Projet Informatique Probabilités Numériques

Antoine Falck
Djahiz Meliani

28 avril 2018

Sommaire

Introduction

Ce projet informatique a été effectué dans le cadre du cours de probabilités numériques dispensé par G. Pagès et V. Lemaire pour le Master 2 Probabilités et Finance de l'Université Pierre et Marie Curie. Il s'appuie

sur l'article *Fast Greeks by Algorithmic Differentiation* de L. Capriotti [1] pour le calcul de sensibilités. Pour l'implémentation de la méthode on a aussi pu se référer à deux autres de ses articles [2, 3].

Sujet

Calcul des sensibilités par différentiation automatique

Étudier la méthode numérique introduit dans l'article de L. Capriotti pour calculer des sensibilités par la méthode du flot (ou processus tangent) en utilisant une technique de différentiation automatique (*algorithm differentiation*).
Vérifier numériquement l'approche en insistant sur le coût numérique (la complexité). Il est possible d'appeler ou d'adapter des librairies existantes de différentiation automatique (cf. le site `http://www.autodiff.org`)

1 Description de la méthode

L'article montre comment la différentiation automatique peut être utilisée pour implémenter efficacement la méthode des trajectoires pour le calcul des sensibilités d'une option avec Monte Carlo. Cette méthode nécessite le calcul de la dérivée de la fonction *payout* de l'option, ce qui peut être long à calculer analytiquement ou par différences finies.

1.1 La méthode des trajectoires

Le prix d'une option peut s'écrire sous la forme d'une espérance sous la probabilité risque neutre de la fonction de payout de l'option. Ici nous notons le prix de l'option :

$$V \quad = \quad \mathbb{E}_{\mathbb{Q}} \left[P(X(T_1), \ldots, X(T_M)) \right] , \tag{1.1}$$

où P est la fonction *payout* de l'option, qui prend comme variable M observations d'une variable $X(T_i)$ avec $i = 1, \ldots, M$, de dimension N où les composantes de cette variable sont des facteurs de marché du sous jacent (prix, taux d'intérêt,...). On note $X = (X(T_1), \ldots, X(T_M))^{\mathrm{T}}$ ce vecteur d'état de dimension $d = M \times N$.

Le calcul du prix par la méthode de Monte Carlo consiste ensuite à simuler sous la probabilité risque neutre N_{MC} échantillons du vecteur X

puis à approcher le prix par :

$$V \approx \frac{1}{N_{\mathrm{MC}}} \sum_{i=1}^{N_{\mathrm{MC}}} P(X[i]), \tag{1.2}$$

avec une erreur $\frac{\Sigma}{\sqrt{N_{\mathrm{MC}}}}$ où $\Sigma^2 = \mathbb{E}_{\mathbb{Q}}\left[P(X)^2\right] - \mathbb{E}_{\mathbb{Q}}\left[P(X)\right]^2$ est la variance de P.

Pour calculer les sensibilités de l'option par rapport à un ensemble de paramètre $\theta = (\theta_1, \ldots, \theta_{N_\theta})$, on utilise le fait que sous condition de régularité de la fonction *payout* de l'option, on a :

$$\bar{\theta}_k = \frac{\partial V}{\partial \theta_k} = \mathbb{E}_{\mathbb{Q}}\left[\frac{\partial P_\theta(X)}{\partial \theta_k}\right]. \tag{1.3}$$

Où $\bar{\theta}_k$ est un estimateur de la k-ième sensibilité du prix. En pratique on obtient cette valeur en prenant la moyenne sur toutes les trajectoires de :

$$\frac{\partial P_\theta(X)}{\partial \theta_k} = \sum_{j=1}^{d} \frac{\partial P_\theta(X)}{\partial X_j} \frac{\partial X_j}{\partial \theta_k} + \frac{\partial P_\theta(X)}{\partial \theta_k}, \tag{1.4}$$

avec

$$\frac{\partial X_j}{\partial \theta_k} = \lim_{\Delta\theta \to 0} \frac{X_j(\theta_1, \ldots, \theta_k + \Delta\theta, \ldots, \theta_{N_\theta}) - X_j(\theta)}{\Delta\theta}. \tag{1.5}$$

Si le calcul de cette dérivée peut se faire efficacement, pour que la méthode reste efficace, il faut aussi pouvoir calculer facilement le dérivée de la fonction *payout*. Pour cela on va utiliser le différentiation automatique.

1.2 La différentiation automatique

L'idée de la différentiation automatique est la suivante. Si on considère un programme comme une suite d'opérations appliquées à des variables d'entrée pour obtenir des variables de sortie, on peut alors appliquer les règles de la dérivation à ces opérations pour obtenir un programme qui sera la dérivée du programme initial.

Si on prend la fonction $F : \mathbb{R}^n \to \mathbb{R}^m$ telle que $(y_1, \ldots, y_m)^{\mathrm{T}} = F(x_1, \ldots, x_n)$ alors calculer la dérivée de F revient à déterminer son Jacobien $J_{i,j} = \frac{\partial F_i(x)}{\partial x_j}$ avec $F_i(x) = y_i$

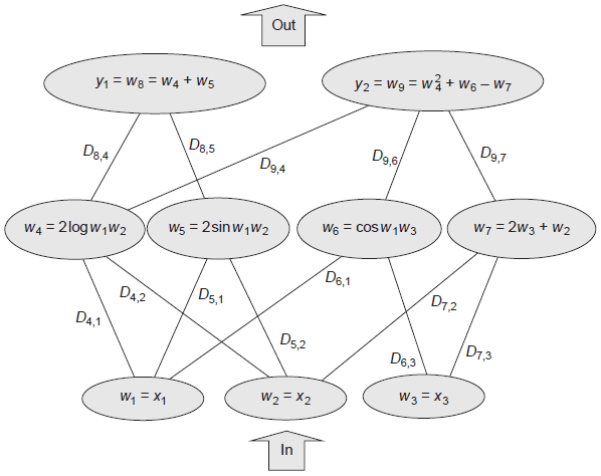

FIGURE 1 – Graphe des instructions.

Pour le calcul, on représente F comme une séquence d'opérations mathématiques simples et on note leurs résultats par des variables intermédiaires $w_1, \ldots, w_N$ de la façon suivante :

$$w_i = x_i \quad \text{si} \quad i = 1, \ldots, N \tag{1.6}$$

$$w_i = \phi_i\left(\{w_j\}_{j<i}\right) \quad \text{si} \quad i = n+1, \ldots, N \tag{1.7}$$

La valeur de ces variables intermédiaires est donc la même que les variables d'entrées pour les n premières puis une combinaison d'opérations notée ϕ_i pour les $N - n - 1$ suivantes. La combinaison donnant w_i doit dépendre uniquement des variables w_j où $j < i$.

Cela permet de représenter le calcul de la fonction F comme une succession de calculs intermédiaires plus simples. De plus cela permet aussi de le représenter sous la forme d'un graphique. Voir ci dessous un exemple pour la fonction F définie par :

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 2\log x_1 x_2 + 2\sin x_1 x_2 \\ 4\log^2 x_1 x_2 + \cos x_1 x_2 - 2x_3 - x_2 \end{pmatrix}. \tag{1.8}$$

1.2.1 Mode tangent

L'étape suivante est de calculer le gradient $\nabla F_i(x)$ pour les m composantes avec :

$$\nabla F_i(x) = (\partial_{x_i} F_i(x), \ldots, \partial_{x_n} F_i(x))^{\mathrm{T}}. \tag{1.9}$$

Pour le mode tangent, on part des variables d'entrée puis on parcourt le graphique pour calculer le gradient de chaque variable intermédiaire et en déduire le résultat.
On pose pour les variables d'entrée $\nabla w_i = e_i$ où e_i est la i-ème composante de la base canonique de $\mathbb{R}^n$.

On parcourt ensuite le graphique en calculant pour $i > n$

$$\nabla w_i \quad = \quad \sum_{j<i} D_{i,j} \nabla w_j \,, \tag{1.10}$$

où $D_{i,j} = \frac{\partial \phi_i}{\partial w_j}$.

Avec cette méthode on voit donc que le graphique doit être parcouru pour chaque colonne de Jacobien, c'est à dire n fois. Le coût de calcul est donc proportionnel au nombre de variable indépendante en entrée.

1.2.2 Mode rétrograde

Pour le mode rétrograde on part du résultat final et on parcourt le graphique dans le sens inverse jusqu'à obtenir les dérivées par rapport aux variables d'entrées.

On part des variables $\bar{y}_i = \lambda_i$ avec $i = 1, \ldots, m$ puis on calcule les variables intermédiaires :

$$\bar{w}_i = \sum_{j>i} D_{j,i} \bar{w}_j. \tag{1.11}$$

Cependant pour utiliser le mode rétrograde, il faut dans un premier temps construire le graphique de calcul pour chaque variable de sortie, c'est à dire m fois. En effet il faut connaître les dépendances entre les différents nœuds.

2 Implémentation de la méthode des flots tangent

2.1 Choix de l'exemple

Avant tout développement sur le code on précise sur quel type de *payoff* on a appliqué cette méthode, ce qui éclairera le lecteur sur les choix fait dans l'implémentation.

2.1.1 Une option *basket*

Nous avons choisi de calculer les grecques d'une option *basket*. C'est-à-dire un payoff de la forme $(I_T - K)^+$ et donc le prix à $t = 0$,

$$P_0 \;=\; \mathbb{E}_{\mathbb{Q}}\left[e^{-rT}(I_T \;-\; K)^+\right], \tag{2.1}$$

où I_t est un indice sur prix défini par, $\forall t \in [0,T]$,

$$I_t \;=\; a_1 S_t^1 \;+\; \ldots \;+\; a_d S_t^d\,, \tag{2.2}$$

avec évidemment $\sum_{i=1}^d a_i = 1$. Dans notre cas on a choisi un modèle Black-Scholes multidimensionnel pour l'évolution des d actifs, *i.e.* pour tout $1 \leq i \leq d$, $(S_t^i)_{t\in[0,T]}$ suit l'EDS

$$\mathrm{d}S_t^i \;=\; rS_t^i \;+\; \sigma_i S_t^i\,\mathrm{d}W_t^i\,, \qquad S_0^t = s_0^i \in \mathbb{R}. \tag{2.3}$$

Où r est le taux d'intérêt, σ_i la volatilité de chaque actif, le processus $(W_t)_{t\in[0,T]}$ est un mouvement brownien multidimentionnel sur $\mathbb{R}^d$ de matrice de corrélation $\rho \in \mathcal{M}_d([-1,1])$ tel que

$$\mathrm{d}\langle W^i, W^j\rangle_t \;=\; \rho_{ij}\,\mathrm{d}t. \tag{2.4}$$

2.1.2 Les paramètres par défaut

On a pris par défaut les paramètres suivants,

$$\begin{cases} d &=& 10 & & \\ T &=& 1 & & \\ r &=& 0.1 & & \\ s_0^i &=& 100 & \forall i & \\ \sigma_i &=& 0.3 & \forall i & \\ \rho_{ij} &=& 0.5 & \text{pour} & i \neq j \\ a_i &=& \frac{1}{d} & \forall i & \end{cases} \tag{2.5}$$

Ces valeurs sont donc celles par défaut lors de la construction d'une classe (voir Section 2.2.1) mais elles sont éventuellement modifiables.

2.2 Architecture du code

2.2.1 Les classes

On a choisi d'implémenter cette technique celon trois classes :
— `Brownian`, qui contient le moteur de génération de nombre (pseudo) aléatoire.
— `TAD`, pour *Tangent Automatic Differentiation*, qui contient tout ce que l'on a besoin pour appliquer les maths de la section précédente.
— `MC`, pour Monte Carlo, une classe générique (pas liée aux deux précédentes) qui fait une simulation de Monte Carlo.

`Brownian.` Cette classe appelle donc le générateur de nombre aléatoire et simule le processus $(W_t)_t$ uniquement à l'arrivée, *i.e.* au temps T. Le but est alors de générer un vecteur Gaussien centré de taille d et de matrice de variance covariance Σ. Grâce à la librairie `random` de C++11 on peut facilement générer $Z \overset{\text{loi}}{=} \mathcal{N}(0, I_d)$. Puis le *package* d'algèbre linéaire `Armadillo` [4] qui nous permet entre autre de faire une décomposition de Cholesky $\Sigma = LL^{\mathrm{T}}$, où L est une matrice triangulaire inférieure. Et finalement on a bien

$$LZ \overset{\text{loi}}{=} \mathcal{N}(0, \Sigma). \tag{2.6}$$

Il était aussi important pour nous d'avoir une classe dédiée à ce brownien, que l'on pourra appeler par la suite sans avoir à reconstruire le moteur de génération de nombres aléatoires.

`TAD.` Cette classe contient tous les éléments permettant de faire les calculs de *Tangent Automatic Differentiation* sur une option *basket*. Les attributs de la classe sont donc tous les éléments déterminants de l'option, *i.e.* la maturité, les volatilités, le *strike*, *etc.* Pour construire un objet on a préféré entrer le chemin vers un fichier texte qui génère la matrice ou le vecteur en question avec `Armadillo`. De cette façon on a pas besoin de recompiler pour changer un paramètre. Évidement cette classe a besoin de la classe précédente pour générer le brownien ; on rappelle que la solution de l'EDS (2.3) est, pour tout $1 \leq i \leq d$, $t \in [0, T]$,

$$S_t^i = s_0^i \exp\left(\left(r - \frac{\sigma_i^2}{2}\right) T + \sigma_i W_t^i\right). \tag{2.7}$$

On applique alors simplement une décomposition du calcul du prix de l'option, en un séquence avec des étapes très simples. On a choisis de pouvoir

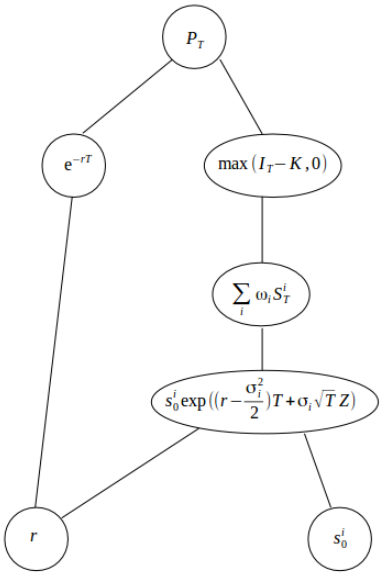

FIGURE 2 – Graphe correspondant aux instructions du code de la classe `TAD`.

dériver par rapport à r et s_0^i $\forall i$. Et donc calculer n'importe quelle combinaison linéaire de ces dérivées. Le graphe de la Figure 2 résume ces étapes élémentaires comme nous les avons définies.

`MC.` Pour cette classe on a voulu créer un Monte Carlo qui ne dépend d'aucun paramètre de l'option ; seule une fonction et une précision souhaitée permet de créer la classe. Cette fonction doit renvoyer un `double` [1], on peut donc appliquer la simulation de Monte Carlo dans n'importe quelle situation. Enfin la précision spécifie l'intervalle à 95% souhaité par l'utilisateur. Une première boucle rapide [2] permet d'approximer la variance de la variable aléatoire (avec l'estimateur biaisé classique des moments), que l'on va noter $\bar{V}$; puis le nombre de simulations du Monte Carlo M est obtenu avec

$$M \quad = \quad 4q_\alpha \frac{\bar{V}}{p^2}\,, \tag{2.8}$$

où p est la précision [3], et $q_\alpha = 2$ pour que l'intervalle de confiance soit à 95%.

1. Donc du type `function<double()>`.
2. De 1000 itérations.
3. Si on a une précision de 0.1, l'intervalle à 95% sera $[m - 0.05, m + 0.05]$.

2.2.2 Les fonctions

La fonction centrale est celle qui calcule une combinaison linéaire des dérivés de l'option par rapport à r ou s_0^i. On l'a appelée `TAD::generate` et elle prend en paramètre un vecteur de taille $d+1$, qui correspond aux poids de la combinaison linéaire [4].

On a aussi implémenté une fonction `TAD::bump` qui de même calcule une combinaison linéaire des dérivés par la méthode du *bump*, elle prend aussi un vecteur en paramètre. On l'utilisera plus tard dans la vérification numérique de nos résultats et pour la réduction de variance.

Enfin la fonction importante est celle qui lance la simulation de Monte Carlo, `MC::compute`.

2.2.3 Les paramètres

Comme on l'a vu précédemment pour spécifier une matrice, *e.g.* la matrice de variance covariance du brownien, on préfère entrer le chemin vers un fichier texte qui contient cette matrice et qui se trouve dans un dossier `data`. Ensuite grâce au *package* `Armadillo` on peut facilement charger la matrice.

Pour créer un objet de la classe `MC` il faut entrer une fonction qui retourne un `double`, on peut donc utiliser la fonction `bind` pour spécifier des paramètres à la fontion d'origine. Par exemple admettons que l'on ait créé un objet `TAD`, que l'on a nommé `T` et `a` est un vecteur de poids. La fonction `T.generate(a)` donne donc une simulation de la combinaison linéaire souhaitée. Pour créer une variable à partir de cette fonction on écrit donc

```
function<double()> f = bind(&TAD::generate, T, a);
```

2.3 Vérification numérique

2.3.1 Formule fermée

Dans un premier temps on peut voir qu'avec les paramètres de notre option, *i.e.* ceux définis en (2.5) on peut avoir l'intuition que le prix sera proche de celle d'une option vanille classique [5]. On a donc aussi des formules

4. Le premier élément correspond au poids sur r puis dans l'ordre sur tous les s_0^i.

5. On ne justifie pas ce point mais l'on considère que c'est une assez bonne approximation pour savoir si nous sommes dans les bons ordres de grandeur.

fermées pour les grecques, pour un *call* sur une option vanille :

$$\frac{\partial C}{\partial r} = KTe^{-rT}\mathcal{N}(d_2)\ ; \tag{2.9}$$

$$\frac{\partial C}{\partial s_0} = \mathcal{N}(d_1). \tag{2.10}$$

L'application donne pour $K = 100$, $T = 1$, $\sigma = 0.3$, $s_0 = 100$, $r = 0.1$, les résultats résumés à la Table 1. Où pour Rho (respectivement Delta) on a le poids $a \in \mathbb{R}^d$ avec $a_1 = 1$ (respectivement $a_1 = 0$) et $a_i = 0$ (respectivement $a_i = 1$) pour $2 \leq i \leq d$.

Comme on pouvait s'y attendre on ne tombe pas exactement sur les valeurs de la formule fermée, mais les ordres de grandeurs sont bons, on peut donc continuer à vérifier numériquement la sortie de notre code.

2.3.2 Méthode *bump*

Pour savoir plus précisément si les résultats de notre programme sont bons, on implémente la méthode du *bump*. On l'a implémentée de sorte que la méthode renvoie aussi une combinaison linéaire des dérivées possibles. On va donc tester sur Rho : avec un poids tel que $a_1 = 1$ et $a_i = 0$ pour $2 \leq i \leq d$; et sur la dérivée par rapport à s_0^1 *i.e.* $a_2 = 1$ et $a_i = 0$ pour $i \neq 2$. Les résultats sont résumés à la Table 2.

Cette méthode permet donc de valider notre code. Mais on va aussi l'utiliser réduire la variance d'une nouvelle variable aléatoire (par rapport à celle de TAD), pour soit gagner du temps, soit être plus précis pour un même coût.

Méthode	**Rho**	**Delta**
Formule fermée vanille	51.823	0.686
Monte Carlo TAD	57.1678	0.711934

TABLE 1 – Résultat de la formule fermée et de la simulation de Monte Carlo avec une précision de 0.1 pour Rho et 0.001 pour Delta.

2.4 Réduction de variance

2.4.1 Comportement avant réduction de variance

On va dans un premier temps regarder comment se comporte nos simulations de Monte Carlo. En fonction de la précision souhaitée [6] notre code en déduit le nombre de simulations nécessaire pour que l'estimation se trouve dans l'intervalle de confiance à 95%. On a tracé à la Figure 3 le comportement de la simulation et de son intervalle de confiance en fonctions du nombre de simulation, et ce pour $\frac{\partial C}{\partial s_0^i}$.

On peut voir qu'avec la méthode *bump* on a la même performance dans cette étude rapide, *i.e.* pour une même précision on aura besoin d'environ autant de simulations qu'avec la différentiation automatique pour parvenir au même intervalle de confiance. C'est aussi ce qu'avait remarqué L. Capriotti dans son article.

2.4.2 Variable de contrôle optimale

La théorie. Dans notre cas on a deux variables aléatoires X et X' (respectivement par la méthode de différentiation automatique et la méthode *bump*) qui estiment une certaine combinaison linéaire des dérivées précédemment cités. Ces deux v.a. ne sont pas identiques [7] dans le sens où $\mathbb{P}\{X \neq X'\} > 0$. On a alors vu en cours qu'en définissant

$$X^\lambda \;:=\; X \;-\; \lambda\Xi\,, \tag{2.11}$$

où $\Xi := X - X'$, on peut trouver un $\lambda_{\min}$ tel que

$$\mathbb{V}\mathrm{ar}\left[X^{\lambda_{\min}}\right] \;\leq\; \min\left(\mathbb{V}\mathrm{ar}\left[X\right], \mathbb{V}\mathrm{ar}\left[X'\right]\right). \tag{2.12}$$

6. On rappelle que l'utilisateur spécifie la précision pour construire un objet de la classe `MC`.

7. Puisque calculées avec deux méthodes différentes.

Méthode	**Rho**	$\boldsymbol{\frac{\partial C}{\partial s_0^1}}$
Bump	57.2363	0.0717528
TAD	57.1938	0.0717492

Table 2 – Résultat la simulation de Monte Carlo pour deux méthodes avec une précision de 0.1 pour Rho et 0.001 pour $\frac{\partial C}{\partial s_0^1}$.

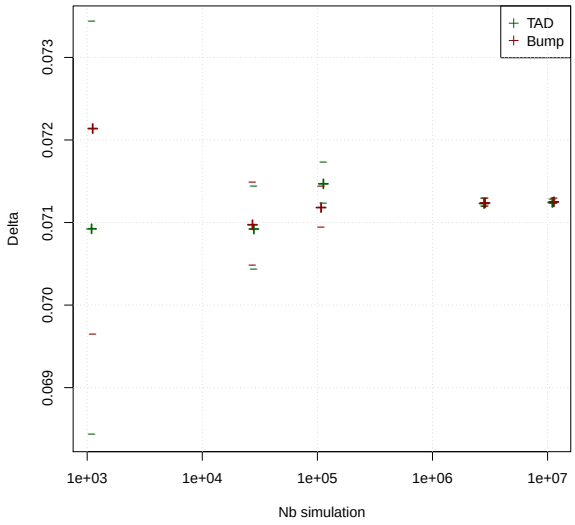

FIGURE 3 – Comportement de la simulation et de l'intervalle de confiance en fonction du nombre de simulation (échelle logarithmique).

Et on a

$$\lambda_{\min} \quad = \quad \frac{\mathbb{C}\mathrm{ov}\left(X, \Xi\right)}{\mathbb{V}\mathrm{ar}\left[\Xi\right]}. \tag{2.13}$$

En pratique. Dans notre code, le constructeur de `MC` prend en fait un troisième argument qui est une fonction `double`[8], la classe Monte Carlo a donc accès à une deuxième fonction (évidemment différente de la première) pour générer la deuxième variable aléatoire X'. On a ensuite besoin de trouver $\lambda_{\min}$, pour cela on fait une boucle rapide de $M = 1000$ itérations et on

8. Si cet argument est laissé vide, par défaut c'est une fonction lambda qui renvoi 0.

estime λ_M de la façon suivante :

$$V_M := \frac{1}{M}\sum_{k=1}^{M}\Xi_k^2 \; ; \tag{2.14}$$

$$C_M := \frac{1}{M}\sum_{k=1}^{M}X_k\Xi_k^2 \; ; \tag{2.15}$$

$$\lambda_M := \frac{C_M}{V_M}. \tag{2.16}$$

On peut ensuite effectuer la simulation de Monte Carlo sur X^{λ_M}, avec une boucle plus grande de N itérations

$$X_N^{\lambda_M} = \frac{1}{N}\sum_{k=1}^{N}\left((1-\lambda_M)X_k + \lambda_M X_k'\right) \tag{2.17}$$

Sur notre simulation on a vu à la Figure 3 qu'il faut le même nombre de simulation suivant les deux méthodes ; suggérant que $\mathbb{V}\mathrm{ar}\left[X\right] \approx \mathbb{V}\mathrm{ar}\left[X'\right]$. On peut alors immédiatement en déduire que $\lambda_{\min} = \frac{1}{2}$ et

$$\mathbb{V}\mathrm{ar}\left[X^{\lambda_{\min}}\right] \approx \mathbb{V}\mathrm{ar}\left[X\right] \tag{2.18}$$

$$\approx \mathbb{V}\mathrm{ar}\left[X'\right]. \tag{2.19}$$

C'est effectivement ce que l'on trouve en faisant les simulations.

Dans ce cas très précise la réduction de variance ne sert donc à rien, mais il suffirait à l'utilisateur d'implémenter une nouvelle méthode de calcul de grecque telle que $\mathbb{V}\mathrm{ar}\left[X\right] \neq \mathbb{V}\mathrm{ar}\left[X'\right]$ pour que la réduction de variance par variable de contrôle optimale ait une utilité.

Références

[1] Capriotti, L. Fast Greeks by algorithmic differentiation. *The Journal of Computational Finance 14* (2011), 3–35.

[2] Capriotti, L., and Giles, M. Fast Correlation Greeks by Adjoint Algorithmic Differentiation.

[3] Capriotti, L., and Giles, M. Algorithmic Differentiation : Adjoint Greeks Made Easy.

[4] Sanderson, C., and Curtin, R. Armadillo : a template-based C++ library for linear algebra. *Journal of Open Source Software 1* (2016), 26.