèle, un langage avec lequel nous sommes plus à l'aise.

Dans cette première partie, nous vous présenterons les étapes de conception et les résultats obtenus avec chaque langage. Ensuite, nous procéderons à une comparaison des performances de VBA et Python, en termes de vitesse de calcul, utilisation de la mémoire, et précision des résultats.

Architecture commune :

- **Class Market** : Cette classe représente le marché financier. Elle contient les informations de base nécessaires pour les calculs de l'arbre, telles que le prix initial de l'actif sous-jacent (*S0*), le taux d'intérêt sans risque (*rate*), la volatilité (*sigma*), ainsi que les dividendes et la date ex-dividende, si applicable.
- **Class Option** : Cette classe modélise une option, incluant son prix d'exercice (*K*), son échéance (*T*), le type d'option (call/put), et son style (américain/européen). Elle inclut une méthode pour calculer le *payoff* de l'option.
- **Class Tree** : Cette classe représente l'arbre trinomial utilisé pour évaluer l'option. Elle contient les méthodes pour construire l'arbre, calculer les probabilités, et effectuer la propagation inverse pour déterminer le prix de l'option.
- **Class Node** : Cette classe représente un nœud dans l'arbre trinomial. Chaque nœud contient les informations sur la valeur de l'actif sous-jacent, les voisins associés (haut, bas, milieu), ainsi que les probabilités et le prix de l'option associé au nœud.
- **Class BlackScholes** : Cette classe calcule le prix d'une option européenne (call ou put) en utilisant la formule de Black-Scholes. Elle prend en compte les paramètres du marché (prix initial de l'actif, taux d'intérêt sans risque, volatilité, dividendes) ainsi que les paramètres de l'option (prix d'exercice, échéance). La classe inclut des méthodes pour calculer les termes *d1* et *d2*, puis utilise ces valeurs dans les formules de Black-Scholes pour obtenir le prix de l'option call ou put.

Différence d'architecture entre VBA et Python :

L'idée était d'avoir une structure au plus similaire pour pouvoir avoir une vraie comparaison entre VBA et Python pour une même méthode. Les différences de code ou de structure de fonctions sont principalement dues aux modifications apportées pour respecter la rigueur recommandée par l'outil de syntaxe.

Extensions supplémentaires sur Python :

- **Class Metrics** : Cette classe calcule et évalue les performances du modèle d'arbre trinomial. Elle permet de comparer les temps de calcul, l'utilisation de la mémoire et la convergence du modèle trinomial par rapport à Black-Scholes. Une fonction supplémentaire en Python est l'option de *light_mode*, qui utilise une version allégée de l'arbre pour réduire la complexité et le nombre de nœuds générés.
- **Class Greeks** : Cette classe calcule les sensibilités des prix d'options (*Delta*, *Gamma*, *Theta*, *Vega*, *Rho*) à l'aide du modèle d'arbre trinomial. Elle utilise une approche numérique par différence finie.

La Class utilise le modèle *Light* pour des calculs plus rapides. Les méthodes utilisent un calcul par dérivation numérique, ce qui permet d'optimiser les calculs avec des ajustements de pas (via la classe **OneDimensionalDerivative**).

- **Class Tree** : En plus des fonctions de construction d'arbre trinomial standard, la version Python inclut une fonction *light_build_tree* pour une construction d'arbre plus rapide en simplifiant la génération des voisins (moins de nœuds générés). La fonction *light_manage_forward_for_trunc* gère les connexions des nœuds de manière plus légère et optimise la construction pour les grands nombres de pas. Les nœuds peu significatif ne sont pas générés grâce au *pruning*, et les nœuds suivant sont supprimés une fois que leur valeur a été prise en compte dans le nœud précédent (On supprime le suivant car on part de la fin pour retourner à la racine de l'arbre).
- **Class Node** : Cette classe gère les nœuds de l'arbre trinomial. Dans la version Python, des méthodes supplémentaires comme *light_forward_neighbor*, *light_generating_neighbors*, et *light_manage_forward_for_trunc* permettent de générer et manipuler les nœuds de manière simplifiée. Ces méthodes sont conçues pour réduire la consommation de mémoire et le temps de calcul.

Extensions

- **Factor** : Nous avons aussi implémenté un factor pour créer une incrémentation de pas de temps (Δt) qui n'est pas constante ce qui nous a obligé à faire varier notre alpha dynamiquement et nous imposant donc un nouveau challenge.
- **Threshold** : Il est également possible d'utiliser du *Pruning* dans Python et VBA, en définissant une probabilité cumulée sous laquelle nous ne voulons pas créer de nouveaux de nœuds. Il existe une seconde méthode de *Pruning* intégrée cette fois ci dans le modèle *Light*, qui permet de générer un certain nombre de nœuds supérieurs et inférieurs à chaque étape, qui dépendra du pas de temps où on se trouve, et du nombre d'écart-type que nous voulons prendre en compte.

Support d'utilisation

- **Excel :** Notre premier support est sur le fichier Excel « TrinomialTreeModel.xlsx » et contient notre méthode VBA, ainsi que la méthode Python, grâce à l'utilisation de « xlswings ». Cela nous permet d'avoir un outil direct de comparaison entre les deux langages.
- **Streamlit** : Nous avons également hébergé notre projet sur Streamlit, nous permettant d'y accéder sur internet à travers ce lien : « https://trinomialtreemodel-272.streamlit.app/ ». Ce support simplifie l'expérience utilisateur et permet de ne pas dépendre d'installation comme « xlswings ». Cependant, la gestion de la mémoire est limitée puisque l'application est hébergé, et ne peut pas être beaucoup saturée en mémoire.

Analyses comparatives Des Résultats entre Python et VBA :

N = 100

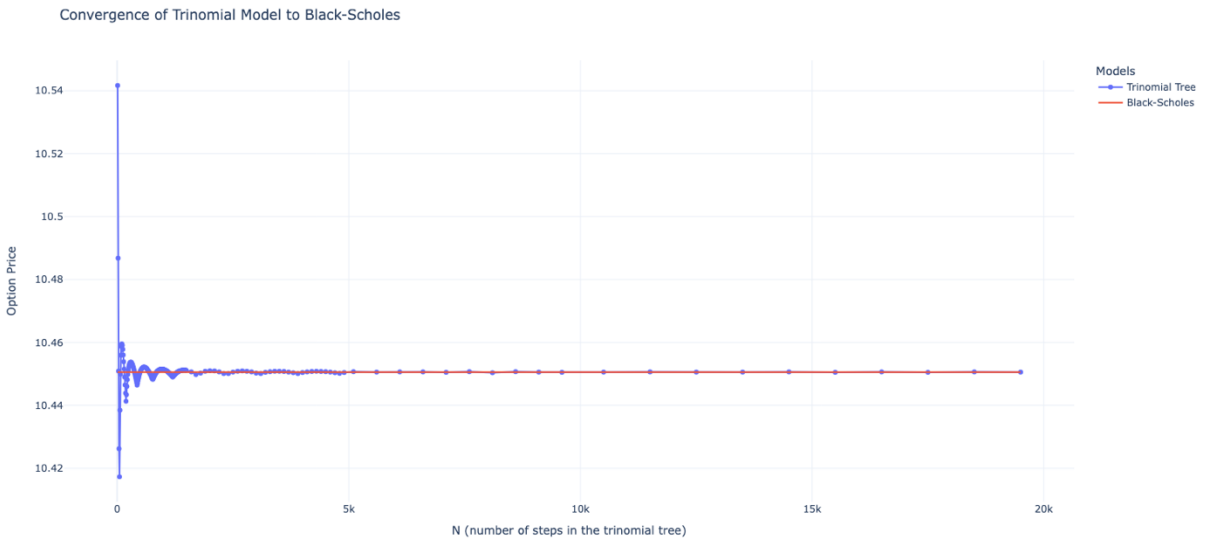
Pricing		
	Python	VBA
Price	**8.419499229**	**8.419499229**
Time (s)	0.021663	0.53125
Gap with B&S	-0.000274094	-0.000274094

N = 500

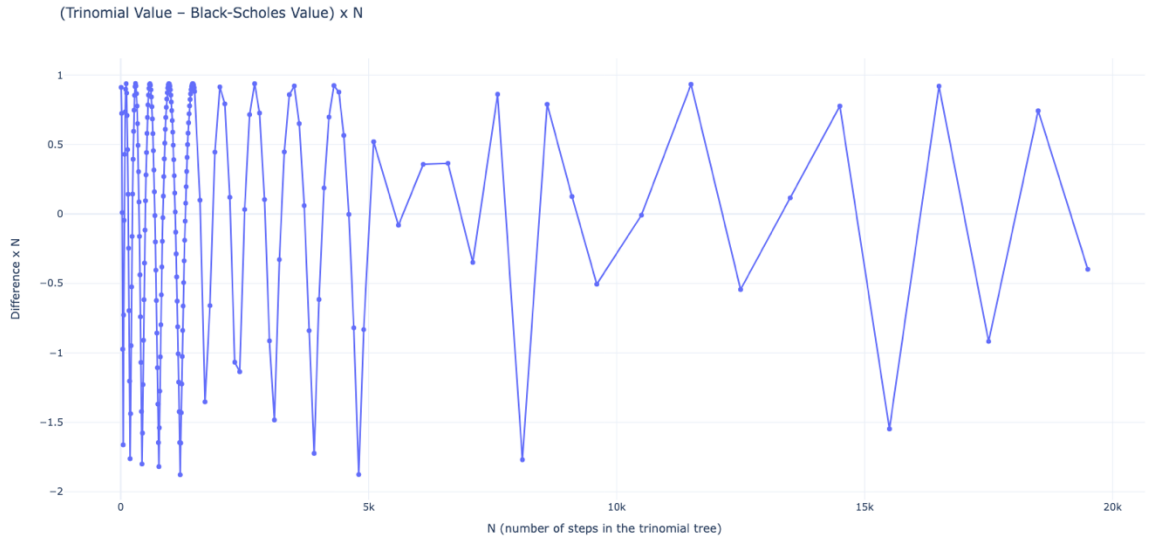
Pricing		
	Python	VBA
Price	**8.419612064**	**8.419612064**
Time (s)	0.646338	12.69140625
Gap with B&S	-0.000386929	-0.000386929

Comme nous pouvons l'observer, la différence de temps de calcul entre Python et VBA est vite conséquente, nous le verrons graphiquement plus tard. Il est intéressant de voir que nos modèles, Python comme en VBA, obtiennent la même valeur.

<u>Convergence du Prix vers Black & Scholes</u> :



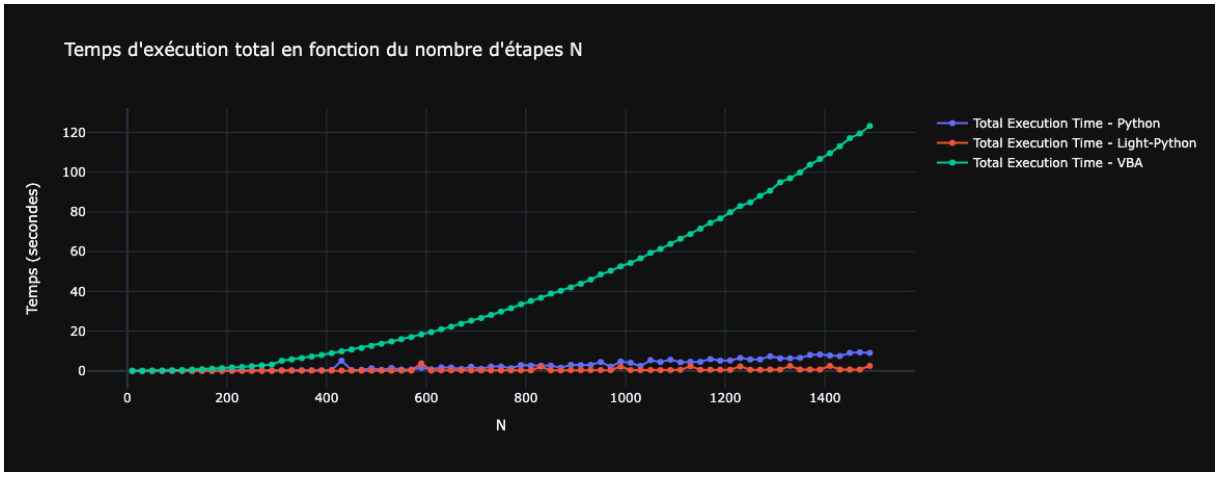
Au fur et à mesure que le nombre d'étapes dans l'arbre trinomial augmente, les oscillations de la courbe s'atténuent progressivement, et la courbe bleue se rapproche de plus en plus de la ligne rouge représentant le modèle de Black-Scholes. Cette tendance indique que le modèle trinomial converge vers le prix calculé par Black-Scholes. Avec un nombre suffisamment élevé d'étapes, les deux modèles fournissent des résultats similaires. Cela montre que le modèle trinomial, en augmentant sa précision avec plus d'étapes, peut approcher les résultats de Black-Scholes, tout en offrant plus de flexibilité pour des options plus complexes, comme les options américaines.



Le graphique montre que la différence entre le modèle trinomial (Tree) et Black-Scholes (BS), multipliée par le nombre d'étapes (Steps), oscille dans un tunnel. Ce comportement traduit une convergence progressive vers le modèle de Black-Scholes, mais avec des fluctuations autour de zéro. Plus le nombre d'étapes augmente, plus le tunnel devient stable, reflétant une meilleure approximation.

Si malheureusement ce tunnel n'est pas parfait après un grand nombre d'étapes, cela s'explique par l'élargissement de l'intervalle entre les étapes. En augmentant cet intervalle, nous perdons en précision sur la capture complète de l'amplitude des oscillations.

<u>Différence de temps d'exécution entre nos différents modèles</u> :

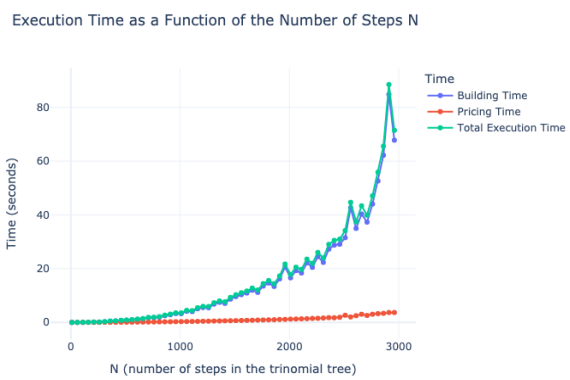


Ce graphique compare le temps d'exécution total en fonction du nombre d'étapes N pour trois méthodes : Python, Light Python, et VBA.

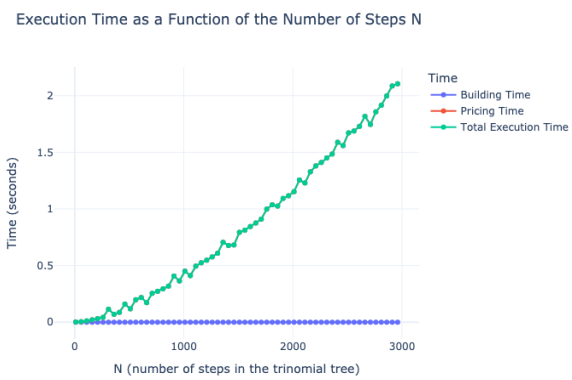
- La courbe verte (VBA) montre une augmentation rapide du temps d'exécution avec N, indiquant une gestion moins efficace des calculs.
- La courbe bleue (Python) est beaucoup plus stable, avec un temps d'exécution presque constant même pour un grand nombre d'étapes.
- La courbe rouge (Light Python) est légèrement plus rapide que Python standard, grâce au *Pruning* principalement.

En résumé, Python est plus performant que VBA, et la version Light Python améliore encore cette efficacité.

Python



Light Python

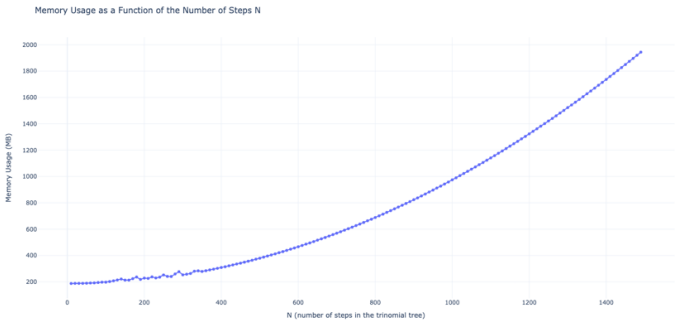


On voit que le temps d'exécution est nettement supérieur pour la version classique sans *Pruning*, puisque le nombre de nœud grandi à chaque étape. Pour la version *Light* avec *Pruning*, le nombre de nœud est défini à chaque étape élimine une quantité significative de nœuds, plus N est grand.

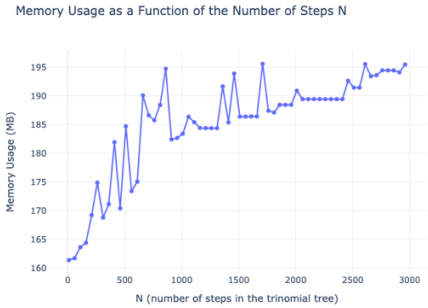
On observe également que la durée du *Pricing* n'est pas ce qui prend le plus de temps, c'est la construction de l'arbre. Dans le second graphique la construction de l'arbre est nulle puisqu'on créer les nœuds en partant de la fin, et on réalise directement le *Pricing*. Le temps d'exécution dans le second cas représente donc uniquement l'étape du *Pricing*.

Différence d'utilisation de mémoire entre la version Light et celle de base :

Python

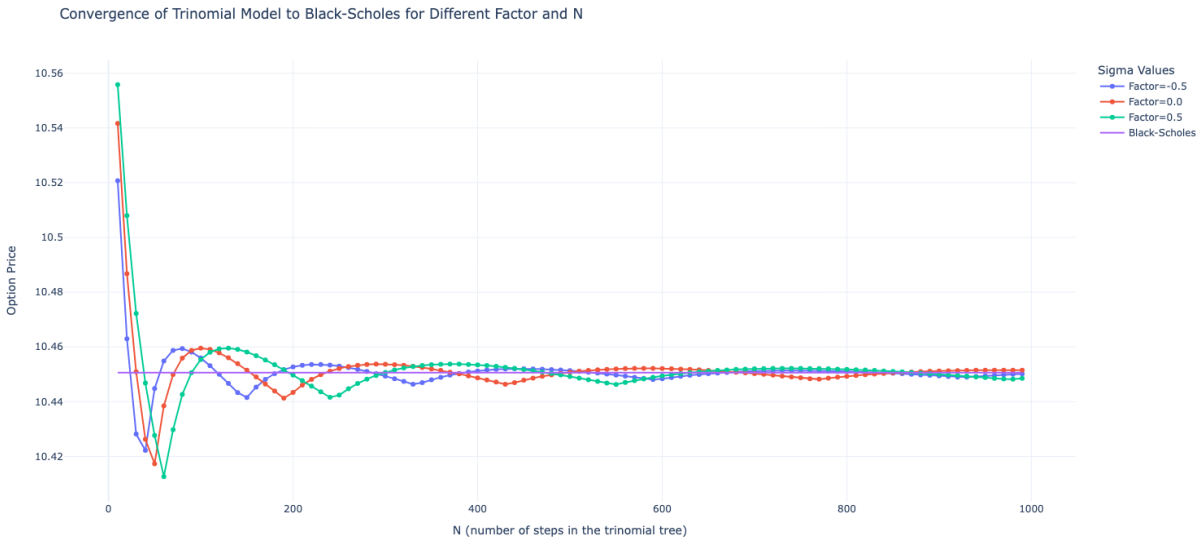


Light Python



Dans notre modèle classique, le nombre de nœuds stockés est exponentiels, ce qui explique un usage de la mémoire exponentiels au plus nous réalisons d'étapes. Pour la version *Light*, les nœuds sont supprimés après utilisation, ce qui nous permet d'être performant sans utiliser une grande quantité de mémoire. Cette méthode nous permet donc d'aller chercher un nombre d'étape très élevée, la seule vraie contrainte restera le temps de calcul (Bien qu'on ait vu qu'il soit moins conséquent avec le *Pruning*).

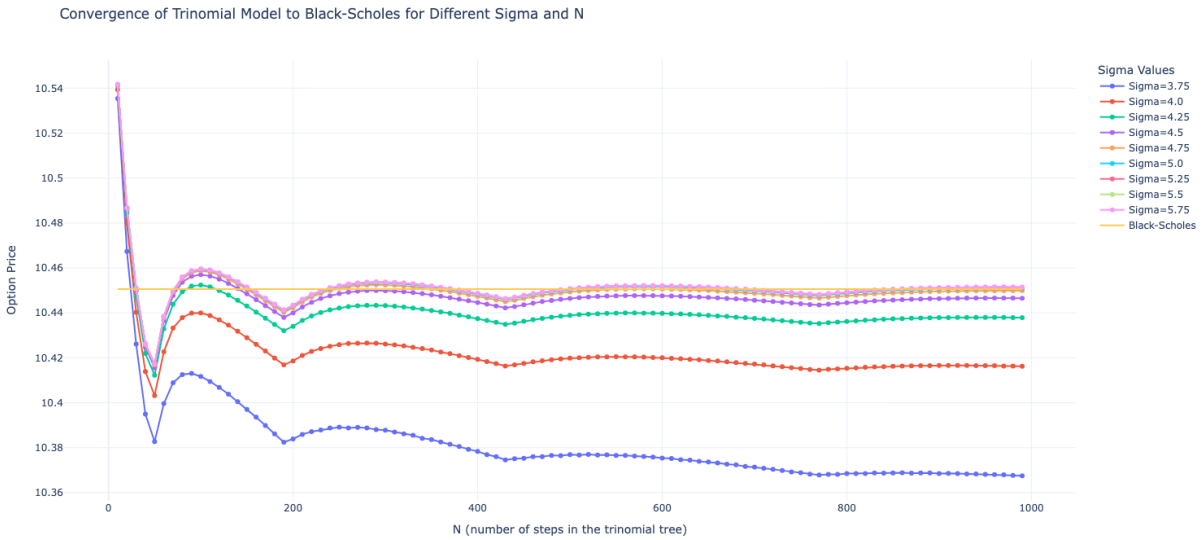
Différence de valorisation en fonction du facteur :

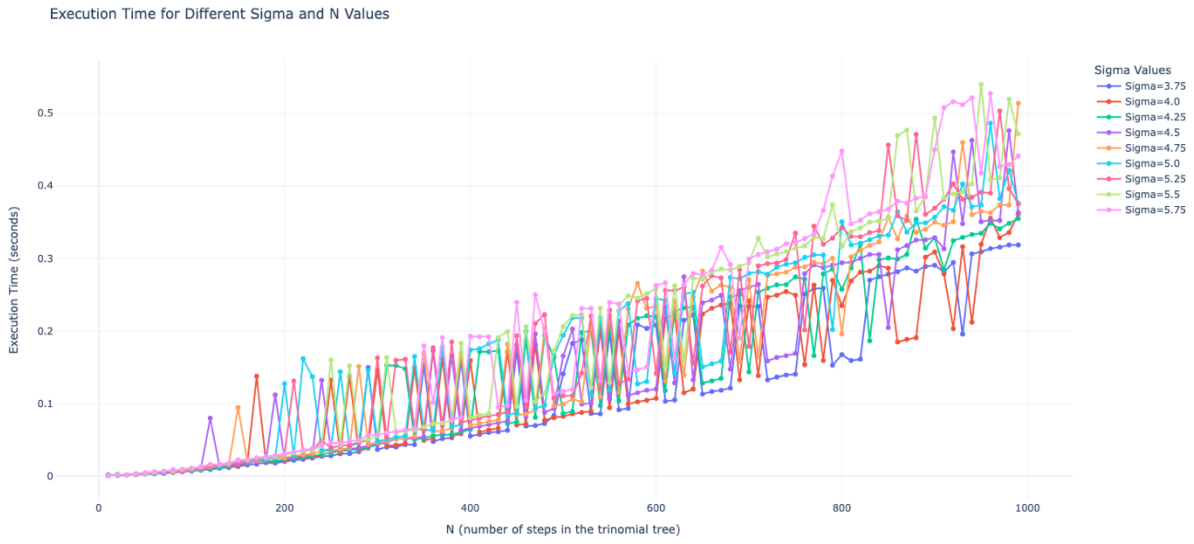


Dans notre modèle, un facteur inférieur à zéro indique un pas de temps de plus en plus faible dans le temps. Un facteur à zéro impliquera un pas de temps de constant dans le temps, et un facteur supérieur à zéro représentera un pas de temps de plus en plus grand.

Il est intéressant de voir qu'un facteur négatif, fait converger notre modèle plutôt, ce qui voudrait dire qu'on pourrait obtenir une valeur plus précise avec potentiellement moins d'étapes N. Un facteur positif retarde la convergence, ce qui impliquerai une plus grande imprécision pour un même nombre d'étapes N. Ces résultats sont concluants, mais mériterais une analyse plus profonde pour voir si nous convergeons bien vers Black & Scholes sans dérives.

<u>Évolution de la convergence en fonction du nombre d'écart-type utilisé pour le *Pruning*</u> :





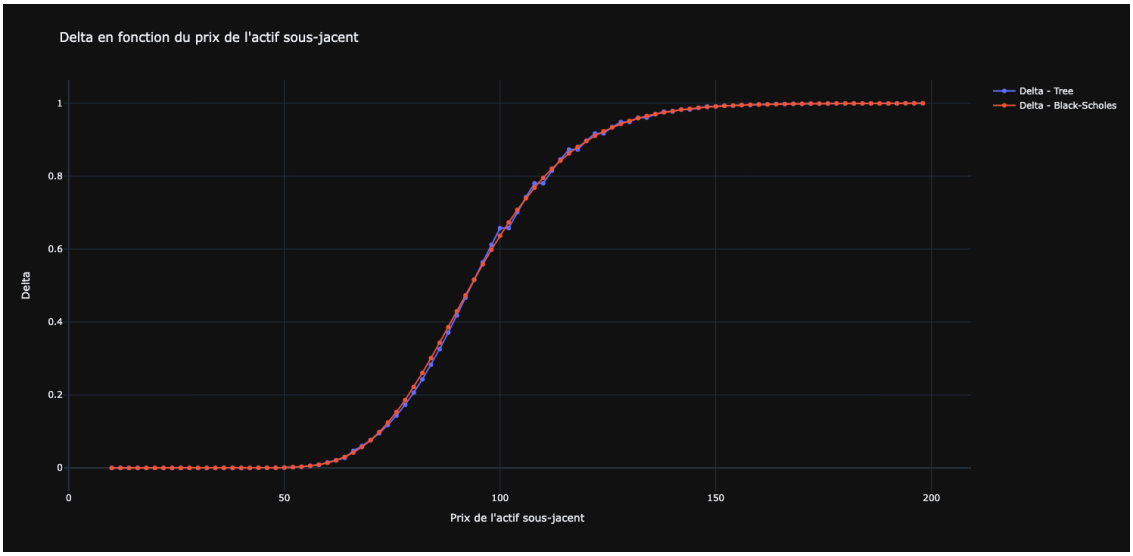
On observe ici que si on sélectionne un nombre d'écart type faible pour le *Pruding*, par exemple 3.75 écart-types représenté par la ligne bleue sur le premier graphique, on aura donc moins de nœud, ce qui représente un temps d'exécution plus court. En revanche on voit que le nombre de nœuds est insuffisant pour obtenir un prix qui converge vers Black & Scholes, car on perd trop de nœuds qui sont important pour obtenir un prix cohérent. On observe graduellement que plus notre nombre d'écart-

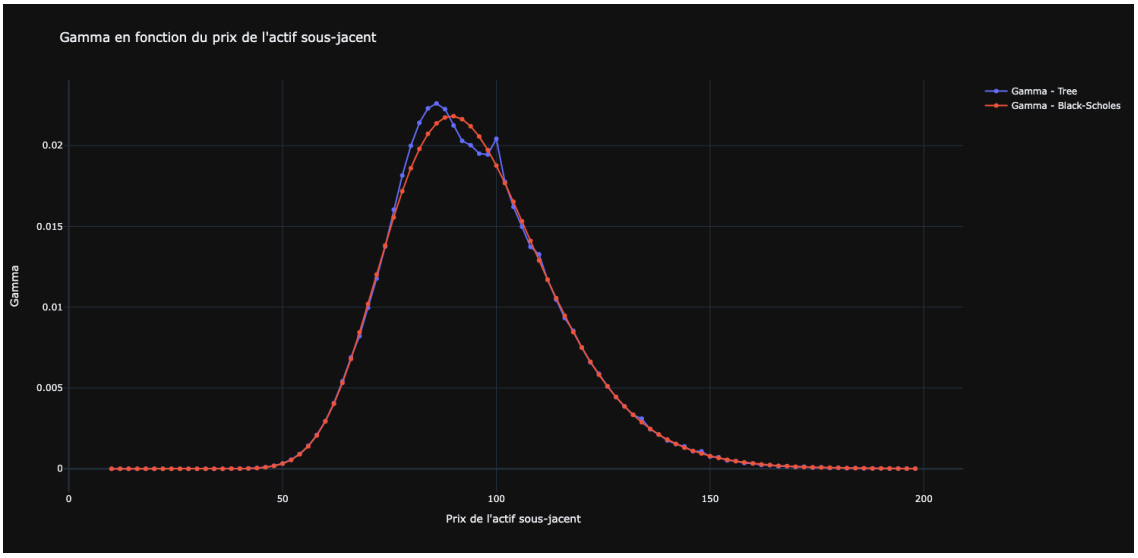
type est grand, plus notre temps d'exécution augmente légèrement, mais meilleur est notre convergence.

On estime qu'un nombre d'écart-type suffisant ce situe en 5 et 6.

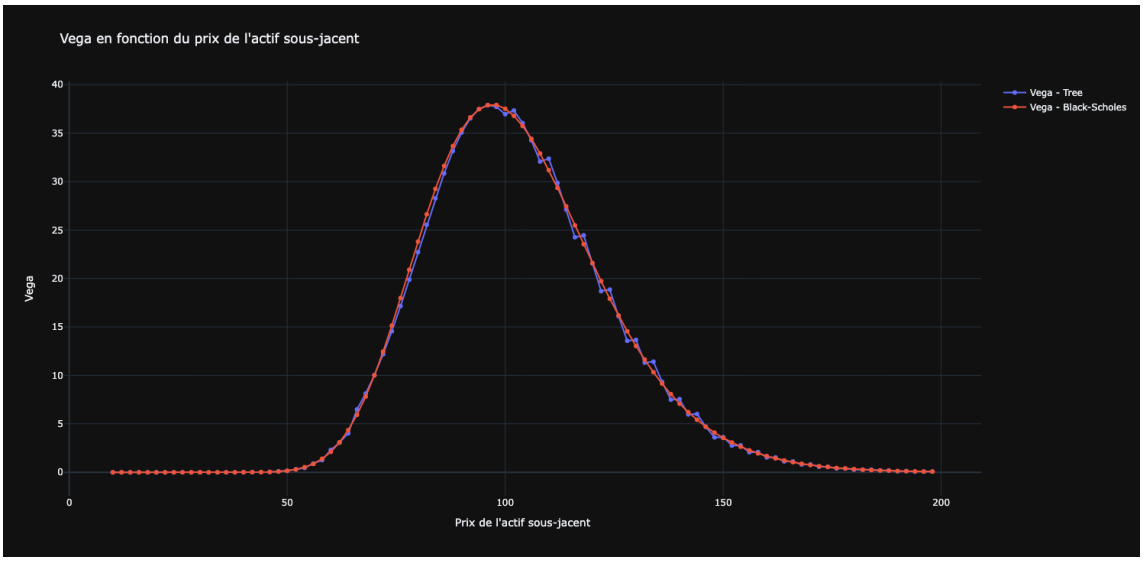
Les différents Grecs comparés à ceux de la formule fermée de Black & Scholes :

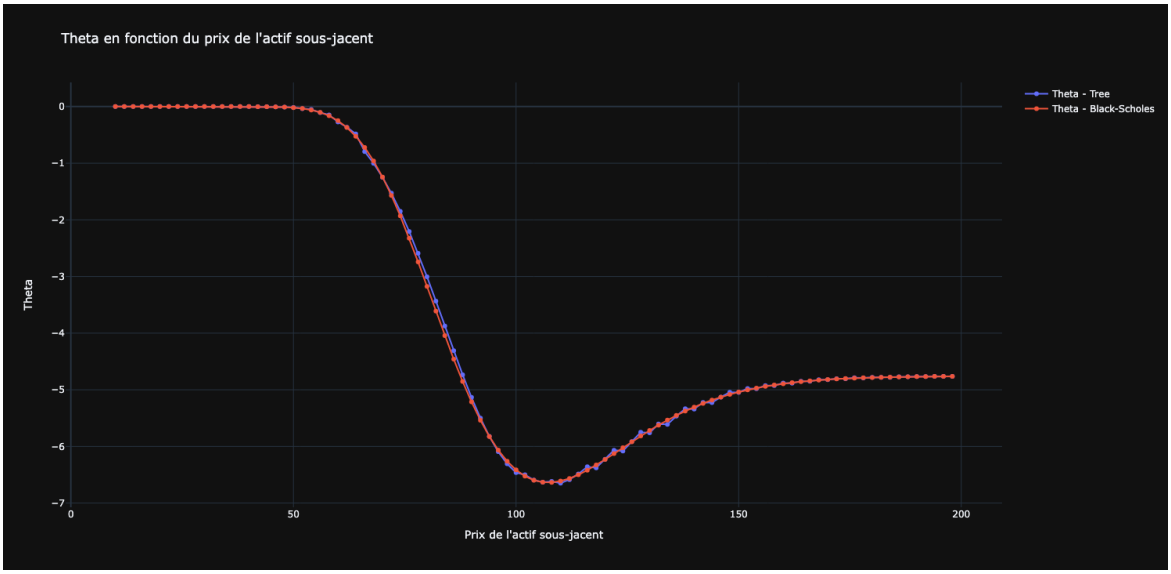
Voici un exemple de l'évolution des *Grecs* pour un Call, en comparant Black & Scholes et les valeurs de notre modèle.

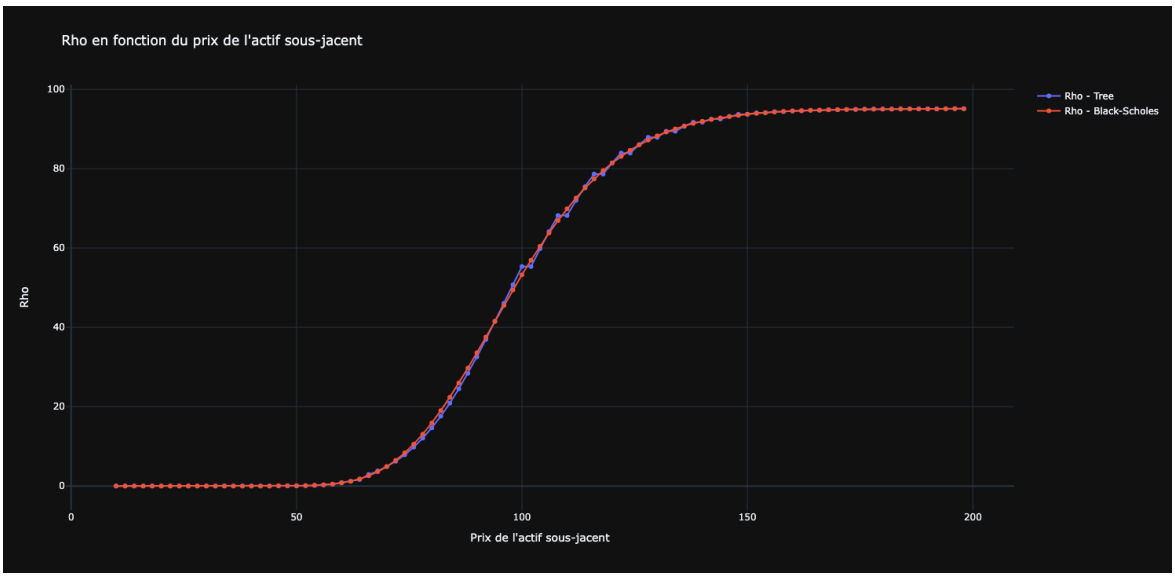




Pour Gamma, la différence entre les deux courbes est due à la dérivée qui n'est pas monotone, nous devons donc utiliser un shift plus grand pour obtenir notre valeur, et la précision est pénalisée. Un travail supplémentaire pourrait être fait pour trouver la valeur de shift optimale.


Vega en fonction du prix de l'actif sous-jacent
Vega - Tree
Vega - Black-Scholes
Vega
Prix de l'actif sous-jacent
0
5
10
15
20
25
30
35
40
0
50
100
150
200


Theta en fonction du prix de l'actif sous-jacent
Theta - Tree
Theta - Black-Scholes
Theta
Prix de l'actif sous-jacent
0
−1
−2
−3
−4
−5
−6
−7
0
50
100
150
200


Rho en fonction du prix de l'actif sous-jacent
Rho - Tree
Rho - Black-Scholes
Rho
Prix de l'actif sous-jacent
0
20
40
60
80
100
0
50
100
150
200

<u>Problématiques Rencontrées</u> :

- **Construction de l'Arbre en Python :** Notre approche initiale visait à développer la construction de l'arbre de tarification en stockant toute nos valeurs dans des *arrays* pour ensuite pourvoir les représentés, mais cette méthode, en plus de ne pas être celle attendue, c'est avéré plus lente lorsque nous avons augmenté notre nombre de nœud. Il a donc fallu penser à une autre méthode pour créer l'arbre.

- **Création de l'arbre en objet :** La principale problématique a été de comprendre comment l'arbre se génère, et comment les nœuds doivent être liés les uns avec les autres. Il nous a fallu du temps pour comprendre le mécanisme et visualiser comment nous allions procéder pour créer notre arbre. Forte heureusement, nous avons su nous débloquer et nous entraider pour pouvoir avancer dans ce projet. La mise en place de graphique sur papier et de schéma a été d'une grande aide pour nous représenter à l'intérieur de l'arbre et ainsi mieux comprendre le mécanisme.

- **Utilité de la représentation graphique :** Le réel gain de temps que nous avons eu a été de créer assez rapidement des outils de visualisations, soit sous forme de graphiques directement dans python, soit dans une feuille Excel. Ce processus nous a permis de mettre le doigts sur de nombreux problèmes tout au long de la construction du modèle.

- **Dividende :** L'implémentation du dividende nous a challengé car nous obtenions des probabilités négatives sur certains de nos nœuds, nous avons donc dû forcer l'attribution de certains nœuds en fonction de leurs valeurs. En implémentant les facteurs, et donc une augmentation (ou diminution) de la taille du pas de temps dans le temps, nous avons rencontré des problèmes de valeurs trop élevés dû à un pas de temps de trop conséquent. Ces valeurs mettaient en péril notre boucle *While* de création de nœuds supérieurs ou inférieurs. Nous avons donc décidé de limiter la taille de l'augmentation du pas de temps (facteur).

- **Implémentation dans VBA :** Une fois notre code bien structuré et organisé, nous avons entrepris de traduire le modèle de base en VBA, et de créer une interface dans Excel utilisant *xlswings*. Nous avons eu un peu de fil à retorde avec l'outil de Syntaxe qui nous demandé de modifier légèrement notre structure comparée à Python.

- **Modèle Light :** Par chance, notre structure en Python nous a permis de créer assez rapidement et aisément un modèle dit « Light » qui correspond à une version plus efficiente en mémoire que le modèle classique. Cependant, même s'il fonctionne pour des dividendes faibles, nous n'avons pas réussi pour l'instant à corriger l'attribution des nœuds lorsque les chocs de prix sur les dividendes sont trop conséquents. Cela résulte par des prix imprécis et peu cohérent dans certaines configurations. Il vaut donc mieux utiliser le modèle standard (en python ou VBA, même si on a montré que Python est plus rapide) si l'on veut s'assurer d'avoir un prix précis.

<u>Conclusion</u> : Ce projet a été extrêmement formateur pour nos deux autant sur l'aspect technique du travail demandé notamment sur le développement de notre compréhension de la programmation orienté objet mais aussi la gestion de travail et la coordination du groupe lors du développement du projet. Ce projet nous a permis de travailler à plusieurs sur un même code nous efforçant à le rendre clair et lisible pour les futurs utilisateurs.