



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 donné 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, dans le fichier `_init_.py`, on a pris $\alpha = 0.95$ pour la construction des intervalles de confiance.

```

1  "Pricing d'actifs énergétiques par Monte Carlo."
2
3  import numpy as np
4  import pandas as pd
5  import scipy.stats as sps
6  import efficientmc.assets as assets
7  import efficientmc.pricemodels as pricemodels
8  import efficientmc.generators as generators
9  from collections import namedtuple
10
11  MCResults = namedtuple('MCResults', ('mean', 'iclow', 'icup'))
12

```

```

13 def runmc(*allassets):
14     """
15     Calcule par simulation Monte-Carlo les cash-flows des actifs de
16     'allassets', sous la probabilité risque-neutre.
17     """
18     allmarkets = set([m for asset in allassets for m in asset.
19                       getmarkets()])
20     marketsdates = set([d for market in allmarkets for d in market.
21                          getdates()])
22     assetsdates = set([d for asset in allassets for d in asset.
23                        getdates()])
24     earnings = {asset.name: {} for asset in allassets}
25     prices = {market.name: {} for market in allmarkets}
26     volumes = {asset.name: {m.name: {} for m in asset.getmarkets()}}
27     \
28     for asset in allassets}
29     alldates = assetsdates.union(marketsdates)
30     for date in alldates:
31         #FIXME: utiliser une notion de 'computation' pour sauvegarder et
32         # agréger les résultats au fur et à mesure.
33         # Mise à jour des marchés :
34         for market in allmarkets:
35             market.simulate(date)
36         # Mise à jour des objets :
37         for asset in allassets:
38             earnings[asset.name][date] = asset.get_discounted_cf(date)
39             for market in asset.getmarkets():
40                 volumes[asset.name][market.name][date] = asset.getvolume(date,
41                                     market)
42         #FIXME: on peut avoir besoin d'autre chose que du spot.
43         if date not in prices[market.name]:
44             prices[market.name][date] = market.getspot(date)
45         earnings = pd.Panel(earnings).transpose(1, 0, 2)
46         #FIXME: utiliser xarray.
47         volumes = {key: pd.Panel(value).transpose(1, 0, 2)
48                    for key, value in volumes.items()}
49         prices = pd.Panel(prices).transpose(1, 0, 2)
50         return earnings, volumes, prices
51
52 def getmtm(cf, alpha=0.95):
53     """
54     Calcule la MtM (i.e. : les cash-flows moyens réalisés et un
55     intervalle
56     de confiance) de chaque actif listé dans 'cf'.

```

```

53 Paramètres
54 -----
55 cf : pandas.Panel
56 Cash-flows réalisés pour chaque simulation ('items'), chaque
57 actif ('major_axis') et chaque date ('minor_axis').
58 alpha : double compris entre 0. et 1.
59 Quantile à utiliser pour le calcul des intervalles de confiances
60 .
61 """
62 nsims = cf.shape[0]
63 cumvalues = cf.sum(axis=2)
64 mean = cumvalues.mean(axis=1)
65 std = cumvalues.std(axis=1)
66 res = {}
67 for key, val in mean.items():
68     res[key] = mkmcresults(val, std[key], nsims)
69 return res

```

Il reste aussi le problème de la queue de distribution de la véritable loi de e_N (on a supposé précédemment qu'elle suivait une loi normale grâce au TCL) qui n'est pas négligeable, nous verrons cela plus en détails à travers les résultats obtenus de notre pricer.

Il y a par exemple, le prix de l'électricité en forward qui varie extrêmement et ne fluctue plus comme une loi gaussienne. La distribution de ses fluctuations appartient au régime des distributions dites « à queue épaisse ».

Nous verrons cela dans le chapitre 3 et 4 de ce rapport.

1.4 Générateur aléatoire

On a besoin de simuler des variables aléatoires. Or un ordinateur ne génère que de suites déterministes. bruit gaussien : réalisation de processus aléatoire suivant une loi normale) Sous Python, nous allons stocké des bruits gaussiens dans un tableau (matrice). Voici la classe GaussianGenerator dans le fichier generators.py correspondant :

```
1 import numpy as np
2 from efficientmc.utils import timecached, DateCache
3 import sobol_seq
4 import ghalton as gh
5 from scipy.stats import norm
6 from math import ceil, fmod, floor, log
7
8 class GaussianGenerator:
9     "Générateur de bruits gaussiens corrélés."
10
11     def __init__(self, nsims, corrmatrix, corrkeys, randomfunc):
12         """
13         Initialise une nouvelle instance de la classe '
14         GaussianGenerator'.
15
16         Paramètres :
17         _____
18         nsims : entier positif
19         Nombre de simulations à générer.
20         corrmatrix : matrice carrée
21         Matrice de corrélation entre les différents bruits gaussiens
22         à simuler. La matrice doit être définie positive.
23         corrkeys
24         Liste des identifiants associés aux différentes lignes /
25         colonnes
26         de la matrice de corrélation 'corrmatrix'. Les identifiants
27         doivent
28         être donnés dans l'ordre dans lequel les bruits
29         correspondants
30         apparaissent dans 'corrmatrix'.
31         randomfunc
32         Fonction permettant de générer des bruits gaussiens indé
33         pendants
34         (typiquement 'np.random.randn').
35         """
36         self.corrmatrix = corrmatrix
```



```

32     self.corrkeys = corrkeys
33     self.nsimns = nsims
34     self.randomfunc = randomfunc
35     self.cache = DateCache()
36     try:
37         np.linalg.cholesky(self.corrmatrix)
38     except np.linalg.LinAlgError:
39         raise np.linalg.LinAlgError("the correlation matrix is not "
40 \
41     "positive definite.") from None
42
43     @property
44     def nnoises(self):
45         "Nombre de bruits gaussiens distincts à simuler."
46         return self.corrmatrix.shape[0]
47
48     @timecached
49     def getallnoises(self, date):
50         """
51         Renvoie 'self.nsimns' réalisations de 'self.nnoises' bruits
52         gaussiens corrélés.
53         """
54         whitenoises = self.randomfunc(self.nnoises, self.nsimns)
55         noises = np.dot(np.linalg.cholesky(self.corrmatrix),
56 whitenoises)
57         return noises
58
59     @timecached
60     def getnoises(self, date, keys):
61         """
62         Renvoie un tableau de taille '(len(keys), self.nsimns)' de
63         bruits
64         gaussiens corrélés correspondants aux aléas identifiés par
65         les
66         clefs 'keys'.
67         """
68         noises = self.getallnoises(date)
69         res = np.empty((len(keys), self.nsimns))
70         for idx, key in enumerate(keys):
71             keyidx = self.corrkeys.index(key)
72             res[idx, :] = noises[keyidx, :]
73         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 génère une suite de points dans l'intervalle $[0,1]$ qui ne se répète jamais.

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

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

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

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 : $0, \frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8}, \frac{1}{16}, \frac{9}{16}, \frac{5}{16}, \frac{13}{16}$

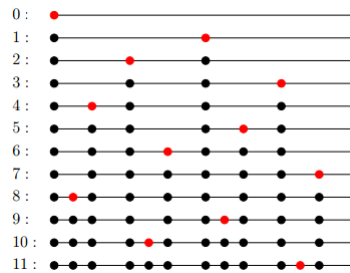


FIGURE 2.1 – Suite de Van Der Corput en base 2 pour $n=12$

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)
```

```

10 vdc += remainder / denom
11 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 :

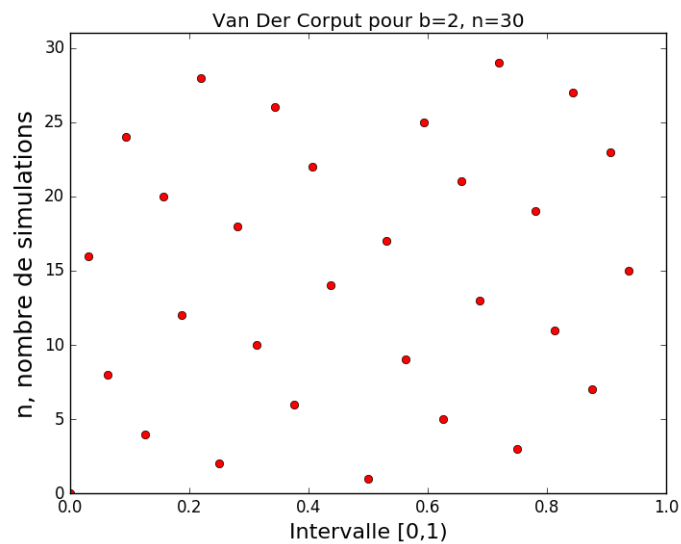


FIGURE 2.2 – Suite de Van Der Corput en base 2 pour n=30

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 : $\phi_b(n) = \sum_{i=0}^I \frac{a_i(n)}{b^{i-1}}$

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

Les $a_i(n)$ sont la réflexion de n en une base b.

$I = \text{int}(\ln(n)/\ln(b))$.

ex :

n=19 et b=3, $19 = 2 \times 3^2 + 0 \times 3^1 + 1 \times 3^0$

$\phi_3(19) = \frac{1}{3} + \frac{0}{9} + \frac{2}{27} = \frac{11}{27}$

Implémentation de la suite de Van Der Corput

Voici l'implémentation sous Python.

```

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     def van_der_corput(nsims,b):
14         """
15         Cette fonction permet de générer la séquence de Van Der
16         Corput en base b
17         """
18         array=np.empty(nsims)
19         i=0
20         for i in range(nsims):
21             array[i]=vdc(i,b)
22         return array
23
24     def van_der_corput_dimension(dim,nsims):
25         array=np.empty((dim,nsims))
26         """
27         Cette fonction génère dans un tableau de taille (dim,nsims)
28         toutes les séquences
29         de la suite de Van der Corput de la base 2 à la base dim+2
30         """
31         for i in range(2,dim+2,1):
32             array[i-2,:]=van_der_corput(nsims, i)
33         return array

```

Discrépance

$$D_N^*(\phi_b(1), \phi_b(2), \dots, \phi_b(N)) = \mathcal{O}\left(\frac{\log N}{N}\right)$$
 avec N=nsims nombres de simulations.

2.2.2 Suite de Halton

La suite de Halton est une extension de la suite de Van der Corput en plusieurs dimensions.

Elle est composée de d 1-dimension suites de Van der Corput en utilisant comme base les premiers nombres premiers.

Un point généré par la suite de Halton est de la forme :

$$x_i = (\phi_{p_1}(i), \phi_{p_2}(i), \dots, \phi_{p_d}(i))$$

avec d la dimension de la suite de Halton et p_1, p_2, \dots, p_d les premiers nombres premiers.

En prenant la logique de la construction de la suite de Halton, nous l'avons implémenté de 2 façons, dont voici la première :

```
1 def haltonF(nnoises, nsims):
2   Prime=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
          53,\
3   59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113,\
4   127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181,\
5   191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251,\
6   257, 263, 269, 271, 277, 281]
7   halton=np.empty((nnoises, nsims))
8   for i in range(0, nsims, 1):
9     for j in range(0, nnoises, 1):
10      prime=Prime[j]
11      halton[j, i]=vdc(i, prime)
12   return halton
```

Elle suit la logique décrite précédemment.

Sous Python, la suite de Halton est déjà existante dans le module ghalton. Voici la deuxième implémentation :

```
1 import ghalton as gh
2 def halton(dim, nsims):
3     """
4     Attention: la fonction crash pour nsims>500, on doit revoir l'
        optimisation de la fonction
5
6     On utilise la librairie Python existante sur la suite de Halton
7
8     GeneralizedHalton produit une suite de nsims dimension (colonnes
9     ),
10    le nombre 68 est utilisé pour faire des permutations, c'est le
        nombre qui permet de se rapprocher
11    le plus des valeurs du Monte Carlo classique
12    """
13    sequence = gh.GeneralizedHalton(nsims, 68)
14    "Une liste de dim sous-listes est produite"
15    points=sequence.get(dim)
16    "Pour lire la liste dans une matrice à plusieurs dimensions (dim
        , nsims)"
17    data=np.array(norm.ppf(points))
18    shape=(dim, nsims)
19    return data.reshape(shape)
```

La suite que nous avons construite marche beaucoup mieux que celle existante. (problème d'allocation de la mémoire, ghalton pas assez optimisé).

Il est intéressant de comparer la suite de Halton avec une génération aléatoire simple. (Monte Carlo classique)

On génère le même nombre de points dans les 2 cas.

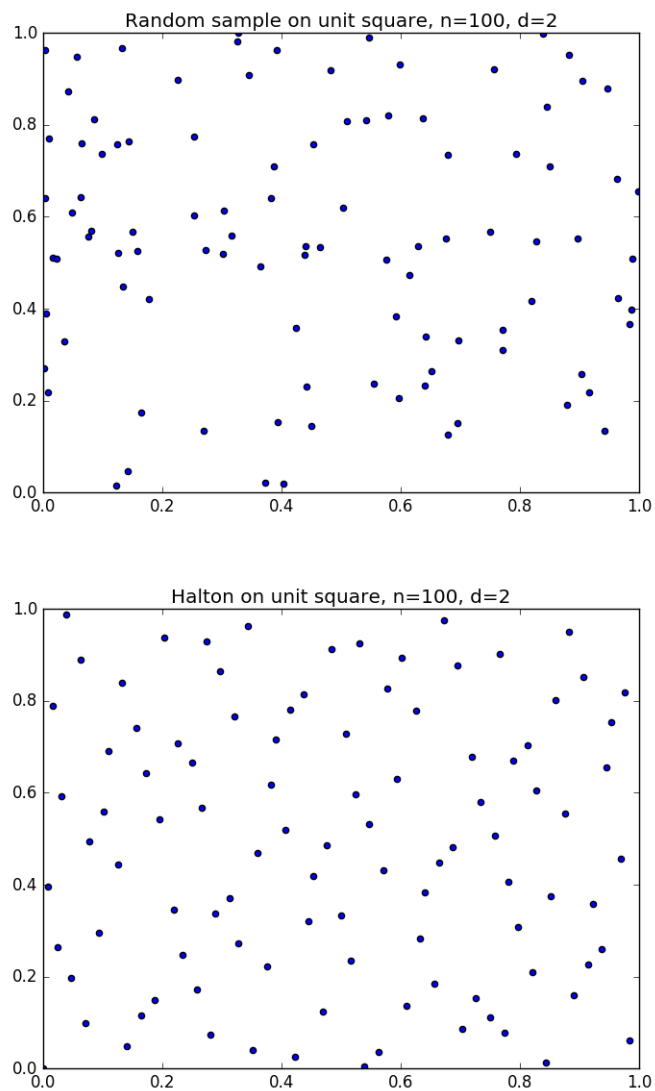
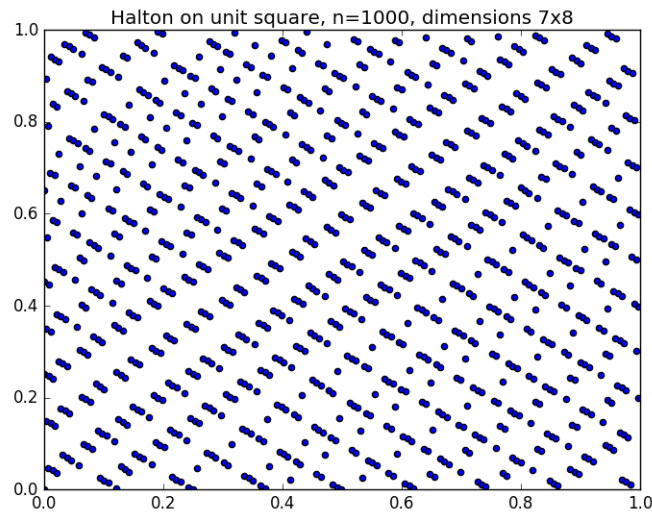


FIGURE 2.4 – Comparaison Monte Carlo classique et suite de Halton

On voit bien la différence et les propriétés d'équirépartition des points générés par la suite de Halton. Malheureusement, plus la dimension devient grande, plus la suite de Halton de Halton marche moins.

La génération de points équiréparties dans l'intervalle $[0, 1)^d$ devient plus difficile quand d la dimension augmente car l'espace à remplir devient trop large.

Dans notre cas, la suite de Halton marche plutôt bien jusqu'à $d=8$.



Après la dimension $d=14$, la suite de Halton laisse des grands espaces.

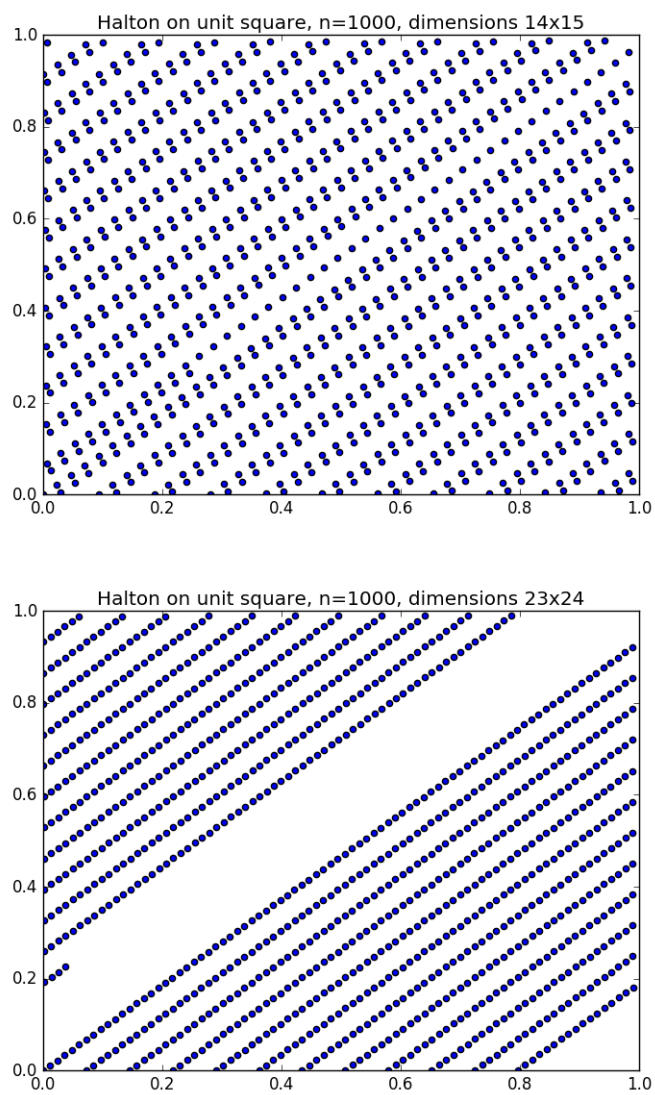


FIGURE 2.5 – Faiblesse de la suite de Halton en grandes dimensions

Discrépance

$$D_N^*(x_1, x_2, \dots, x_N) = \mathcal{O}\left(\frac{(\log N)^d}{N}\right)$$

d la dimension de la suite de halton tel que : p_1, p_2, \dots, p_d d premiers nombres premiers et N=nsims nombre de simulations.

2.2.3 Suite de Hammersley

La suite de Hammersley est une simple modification de la suite de Halton. Elle est de la forme :

$$x_i = (i/n, \phi_{p1}(i), \phi_{p2}(i), \dots, \phi_{d-1}(i))$$

Discrépance

L'avantage par rapport à la suite de Halton est que la discrépance est meilleure pour Hammersley. $D_N^*(x_1, x_2, \dots, x_N) = \mathcal{O}(\frac{(\log N)^{d-1}}{N})$ d la dimension de la suite de halton tel que : p_1, p_2, \dots, p_{d-1} d-1 premiers nombres premiers et N=nsims nombre de simulations.

Remarque

La suite de Hammersley ne sera utile que en moyenne dimensions et ne sera donc pas intrèss intéressante pour nous à cause de la séquence de la 1ère dimension.

Nous l'avons quand même implémenté vu la simple modification de la suite de Halton pour avoir une comparaison entre plusieurs suites à discrèpances faibles.

Voici la comparaison entre une génération aléatoire simple, la suite de Halton et la suite de Hammersley :

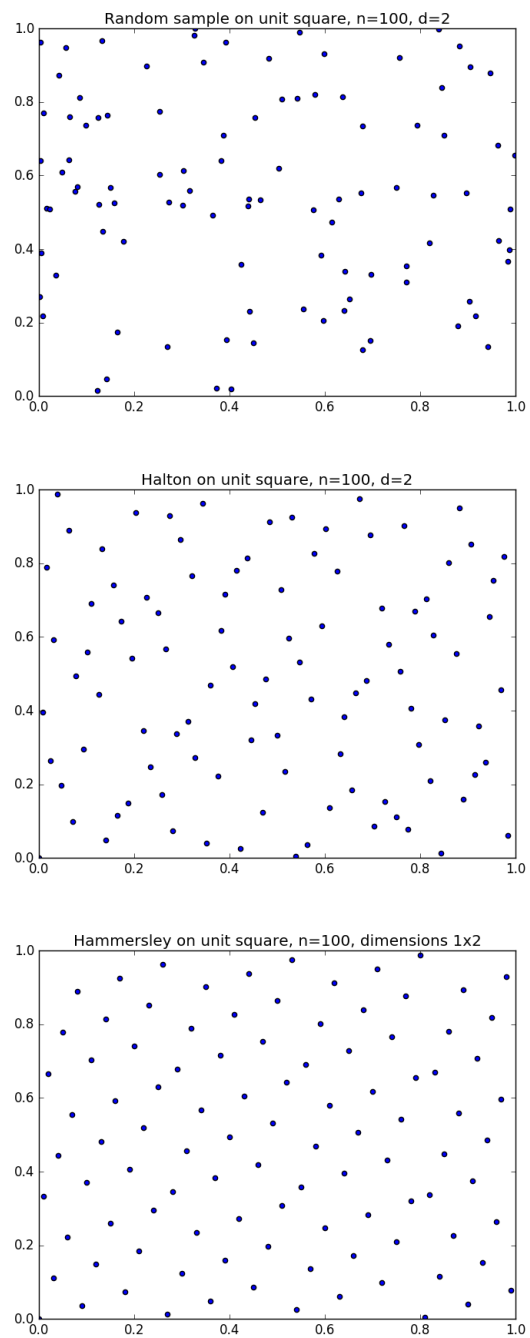


FIGURE 2.6 – Comparaisons Monte Carlo classique-Suite de Halton-Suite de Hammersley