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 soous 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 :

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

Il faut donc évaluer cette espérance pour connaître le prix d'un produit dérivé. Nous allons utilize les méthodes de Monte Carlo qui consiste à simuler un grand nombre de realisations, N indépendantes et identiquements 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 :

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

la loi des grand 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 utilisé le cas où on ne connait pas la loi de e_p. Si la loi de e_p est inconnue (ou non normale), alors $\bar{e_N} \hookrightarrow N(\mu, \frac{\sigma}{\sqrt{N}})$ pour N assez grand, avec $\sigma = \sqrt{(Var(f(X)))}$ et $\mu = e_p = \mathbb{E}[f(X)]$ alors on a :*

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

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

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

ou :

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

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 élevé, plus N doit être grand.

Or N grand a pour conséquence une augmentation du temps de calcul. Donc plus $Var(f(X^i))$, 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 :

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

En supposant que la variance est connue, on a :

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

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

$$\mathbb{P}(-u\frac{\sqrt{Var(f(X))}}{\sqrt{N}} + \bar{e_N} \leq e_p \leq u\frac{\sqrt{Var(f(X))}}{\sqrt{N}} + \bar{e_N}) = \alpha \tag{1.8}$$

et donc :

$$\mathrm{IC}_\alpha(e_p) = [-u\frac{\sqrt{Var(f(X))}}{\sqrt{N}} + \bar{e_N}; u\frac{\sqrt{Var(f(X))}}{\sqrt{N}} + \bar{e_N}] \tag{1.9}$$

L'erreur donné par un intervalle de confiance est :

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

Il a un problème c'est qu'on ne peut appliquer cette méthode que si la variance de f(X), Var(f(X)) est connue. Or ici, 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 :

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

Voici l'implémentation sous Python de tout ce que nous avons énoncé dans cette partie I, dans le fichier *_init_.py*, on a pris α =0.95 pour la contruction des intervalles de confiance.

```
"Pricing d'actifs énergétiques par Monte Carlo."

import numpy as np
import pandas as pd
import scipy.stats as sps
import efficientmc.assets as assets
import efficientmc.pricemodels as pricemodels
import efficientmc.generators as generators
from collections import namedtuple

MCResults = namedtuple('MCResults', ('mean', 'iclow', 'icup'))

```

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

```
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  nsims = cf.shape[0]
62  cumvalues = cf.sum(axis=2)
63  mean = cumvalues.mean(axis=1)
64  std = cumvalues.std(axis=1)
65  res = {}
66  for key, val in mean.items():
67  res[key] = mkmcresults(val, std[key], nsims)
68  return res
```

Il reste aussi le problème de la queue de distribution de la veritable loi de $\bar{e_N}$ (on a supposé précedemment 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 : realisation 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 `
      GaussianGenerator`.
14
15    Paramètres :
16    ------------
17    nsims : entier positif
18    Nombre de simulations à générer.
19    corrmatrix : matrice carrée
20    Matrice de corrélation entre les différents bruits gaussiens
21    à simuler. La matrice doit être définie positive.
22    corrkeys
23    Liste des identifiants associés aux différentes lignes /
      colonnes
24    de la matrice de corrélation `corrmatrix`. Les identifiants
      doivent
25    être donnés dans l'ordre dans lequel les bruits
      correspondants
26    apparaissent dans `corrmatrix`.
27    randomfunc
28    Fonction permettant de générer des bruits gaussiens indé
      pendants
29    (typiquement `np.random.randn`).
30    """
31    self.corrmatrix = corrmatrix
```

```
32      self.corrkeys = corrkeys
33      self.nsims = 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      "positive definite.") from None
41
42      @property
43      def nnoises(self):
44      "Nombre de bruits gaussiens distincts à simuler."
45      return self.corrmatrix.shape[0]
46
47      @timecached
48      def getallnoises(self, date):
49      """
50      Renvoie `self.nsims` réalisations de `self.nnoises` bruits
51      gaussiens corrélés.
52      """
53      whitenoises = self.randomfunc(self.nnoises, self.nsims)
54      noises = np.dot(np.linalg.cholesky(self.corrmatrix),
        whitenoises)
55      return noises
56
57      @timecached
58      def getnoises(self, date, keys):
59      """
60      Renvoie un tableau de taille `(len(keys), self.nsims)` de
        bruits
61      gaussiens corrélés correspondants aux aléas identifiés par
        les
62      clefs `keys`.
63      """
64      noises = self.getallnoises(date)
65      res = np.empty((len(keys), self.nsims))
66      for idx, key in enumerate(keys):
67      keyidx = self.corrkeys.index(key)
68      res[idx, :] = noises[keyidx, :]
69      return res
```

Chapitre 2

Méthodes de réduction de la variance

2.1 Variables antitthétiques

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

2.1.1 Idées globales

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

2.1.2 Mathématiquement

On veut estimer $\mathrm{E}[Y]$ en utilisant l'implémentation d'échantillons antithétiques c'est-à-dire : $(Y_1, Y_{1ant}), (Y_2, Y_{2ant}), ..., (Y_N, Y_{Nant})$ 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 moyene des 2N réalisations :

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

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

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

On pose $\sigma^2_{ant} = Var(Y) = Var(\frac{Y_i+Y_{iant}}{2})$

Comme nous venons de le voir dans les idées globales :

$\sigma^2_{ant} = \frac{Var(Y_i)+Var(Y_{iant})+2Cov(Y_i,Y_{iant})}{4}$

$= \frac{2Var(Y_i)+2Cov(Y_i,Y_{iant})}{4}$ car Y_i et Y_{iant} sont de même loi de distribution

$= \frac{Var(Y_i)+2Cov(Y_i,Y_{iant})}{2}$

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 $Y_i, Y_2, ..., Y_N$ (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 : $-Y_1, -Y_2, ..., -Y_N$.

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 :

```
def antithetic_randn(nnoises, nsims):
"""
Renvoie un tableau de bruits gaussiens non corrélés :math:`(G_{
    _i,j})`
de taille `(nnoises, nsims)`, et tel que, pour tout :math:`i` :
:math:`\forall 1 \leq j \leq n / 2, G_{i,n/2+j} = -G_{i,j}`

Paramètres :
-----------
nnoises : entier positif
Nombre de bruits à simuler.
nsims : entier positif, pair
Nombre de simulations à effectuer par prix.
"""
if nsims % 2 != 0:
raise ValueError("the number of simulations used with antithetic
    "\
"variables should be even.")
half = int(0.5 * nsims)
noises = np.empty((nnoises,nsims))
noises[:, :half] = np.random.randn(nnoises, half)
noises[:, half:] = -noises[:, :half]
return noises
```

2.2 Suites à discrépance 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 discrépance 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 :

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

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 discrépance faible permettant de générer ces points.

Discrépance 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 :
$\phi_b(n) = \frac{a_o}{b} + \frac{a_1}{b^2} + ... + \frac{a_r}{b^{r+1}}$
avec n=$a_0 + a_1 + ... + a_r b^r$, $a_r > 0$, $0 \leq a_i \leq p$, $0 \leq i \leq r$

ou

$\phi_b : \mathbb{N} \to [0,1)]$
$\forall\ b \in \mathbb{N}$, $\phi_b(n) = \sum_{i=0}^{I} \frac{a_i(n,b)}{b^{i-1}}$

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

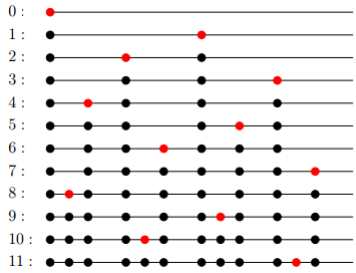


FIGURE 2.1 – Suite de Van Der Corput en base 2 pour n=12

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 permmettant de générer la séquence :

```
1 def vdc(n, base):
2 """
3 Cette fonction permet de calculer le n-ieme nombre de la base b
     de la
4 séquence de Van Der Corput
5 """
6 vdc, denom = 0,1
7 while n:
8 denom *= base
9 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
        en base b
4   """
5   array=np.empty(nsims)
6   i=0
7   for i in range(nsims):
8   array[i]=vdc(i,b)
9   return array
```

On a testé ces fonctions et on a tracé ceci :

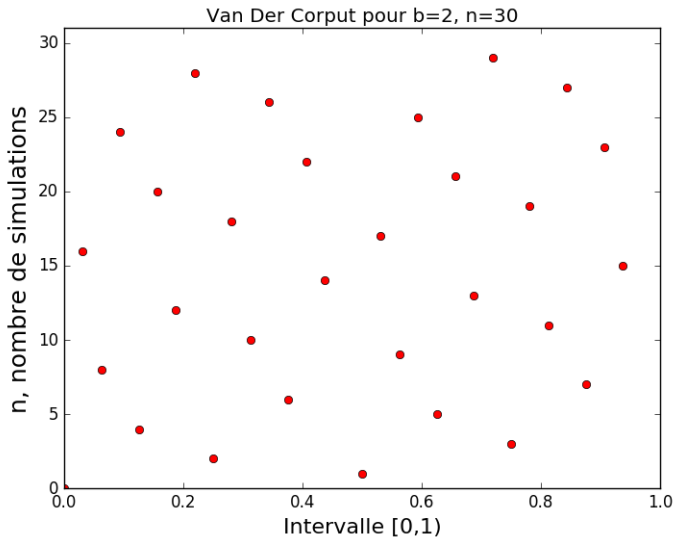


FIGURE 2.2 – Suite de Van Der Corput en base 2 pour n=30

On voit bien sur la figure 2.2 que le 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=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
        base b de la
4       séquence de Van Der Corput
5       """
6       vdc, denom = 0,1
7       while n:
8       denom *= base
9       n, remainder = divmod(n, base)
10      vdc += remainder / denom
11      return norm.ppf(vdc)
12
```

```
13      def van_der_corput(nsims,b):
14      """
15      Cette fonction permet de générer la séquence de Van Der
        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)
        toutes les séquences
27      de la suite de Van der Corput de la base 2 à la base dim+2
28      """
29      for i in range(2,dim+2,1):
30      array[i-2,:]=van_der_corput(nsims, i)
31      return array
```

Discrépance

$D_N^*(\phi_b(1), \phi_b(2), ..., \phi_b(N)) = \mathcal{O}(\frac{logN}{N})$
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 :
$x_i = (\phi_{p_1}(i), \phi_{p_2}(i), ..., \phi p_d(i))$
avec d la dimension de la suite de Halton et $p_1, p_2, ..., p_d$ 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 précedemment.

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

```
import ghalton as gh
def halton(dim,nsims):
"""
Attention: la fonction crash pour nsims>500, on doit revoir l'
    optimisation de la fonction

On utilise la librairie Python existante sur la suite de Halton

GeneralizedHalton produit une suite de nsims dimension (colonnes
    ),
le nombre 68 est utilisé pour faire des permutations, c'est le
    nombre qui permet de se rapprocher
le plus des valeurs du Monte Carlo classique
"""
sequence = gh.GeneralizedHalton(nsims,68)
"Une liste de dim sous-listes est produite"
points=sequence.get(dim)
"Pour lire la liste dans une matrice à plusieurs dimensions (dim
    ,nsims)"
data=np.array(norm.ppf(points))
shape=(dim,nsims)
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.

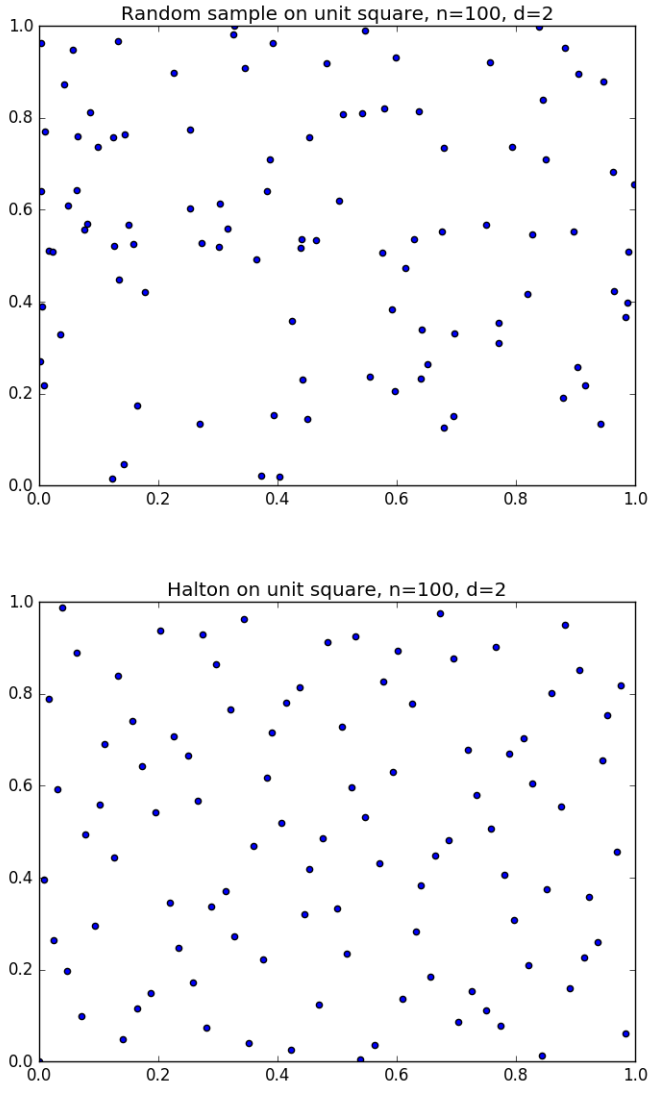
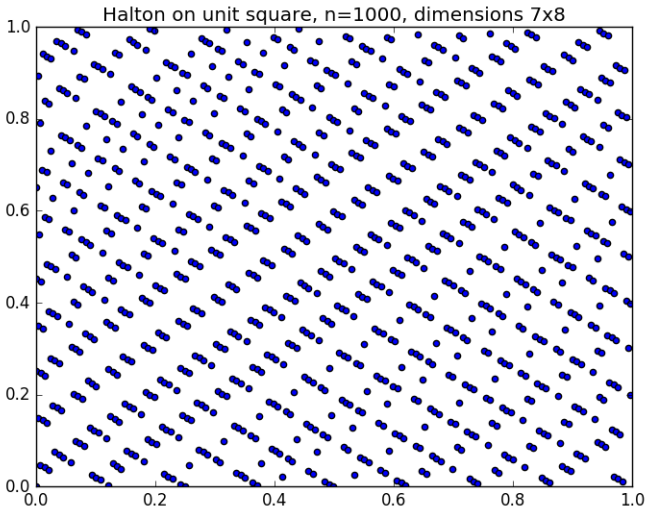


FIGURE 2.4 – Comparaison Monte Carlo classique et suite de Halton

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.

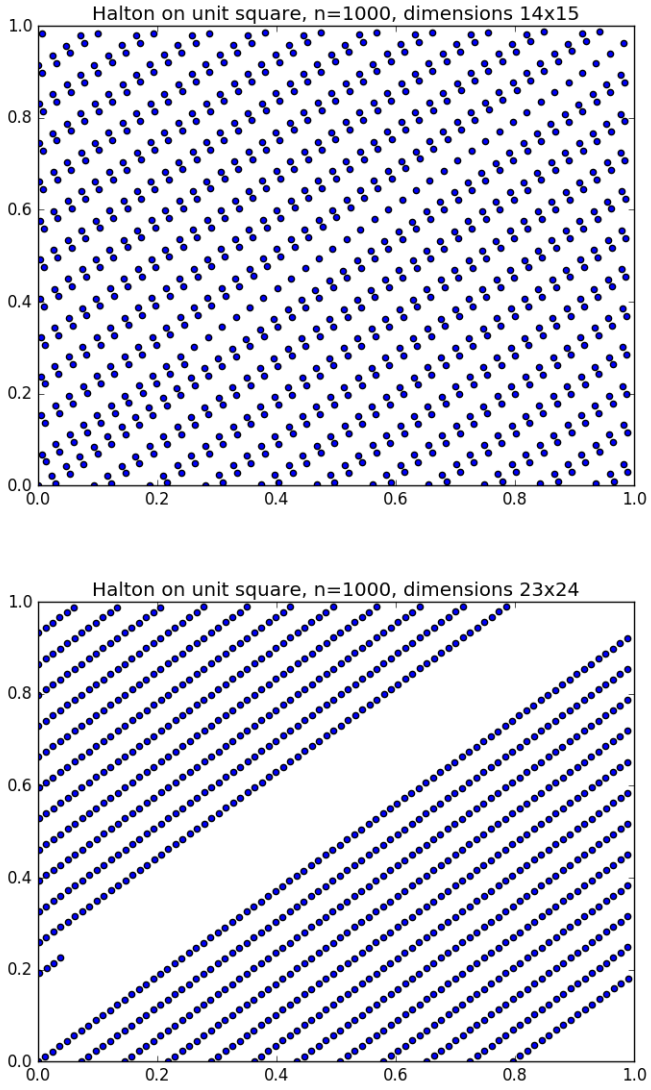


FIGURE 2.5 – Faiblesse de la suite de Halton en grandes dimensions

Discrépance

$D_N^*(x_1, x_2, ..., x_N) = \mathcal{O}(\frac{(logN)^d}{N})$
d la dimension de la suite de halton tel que : $p_1, p_2, ..., p_d$ d premiers nombres premiers et N=nsims nombre de simulations.