



Pricer efficace d'actifs énergétiques

Team 16 :

Anthony COURILLON

Denis BARRANCO

Ousman ASADULLAH

Hicham KORTBI

Partenaire :

Arnaud DE LATOUR

Mentor :

Martino GRASSELLI

Mars 2017

Introduction

Etant étudiants en quatrième année à l'ESILV dans la majeure finance et dans le cadre du projet PI2, nous avons décidé de travailler sur la réalisation d'un pricer d'actifs énergétiques proposé par notre partenaire EDF R&D et encadré par Arnaud DE LATOUR

Un certain nombre d'actifs utilisés dans le domaine de l'énergie (par exemple les centrales électriques) se prête facilement à une valorisation sous la forme de produits dérivés complexes, de nature américaine (contrairement à une option européenne, l'option américaine peut être exercée n'importe quand).

Le pricing de ces actifs met en oeuvre des méthodes numériques avancées qui peuvent être coûteuses en temps de calcul. Lorsque ces méthodes sont fondées sur une simulation Monte-Carlo (par exemple l'algorithme de Longstaff-Schwartz), on a intérêt à employer des techniques de réduction de variance pour accélérer la convergence de l'algorithme - et donc réduire le temps de calcul.

Cependant, l'efficacité de ces techniques dépend généralement du payoff de l'actif qu'on cherche à valoriser. L'objectif de ce projet est donc d'implémenter et de tester différentes méthodes de réduction de variance appliquées à des actifs énergétiques, afin d'identifier les méthodes les plus pertinentes.

Nous allons réaliser ce pricer sous Python et nous allons utiliser exclusivement ces 3 logiciels afin de réaliser ce projet :

- Anaconda ;
- LiClipse ;
- Github.

Chapitre 1

Evaluation du prix d'un produit dérivé par simulation

1.1 Principes des méthodes de Monte Carlo

Le prix d'un produit dérivé d'un sous-jacent $X = (X_t)_{t \geq 0}$ de payoff actualisé $f(X)$, d'un marché financier sous probabilité risque neutre, s'écrit comme l'espérance des flux futurs actualisés qu'il génère, e_p :

$$e_p = \mathbb{E}[f(X)] \quad (1.1)$$

Il faut donc évaluer cette espérance pour connaître le prix d'un produit dérivé. Nous allons utiliser les méthodes de Monte Carlo qui consiste à simuler un grand nombre de réalisations, N indépendantes et identiquement distribuées, de $f(X)$, $f(X^i)$ avec $i \in \mathbb{N}$, puis on calcule chaque $f(X^i)$.

Enfin, on prend la moyenne empirique de ces réalisations, \bar{e}_N estimateur de e_p :

$$\bar{e}_N = \frac{1}{N} \sum_{i=1}^N f(X^i) \xrightarrow[N \rightarrow \infty]{\text{p.s.}} e_p \text{ avec } \mathbb{E}[|f(X)|] < \infty \quad (1.2)$$

la loi des grands nombres assure la convergence de \bar{e}_N vers e_p . Autrement dit, pour N assez grand, $\bar{e}_N \simeq e_p$.

1.2 Limite pratique des méthodes de Monte Carlo

Théorème 1.2.1 (Théorème Central Limite). *Nous allons utiliser le cas où on ne connaît pas la loi de e_p . Si la loi de e_p est inconnue (ou non normale), alors $e_N \hookrightarrow N(\mu, \frac{\sigma}{\sqrt{N}})$ pour N assez grand, avec $\sigma = \sqrt{\text{Var}(f(X))}$ et $\mu = e_p = \mathbb{E}[f(X)]$ alors on a :*

$$\frac{\sqrt{N}(\bar{X}_n - \mu)}{\sigma} \hookrightarrow N(0, 1) \quad (1.3)$$

Donc le théorème central limite permet d'avoir l'approximation de l'erreur réalisée :

$$\sqrt{N}(\frac{1}{N} \sum_{i=1}^N f(X^i) - e_p) \hookrightarrow N(0, \sqrt{\text{Var}(f(X))}) \quad (1.4)$$

ou :

$$\sqrt{N}(e_N - e_p) \hookrightarrow N(0, \sqrt{\text{Var}(f(X))}) \quad (1.5)$$

Cela montre le problème du nombre de simulations N nécessaires pour avoir un bon estimateur de e_p , plus la variance de $f(X^i)$ est élevée, plus N doit être grand.

Or N grand a pour conséquence une augmentation du temps de calcul. Donc plus $\text{Var}(f(X^i))$, plus l'obtention de l'évaluation de e_p demandera un temps de calcul important.

Il est alors intéressant d'employer des techniques de réduction de variance pour accélérer la convergence de l'algorithme et donc réduire le temps de calcul. C'est l'un des problèmes que nous allons tenter de résoudre dans la réalisation de notre pricer.

Nous allons voir les différentes techniques de réduction de variance et tenter de les implémenter dans notre pricer dans le chapitre 2 de ce document.

1.3 Contrôle de la convergence

Il faut évaluer l'erreur d'approximation de (1.4) grâce à la construction d'un intervalle de confiance :

$$Z = \frac{\sqrt{N}}{\sqrt{\text{Var}(f(X))}} (\frac{1}{N} \sum_{i=1}^N f(X^i) - e_p) \hookrightarrow N(0, 1) \quad (1.6)$$

En supposant que la variance est connue, on a :

$$\mathbb{P}(-u \leq Z \leq u) = \alpha \quad (1.7)$$

avec u fractile d'ordre $1-\frac{\alpha}{2}$ de la $N(0,1)$. Ce qui revient à :

$$\mathbb{P}\left(-u \frac{\sqrt{\text{Var}(f(X))}}{\sqrt{N}} + e_N^- \leq e_p \leq u \frac{\sqrt{\text{Var}(f(X))}}{\sqrt{N}} + e_N^-\right) = \alpha \quad (1.8)$$

et donc :

$$IC_\alpha(e_p) = \left[-u \frac{\sqrt{\text{Var}(f(X))}}{\sqrt{N}} + e_N^-; u \frac{\sqrt{\text{Var}(f(X))}}{\sqrt{N}} + e_N^-\right] \quad (1.9)$$

L'erreur donnée par un intervalle de confiance est :

$$\mathbb{P}(e_p \in IC_\alpha) \simeq \alpha \quad (1.10)$$

Il a un problème c'est qu'on ne peut appliquer cette méthode que si la variance de $f(X)$, $\text{Var}(f(X))$ est connue. Or ici, $\text{Var}(f(X))$ est inconnue, et il faut l'estimer grâce à un estimateur.

On peut construire un estimateur non biaisé tel que celui-ci :

$$v_N^- = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (f(X^i) - e_N^-)^2} \quad (1.11)$$

Voici l'implémentation sous Python de tout ce que nous avons énoncé dans cette partie I, on a pris $\alpha = 0.95$ pour la construction des intervalles de confiance.

```

1 class GaussianGenerator:
2     "Générateur de bruits gaussiens corrélés."
3
4     def __init__(self, nsims, corrmatrix, corrkeys, randomfunc):
5         """
6         Initialise une nouvelle instance de la classe 'GaussianGenerator'
7
8         Paramètres :
9         _____
10        nsims : entier positif
11        Nombre de simulations à générer.
```

```

12 corrmatrix : matrice carrée
13 Matrice de corrélation entre les différents bruits gaussiens
14 à simuler. La matrice doit être définie positive.
15 corkeys
16 Liste des identifiants associés aux différentes lignes /
   colonnes
17 de la matrice de corrélation 'corrmatrix'. Les identifiants
   doivent
18 être donnés dans l'ordre dans lequel les bruits correspondants
19 apparaissent dans 'corrmatrix'.
20 randomfunc
21 Fonction permettant de générer des bruits gaussiens indépendants
22 (typiquement 'np.random.randn').
23 """
24 self.corrmatrix = corrmatrix
25 self.corkeys = corkeys
26 self.nsim = nsims
27 self.randomfunc = randomfunc
28 self.cache = DateCache()
29 try:
30     np.linalg.cholesky(self.corrmatrix)
31 except np.linalg.LinAlgError:
32     raise np.linalg.LinAlgError("the correlation matrix is not "\
33     "positive definite.") from None
34
35 @property
36 def nnoises(self):
37     "Nombre de bruits gaussiens distincts à simuler."
38     return self.corrmatrix.shape[0]
39
40 @timecached
41 def getallnoises(self, date):
42     """
43     Renvoie 'self.nsim' réalisations de 'self.nnoises' bruits
44     gaussiens corrélés.
45     """
46     whitenois = self.randomfunc(self.nnoises, self.nsim)
47     noises = np.dot(np.linalg.cholesky(self.corrmatrix), whitenois)
48     return noises
49
50 @timecached
51 def getnoises(self, date, keys):
52     """
53     Renvoie un tableau de taille '(len(keys), self.nsim)' de bruits

```

```
54 gaussiens corrélés correspondants aux aléas identifiés par les
55 clefs 'keys'.
56 """
57 noises = self.getallnoises(date)
58 res = np.empty((len(keys), self.nsim))
59 for idx, key in enumerate(keys):
60     keyidx = self.corrkeys.index(key)
61     res[idx, :] = noises[keyidx, :]
62 return res
```

Chapitre 2

Méthodes de réduction de la variance

2.1 Variables antithétiques

L'idée des variables antithétiques repose sur la symétrie de la distribution de la loi uniforme et la corrélation entre 2 variables aléatoires.

2.1.1 Idées globales

Calculer $\theta = E[Y] = E[f(X)]$
 $E[Y] = \frac{1}{2}E[Y_1] + E[Y_2] = E\left[\frac{Y_1 + Y_2}{2}\right]$
 $Var\left(\frac{Y_1 + Y_2}{2}\right) = \frac{Var(Y_1) + Var(Y_2) + 2Cov(Y_1, Y_2)}{4}$
- Si Y_1 et Y_2 sont indépendantes, $Cov(Y_1, Y_2) = 0$
- Si $Cov(Y_1, Y_2) < 0$, alors $Var\left(\frac{Y_1 + Y_2}{2}\right) < Var\left(\frac{Y}{2}\right)$ donc la variance est réduite si Y_1 et Y_2 sont corrélés négativement, $Cov(Y_1, Y_2) < 0$

2.1.2 Mathématiquement

On veut estimer $E[Y]$ en utilisant l'implémentation d'échantillons antithétiques c'est-à-dire : $(Y_1, Y_{1ant}), (Y_2, Y_{2ant}), \dots, (Y_N, Y_{Nant})$ avec $\forall i \in \mathbb{N}$, Y_i et Y_{iant} de même loi de distribution et i.i.d. (indépendantes et identiquement distribuées).

L'estimateur de $E[Y]$ est simplement la moyenne des $2N$ réalisations :

$$e_{Nant}^- = \frac{1}{2N} (\sum_{i=1}^N Y_i + \sum_{i=1}^N Y_{iant}) = \frac{1}{N} \sum_{i=1}^N \left(\frac{Y_i + Y_{iant}}{2} \right)$$

En utilisant les mêmes méthodes que dans le chapitre I, on par TCL :

$$\frac{\sqrt{N}(\bar{e}_{Nant} - E[Y])}{\sqrt{Var(Y)}}$$

$$\text{On pose } \sigma_{ant}^2 = Var(Y) = Var\left(\frac{Y_i + Y_{iant}}{2}\right)$$

Comme nous venons de le voir dans les idées globales :

$$\begin{aligned} \sigma_{ant}^2 &= \frac{Var(Y_i) + Var(Y_{iant}) + 2Cov(Y_i, Y_{iant})}{4} \\ &= \frac{2Var(Y_i) + 2Cov(Y_i, Y_{iant})}{4} \text{ car } Y_i \text{ et } Y_{iant} \text{ sont de même loi de distribution} \\ &= \frac{Var(Y_i) + 2Cov(Y_i, Y_{iant})}{2} \end{aligned}$$

$$\text{Alors, si } Cov(Y_i, Y_{iant}) < 0, \sigma^2 < \frac{Var(Y_i)}{2}$$

Donc nous avons besoin de construire les paires Y_i et Y_{iant} de façon à ce que $Cov(Y_i, Y_{iant}) < 0$ pour réduire la variance.

2.1.3 Implémentation

Comme nous simulons une matrice de bruits gaussiens indépendants, on peut implémenter la méthode des variables antithétiques en construisant une séquence Y_1, Y_2, \dots, Y_N (rappel : $f(X^i)$ avec $i \in \mathbb{N}$) puis construire une autre séquence opposé à la première en prenant la négation de celle-ci : $-Y_1, -Y_2, \dots, -Y_N$.

Pseudo-code :

1. Créer une matrice vide de taille (nnoises, nsims) avec
nnoises : nombre de bruits gaussiens indépendants
nsims : nombre de simulations

2. Générer dans la moitié de la matrice des bruits gaussiens.
3. Prendre l'opposé de la première moitié de la matrice, pour remplir la deuxième moitié de la matrice.

Voici l'implémentation sous Python de la méthode des variables antithétiques :

```

1 def antithetic_randn(nnoises , nsims):
2     """
3     Renvoie un tableau de bruits gaussiens non corrélés :math: '(G_{\{
4         \_i,j\}})'
5     de taille '(nnoises , nsims)', et tel que, pour tout :math: 'i' :
6     :math: '\forall 1 \leq j \leq n / 2, G_{\{i,n/2+j\}} = -G_{\{i,j\}}'
7     Paramètres :
8     -----
9     nnoises : entier positif
10    Nombre de bruits à simuler.
11    nsims : entier positif, pair
12    Nombre de simulations à effectuer par prix.
13    """
14    if nsims % 2 != 0:
15        raise ValueError("the number of simulations used with antithetic
16            "\
17            "variables should be even.")
18    half = int(0.5 * nsims)
19    noises = np.empty((nnoises , nsims))
20    noises[:, :half] = np.random.randn(nnoises , half)
21    noises[:, half:] = -noises[:, :half]
22    return noises

```

2.2 Suites à discrédance faible (Quasi Monte Carlo)

Nous allons étudier les méthodes de Quasi Monte Carlo qui contrairement au Monte Carlo classique n'est pas basé sur la génération de nombres pseudo-aléatoires mais sur la génération de nombre déterministe aux propriétés spéciales générés par des suites à discrédance faible.// On peut voir le problème de l'évaluation du prix d'un produit dérivé vu dans le chapitre I (1) comme une intégrale à la place d'une espérance :

$$e_p^{QMC} = \int_{[0,1]^s} f(u) du = \frac{1}{N} \sum_{i=1}^N f(X^i)$$

On cherche à approximer l'intégrale.

Pour cela, il faut générer les points X^i dans l'hypercube $[0, 1]^s$ avec s la dimension de l'espace. Ces points doivent être équiréparties. On ne se souciait pas vraiment de la dimension jusqu'à présent mais plus la dimension sera grande, plus il y aura des irrégularités dans les projections.

Nous allons voir les différentes suites à discrédance faible permettant de générer ces points.

Discrédance et propriétés d'équirépartition à détailler.(a faire)

2.2.1 Suite de Van Der Corput

La suite de Van der Corput en base b est de la forme :

$$\phi_b(n) = \frac{a_0}{b} + \frac{a_1}{b^2} + \dots + \frac{a_r}{b^{r+1}}$$

avec $n = a_0 + a_1 + \dots + a_r b^r$, $a_r > 0$, $0 \leq a_i \leq b-1$, $0 \leq i \leq r$

ou

$$\phi_b : \mathbb{N} \rightarrow [0, 1]$$

$$\forall b \in \mathbb{N}, \phi_b(n) = \sum_{i=0}^{\infty} \frac{a_i(n, b)}{b^{i+1}}$$

Nous allons voir les suites de Van Der Corput avec deux exemples ayant chacun leur approche. C'est important de bien comprendre cette suite car elle est la clé de la compréhension des suites qui vont suivre.

Exemple 1 : l'approche classique

Nous allons prendre un exemple d'une suite de Van Der Corput en base 2 distribué sur l'intervalle $[0,1)$.

Pour $n=12$, les nombres générés sont : $0, \frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8}, \frac{1}{16}, \frac{9}{16}, \frac{5}{16}, \frac{13}{16}$

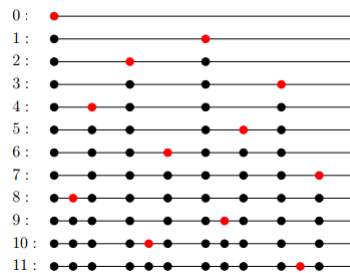


FIGURE 2.1 – Suite de Van Der Corput en base 2 pour $n=12$

Les nombres générés séquentiellement remplissent les larges espaces entre les nombres précédents de la séquence.

En prenant la logique de la construction d'une suite de Van Der Corput, nous avons implémenté sous Python les fonctions permettant de générer la séquence :

```
1 def vdc(n, base):
2     """
3     Cette fonction permet de calculer le n-ieme nombre de la base b
4     de la
5     séquence de Van Der Corput
6     """
7     vdc, denom = 0,1
8     while n:
9         denom *= base
10        n, remainder = divmod(n, base)
11        vdc += remainder / denom
12    return vdc
```

```

1 def van_der_corput(nsims,b):
2     """
3     Cette fonction permet de générer la séquence de Van Der Corput
4     en base b
5     """
6     array=np.empty(nsims)
7     i=0
8     for i in range(nsims):
9         array[i]=vdc(i,b)
10    return array

```

On a testé ces fonctions et on a tracé ceci :

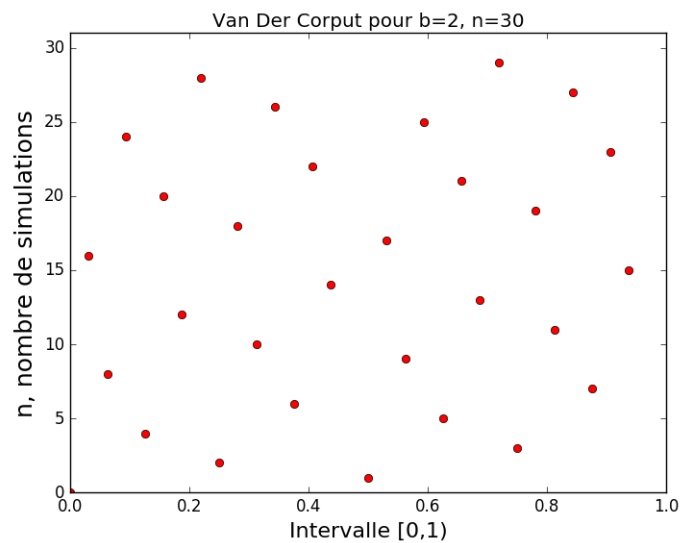


FIGURE 2.2 – Suite de Van Der Corput en base 2 pour n=30

On voit bien sur la figure 2.2 que les points générés par la séquence de Van Der Corput sont plutôt bien équiréparties.

Exemple 2 :l'approche binaire

On peut calculer le n-ième terme de la suite de Van Der Corput de base b comme ceci : (à expliquer plus en détails)

n	$b = 2$	$\phi_b(n)$
0	$(0)_2$	$0 \times 2^{-1} = 0,0000$
1	$(1)_2$	$1 \times 2^{-1} = 0,5000$
2	$(10)_2$	$1 \times 2^{-2} + 0 \times 2^{-1} = 0,2500$
3	$(11)_2$	$1 \times 2^{-2} + 0 \times 2^{-1} = 0,7500$
4	$(100)_2$	$1 \times 2^{-3} + 0 \times 2^{-2} + 0 \times 2^{-1} = 0,1250$
5	$(101)_2$	$1 \times 2^{-3} + 0 \times 2^{-2} + 1 \times 2^{-1} = 0,6250$
6	$(110)_2$	$1 \times 2^{-3} + 1 \times 2^{-2} + 0 \times 2^{-1} = 0,3750$
7	$(111)_2$	$1 \times 2^{-3} + 1 \times 2^{-2} + 1 \times 2^{-1} = 0,8750$
8	$(1000)_2$	$1 \times 2^{-4} + 0 \times 2^{-3} + 0 \times 2^{-2} + 0 \times 2^{-1} = 0,0625$

FIGURE 2.3 – Suite de Van Der Corput en base 2 pour n=30

Implémentation de la suite de Van Der Corput

Voici l'implémentation sous Python.

remarque : comme la suite de Van Der Corput génère des points dans l'intervalle $[0,1)$, il a fallu modifier la fonction vdc (sans cette modification, on n'obtenait pas des résultats ce rapprochant du Monte Carlo classique) en prenant l'inverse de la CDF(cumulative distribution function), à vérifier

```

1  def vdc(n, base):
2      """
3      Cette fonction permet de calculer le n-ieme nombre de la
4      base b de la
5      séquence de Van Der Corput
6      """
7      vdc, denom = 0,1
8      while n:
9          denom *= base
10         n, remainder = divmod(n, base)
11         vdc += remainder / denom
12         return norm.ppf(vdc)
13
14  def van_der_corput(nsims, b):
15      """
16      Cette fonction permet de générer la séquence de Van Der
17      Corput en base b

```

```

16     """
17     array=np.empty( nsims)
18     i=0
19     for i in range( nsims):
20         array[i]=vdc(i,b)
21     return array
22
23     def van_der_corput_dimension(dim, nsims):
24         array=np.empty((dim, nsims))
25         """
26         Cette fonction génère dans un tableau de taille (dim, nsims)
27         toutes les séquences
28         de la suite de Van der Corput de la base 2 à la base dim+2
29         """
30         for i in range(2, dim+2, 1):
31             array[i-2,:]=van_der_corput( nsims, i)
32         return array

```

2.2.2 Suite de Sobol

Explication de la suite de Sobol à faire

```

1  import sobol_seq
2
3  def sobol(nnoises, nsims):
4      """
5      Renvoie un tableau de valeurs générés par la suite de Sobol
6      de taille (nnoises, nsims)
7      """
8      noises = np.empty((nsims))
9      """
10     Utilisation de la fonction sobol_seq.
11     i4_sobol_generate_std_normal
12     pour générer des variables quasi-aléatoires suivant une loi
13     normale
14     """
15     noises=sobol_seq.i4_sobol_generate_std_normal(nnoises, nsims)
16     return (noises.reshape(nnoises, nsims))

```