

Pricing d'options en Quasi Monte Carlo

Grégoire Jauvion,
Ecole Centrale Paris
Momentum Consulting, Paris, France
Arbitragis Trading, Paris, France

29 Août, 2008

Résumé

Nous allons décrire l'implémentation sur GPU (graphic processor unit) d'une méthode de pricing permettant de pricer différents types d'options assez complexes. Nous verrons d'une part le pricing d'options exotiques (asiatiques et lookbacks) portant sur un sous-jacent, et d'autre part le pricing d'options européennes portant sur plusieurs sous-jacents corrélés. Nous implémenterons également des modèles à volatilité locale.

La méthode du Monte Carlo est une méthode de pricing bien connue et assez simple à mettre en oeuvre. Elle consiste à approcher la valeur de l'espérance mathématique d'une variable aléatoire X par la moyenne d'un grand nombre w de réalisations de cette variable aléatoire. Par exemple, on peut estimer la valeur d'un call européen de strike K et d'échéance T (de valeur $V_0 = (S_T - K)_+$, S_T étant la valeur de l'actif à l'échéance) par $v = \frac{1}{w} \sum_{i=1}^w (S_T^i - K)_+$, S_T^i étant une réalisation de la variable aléatoire S_T .

Table des matières

1 Méthode de Quasi Monte Carlo

1.1 Principe Général

Rappelons que dans une méthode de Monte Carlo, on a :

$$V_0 = \frac{1}{w} \sum_{i=1}^w V_0^i \text{ où } V_0^i \text{ est une réalisation de } V_0 \quad (1)$$

On suppose maintenant que l'actif sous-jacent suit le modèle de Black-Scholes, on a alors :

$$S_T = S_0 \cdot e^{(r - \frac{\sigma^2}{2}) \cdot T + \sigma \cdot B_T} \text{ où } (B_t)_{t \geq 0} \text{ est un mouvement brownien standard} \quad (2)$$

Supposons que l'on veuille pricer un call européen, on veut donc estimer l'espérance de la variable aléatoire

$$V_0 = (S_T - K)_+ \quad (3)$$

On va alors générer un grand nombre w de réalisations de $S_T = S_0 \cdot e^{(r - \frac{\sigma^2}{2}) \cdot T + \sigma \cdot B_T}$, et on aura :

$$c \cong \frac{1}{w} \cdot \sum_{i=1}^w (S_T^i - K)_+ \text{ pour } w \text{ assez grand} \quad (4)$$

On voit alors que la seule difficulté restante est de simuler un mouvement brownien. C'est là que les méthodes de Monte Carlo et de quasi Monte Carlo se distinguent. Dans un Monte Carlo classique, on génère w réalisations g_i d'une gaussienne centrée réduite, puis on en déduit une réalisation d'un mouvement brownien par la formule $B_T^i = \sqrt{T} \cdot g_i$.

Dans un quasi Monte Carlo, on procède d'une manière un peu plus subtile : les réalisations du mouvement brownien ne sont pas générées de façon aléatoire, mais de façon pseudo-aléatoire. On ne va pas utiliser des variables aléatoires uniformes (permettant ensuite de générer des gaussiennes par la transformation de Box-Muller), mais des variables déterministes qui sont plus uniformément distribuées. En fait, on s'arrange pour que les nombres aléatoires choisis soient bien uniformes, et ceci améliore nettement la convergence du Monte Carlo.

Il existe plusieurs types de suites de nombres pseudo-aléatoires. Nous avons choisi les nombres de Sobol. Dans un premier temps, nous allons voir comment générer des suites de Sobol. Ensuite, nous verrons une méthode de génération des mouvements browniens permettant d'améliorer la vitesse de convergence de l'algorithme : *le pont brownien*

1.2 Construction des suites de Sobol

Nous allons présenter une méthode de construction des suites de Sobol, qui sont des suites de nombres pseudo-aléatoires. Dans la suite, on va générer une suite de Sobol de dimension d (c'est-à-dire qu'elle contient d nombres pseudo-aléatoires).

Soit b un entier. On va générer d nombres entiers pseudo-aléatoires x_i compris entre 1 et $(2^b - 1)$, et donc codés sur b bits. b représente en quelque sorte la précision de la génération des nombres de Sobol.

Pour chaque dimension, on va donc générer un nombre qui s'écrit sur b bits. Pour cela, on va avoir besoin de b entiers appelés *entiers de direction* (il y en aura donc $b \times d$ pour générer une suite de Sobol de dimension d).

1.2.1 Construction des entiers de direction

Pour une dimension k donnée, on considère un polynôme primitif P à coefficients dans $\{0,1\}$ de degré quelconque g . Un polynôme de degré g à coefficients dans $\{0,1\}$ est dit *primitif modulo 2* lorsqu'il est d'ordre $2^g - 1$, c'est-à-dire que $(2^g - 1)$ est le plus petit entier q pour lequel $P(x)$ divise $(x^q - 1)$.

On utilisera lors de l'implémentation des suites de Sobol un fichier contenant les polynômes primitifs modulo 2 jusqu'au degré vingt-sept (il y en a plus de deux millions). Les polynômes sont classés par degré croissant, et sont représentés par un entier, dont l'écriture en binaire donne les coefficients du polynôme.

On ne tient pas compte des premiers et derniers coefficients, qui valent forcément 1 : le premier car il donne le degré du polynôme, qui est déjà connu, et le dernier car le polynôme est primitif. Par exemple, le polynôme primitif de degré 5 représenté par l'entier 13 (en binaire 1101) est : $P(x) = x^5 + x^4 + x^3 + x + 1$.

Soit $P(x) = \sum_{j=0}^g a_j \cdot x^{g-j}$ un polynôme primitif de degré g . On se donne g entiers v_1, \dots, v_g (appelés *entiers de direction*) tels que seuls les l bits les plus forts de v_l puissent être non nuls, et que le l^{eme} bit le plus fort de v_l soit égal à 1.

On va alors en déduire par la relation de récurrence suivante les *entiers de direction* v_{g+1}, \dots, v_b :

$$v_l = \frac{v_{l-g}}{2^g} \oplus \sum_{j=1}^g a_j \cdot v_{l-j} \text{ pour } l > g_k \quad (5)$$

Le symbole \oplus est ici le ou exclusif, qui est l'addition en binaire (le *sigma* correspond aussi à un ou exclusif). C'est de cette relation de récurrence que vient le caractère pseudo-aléatoire des suites de Sobol : les bits les plus faibles sont générés à partir des bits les plus forts.

Pour chaque dimension, on est alors capables de générer b nombres de direction. On notera par la suite v_{kl} le l^{eme} entier de direction associé à la dimension k .

1.2.2 Génération des nombres de Sobol

On se donne maintenant un entier n strictement positif, et on pose :

$$\forall k \in [1, d], x_k = \sum_{l=1}^b v_{kl} \cdot \mathbf{1}_{\{\text{le } l^{eme} \text{ bit de } n \text{ vaut } 1\}} \quad (6)$$

Pour chaque dimension, on fait la somme des *entiers de direction* correspondant aux bits de l'écriture de n . En posant alors $y_k = \frac{x_k}{2^b}$, on obtient une suite de nombres pseudo-aléatoires compris entre 0 et 1. Ces nombres pseudo-aléatoires générés dépendent d'un entier n strictement positif. Ceci nous permet alors de générer différentes suites de Sobol en modifiant simplement n . Lors du quasi Monte Carlo, l'entier n sera tout simplement utilisé pour générer la *neme* trajectoire.

1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
2	0.75	0.25	0.75	0.75	0.25	0.75	0.25	0.25	0.75	0.25
3	0.25	0.75	0.25	0.25	0.75	0.25	0.75	0.75	0.25	0.75
4	0.875	0.375	0.125	0.125	0.625	0.625	0.875	0.375	0.125	0.125
5	0.375	0.875	0.625	0.625	0.125	0.125	0.375	0.875	0.625	0.625
6	0.625	0.625	0.375	0.375	0.375	0.875	0.125	0.625	0.375	0.875
7	0.125	0.125	0.875	0.875	0.875	0.375	0.625	0.125	0.875	0.375
8	0.9375	0.0625	0.3125	0.1875	0.0625	0.0625	0.8125	0.8125	0.5625	0.1875
9	0.4375	0.5625	0.8125	0.6875	0.5625	0.5625	0.3125	0.3125	0.0625	0.6875
10	0.6875	0.8125	0.0625	0.4375	0.8125	0.3125	0.0625	0.0625	0.8125	0.9375

TAB. 1 – Les 10 premières suites de Sobol en 10 dimensions

On utilise maintenant la transformation de Box-Muller pour obtenir une suite de réalisations de lois normales : si u et v sont deux lois uniformes indépendantes, alors $x = \sqrt{-2\ln(u)}\sin(2\pi v)$ et $y = \sqrt{-2\ln(u)}\cos(2\pi v)$ sont deux lois normales indépendantes. On va alors parcourir deux par deux les nombres de Sobol pour les transformer en pseudo-gaussiennes.

1.2.3 Remarques

- la génération des suites de Sobol dépend simplement d'un entier n . N'oublions pas que cet entier n doit être codé sur b bits, et on doit alors avoir $n \leq 2^b - 1$. Ainsi, le nombre de trajectoires générées dans le quasi Monte Carlo ne peut excéder cette valeur. Pour simuler plus de trajectoires, il faut augmenter b .

- les nombres générés dépendent des *entiers de direction*, dont les g premiers (g étant le degré du polynôme primitif choisi) doivent vérifier deux contraintes : seuls les k bits les plus forts du k^{eme} entier de direction peuvent être à 1, et le k^{eme} est forcément à 1. Il existe plusieurs possibilités pour les choisir. Dans la suite, nous choisirons $v_i = 2^{b-i} \text{int}(u_i 2^i)$, où u_i est un nombre tiré uniformément entre 0 et 1, tel que $\text{int}(u_i 2^i)$ soit impair (si ce n'est pas le cas, on tire un autre u_i).

1.3 Génération des mouvements browniens

Nous allons voir maintenant comment générer un mouvement brownien sur une trajectoire. Soit $t_0 = 0, \dots, t_n = T$ la discrétisation de la trajectoire. Il sera utile par la suite de discrétiser les trajectoires, pour pricer des options exotiques, pour intégrer les dividendes, ou pour utiliser des modèles à volatilité locale. On cherche donc à simuler une trajectoire d'un mouvement brownien standard $(B_t)_{t \geq 0}$ sur cette discrétisation. On suppose que l'on a généré n lois pseudo-normales de la manière vue au paragraphe précédent g_1, \dots, g_n , qui vont nous servir à générer les n étapes du mouvement brownien.

La méthode la plus simple serait d'utiliser la formule : pour $i > 0$, $B_{t_i} = \sqrt{t_i - t_{i-1}} \cdot g_i$.

Nous allons utiliser une méthode plus performante, appelée pont brownien, et qui présente une meilleure convergence (moins de trajectoires seront nécessaires pour avoir la précision recherchée). Cette méthode est basée sur la formule suivante :

$$\text{Si } t_i < t_j < t_k, \text{ alors } W_{t_j} = \frac{t_k - t_j}{t_k - t_i} W_{t_i} + \frac{t_j - t_i}{t_k - t_i} W_{t_k} + \sqrt{\frac{(t_j - t_i)(t_k - t_j)}{t_k - t_i}} g \text{ où } g \text{ est une loi normale} \quad (7)$$

D'après cette formule, on peut déduire B_{t_j} de B_{t_i} et B_{t_k} .

On commence par générer la trajectoire au pas de temps final t_n à l'aide de g_1 : $B_{t_n} = \sqrt{t_n} g_1$. g_2 va maintenant être utilisé pour déterminer un B_{t_j} avec $1 \leq i \leq n-1$ à l'aide de la formule précédente, connaissant B_{t_n} qui vient d'être déterminé, et $B_{t_0} = B_0$ qui est nul.

On va maintenant expliciter l'ordre de génération des B_{t_j} . Si n est une puissance de deux, la construction du mouvement brownien est assez simple à mettre en oeuvre : chaque intervalle est divisé en deux intervalles contenant le même nombre de noeuds, et on procède par dichotomie. Dans le cas général, on procède de la même manière, sauf que les deux intervalles issus d'une subdivision n'auront pas forcément le même nombre de noeuds.

Maintenant que l'on sait générer des mouvements browniens, on est en mesure (voir 1.1) de calculer une valeur approchée du prix de l'option.

1.4 Prise en compte de dividendes discrets

On suppose ici que l'on dispose d'une discrétisation $t_0 = 0, \dots, t_n = T$. On suppose également que l'actif sous-jacent à l'option verse des dividendes discrets de valeurs c_1, \dots, c_k aux temps d_1, \dots, d_k . On va alors modifier l'échéancier pour tenir compte des dividendes : pour $i \in \llbracket 1, k \rrbracket$, si d_i correspond à une tombée de dividende et n'est pas déjà un pas de temps t_j , on l'ajoute à l'échéancier.

Soit $t'_1 = 0, \dots, t'_m = T$ le nouvel échéancier ainsi formé : chaque pas de temps t'_j peut avoir trois statuts différents : pas de discrétisation, tombée de dividende ou les deux.

On parcourt cet échéancier de 0 à T , et pour chaque pas t'_i , on ajuste le spot en tenant compte d'éventuels dividendes, ce qui nous permet de calculer une nouvelle valeur de S (valeur du sous-jacent sur la trajectoire) à chaque pas de discrétisation :

- dividende c_i tombant en d_i : $spot = spot - c_i \cdot e^{-(r - \frac{\sigma^2}{2})d_i - \sigma B_{a_i}}$
- t'_i correspond au k^{eme} pas de discrétisation ($t'_i = t_k$) : $S_{t'_i} = spot \cdot e^{(r - \frac{\sigma^2}{2})t'_i + \sigma B_{t'_i}}$, et $S_{t'_i}$ prend alors en compte les dividendes versés avant t'_i

1.5 Utilisation de modèles à volatilité locale

Jusque là, nous avons considéré que la volatilité du sous-jacent restait contante. Ceci est loin d'être vrai en pratique : on va supposer que la volatilité dépend du spot : $\sigma = \sigma(S) = \sigma_0 \cdot f(S)$.

On suppose toujours que les sous-jacents suivent le modèle de Black-Scholes, on a donc :

$$\frac{dS_t}{S_t} = \mu \cdot dt + \sigma(S_t)B_{dt} \text{ où } \mu \text{ est l'espérance de rentabilité du sous-jacent} \quad (8)$$

On peut donc écrire :

$$S_T = S_0 + \int_0^T dS_t = S_0 + \int_0^T S_t \cdot (\mu dt + \sigma(S_t)dB_t) \quad (9)$$

On ne peut pas résoudre cette équation différentielle stochastique dans le cas général. On va maintenant considérer deux schémas : le schéma d'Euler, du premier ordre, et le schéma de Milhstein, du second ordre.

1.5.1 Le schéma d'Euler

On fait dans ce schéma une approximation : on suppose que si $t \in [0, T - \Delta T]$ et si Δt est suffisamment petit, alors le spot et la volatilité ne varient pas sur l'intervalle $[t, t + \Delta t[$ et valent respectivement S_t et $\sigma(S_t)$. On trouve alors :

$$\forall t \in [0, T - \Delta T], S_{t+\Delta t} = S_t + \int_t^{t+\Delta T} S_t \cdot (\mu dt' + \sigma(S_t)dB_{t'}) \quad (10)$$

On subdivise alors de manière régulière l'intervalle $[0, T]$ à l'aide de t_0, \dots, t_n , la subdivision étant assez fine pour pouvoir supposer que S_t et $\sigma(S_t)$ restent constants sur $[t_k, t_{k+1}[$ et valent respectivement S_{t_k} et $\sigma(S_{t_k})$. On peut donc écrire :

$$\forall k \in [0, n - 1], S_{t_{k+1}} = S_{t_k}(1 + \mu\Delta T + \sigma(S_{t_k})(B_{t_{k+1}} - B_{t_k})) \quad (11)$$

En partant de S_0 qui est connu, on va alors calculer les S_{t_i} de proche en proche, pour en déduire $S_{t_n} = S_T$.

Ce schéma est aussi applicable lorsque l'on considère plusieurs sous-jacents corrélés : on peut traiter les sous-jacents de manière indépendante en utilisant les mouvements browniens corrélés, ce qui ne sera pas aussi simple pour le schéma de Milhstein.

1.5.2 Le schéma de Milhstein pour un sous-jacent

Lorsqu'il n'y a qu'un sous-jacent, le schéma de Milhstein est défini par le schéma suivant :

$$\forall k \in [0, n - 1], S_{t_{k+1}} = S_{t_k}(1 + \mu\Delta T + \sigma(S_{t_k})(B_{t_{k+1}} - B_{t_k})) + \frac{\partial \sigma}{\partial S}(S_{t_k}) \cdot \sigma(S_{t_k}) \int_{t_k}^{t_{k+1}} (B_s - B_{t_k})dB_s \quad (12)$$

Essayons de comprendre l'apparition du terme $\frac{\partial \sigma}{\partial S}(S_{t_k}) \cdot \sigma(S_{t_k}) \int_{t_k}^{t_{k+1}} (B_s - B_{t_k})dB_s$ en considérant l'équation $dS_t = \sigma(S_t) \cdot dB_t$, qui donne $S_{t_{k+1}} = S_{t_k} + \int_{t_k}^{t_{k+1}} \sigma(S_t)dB_t$.

On remplace alors la variable S_t par sa valeur dans un schéma d'Euler : $S_t = S_{t_k} + \sigma(S_{t_k})(B_t - B_{t_k})$. Un développement de Taylor de $\sigma(S_t)$ donne alors :

$$\sigma(S_t) = \sigma(S_{t_k} + \sigma(S_{t_k})(B_t - B_{t_k})) \quad (13)$$

$$\cong \sigma(S_{t_k}) + \frac{\partial \sigma}{\partial S}(S_{t_k}) \cdot \sigma(S_{t_k})(B_t - B_{t_k}) \quad (14)$$

Dans le cas de l'équation de Black-Scholes, on retrouve donc bien le schéma proposé. On calcule maintenant le terme $\int_{t_k}^{t_{k+1}} (B_t - B_{t_k})dB_t$ à l'aide de la formule d'Itô :

$$\int_{t_k}^{t_{k+1}} (B_t - B_{t_k})dB_t = \frac{(B_{t_{k+1}} - B_{t_k})^2 - (t_{k+1} - t_k)}{2} \quad (15)$$

On trouve donc finalement :

$$S_{t_{k+1}} = S_{t_k}(1 + r \cdot \Delta T + \sigma(S_{t_k})((B_{t_{k+1}} - B_{t_k}) + \frac{1}{2} S_{t_k} \frac{\partial \sigma}{\partial S}(S_{t_k})((B_{t_{k+1}} - B_{t_k})^2 - \Delta T))) \quad (16)$$

1.5.3 Le schéma de Milhstein en multi sous-jacent

Comme dans le cas un sous-jacent, on utilise la formule de Taylor pour déterminer le schéma de Milhstein. Le schéma associé au n^{eme} sous-jacent est le suivant :

$$S_{t_{k+1}}^n = S_{t_k}^n (1 + \mu \cdot \Delta T + \sigma(S_{t_k})(B_{t_{k+1}} - B_{t_k})) + \sum_{j,l=1}^p (\partial \sigma_j \sigma_l)(S_{t_k}) \int_{t_k}^{t_{k+1}} (B_s^j - B_{t_k}^j) dB_s^l \quad (17)$$

avec, pour $1 \leq i \leq n$

$$(\partial \sigma_j \sigma_l)_i(x) = \sum_{r=1}^n \frac{\partial \sigma_{ij}(x)}{\partial x_r} \quad (18)$$

et où $(B_t^k)_{t \geq 0}$ est le mouvement brownien associé au k^{eme} sous-jacent.

Pour plus de précision sur le schéma de Milhstein en plusieurs sous-jacents, voir le cours sur le Monte Carlo du DEA de Paris VII, qui est sur le wiki.

2 Implémentation du quasi Monte Carlo

Deux types d'options ont été implémentés : les options asiatiques et les options européennes sur plusieurs sous-jacents.

2.1 Pricing d'options asiatiques

Une option asiatique est une option exotique qui ne peut être exercée qu'à l'échéance et dont le payoff vaut $(S_{moy} - K)_+$, où S_{moy} est la moyenne arithmétique du spot sur $[0, T]$. On considère dans la suite que l'on veut pricer un call. On va d'abord discrétiser l'intervalle $[0, T]$ en n intervalles de même taille : soit $t_i = i \cdot \frac{T}{n}$. On génère alors un nombre important w de réalisations de mouvements browniens (on génère leurs valeurs aux t_i) par la méthode décrite précédemment. Ceci nous permet de générer les valeurs aux t_i de w trajectoires du cours de l'actif sous-jacent (en tenant compte d'éventuels dividendes).

On calcule alors sur chaque trajectoire le payoff dû à l'exercice de l'option en approchant S_{moy} par $S_{moy}^n = \frac{1}{n} \cdot \sum_{i=1}^n S_{t_i}$.

Une valeur approchée du prix de l'option calculée avec n pas de temps et w trajectoires s'en déduit facilement : $c_{wn} = \frac{1}{w} \cdot \sum_{i=1}^w (S_{moy}^n(i) - K)_+$

2.2 Pricing d'options en multi sous-jacent

Nous allons maintenant voir comment pricer en quasi Monte Carlo une option européenne portant sur n sous-jacents.

On veut pricer une option européenne : il n'est donc pas nécessaire a priori de discrétiser l'intervalle $[0, T]$. Cependant, on va le faire pour prendre en compte des dividendes discrets. Un instant t sera un pas de discrétisation dès lors qu'il correspond à une tombée de dividende de l'un des sous-jacents : ceci nous permettra d'ajuster le spot de ce sous-jacent.

On va maintenant voir comment générer $w \times n$ mouvements browniens (w pour chaque sous-jacent). On va en fait générer n mouvements browniens d'un coup (et pour chacun d'entre eux, on génère les valeurs aux d pas de discrétisation). Pour cela, on commence par générer une suite de sobol de dimension $d \times n$, que l'on transforme ensuite en réalisations de lois normales par la transformation de Box-Muller. On applique alors la méthode du pont brownien sur chaque sous-jacent en utilisant les pseudo-gaussiennes correspondant aux dimensions de Sobol les plus basses (qui sont en fait les variables les plus uniformes) pour les points qui doivent être construits en premier (le tout premier correspondant à la maturité de l'option).

On dispose alors de w matrices $n \times d$ de mouvements browniens indépendants B_{ind}^i . Supposons que les actifs soient corrélés : soit C la matrice de corrélation $n \times n$. On calcule sa racine carrée L par la

décomposition de Cholesky, puis pour chaque trajectoire w , on peut alors construire une matrice de mouvements browniens corrélés $B_{cor}^i = L \times B_{ind}^i$.

On peut maintenant générer w trajectoires du cours des n sous-jacents à l'aide des mouvements browniens corrélés générés. On calcule ensuite le payoff de l'option sur chaque trajectoire en parcourant l'échéancier pour tenir compte d'éventuels dividendes, et on en déduit une valeur approchée du prix de l'option.

3 Implémentation en GPU

Une GPU est composée de plusieurs multiprocesseurs. Chacun de ces multiprocesseurs peut traiter parallèlement des groupes de tâches. Il y a donc deux niveaux de parallélisation. Tandis que la synchronisation des tâches effectuées par un même multiprocesseur est rapide, il est beaucoup plus long de synchroniser les multiprocesseurs entre eux, ce qui oblige à séparer le problème que l'on veut implémenter en sous-problèmes indépendants.

Chaque multiprocesseur d'une GPU contient de la *mémoire partagée*, à laquelle peuvent accéder simultanément les tâches effectuées par ce multiprocesseur. L'accès à cette mémoire est extrêmement rapide, mais elle est de taille faible (16Kb pour une Nvidia Geforce 8000, ce qui permet de stocker 4000 entiers ou *floats*), ce qui nous le verrons a posé certains problèmes. De plus, la mémoire principale de la GPU est de l'ordre de 800Mo, ce qui est insuffisant lorsque la dimension ou le nombre de trajectoires simulées est trop grand.

3.1 Génération des suites de Sobol

Voyons comment générer à l'aide de la GPU w suites de Sobol de dimension d , les entiers générés étant codés sur b bits. On commence par générer sur CPU les *entiers de direction*. Ça ne vaut pas vraiment le coup de paralléliser ce calcul, car il n'y en a que $b \times d$. On copie ensuite ces nombres dans la mémoire principale de la GPU. La première étape est de copier les *entiers de direction* dans la mémoire partagée de chaque multiprocesseur (ce qui permettra d'y accéder de manière simultanée). Chaque multiprocesseur va alors traiter une trajectoire, et sur chaque trajectoire, on traite les dimensions de manière parallèle également (ce qui est possible car les dimensions sont indépendantes). On applique enfin la transformation de Box-Muller sur les nombres de Sobol pour obtenir des gaussiennes.

Mais il faut faire ici attention à la taille de la mémoire partagée. En effet, on y copie $b \times d$ *floats*, qui doit être inférieur à la taille de la mémoire partagée (4000 *floats* pour une Geforce 8000). En choisissant $b = 20$, on se limite alors à $d = 200$ dimensions, ce qui est loin d'être suffisant pour les applications pratiques. On va alors découper la génération des suites de Sobol en traitant les dimensions par paquets. La taille des paquets a été prise égale à 128 : il est optimal de choisir une puissance de 2 pour le traitement parallèle des dimensions au sein d'un paquet. Pour chaque paquet de dimensions, on copie en mémoire partagée les direction numbers correspondants, puis on crée les nombres de Sobol dans la mémoire principale de la GPU.

3.2 Construction du pont brownien

On va maintenant générer un tableau de w mouvements browniens par la méthode du pont brownien, à partir des nombres de Sobol générés précédemment. Pour cela, on va commencer par construire des tableaux contenant toutes les données utiles à la construction des ponts browniens. Soit $t_0 = 0, \dots, t_n = T$ la discrétisation du temps.

On a en fait besoin de 6 tableaux :

- `bridgeIndex` : $bridgeIndex[i] = l$ signifie que le i^{eme} point construit (à l'aide de la i^{eme} gaussienne) correspond au l^{eme} pas de temps
- `leftIndex` et `rightIndex` : $leftIndex[i] = j$ et $rightIndex[i] = k$ signifient que la valeur du brownien au i^{eme} point construit (correspondant donc au pas de temps $bridgeIndex[i]$) sera générée à partir des valeurs au j^{eme} et au k^{eme} pas de temps

- leftWeight et rightWeight : si $leftIndex[i] = j$ et $rightIndex[i] = k$, on a alors $leftWeight[i] = \frac{t_k - t_i}{t_k - t_j}$ et $rightWeight[i] = \frac{t_l - t_j}{t_k - t_j}$, qui correspondent aux poids de j et k dans la construction du i^{eme} point
- spread : $spread[i] = \sqrt{\frac{(t_l - t_j)(t_k - t_l)}{t_k - t_j}}$ et correspond au coefficient devant la gaussienne dans la formule du pont brownien

Après avoir construit ces tableaux (sur CPU : la encore ça ne vaut pas le coup de le faire sur GPU), on les copie dans la mémoire partagée de chaque multiprocesseur. On choisit cette fois un autre schéma de parallélisation que celui utilisé pour générer les suites de Sobol : un multiprocesseur traitera cette fois plusieurs trajectoires en même temps : le traitement d'une trajectoire sera assimilé à une tâche. Sur chaque trajectoire, on va alors parcourir dans l'ordre le tableau bridgeIndex, et construire chaque point en se servant des autres tableaux.

On est confronté ici encore au problème de la taille de la mémoire partagée. On y copie 6 tableaux qui ont pour taille le nombre de dimensions, ce qui limite le nombre de dimensions à 666 (pour une Nvidia Geforce 8000). On va alors traiter les dimensions par paquet de taille 512 : pour chaque paquet, on copie en mémoire partagée les parties des 6 tableaux correspondant aux dimensions concernées, puis on peut traiter la partie correspondante du pont brownien.

3.3 Pricing d'options asiatiques

Maintenant que l'on est en mesure de générer des mouvements browniens, voyons comment paralléliser l'étape finale du Monte Carlo : le calcul du payoff sur chaque trajectoire. Le schéma de parallélisation utilisé est le même que pour le pont brownien : chaque trajectoire correspond à une tâche, ainsi plusieurs trajectoires peuvent être traitées en même temps par un même multiprocesseur.

On crée dans la mémoire partagée de chaque multiprocesseur un tableau *prices* destiné à contenir les payoffs de l'option sur chaque trajectoire traitée par ce multiprocesseur. Ce tableau a pour taille le nombre de trajectoires pouvant être traitées en même temps par le multiprocesseur. On calcule alors le payoff de l'option sur chaque trajectoire en parcourant l'échéancier et en tenant compte des éventuels dividendes. Les payoffs sont accumulés dans le tableau *prices*. On copie ensuite les tableaux *prices* correspondant à chaque multiprocesseur dans la mémoire de la CPU, puis on en déduit le prix de l'option.

3.4 Pricing d'options en multi sous-jacent

La seule chose à rajouter pour pricer une option portant sur plusieurs sous-jacents est la prise en compte de la matrice de corrélation. On a déjà vu comment générer des mouvements browniens indépendants. Il faut maintenant multiplier chacun d'eux par la racine carrée de la matrice de corrélation afin d'obtenir des mouvements browniens corrélés.

L'obtention de la racine carrée de la matrice de corrélation par décomposition de Cholesky n'a pas été parallélisée. En effet, ce calcul ne prend que quelques millisecondes, ce qui est très faible par rapport aux temps de calcul mis en jeu, plutôt de l'ordre de la minute. On calcule la racine carrée de la matrice de corrélation à l'aide de la gsl ¹, puis on la copie sur la GPU.

Pour ce qui est des multiplications matricielles, on choisit le même schéma de parallélisation que pour la génération des nombres de Sobol : chaque multiprocesseur traite les trajectoires une par une. On copie tout d'abord la racine carrée de la matrice de corrélation dans la mémoire partagée de chaque multiprocesseur, qui peut alors traiter les trajectoires.

On pourrait penser à un autre schéma de parallélisation : la matrice de corrélation est copiée dans la mémoire partagée de chaque multiprocesseur, et un multiprocesseur traite ensuite plusieurs chemins en même temps. Ce schéma est cependant beaucoup moins performant à cause d'accès concurrentiels à la matrice de corrélation.

¹GNU scientific library : librairie mathématique en C++

4 Etude des résultats renvoyé par l'algorithme

4.1 Pricing d'options asiatiques

On va supposer pour simplifier l'étude que les caractéristiques du sous-jacent (spot, strike, volatilité, dividendes, taux d'intérêt) n'ont aucun impact sur la complexité et la précision de l'algorithme. Les paramètres à étudier de plus près sont la discrétisation et le nombre de chemins utilisés. On peut montrer que si d est le nombre de dimensions du quasi Monte Carlo (c'est-à-dire le nombre de pas de discrétisations) et si w est le nombre de chemins générés, alors l'erreur maximale du pricing est de l'ordre de $c(d) \cdot \frac{\log(w)^d}{w}$, où $c(d)$ est une constante ne dépendant que de la dimension. On voit bien que si le nombre de dimensions augmente, il faut aussi augmenter w pour faire tendre cette erreur vers 0.

On trouve empiriquement que pour une dimension de l'ordre de 10000, le prix commence à converger vers une valeur stable lorsque le nombre de chemins est de l'ordre de 100000 (la précision relative est alors de l'ordre de 0.01%). Il n'est donc pas nécessaire d'en utiliser plus.

4.2 Pricing d'options européennes en multi sous-jacent

Ici encore, les spots et volatilités des sous-jacents, ainsi que le strike et la maturité de l'option n'interviennent pas dans la précision des résultats. Par contre, les dividendes, qui définissent la discrétisation donc le nombre de dimensions, interviennent. Le paramètre à considérer est le nombre de dimensions, qui est la taille d'un vecteur sans répétitions contenant les dividendes de tous les sous-jacents multiplié par le nombre de sous-jacents.

On va donc étudier empiriquement la précision des résultats en faisant varier le nombre de dimensions ainsi que le nombre de sous-jacents.

Lorsqu'il n'y a pas de dividendes, le nombre de dimensions est alors égal au nombre de sous-jacents. Un faible nombre de chemins (de l'ordre de 50000) est alors suffisant pour avoir une estimation du prix avec une précision relative de 0.01%, même s'il y a beaucoup de sous-jacents (de l'ordre de 50).

Si les sous-jacents versent des dividendes, le nombre de dimensions peut grimper extrêmement vite, mais un nombre de chemins de l'ordre de 100000 est encore amplement suffisant pour une précision de 0.01%.

4.3 Etude de modèles à volatilité locale : le modèle de Dupire

On va maintenant étudier les résultats renvoyés par le modèle à volatilité locale. Le but sera de déterminer des conditions d'utilisation de ce modèle. On fait un développement limité de la formule de Black-Scholes (à l'ordre 1 dans le schéma d'Euler et à l'ordre 2 dans le schéma de Milhstein), donc il faut que la discrétisation soit suffisamment fine. Il sera également intéressant de déterminer dans quels cas le schéma de Milhstein converge plus vite que le schéma d'Euler.

4.3.1 Le schéma d'Euler

Le schéma d'Euler revient à effectuer un développement limité au premier ordre de $e^{r \cdot \Delta T + \sigma B_{\Delta T}}$, où ΔT est la finesse de la discrétisation, c'est-à-dire $\Delta T = \frac{T}{n}$, n étant le nombre de discrétisations. $\sigma B_{\Delta T}$ est de l'ordre de $\sigma \sqrt{\Delta T}$. Si l'on prend σ de l'ordre de 0.2 et r de l'ordre de 0.05, on voit alors que le terme $\sigma B_{\Delta T}$ prédomine sur $r \cdot \Delta T$ dès que $\Delta T \leq 16$, ce qui peut être supposé vrai.

On va donc supposer par la suite que le choix du schéma dépend que de $\sqrt{\frac{T}{n}}$.

L'erreur du schéma d'Euler a 2 principales causes : une erreur statistique et une erreur liée à la discrétisation (car l'on suppose que la volatilité locale ne varie pas sur un intervalle de longueur ΔT et car l'on fait un développement limité).

On s'intéresse ici à l'erreur générée par le développement limité. Cherchons dans un premier temps la valeur maximale de la variable x pour laquelle $|e^x - (1 + x)|$ est plus petit qu'une valeur fixée. Le développement limité étant fait à chaque pas de temps, il faut une très bonne précision à chaque développement limité pour obtenir une bonne précision sur le pricing. Si l'on veut par exemple une précision relative de 10^{-4} sur chaque développement limité, on doit avoir $x \leq 10^{-2}$, c'est-à-dire dans

notre problème $\sigma\sqrt{\frac{T}{n}} \leq 10^{-2}$, et en prenant $\sigma = 0.2$, on trouve $\frac{T}{n} \leq 2.10^{-3}$. On arrive alors à la condition $\frac{n}{T} \geq 400$. Remarquons qu’une précision relative de 10^{-4} sur chaque développement limité donne, avec 500 pas de temps et 1 sous-jacent (donc 500 développements limités effectués), une précision relative de l’ordre de 10^{-3} sur le prix. En revanche, exiger une meilleure précision au niveau du développement limité fait exploser le nombre de pas de temps nécessaires, donc le temps de calcul. Par exemple, une précision de 10^{-6} sur le développement limité entraîne la condition $\frac{n}{T} \geq 40000$.

On remarque empiriquement qu’augmenter le nombre de sous-jacents ne diminue pas trop la précision, bien que le nombre de développements limités effectués soit multiplié par le nombre de sous-jacents.

On supposera donc que le schéma d’Euler est utilisable dès lors que la condition $\frac{n}{T} \geq 500$ est vérifiée.

4.3.2 Le schéma de Milhstein

On peut montrer que dans la plupart des cas, les schémas d’Euler et de Milhstein ont la même vitesse de convergence. Le schéma de Milhstein nécessitant plus de calculs, on utilisera donc plutôt le schéma d’Euler.

Sur un sous-jacent, il n’est pas forcément préférable d’utiliser le schéma de Milhstein : pour une précision donnée, il converge vers un prix pour à peu près les mêmes paramètres que le schéma d’Euler, et son temps d’exécution est un petit peu plus long. Pour plusieurs sous-jacents, le schéma que j’ai implémenté est je pense trop simplifié, et ne permet donc pas d’obtenir de résultats concluants.

5 Remarques pratiques et développements futurs

5.1 Remarques pratiques

Pour compiler le projet, voir le *readme.txt*. Ne pas oublier de copier le fichier *PrimitivePolynomials.txt* dans le dossier contenant l’exécutable *MonteCarloGPU*, car c’est la qu’il sera lû (si ce n’est pas fait, le projet compile, mais l’exécution provoque une *segmentation fault*).

Il faut faire attention à la taille de la mémoire principale de la GPU sur laquelle sont faits les calculs. La taille de la mémoire d’une Nvidia Geforce 8000 est d’environ 800Mo. Si d est le nombre de dimensions du Monte Carlo et w le nombre de trajectoires simulées, pour faire un pricing d’une option sur un sous-jacent, on a besoin de stocker deux tableaux de taille $w \times d$ (un pour les nombres de Sobol et un pour les mouvements browniens), les autres tableaux étant de taille négligeable, et on doit donc avoir $2wd \times \text{sizeof(float)} \leq 800.10^6$. Or, $\text{sizeof(float)} = 4 \text{ octets}$, donc $wd \leq 100.10^6$. Lorsque wd est trop grand et provoque le remplissage de la mémoire de la GPU, soit le programme renvoie la valeur *inf*, ou soit il y a un *core dump*.

Pour un pricing d’une option portant sur plusieurs sous-jacents, on a besoin de stocker trois tableaux de taille $w \times d$ (le troisième contient les mouvements browniens corrélés), et on doit donc avoir $3wd \times \text{sizeof(float)} \leq 800.10^6$, c’est-à-dire $wd \leq 67.10^6$.

Ces limites ne sont pas suffisantes en pratique, et il va falloir traiter les trajectoires par paquets dont la taille dépendra de la dimension du Monte Carlo. On choisit toujours des paquets dont la taille est une puissance de 2, pour optimiser le partage des calculs sur la GPU. Par exemple, si l’on price une option asiatique avec 200 pas de discrétisation, on doit donc avoir $200.w \leq 100.10^6$, soit $w \leq 500.000$, et on traite alors les trajectoires en paquets de taille $2^{18} = 262144$.

Voyons la structure du projet pour l’exemple du pricing d’une option asiatique. Elle est la même pour les autres types de pricing (option multi sous-jacent, pricing avec volatilité locale). Le main appelle la fonction *asianOption* qui se trouve dans *asianOption.cpp*. Dans cette fonction, l’échéancier est calculé, et on calcule la taille des paquets de chemins qui seront envoyés à la fonction *asianOptionGPU* qui se trouve dans *asianOptionGPU.cu*. C’est cette fonction qui va effectuer le traitement des chemins étape par étape, en appelant différents kernels (fonctions qui effectuent des calculs parallélisés sur la GPU).

Dans le projet, les schémas d’Euler et de Milhstein ont été implémentés dans le cas d’options européennes sur plusieurs sous-jacents. Le schéma d’Euler est très simple à implémenter : une fois que

l'on a des mouvements browniens corrélés, on peut traiter les sous-jacents de manière indépendante et effectuer pour chacun d'eux un développement limité au premier ordre de la formule de Black Scholes.

Le schéma de Milhstein a été implémenté de la même manière (en faisant un développement limité au second ordre). Mais comme le montre la formule du 1.5.3, il faut tenir compte du terme $\int_{t_k}^{t_{k+1}} (B_s^j - B_{t_k}^j) dB_s^l$ où j et l sont deux sous-jacents. Le calcul de ce terme n'est peut être pas évident à implémenter, et je n'ai pas eu le temps de me pencher sur la question. Le schéma de Milhstein utilisé dans le projet est à mon avis faux s'il y a plusieurs sous-jacents, et d'ailleurs il ne converge pas vers la même valeur que si l'on utilise le schéma d'Euler (pour 20 sous-jacents, l'écart relatif est d'environ 10^{-2}).

Si le programme plante, c'est sûrement à cause d'un problème de mémoire (par exemple si la GPU n'est pas la même que celles sur lesquelles j'ai fait mes tests : *Nvidia Geforce 8000*, et *Macbook Pro*). Les caractéristiques de la GPU utilisée sont prises en compte dans l'algorithme, mais je ne peux pas être sûr que tout marche bien sur une autre GPU.

Le problème peut être dû au remplissage de la mémoire partagée des multiprocesseurs dans un kernel (fonction s'exécutant sur la GPU). Il faut alors diminuer les bons paramètres, par exemple la taille des paquets de dimensions pour la génération des nombres de Sobol ou pour le pont brownien, ou le nombre de sous-jacents pour la corrélation.

Mais la plupart des bugs viennent du remplissage de la mémoire principale de la GPU : il faut alors diminuer la taille des paquets de chemins. Pour un pricing d'option asiatique par exemple, celle-ci est déterminée dans *asianOption.cpp* : le paramètre *pathsDiscretization* correspond à la taille des paquets de chemins. Regardons la ligne de code suivante intervenant dans le calcul de ce paramètre :

$$intGPU\ MemorySize = static_cast < int > (ceil(0.9 * deviceProperties(0) * 1000000)); \quad (19)$$

Le paramètre 0.9 représente la part de la mémoire de la GPU qui sera remplie par les données du Monte Carlo : le diminuer revient à diminuer la taille des paquets de chemins, et corrige en général le bug.

5.2 Développements futurs

Il y a pour l'instant un problème dans l'implémentation des modèles à volatilité locale : la volatilité n'est pas considérée comme une fonction. Pour l'instant, elle est supposée être de la forme $\sigma = \sigma_0.S^\beta$. On passe donc en argument de la fonction de pricing un tableau contenant les coefficients σ_0 des sous-jacents, et un autre tableau contenant les coefficients β des sous-jacents. Or, en pratique, ce n'est pas du tout le cas : le modèle de volatilité utilisé est déterminé en fonction des données de marché.

La meilleure méthode est d'utiliser un pointeur sur la fonction volatilité. Ceci n'est pas possible sur CUDA (les différentes tâches de différents multiprocesseurs ne pourront pas accéder à la fonction volatilité en même temps). Il y a donc deux solutions : passer en argument de la fonction de pricing différents paramètres permettant de reconstituer la fonction de volatilité (qui pourra donc être calculée de manière parallèle), ou déparalléliser la dernière partie du Monte Carlo, dans laquelle on parcourt l'échéancier sur chaque chemin (la seule partie où on ait besoin de la volatilité).

Un autre développement intéressant serait d'implémenter le schéma de Milhstein sur plusieurs sous-jacents, afin de voir s'il est plus performant que le schéma d'Euler. Cela dit, dans le cas général, c'est le schéma d'Euler qui est plus performant en terme de temps de calcul (voir le cours Méthodes de Monte Carlo pour la finance du DEA de Paris VII), et le schéma de Milhstein n'est pas facile à implémenter avec plusieurs sous-jacents.

Dans le projet, le problème a été découpé de façon à ce que le nombre de chemins et la dimension puissent être théoriquement illimités. Par contre, le nombre de sous-jacents est limité à la racine carrée du nombre de *floats* que peut contenir la mémoire partagée de la GPU utilisée. Pour une Nvidia Geforce 8000, on se limite donc à 63 sous-jacents. Cette limitation vient du kernel *correlation_kernel* (ou *correlation.LocalVolatility_kernel*). Ce kernel remplit une matrice de mouvements browniens corrélés à partir d'une matrice de mouvements browniens corrélés.

Il n'est pas très compliqué d'adapter ce kernel pour qu'il tourne avec plus de sous-jacents (je n'ai pas eu le temps de le faire). Soit n le nombre de sous-jacents, d la dimension et t la taille de la mémoire partagée. On va générer les matrices $n \times d$ de mouvements browniens corrélés par paquets de lignes, chaque paquet ayant moins de \sqrt{t} lignes (les lignes correspondent aux sous-jacents), en ne copiant en mémoire partagée que les lignes utiles de la matrice de corrélation.

Enfin, une autre amélioration relativement simple consisterait à adapter l'algorithme pour qu'il puisse tourner sur plusieurs GPUs. Pour pricer une option asiatique par exemple, il suffit de rajouter un argument à la fonction `asianOption` représentant la GPU sur laquelle seront fait les calculs (cet argument est un entier compris entre 0 et $n-1$, où n est le nombre de GPUs installées). Si l'on a deux GPUs, on appelle la fonction `asianOptionGPU` appliquée à la moitié des chemins sur une GPU (à l'aide de l'instruction `cudaSetDevice(int)` qu'il faut placer au début de `asianOptionGPU`), et à l'autre moitié des chemins sur l'autre GPU. Ceci divise par deux le temps de calcul.

Références

- [1] Peter Jäckel : Monte Carlo methods in finance
Wiley finance
- [2] Emmanuel Temam : Méthodes de Monte Carlo pour la finance
DEA Statistiques et modèles aléatoires en économie et finance
Université Paris VII, Avril 2004
- [3] Victor Podlozhnyuk, Mark Harris
Monte Carlo Option Pricing
Nvidia, November 2007