Pricing d'Option & Arbre Trinomial

VBA & Python POO appliqué à la finance

Guide d'utilisation et Description

Par Seguin Milo,
Othmane Ayman
M2 272 parcours Quant

Prof: Emmanuel Fruchard

1. Classe Tree

Cette classe est responsable de la construction de l'arbre. Elle contient des méthodes pour construire l'arbre en créant les nœuds et effectuant le Pruning sur les nœuds avec une probabilité négligeable.

Pruning(node, step): Nous avons décidé de ne pas construire les nœuds avec une proba inférieure à 10^{-7}

Build_Tree(): Méthode principale pour construire l'arbre. Nous construisons à chaque step le nœud central ainsi que ses 3 fils puis nous construisons l'arbre vers le haut dans une première boucle while, puis vers le bas.

2. Classe Node

Représente un nœud dans l'arbre. Chaque nœud a pour attribut, à instanciation, sur son prix spot, ses probabilités et ses connexions avec les nœuds voisins.

Dividende(step): Vérifie la présence d'un dividende à l'étape actuelle.

forward_adjust(step): Ajuste le prix forward en cas de dividende.

connect_node(): Connecte les nœuds de différentes manières (haut, milieu, bas, voisin haut, voisin bas). Les arguments sont optionnels afin d'avoir une plus grande flexibilité.

node_proba(step): Calcule les probabilités pour les nœuds suivant en partant du nœud actuel. Les probas sont calculés en prenant en compte l'effet d'un dividende et la présence de pruning, c'est-à-dire que si un nœud ne construit pas son next_up ou next_down si les probas de ces derniers sont négligeables

proba_next_node(step): Calcule les probabilités cumulatives pour les nœuds suivants.

compute_pricing_node(opt): Calcule le prix de l'option pour le nœud actuel. Pricing récrusif

compute_price_exo(opt, price_euro): Calcule le prix des options exotiques (américaines, down-and-out, up-and-out).

3. Classe Option

type(): Vérifie si l'option est un call ou un put.

payoff(spot): Calcule le payoff de l'option en fonction du prix spot.

verif_nat(): Vérifie la validité de la nature de l'option.

verif_barrier(tree): Vérifie la validité de la barrière de l'option.

4. Classes erreurs

Contient des exceptions personnalisées (OptionNatureInvalide, BarrierInvalide) pour gérer les erreurs liées à la nature de l'option et à la barrière.

1. plot_tree(self)

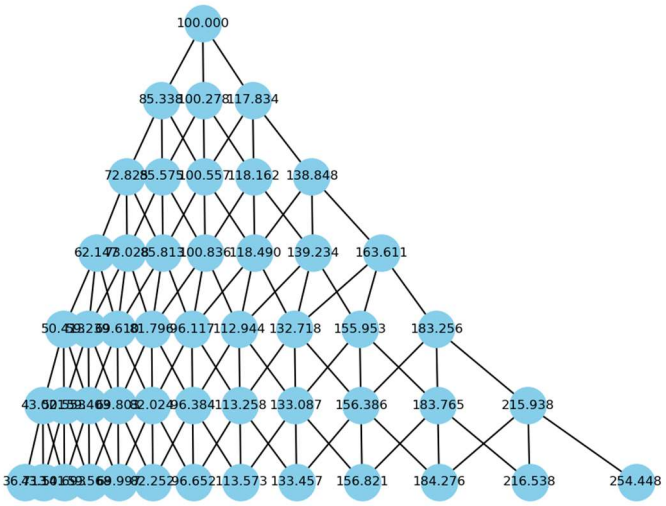
Cette méthode visualise l'arbre trinomial en utilisant la bibliothèque NetworkX, affichant les prix spot à chaque nœud. Il y a la possibilité d'afficher également les probas ou les prix des options

Nous avons ensuite utilisé XlWings et son addin avec @xw.func pour lier le code python sur excel. Il faut dans le ruban parametrer l'interpreter le python path et l'UDF Module. Notre fichier excel comporte une interface pour VBA, une interface pour le code python, une worksheet pour afficher l'arbre avec vba, et une worksheet caché pour les menus déroulants. **Nous vous avons également mis dans le dossier le code python en un seul script si jamais vous avez des problèmes avec Xlwings et le refresh des formules liées au code python.** Ceci fichier sert de Backup, il se nomme Trinomiale.py

Dans le code python contenant l'intégralité du code en une seule page il suffit de run la Main et d'enlever les dièses pour afficher les fonctions que vous voulez (Grecques, temps de pricing, convergence, sensibilité, nombre de nœuds et Prix B&S)

Note pour les options KO : Pour Pricer des options KO (UO ou DO) vous devez renseigner la barrière comme tel : 1.1 pour 110%, 0.8 pour 80% etc.. et il faut faire attention à la validité de la barrière, c'est-à-dire à son positionnement par rapport au strike et/ou au spot, cela expliqué dans la section options KO.

Affichage de l'arbre avec tombée de dividendes sur Python (fonction disponible dans le code python et VBA)



Temps de Pricing et résultats avec Python

Python		
Steps	Temps de Run Classique	Temps de Run avec les Grecques (Delta Gamma, Vega, Theta et Rho)

100	0,1350s	0,94s
200	0,3273s	2,5s
500	1,1600s	9,17s
1000	3,0800s	30,4s

Temps de Pricing 10 000 Steps en Python pour :

Call Eu
K= 101
Vol= 25%
Div= 0
Rate= 2%
Maturité : 01/07/2024
Pricing Date= 01/09/2023
Div Ex Date = 01/03/2024

Temps d'execution 102s soit 1,7 min
Prix = 9.390129151697236

Temps de Pricing 10 000 Steps en Python

Call Eu
K= 101
Vol= 25%
Div=3
Rate= 2%
Maturité : 01/07/2024
Pricing Date= 01/09/2023
Div Ex Date = 01/03/2024

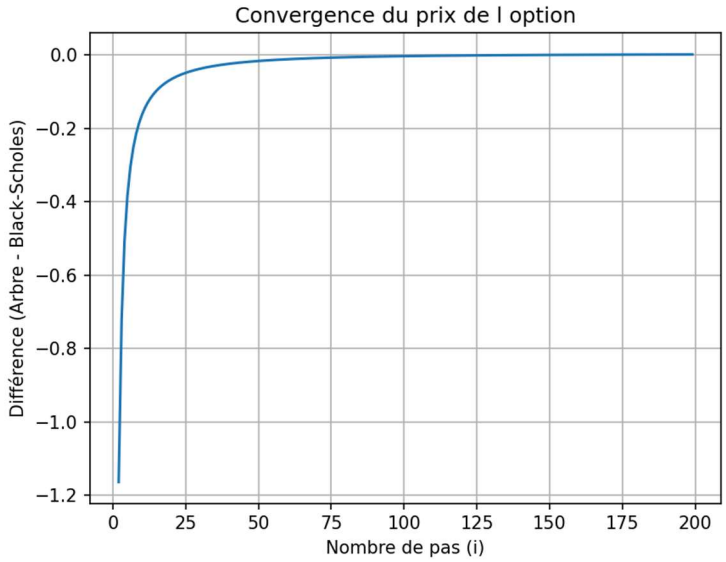
105s soit 1,7 min,
Prix = 7.975170267054077.
Pour 200 steps : 7.973917429539263

Grecques pour une option avec les mêmes paramètres mais avec 0 div et avec 1000 steps :

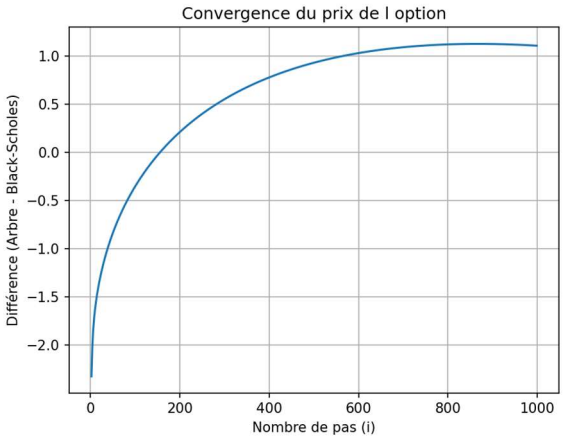
Delta	0.5571737666297141
Gamma	0.016061070689140644
Vega	0.3603247527306319
Theta	-0.01736612066135379
Rho	0.3889386424091512

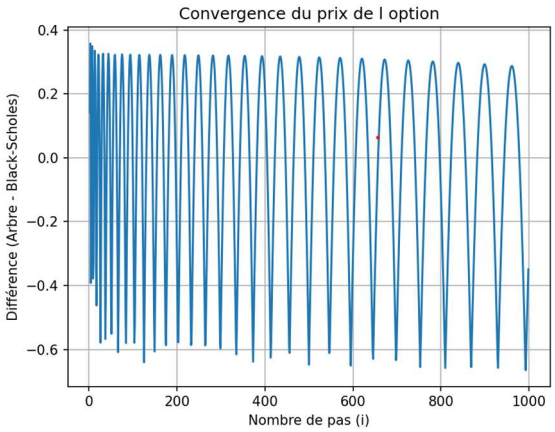
<u>Convergence du Prix vers B&S :</u>

<u>Convergence du prix vers B&S sans multiplication par le nombre de steps</u>

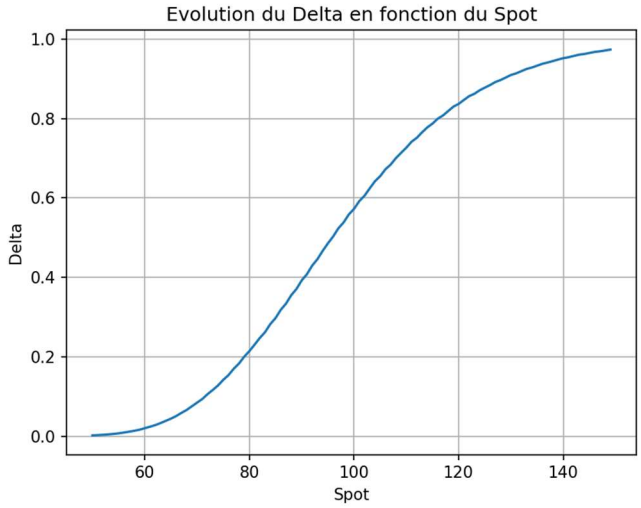


Convergence du prix vers B&S avec multiplication par le nombre de steps (K=101 à gauche, K=150 toutes choses égales par ailleurs, à droite

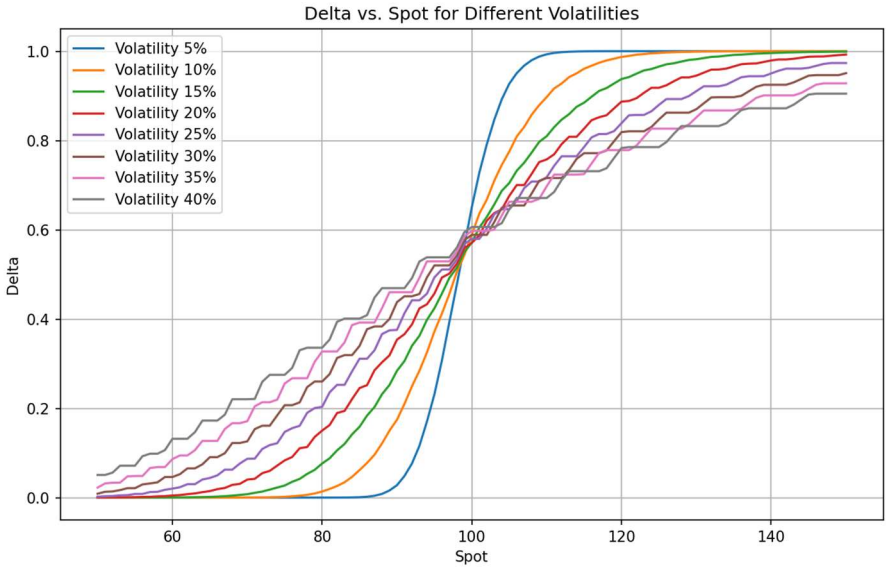




Evolution du Delta en fonction du Spot pour un Call à la monnaie avec 300 steps et une Vol de 25%



Evolution du Delta en fonction du Spot pour différents niveaux de vol pour un Call Européen avec 100 steps.



Nous voyons ci-dessus que plus la volatilité est importante et plus le delta des options OTM est important contrairement aux options ITM. En effet, une volatilité plus importante augmente la probabilité d'une option OTM de devenir ITM et augmente la probabilité d'une option ITM de devenir OTM. Pour une volatilité très faible, le delta varie énormément pour les options ayant un spot autour de la monnaie. Nous voyons également que plus la volatilité est élevée, plus le delta est instable.

Pricing Option Down and Out, méthode: Compute_price_exo

Nous avons ajouté la possibilité de faire intervenir une barrière dans les caractéristiques de l'option pour pricer des options KO, nous reviendrons sur les options KI par la suite. Pour les Options KO, cela signifie que lorsque la barrière est franchie par le sous-jacent, l'option expire et ne vaut plus que 0. L'augmentation de la volatilité pour ce genre d'option fais baisser son prix car la proba d'expirer à 0 est bien plus importante.

Nous avons eu 2 approches :

1) La première a été de nous dire que dès la création de l'arbre, lorsque le spot d'un nœud a franchi la barrière, alors ses probabilités d'aller au milieu, en haut ou en bas sont égales à 0, et le prix de ce nœud vaut 0. Comme ses probabilités pour les nœuds suivant valent 0, alors cela annule la possibilité qu'un nœud franchisse la barrière puis la refranchisse et expire avec une valeur positive ce qui serait faux car lorsque la barrière est atteinte, l'option expire et vaut 0.

2) La seconde approche consiste à ne pas intervenir sur les probas mais uniquement sur les prix lors du pricing récursif et de déclarer que lorsque que le Spot du nœud est supérieur à la barrière alors son prix vaut 0, ce qui revient sensiblement au même résultat que la première approche.

```
def compute_price_exo(self,opt,price_euro):
    opt.verif_barrier(self.tree)
    if opt.Nat=="UO": #les noeuds supérieurs à la barrière valent 0, explication dans le word
        if self.spot>=opt.Barrier*self.tree.root.spot:
            self.price=0
    if opt.Nat=="Am":
        self.price=max(price_euro,opt.payoff(self.spot))
    if opt.Nat=="DO":
        if self.spot<=opt.Barrier*self.tree.root.spot:
            self.price=0
```

La variable price_euro fait évidemment référence au prix européen du nœud.

Test de réalisme de la barrière :

```
2 usages (2 dynamic)
def verif_barrier(self,tree):
    if self.Nat=="UO":
        if self.Barrier * tree.root.spot < tree.root.spot or self.Barrier*tree.root.spot<self.strike:
            raise BarrierInvalide("La barrier ne peut pas être inférieur au spot ou la barrière ne peut pas être inf au strike")
    elif self.Nat=="DO":
        if self.Barrier * tree.root.spot > tree.root.spot or self.Barrier*tree.root.spot>self.strike:
            raise BarrierInvalide("La barrier ne peut pas être supérieur au spot ou la barriere ne peut pas être sup au strike")
```

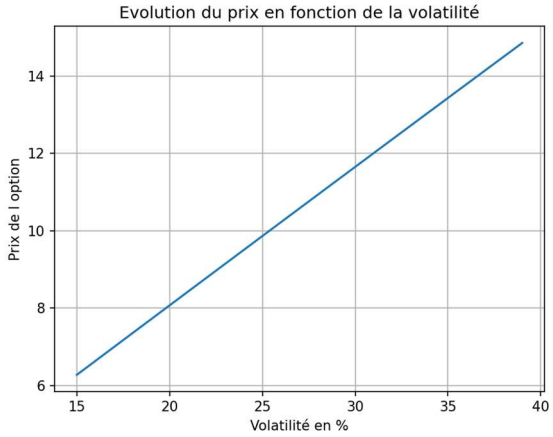
En effet, la barrière ne peut pas être égale à n'importe quelle valeur car cela pourrait ne pas avoir de sens :

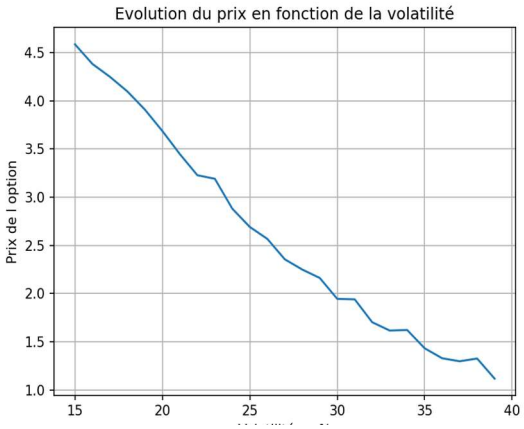
Call et Put Up and Out : La barrière ne peut pas être inférieur au Spot car cela signifierait que l'option serait désactivée instantanément. La barrière ne peut également pas être inférieur au Strike, sinon l'option vaut 0 en toutes circonstances.

Call et Put Down and Out: Logique inverse.

Evolution du Prix d'un Call EU à la monnaie avec 300 steps en fonction de la vol avec une variation de 1% de cette dernière, toutes choses égales par ailleurs. *(Graphique de gauche)***. Evolution du même Call mais avec une barrière désactivante à 130%** *(Graphique de droite)***.**

Nous voyons qu'effectivement pour les Options KO, l'augmentation de la vol toutes choses égales par ailleurs a pour effet de diminuer le prix de l'option.





<u>Pricing Option KI</u>

Nous avons passé beaucoup de temps à essayer de calculer un prix pour les options KI, cela est d'une difficulté bien plus importante et nous avons dû abandonner. En effet, pour une option Down and In ou Up and In, le prix de l'option devient le prix d'une option européenne avec les paramètres de départ lorsque la barrière est atteinte au moins une fois dans la vie de l'option. C'est à dire que l'option « vient à la vie » que quand le sous-jacent a franchi la barrière mais cela ne signifie pas que l'option n'a pas de valeur avant !

Par exemple pour un Put Down and In (PDI) qui est une des options exotiques les plus utilisées notamment dans les produits structurés. Un PDI avec un K=100, Spot=100 et barrière à 40% (soit 40), l'option a une valeur européenne à partir du moment où le spot atteint 40, l'option obtient la valeur d'un Put très à la monnaie puisque le Strike est égal à 100.

Cependant, lorsque le cours du sous-jacent n'a pas encore atteint la barrière, l'option a bien un prix positif qui est égale à la probabilité de franchir la barrière un jour puis au pricing européen classique à partir du passage de cette barrière.

<u>Nous avons eu l'approche suivante :</u>

Nous avons décidé de donner des prix à des nœuds dès l'initialisation de l'arbre. En effet, nous avons segmenter les nœuds en 3 catégories :

1) Les nœuds qui lorsqu'ils sont atteints, ne pourront jamais atteindre la barrière et ne pourront jamais être atteint par des nœuds ayant atteint la barrière donc valent 0.
2) Les nœuds ayant atteint la barrière qui ont un prix européen
3) Les nœuds pouvant atteindre la barrière un jour ou pouvant être atteint par des nœuds ayant atteint la barrière.

Nous avons dû pour commencer rendre notre objet arbre itérable.

```python
def __iter__(self):  # méthode qui permet de rendre l'objet arbre itérable
    visited = set()  # Créer un ensemble pour suivre les nœuds visités
    return self._traverse(self.root, visited)

4 usages
def _traverse(self, node, visited):
    if node is not None:
        if node not in visited:  # Vérifier si le nœud a déjà été visité
            visited.add(node)  # Ajouter le nœud à l'ensemble des nœuds visités
            yield node  # générateur yield
            if node.next_up:
                yield from self._traverse(node.next_up, visited)
            if node.next_mid:
                yield from self._traverse(node.next_mid, visited)
            if node.next_down:
                yield from self._traverse(node.next_down, visited)
```

Puis nous avons construit notre fonction pour pricer un PDI, seulement le prix trouvé n'a pas fait sens et nous avons comparé avec des pricing de PDI en banque et cela ne matchait pas.

```python
def compute_price_pdi(self,Barrier):
    Barrier_value=Barrier*self.root.spot
    node_spot_list=[]
    for Node in self:
        node_spot_list.append(Node.spot)
    for Node in self:
        new_node=Node
        while new_node.next_down is not None:
            new_node=new_node.next_down
        if new_node.spot>Barrier_value:
            node_mixte_spot=Node.spot
            node_mixte_spot_2=Node.spot
            while node_mixte_spot and node_mixte_spot_2 in node_spot_list:
                node_mixte_spot= node_mixte_spot / np.exp(self.rate * self.timestep) / self.alpha
                node_mixte_spot_2=node_mixte_spot_2 / np.exp(self.rate * self.timestep)
                if node_mixte_spot or node_mixte_spot_2 not in node_spot_list:
                    node_mixte_spot=node_mixte_spot*self.alpha
                    if node_mixte_spot not in node_spot_list:
                        node_mixte_spot=np.exp(self.rate * self.timestep)
                    break
            if node_mixte_spot>Barrier_value:
                Node.price=0
```

Le but du code était de vérifier à chaque nœud s'il est possiblement atteignable par un nœud ayant atteint la barrière ou bien s'il pourra un jour atteindre la barrière activante.