Implémentation de l'arbre trinomial sur Hull-White en python

Ali Broma Sidibe [1]

Ayten Yagbassan [2]

Demo online : http ://www.fmgrlab.com/test/

1. bromasidibe@gmail.com
2. ayten.yagbasan92@gmail.com

0.1 Introduction

La connaissance de la forme future du taux relève d'une importance capitale en finance pour valoriser les actifs dérivées. Certains actifs financiers tels que les obligations incluent des options (tel qu'options de rachat, de prolongation, de rétraction ou même de conversion) qui les rendent plus attrayantes pour l'émetteur ou l'investisseur selon les cas. Ces options qui ne peuvent exister qu'a travers les obligations qui les portent sont difficile à évaluer au moment de l'émission puisqu'ils dépendent des taux d'intérêts futurs. Leur évaluations nécessite donc la connaissance de la forme du taux futur d'où la nécessité des modèles de taux.

Les modèles de taux sont utilises principalement pour évaluer les produits dérivées financiers. Elles sont multiples et variés, on peut tout de même catégoriser en deux catégories : modèles d'équilibre et modèles d'arbitrages. Les modèles d'équilibre ne cherchent pas à ajuster automatiquement à la structure de taux d'intérêt et requièrent de choisir judicieusement les paramètres afin de s'ajuster à toute sorte de structure par termes. Dans cette catégories de modèles de taux, nous pouvons citer le modèle de Vasicek. D'une part, nous avons les modèles d'équilibre qui en se basant sur la théorie d'arbitrage parvienne à prédire le taux futur tout en restant cohérent avec le taux actuel. Dans cette catégorie, on peut citer le modèle de Hull White, Ho et Lee ...etc. Dans ce présent document, notre travail consiste à implémenter le modèle de Hull-White en python à travers l'arbre trinomial. Pour mener à bien ce projet, nous allons procéder à la résolution analytique du modèle de Hull-White. Dans la deuxième partie, nous allons discrétiser le modèle de Hull-White par la méthode de différences finies afin de le simuler avec l'arbre trinomial sous python.

0.2 Résolution analytique

0n se place dans un espace de probabilité (Ω, F, P) muni d'une filtration $(F_t)_{0\leq t\leq T}$. On note Q la probabilité risque neutre.
Par définition sous le modèle Hull-White, le taux court instantané r_t est solution de l'équation différentielle stochastique suivante :

$$dr_t = (b(t) - ar_t)dt + \sigma dW_t \tag{1}$$

avec a et σ réels positifs
$b : [0, T] \to R$ une fonction déterministe du temps suffisamment régulière et à croissance linéaire et lipschitzienne.
W_t un mouvement brownien sous P.

Notre objectif est de déterminer une solution analytique mathématique à l'équation 18. D'abord l'existence d'une solution est assurée par le fait que σ est constante et $(b(t) - ar_t)$ est à croissance linéaire. L'unicité découle de l'aspect de leur caractère lipschitzien.
Pour trouver l'expression analytique du taux $r(t)$, on définit une fonction $f(t, r_t) = r_t e^{at}$. f est

continue et deux fois derivable, on peut appliquer le Lemme d'Ito.

$$\begin{aligned} df &= f_t dt + f_r dr_t + \frac{1}{2} f_{rr} < dr, dr >_t \\ &= af(r,t)dt + e^{at}[(b - ar)dt + \sigma dW_t] \\ &= e^{at}[bdt + \sigma dW_t] \end{aligned} \tag{2}$$

En intégrant l'expression entre l'intervalle [s,t], on trouve :

$$\begin{aligned} \int_s^t df &= \int_s^t e^{au} b(u)du + \int_s^t \sigma e^{au} dW_u \\ f(t) - f(s) &= \int_s^t be^{au} du + \int_s^t \sigma e^{au} dW_u \\ r(t)e^{at} - r(s)e^{as} &= \int_s^t b(u)e^{au} du + \int_s^t \sigma e^{au} dW_u \\ r(t) = r(s)e^{-a(t-s)} &+ \int_s^t b(u)e^{-a(t-u)} du + \int_s^t \sigma e^{-a(t-u)} dW_u \end{aligned} \tag{3}$$

0.3 Distribution

Par la solution analytique trouvée dans la section précédente (3), on cherche à déterminer la distribution qui régit la dynamique du taux r(t) :

$$r(t) = r(s)e^{-a(t-s)} + \int_s^t b(u)e^{-a(t-u)} du + \int_s^t \sigma e^{-a(t-u)} dW_u \tag{4}$$

On constate que r_t est une intégrale stochastique (de Wiener), une combinaison linéaire de variables loi normale indépendantes donc par conséquent r_t suit la loi normale.

Pour tout $s < t$, la variance du taux déterminé par l'équation précédente est définie par :

$$Var(r_t) = \frac{\sigma^2}{2a}(1 - e^{-2a(t-s)}) \tag{5}$$

Sachant que l'espérance d'un mouvement brownien est 0 et $b(t)$ est déterministe, alors l'espérance est :

Pour tout $s < t$,

$$E_s(r_t) = r(s)e^{-a(t-s)} + \int_s^t b(u)e^{-a(t-u)} du \tag{6}$$

0.4 Calibration

Il s'agit de trouver une relation entre $b(t)$ et $f(0,t)$ dans le modèle, et de déclarer que le modèle est calibré sur les données de marché si

$$f(0,T) = f^M(0,T)$$

pour toutes les maturités T. Le modèle de Hull-White reproduit exactement la courbe des taux zéro coupon de marché si l'on pose

$$b(t) = \frac{\partial f^M}{T}(0,T) + af^M(0,t) + \frac{\sigma^2}{2a}(1 - e^{-2at})$$

Démonstration Le prix d'une obligation zéro coupon est une fonction de temps et du taux suffisamment dérivable. On peut poser

$$P(t,T) = f(t,r_t)$$

En appliquant Ito à la fonction f on a :

$$\frac{\partial f}{\partial t} = rf + (b(t) - ar)\frac{\partial f}{\partial r} + \frac{1}{2}\frac{\partial^2 f}{\partial \sigma^2} \tag{7}$$

Les conditions terminales sont : $f(T,r) = P(T,T) = 1$

On peut chercher une solution de la forme $P(t,T) = A(t,T)e^{-B(t,T)r(t)}$ on obtient le système d'équation suivant :

$$\begin{cases} \dfrac{\partial A}{\partial t} - b(t)AB + \dfrac{1}{2}\sigma^2 AB^2 = 0, \\ \dfrac{\partial A}{\partial t} - aB + 1 = 0, \\ A(T,T) = 1, \\ B(T,T) = 0 \end{cases} \tag{8}$$

La seconde équation donne directement :

$$B(t,T) = \frac{1}{a}(1 - e^{-a(T-t)})$$

Et la première s'intègre par :

$$\ln A(t,T) = -\int_t^T b(u)B(u,T)du - \frac{\sigma^2}{2a^2}(B(t,T) - (T-t)) - \frac{\sigma^2}{4a}B^2(t,T)$$

En remplaçant A et B par leur expression dans l'équation de P(t,T), on a donc l'expression des prix de zéro-coupon dans le modèle de Hull & White, et on en déduit une expression des taux forward à la date 0 :

$$\begin{aligned} f(0,T) &= -\frac{\partial}{\partial T}\ln P(0,T) \\ &= \int_0^T b(u)\frac{\partial B}{\partial T}(u,T)du + \frac{\partial B}{\partial T}(0,T)r_0 - \frac{\sigma^2}{2a}B(0,T)\left(1 - \frac{\partial B}{\partial T}(0,T)\right) \end{aligned} \tag{9}$$

En dérivant encore une fois par rapport à T, on obtient :

$$\begin{aligned} f(0,T) = \int_0^T b(u)\frac{\partial^2 B}{\partial T^2}(u,T)du + \frac{\partial^2 B}{\partial T^2}(0,T)r_0 - \\ \frac{\sigma^2}{2a}\left(\frac{\partial B}{\partial T}(0,T) - \left(\frac{\partial B}{\partial T^2}(0,T)\right)^2 B(0,T) - \frac{\partial^2 B}{\partial T^2}(0,T)\right) \end{aligned} \tag{10}$$

En combinant ces deux résultats, puis en remplaçant B par l'expression calculée par l'équation et en simplifiant, on obtient :

$$af(0,T) + \frac{\partial f}{\partial T}(0,T) = b(T) + \frac{\sigma^2}{2a}(1 - e^{-2aT}) \tag{11}$$

Finalement, on considère que le modèle est calibré sur les données du marché si pour toutes les maturités T, on a

$$f(0,T) = f^M(0,T)$$

.

0.5 Évaluation des obligations

Dans le modèle de Hull et White, le prix $P(t,T)$ du zéro-coupon de maturité T à la date t s'écrit :

$$P(t,T) = A(t,T)e^{B(t,T)r(t)} \tag{12}$$

avec,

$$\begin{cases} A(t,T) = \dfrac{P^M(0,T)}{P^M(0,t)} \exp\left[B(t,T)f^M(0,t) - \dfrac{\sigma^2}{4\alpha}(1 - e^{-2\alpha t})B(t,T)^2\right] \\ B(t,T) = \dfrac{1}{a}(1 - e^{-a(T-t)}) \end{cases} \tag{13}$$

Démonstration Nous venons de calculer directement $B(t,T)$ dans la partie calibration. Pour $A(t,T)$, on repart de l'équation (2.9) et on remplace b par son expression. Par le calcul direct de $\int_s^t b(u)du$ et on observant en particulier $\ln \frac{P^M(0,T)}{P^M(0,t)} = -\int_t^T f^M(0,u)du$, on obtient le résultat voulu.

0.6 Prix d'une option sur zéro-coupon

Dans la partie précédente, nous avons démontré que le taux court suit la gaussienne. Ce caractère gaussien du taux court r permet de calculer le prix des produits dérivés simples sur l'obligation zéro-coupon. Dans le cas du call européen, le prix est donné par la proposition suivante.

Soit une option de sous-jacent une obligation zéro coupon de maturité S, de prix d'exercice K, de maturité T (S>T) alors la valeur d'une telle option peut être déterminée par :

$$C = P(t,S)N(d1) - KP(t,N)N(d2) \tag{14}$$

avec

$$d1 = \frac{\ln\left(\frac{P(t,S)}{KP(t,T)}\right) + \frac{1}{2}\sigma_h^2(T-t)}{\sigma_p\sqrt{(T-t)}} \tag{15}$$

$$d2 = d1 - \sigma_h\sqrt{(T-t)} \tag{16}$$

$$\sigma_p = \frac{\sigma}{\sqrt{T-t}} B(T,S) \sqrt{\frac{1-e^-2a(T-t)}{2a}} \tag{17}$$

où N est la fonction de répartition de la loi normale centrée réduite.

Démonstration Par Feyman Kac sous la probabilité risque neutre, le prix d'un tel call s'écrit comme suit

$$C = E_Q \left[e^{-\int_t^T r_u du} (P(T,S) - K)^+ | F_t \right]$$

Avec Q probabilité risque neutre. Par changement de numéraire, on définit la probabilité Q^T dite "T-forward" associé au numéraire P(.,T) de derivée de Radon-Nikodym :

$$\frac{dQ^T}{dQ} = \frac{e^{-\int_t^T r_u du}}{P(t,T)}$$

Par ce changement de numéraire, il vient alors que sous la nouvelle mesure de probabilité

$$C = E_{Q^T} \left[\left(P(t,T) A(t,S) e^{-B(t,S) r_T} - K P(t,T) \right)^+ | F_t \right]$$

Il suffit de vérifier les hypothèses de Black Scholes pour déduire le résultat c'est à dire que le processus défini par $P(t,T)A(t,S)e^{-B(t,S)r_T}$ est log normal. Or, on a bien dit que r_t est gaussienne conditionnellement a sa filtration naturelle F_t sous Q^T de variance définie (en haut), alors le processus $P(t,T)A(t,S)e^{-B(t,S)r_T}$ est log normale conditionnellement à sa filtration naturelle F_t sous Q^T de variance définie. On applique la formule de Black Scholes pour déduire le résultat cherché.

0.7 Résolution numérique

0.7.1 Équation

Il s'agit ici de discrétiser à l'aide de la méthode de différence finie l'équation vérifiée par le taux

$$dr_t = (\theta(t) - a r_t) dt + \sigma dW_t \tag{18}$$

On admet que le sous-jacent θ suit la dynamique suivante :

$$d\theta = \mu(\theta, t)\theta dt + \sigma(\theta, t)\theta dW_t \tag{19}$$

Soit f le prix du sous-jacent, alors f est une fonction de θ et de t.

f est continue et suffisamment derivable pour appliquer le Lemme d'Ito.

En appliquant le Lemme d'Ito à f on a

$$\begin{aligned} df &= \frac{\partial f}{\partial t} dt + \frac{\partial f}{\partial \theta} d\theta + \frac{1}{2} \frac{\partial^2 f}{\partial \theta^2} < d\theta, d\theta >_t \\ df &= \frac{\partial f}{\partial t} dt + \frac{\partial f}{\partial \theta} (\mu dt + \sigma dw)\theta + \frac{1}{2} \frac{\partial f}{\partial \theta^2} \sigma^2 \theta^2 dt \\ df &= (\frac{\partial f}{\partial t} + \frac{\partial f}{\partial \theta} \mu + \frac{1}{2} \frac{\partial f}{\partial \theta^2} \sigma^2 \theta^2) dt + \frac{\partial f}{\partial \theta} \sigma dw \end{aligned} \tag{20}$$

On construit une portefeuille constitué d'une fraction du sous-jacent :

$$\pi = f + \delta\theta \tag{21}$$

En veillant à choisir la fraction du sous-jacent $\delta = -\frac{\partial f}{\partial \theta}$, le porte-feuille π devient sans risque, il en résulte que sa variation suit la dynamique suivante ;

$$\begin{aligned} d\pi &= df - \frac{\partial f}{\partial \theta} d\theta \\ &= (\frac{\partial f}{\partial t} + \frac{\partial f}{\partial \theta}\mu + \frac{1}{2}\frac{\partial f}{\partial \theta^2}\sigma^2\theta^2)dt + \frac{\partial f}{\partial \theta}\sigma dw - \frac{\partial f}{\partial \theta}(\mu dt + \sigma dw)\theta \\ &= (\frac{\partial f}{\partial t} + \frac{1}{2}\frac{\partial f}{\partial \theta^2}\sigma^2\theta^2)dt \end{aligned} \tag{22}$$

Sous la probabilité risque neutre, tout actif non risqué actualise avec le taux sans risque ;

$$d\pi = r\pi dt \tag{23}$$

En combinant 21, 22 et 23 nous retrouvons la dynamique suivant de f

$$\frac{\partial f}{\partial t} + \frac{\partial f}{\partial \theta}(\mu - \lambda\sigma)\theta + \frac{1}{2}\theta^2\sigma^2\frac{\partial f}{\partial \theta^2} - rf = 0 \tag{24}$$

0.7.2 Discrétisation

Il s'agit maintenant de discrétiser à l'aide de la méthode des différences l'égalité 24. Pour cela nous procédons comme suit ;

1. Le pas de temps Δt, tel que le temps continu sera discrétisé comme :

$$t = [t_0, t_0 + \Delta t, \ldots, t_0 + i\Delta t]$$

2. Les possibles valeurs de θ sont discretisées avec un intervalle $\Delta\theta$ tel que :

$$\theta = [\theta_0, \theta_0 + \Delta\theta, \ldots, \theta_0 + j\Delta\theta]$$

;

On passe du temps continu au temps discret comme :

$$\begin{aligned} \frac{\partial f}{\partial t} &\to \frac{f_{i,j} - f_{i-1,j}}{\Delta t} \\ \frac{\partial f}{\partial \theta} &\to \frac{f_{i,j+1} - f_{i,j-1}}{2\Delta\theta} \\ \frac{\partial^2 f}{\partial \theta^2} &\to \frac{(f_{i,j+1} - f_{i,j}) - (f_{i,j} - f_{i,j-1})}{\Delta\theta^2} \\ \frac{\partial^2 f}{\partial \theta^2} &\to \frac{f_{i,j+1} + f_{i,j-1} - 2f_{i,j}}{\Delta\theta^2} \end{aligned} \tag{25}$$

En remplaçant les dérivées par leurs expressions obtenue par la différence finie dans l'équation 24 on passe à :

$$\frac{f_{i,j} - f_{i-1,j}}{\Delta t} + (f_{i,j+1} - f_{i,j-1})\frac{(\mu - \lambda\sigma)\theta_j}{2\Delta\theta} + \frac{1}{2}\theta_j^2\sigma^2\frac{f_{i,j+1} + f_{i,j-1} - 2f_{i,j}}{\Delta\theta^2} - rf_{i-1,j} = 0$$

$$f_{i,j} + (f_{i,j+1} - f_{i,j-1})\frac{(\mu - \lambda\sigma)\theta_j \Delta t}{2\Delta\theta} + \frac{1}{2}\frac{\theta_j^2 \sigma^2 \Delta t}{\Delta\theta^2}(f_{i,j+1} + f_{i,j-1} - 2f_{i,j}) = f_{i-1.j} + r f_{i-1,j}\Delta t$$

$$f_{i-1.j}(1 + r\Delta t) = f_{i,j-1}\left[\frac{1}{2}\frac{\theta_j^2 \sigma^2 \Delta t}{\Delta\theta^2} - \frac{(\mu - \lambda\sigma)\theta_j \Delta t}{2\Delta\theta}\right] + f_{i,j}\left[1 - \frac{\theta_j^2 \sigma^2 \Delta t}{\Delta\theta^2}\right]$$

$$+ f_{i,j+1}\left[\frac{(\mu - \lambda\sigma)\theta_j \Delta t}{2\Delta\theta} + \frac{1}{2}\frac{\theta_j^2 \sigma^2 \Delta t}{\Delta\theta^2}\right]$$

On pose

$$\begin{aligned} a_{j-1} &= \frac{1}{2}\frac{\theta_j^2 \sigma^2 \Delta t}{\Delta\theta^2} - \frac{(\mu - \lambda\sigma)\theta_j \Delta t}{2\Delta\theta} \\ a_j &= 1 - \frac{\theta_j^2 \sigma^2 \Delta t}{\Delta\theta^2} \\ a_{j+1} &= \frac{(\mu - \lambda\sigma)\theta_j \Delta t}{2\Delta\theta} + \frac{1}{2}\frac{\theta_j^2 \sigma^2 \Delta t}{\Delta\theta^2} \end{aligned} \tag{26}$$

On obtient

$$f_{i-1.j}(1 + r\Delta t) = a_{j-1} f_{i,j-1} + a_j f_{i,j} + a_{j+1} f_{i,j+1} \tag{27}$$

On vérifie aisément que

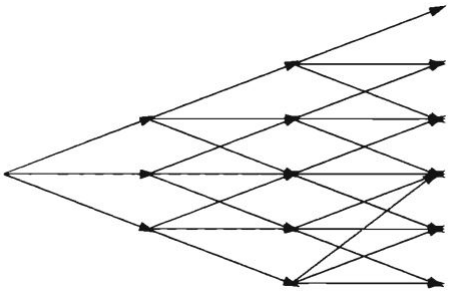
— Les valeurs a_{j-1}, a_j et a_{j+1} sont tous positifs.

— Leur somme est égale à 1 $\sum_{p=j-1}^{j+1} a_p = 1$

Alors a_p peut-être considéré alors comme un coefficient de probabilité de chaque état de transition de f.

0.8 Arbre trinomial

Lorsque nous abordons le pricing des options dans les classes inférieures, le premier outils que nous utilisons est l'arbre binomial. L'arbre binomial sous-entend que le prix d'un actif ne peut qu'augmenter ou baisser avec une certaines probabilité. Il n'accorde aucune chance aux pris de rester identique pendant un certains temps. Développé par Phelim Boyle en 1986, l'arbre trinomial est aperçu comme une extension de l'arbre binomial en ajoutant un état de transition.



0.8.1 Branchement

L'arbre trinomial possède trois sortes de branchement. En fixant un intervalle de temps égal à Δt sur la maturité et un pas $j\Delta r$ de tel sorte que chaque noeud (i,j) correspond au pas temps $t = i\Delta t$ et valeur $r_{0,0} + j\Delta r$. Un point (i, j) est alors positionné dans le graphe par

$$n(i, j) = [i\Delta t, r_{0,0} + j\Delta r] \tag{28}$$

Les différentes valeurs possibles de j sont $j - 1$, j, $j + 1$.

— Lorsque j fait un seul saut c'est à dire de quitter de $j - 1$ vers j ou de quitter j à j+1, nous obtenons un branchement standard qui est la forme A dans le graphe suivant. Le branchement du type A est du type standard dans lequel on peut augmenter la valeur actuelle de r par Δr, ou rester égal ou bien descendre par Δr.

— Lorsque j fait un saut positif de deux pas pendant la même période, c'est à dire quitter de $j - 1$ vers $j + 1$ nous obtenons le branchement B, soit un saut de $2\Delta r$.

— Le branchement du type C est exactement l'opposé du type B c'est à dire quitter $j + 1$ vers $j - 1$ soit un saut de $-2\Delta r$

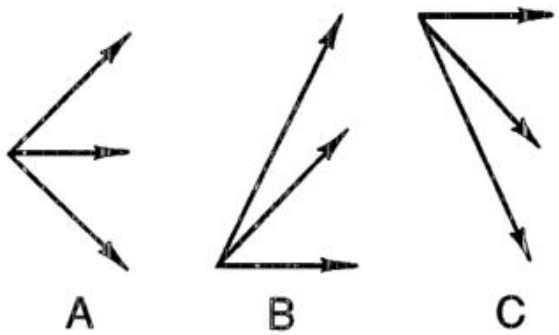
Le nœud central est arrangé de telle manière que la valeur moyenne correspond à la valeur attendue de r. Le choix d'une orientation est décidé selon la valeur du j. Il s'agirait alors de définir la valeur maximal que j peut prendre, notons le j_{max} Empiriquement, toujours par Hull-White, on a ;

$$\frac{0.184}{M} < j_{max} < \frac{0.186}{M}$$

De même on définit , j_{min} comme la valeur de j pour laquelle on change de la forme A pour la forme B, tel que

$$j_{min} = -j_{max}$$

$$M = a\Delta t \tag{29}$$



0.8.2 Stabilité de l'arbre

Le choix de de pas de temps est arbitraire. Il peut être chaque jour, six mois ou an. Cependant afin de garder une certaine stabilité, le choix du pas est empiriquement choisi de telle sorte que la grandeur $r(t+\Delta t) - r(t)$ suit une distribution normal. Par des simulations empirique, Hull White ont estimé qu'en choisissant le pas Δr comme suit, la convergence, stabilité, la minimisation de l'erreur est assurée.

$$\Delta r = \sigma\sqrt{3\Delta t}$$

0.8.3 Probabilités de transition

Nous avons démontré que a_{j-1}, a_j et a_{j-1} sont des coefficients de probabilité. On définit (i,j) comme le nœud pour lequel $t = i\Delta t$ et $r = j\Delta r$, et les probabilités de croissance, moyenne et baisse comme $p_u,\ p_m,\ et\ p_d$, respectivement.

Nous avons trois valeurs de probabilités ;

— P_u la probabilité que le taux monte.

— P_m la probabilité que le taux reste identique.

— P_d la probabilité que le taux baisse.

Pour les déterminer les probabilités de transitions dans l'équation 27, à travers le calcul de l'espérance, la variance et la définition de la probabilité, nous déduisons les valeurs suivantes selon les types de branchement

Forme A, on a :

$$\begin{aligned} P_u &= \frac{1}{6} + \frac{j^2M^2 + jM}{2} \\ P_m &= \frac{2}{3} - j^2M^2 \\ P_d &= \frac{1}{6} + \frac{j^2M^2 - jM}{2} \end{aligned}$$

Forme B :

$$\begin{aligned} P_u &= \frac{1}{6} + \frac{j^2M^2 + jM}{2} \\ P_m &= -\frac{1}{3} - j^2M^2 + 2jM \\ P_d &= \frac{7}{6} + \frac{j^2M^2 - 3jM}{2} \end{aligned}$$

Forme C :

$$\begin{aligned} P_u &= \frac{7}{6} + \frac{j^2M^2 + 3jM}{2} \\ P_m &= -\frac{1}{3} - j^2M^2 - 2jM \\ P_d &= \frac{1}{6} + \frac{j^2M^2 + jM}{2} \end{aligned}$$

0.9 Algorithme et implémentation en python

0.9.1 Le Graphe

Un Graphe est un ensemble de noeud lié entre eux par une relation père-fils. Nous le décomposerons en deux objets

— Un GraphNode qui représente un point du graphe (i,j).

— Un GraphStep qui représente l'ensemble des possibilités de taux à une période donnée.

shadecolorLightGray

```
# GraphStep est constitué de l'ensemble des noeud pour une période donnée
class GraphStep:
    def __init__(self, i):
        self.i = i
        self.nodes = []
```

Un Noeud du graphe est un point (i,j) du graphe, il est possède les caractéristiques suivantes ;

— La ième période (i)

— Le jème pas (j)

— Le taux appliqué à ce noeud (rate)

— Les probabilités de monter p_u, de descente p_d ou de maintien p_m

— Les prochains noeud fils en haut (j_up), en bas (j_d) et de même niveau j_d

— L'actif d'Arrow à ce noeud.

En python on peut décrire cela sous forme de code comme ce qui suit shadecolorLightGray

```
class GraphNode:
    def __init__(self, i=0, j=0, pu="", pm="", pd="", rate=0, j_up=0, j_m=0, j_d=0):
        self.id = get_id(i, j)
        self.i = i
        self.j = j
        self.rate = rate
        self.j_up = j_up
        self.j_m = j_m
        self.j_d = j_d
        self.pu = pu
        self.pd = pd
        self.pm = pd
        if i == 0 and self.j == 0:
            self.q = 1
        else:
            self.q = 0
```

0.9.2 Calcul des probabilité de transition

La relation qui lie chaque noeud à un autre noeud dépend de sa position dans le graphe. Connaissant tous les noeuds, nous allons déterminer les probabilités de transition d'un noeud à un autre noeud et créer au passage le lien qui existe entre les noeuds. shadecolorLightGray

```
def compute_transition_probability(self, N, jmax, jmin, M, hw_steps):
        for i in range(0, N, 1):
            top_node = min(i, jmax)
            for j in range(-top_node, top_node + 1, 1):
                node = hw_steps[i].nodes[j + top_node]
                if j == jmax:
                    # Branching C
```

```
                    node.pu = 7.0 / 6.0 + (j * j * M * M + 3 * j * M) / 2
                    node.pm = -1.0 / 3.0 - j * j * M * M - 2 * j * M
                    node.pd = 1.0 / 6.0 + (j * j * M * M + j * M) / 2


                if j == jmin:
                    # Branching B
                    node.pu = 1.0 / 6.0 + (j * j * M * M - j * M) / 2
                    node.pm = -1.0 / 3.0 - j * j * M * M + 2 * j * M
                    node.pd = 7.0 / 6.0 + (j * j * M * M - j * M) / 2


                if (j != jmin) and (j != jmax):
                    # Branching A
                    node.pu = 1.0 / 6.0 + (j * j * M * M + j * M) / 2
                    node.pm = 2.0 / 3.0 - (j * j * M * M)
                    node.pd = 1.0 / 6.0 + (j * j * M * M - j * M) / 2

        return 0
```

0.9.3 Détermination du prochain noeud connectés

La relation qui lie chaque noeud à un autre noeud dépend de sa position dans le graphe. Connaissant tous les noeuds, nous allons déterminer les prochains noeuds connecté. Il s'agit de vérifier la position courante. Si j a atteint la j_{max}, alors on force le prochain noeud à avoir une orientation B. Si j a atteint le j_{min}, alors on force le prochain noeud à avoir un branchement de type C.

shadecolorLightGray

```
def find_next_node(self, N, jmax, jmin, M, hw_steps):
        for i in range(0, N, 1):
            top_node = min(i, jmax)
            for j in range(-top_node, top_node + 1, 1):
                node = hw_steps[i].nodes[j + top_node]
                if j == jmax:
                    # Branching C

                    node.j_up = j
                    node.j_m = j - 1
                    node.j_d = j - 2

                if j == jmin:
                    # Branching B

                    node.j_up = j + 2
                    node.j_m = j + 1
                    node.j_d = j

                if (j != jmin) and (j != jmax):
                    # Branching A
                    node.j_up = j + 1
                    node.j_m = j
                    node.j_d = j - 1
        return 0
```

0.9.4 Détermination des noeuds connectés

Il s'agit de trouver pour un noeud quel sont ses fils ?. Cela semble facile à priori mais tel n'est pas le cas. Les fil d'un noeud (i, j) ne sont pas obligatoirement $(i+1, j-1)$, $(i+1, j)$,$(i+1, j)$. On sait par construction que les noeud-fils sont au nombre de 3 mais qui sont-ils ?. Pour le déterminer, nous faisons la marche inverse, nous parcourons tout les noeuds de la période (i+1) et vérifions si leur parent est notre noeud (i,j). shadecolorLightGray

```
def find_connected_node(self, current_node, node_of_previous_step):
        connected_nodes = []
        for node in node_of_previous_step:
            up_id = get_id(node.i + 1, node.j_up)
            m_id = get_id(node.i + 1, node.j_m)
            d_id = get_id(node.i + 1, node.j_d)
            if up_id == current_node.id or m_id ==
                    current_node.id or d_id == current_node.id:
                connected_nodes.append(node)
        return connected_nodes
```

0.9.5 Calcul d'Actif d'Arrow

Étant donnée que nous sommes à un noeud et connaissant les probabilités de transition et le chemin qui quitte de l'origine à ce noeud, nous pouvons calculer l'actif d'Arrow à ce noeud et par ricochet le taux appliqué à ce noeud.

shadecolorLightGray

```
def compute_arrow(self, current_node, connected_nodes, previous_a, dt, dr):
        for node in connected_nodes:
            if node.j_up == current_node.j:
                current_node.q += node.q * node.pu * math.exp(-(previous_a +
                node.j_up * dr) * dt)

            if node.j_m == current_node.j:
                current_node.q += node.q * node.pm * math.exp(-(previous_a +
                node.j_m * dr) * dt)

            if node.j_d == current_node.j:
                current_node.q += node.q * node.pd * math.exp(-(previous_a +
                node.j_d * dr) * dt)
```

0.10 Résultat

En supposant que le taux forward suit l'équation suivante [3]

$$\begin{aligned} \alpha &= 0.01 \\ \sigma &= 0.01 \\ T &= 18ans \\ \Delta &= 1ans \\ R(t_i) &= 0.08 - 0.05e^{-0.18t_i} \end{aligned} \tag{30}$$

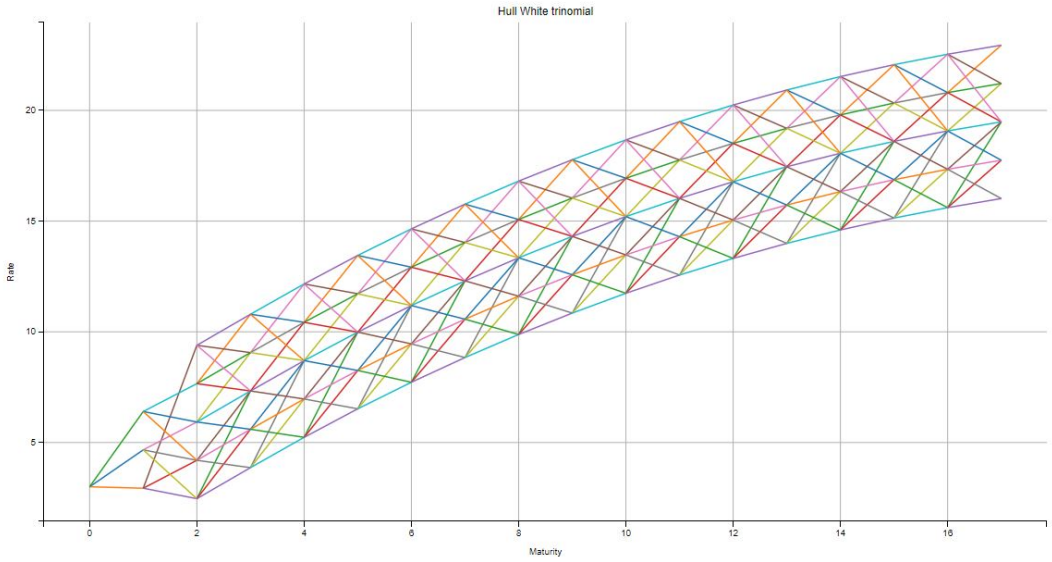


FIGURE 1 – Arbre trinomial appliqué à Hull-White

3. In this test, we had generated the interest rate by : 0.08 - 0.05Exp(-0.18ti). According the publication of Hull and White (1994), this corresponds approximately to the U.S. term structure at the beginning of 1994

Annexe A : Le code complet

shadecolorLightGray

```
import math
import matplotlib.pyplot as plt

class GraphStep:
    def __init__(self, i):
        self.i = i
        self.nodes = []
        self.a = 0

class GraphNode:
    def __init__(self, i=0, j=0, pu="", pm="", pd="", rate=0, j_up=0, j_m=0, j_d=0):
        self.id = str(i) + "," + str(j)
        self.i = i
        self.j = j
        self.rate = rate
        self.j_up = j_up
        self.j_m = j_m
        self.j_d = j_d
        self.pu = pu
        self.pd = pd
        self.pm = pd
        if i == 0 and self.j == 0:
            self.q = 1
        else:
            self.q = 0



class HWCalculator:
    def __init__(self):
        self.steps = []
        self.jmax = 0
        self.maturity = 0
        self.period = 'year'
        self.nbr_steps = 1
        self.volatility = 0.0
        self.alpha = 0.0
        self.rates = []

    def execute(self):
        if self.nbr_steps == 0:
            self.nbr_steps = 1.0
        dt = float(self.maturity) / float(self.nbr_steps)
        return self.process(self.volatility, self.alpha, dt, self.nbr_steps, self.rates)

    def process(self, sig, alpha, dt, N, rates):
        # Init parameter

        M = -alpha * dt
        dr = sig * math.sqrt(3 * dt)
        jmax = int(math.ceil(-0.1835 / M))
        jmin = -jmax
        self.jmax = jmax

        # Create empty graph
```

```
        self.steps = self.create_graph(N, jmax)

        # Calculate probability

        self.init_standard_move(N, jmax, self.steps, dr)

        self.compute_transition_probability(N, jmax, jmin, M, self.steps)
        self.find_next_node(N, jmax, jmin, M, self.steps)

        P = []
        P.append(1)
        for i in range(1, len(rates) + 1):
            P.append(math.exp(-rates[i - 1] * i * dt))

        self.steps[0].a = -math.log(P[1]) / dt
        for node in self.steps[0].nodes:
            node.rate += self.steps[0].a

        for i in range(1, N, 1):
            top_node = min(i, jmax)
            current_step = self.steps[i]
            for j in range(-top_node, top_node + 1, 1):
                current_node = current_step.nodes[j + top_node]
                connected_nodes = self.find_connected_node(current_node, self.steps[i - 1].nodes)
                self.compute_arrow(current_node, connected_nodes, self.steps[i - 1].a, dt, dr)
            sum = 0
            for j in range(-top_node, top_node + 1, 1):
                node = self.steps[i].nodes[j + top_node]
                sum += node.q * math.exp(-j * dt * dr)

            current_step.a = (math.log(sum) - math.log(P[i + 1])) / dt

            for j in range(-top_node, top_node + 1, 1):
                current_step.nodes[j + top_node].rate += current_step.a

    def create_graph(self, N, jmax):
        hwsteps = []
        for i in range(0, N, 1):
            top_node = min(i, jmax)
            current_step = GraphStep(i)
            for j in range(-top_node, top_node + 1, 1):
                node = GraphNode(i, j)
                current_step.nodes.insert(j + top_node, node)
            hwsteps.insert(i, current_step)
        return hwsteps

    def init_standard_move(self, N, jmax, hw_steps, dr):
        for i in range(0, N, 1):
            top_node = min(i, jmax)
            for j in range(-top_node, top_node + 1, 1):
                hw_steps[i].nodes[j + top_node].rate = j * dr
        return 0

    def compute_transition_probability(self, N, jmax, jmin, M, hw_steps):
        for i in range(0, N, 1):
            top_node = min(i, jmax)
            for j in range(-top_node, top_node + 1, 1):
                node = hw_steps[i].nodes[j + top_node]
                if j == jmax:
                    # Branching C
                    node.pu = 7.0 / 6.0 + (j * j * M * M + 3 * j * M) / 2
```

```
                    node.pm = -1.0 / 3.0 - j * j * M * M - 2 * j * M
                    node.pd = 1.0 / 6.0 + (j * j * M * M + j * M) / 2


                if j == jmin:
                    # Branching B
                    node.pu = 1.0 / 6.0 + (j * j * M * M - j * M) / 2
                    node.pm = -1.0 / 3.0 - j * j * M * M + 2 * j * M
                    node.pd = 7.0 / 6.0 + (j * j * M * M - j * M) / 2


                if (j != jmin) and (j != jmax):
                    # Branching A
                    node.pu = 1.0 / 6.0 + (j * j * M * M + j * M) / 2
                    node.pm = 2.0 / 3.0 - (j * j * M * M)
                    node.pd = 1.0 / 6.0 + (j * j * M * M - j * M) / 2

        return 0

    def find_next_node(self, N, jmax, jmin, M, hw_steps):
        for i in range(0, N, 1):
            top_node = min(i, jmax)
            for j in range(-top_node, top_node + 1, 1):
                node = hw_steps[i].nodes[j + top_node]
                if j == jmax:
                    # Branching C

                    node.j_up = j
                    node.j_m = j - 1
                    node.j_d = j - 2

                if j == jmin:
                    # Branching B

                    node.j_up = j + 2
                    node.j_m = j + 1
                    node.j_d = j

                if (j != jmin) and (j != jmax):
                    # Branching A
                    node.j_up = j + 1
                    node.j_m = j
                    node.j_d = j - 1
        return 0

    def find_connected_node(self, current_node, node_of_previous_step):
        connected_nodes = []
        for node in node_of_previous_step:
            up_id = str(node.i + 1) + "," + str(node.j_up)
            m_id = str(node.i + 1) + "," + str(node.j_m)
            d_id = str(node.i + 1) + "," + str(node.j_d)
            if up_id == current_node.id or m_id == current_node.id or d_id == current_node.id:
                connected_nodes.append(node)
        return connected_nodes

    def compute_arrow(self, current_node, connected_nodes, previous_a, dt, dr):
        for node in connected_nodes:
            if node.j_up == current_node.j:
                current_node.q += node.q * node.pu * math.exp(-(previous_a + node.j_up * dr) * dt)

            if node.j_m == current_node.j:
```

```
                current_node.q += node.q * node.pm * math.exp(-(previous_a + node.j_m * dr) * dt)

            if node.j_d == current_node.j:
                current_node.q += node.q * node.pd * math.exp(-(previous_a + node.j_d * dr) * dt)

#Draw
def draw_data(hw):
    N = hw.nbr_steps
    min_rate = hw.steps[0].nodes[0].rate
    max_rate = min_rate
    for i in range(0, N - 1, 1):
        top_node = min(i, hw.jmax)
        for j in range(-top_node, top_node + 1, 1):
            node = hw.steps[i].nodes[j + top_node]
            if node.rate < min_rate:
                min_rate = node.rate

            if node.rate > max_rate:
                max_rate = node.rate

            up = hw.steps[i + 1].nodes[node.j_up + top_node]
            m = hw.steps[i + 1].nodes[node.j_m + top_node]
            dw = hw.steps[i + 1].nodes[node.j_d + top_node]
            plt.plot([i, i + 1], [node.rate * 100, up.rate * 100])
            plt.plot([i, i + 1], [node.rate * 100, m.rate * 100])
            plt.plot([i, i + 1], [node.rate * 100, dw.rate * 100])

    # get_min_rate

    plt.xlabel('Maturity')
    plt.ylabel('Rate')
    plt.title('Hull White')
    plt.show()

#Test
hwc = HWCalculator()
hwc.maturity = 18
hwc.alpha = 0.1
hwc.volatility = 0.01
hwc.period ="year"
hwc.nbr_steps = 30
hwc.rates = []
for i in range(0,30,1):
    hwc.rates.append(0.08 - 0.05 * math.exp(-0.18 * i))
hwc.execute()
draw_data(hwc)
```