Monte Carlo project

Stratification

Marina BLAZEVIC, Mehdi EL KACEMY, Samy KOBBITE

ENSAE Paris

April 25, 2023

- We want to estimate the integral of the following function :

$$u \in [0,1]^d,\ \text{pour } d \geq 1 : f(u) = 1 + \sin\left(2\pi\left(\frac{1}{d}\sum_{i=1}^{d} u_i - \frac{1}{2}\right)\right)$$

- For this, we will use several stratification methods..

- **Idea** : Integrate numerically by subdividing the integration interval, and performing a Monte Carlo simulation in each subinterval with a different number of random points.
- **Advantage** : Reduce the variance of the estimate of the integral compared to a simple Monte Carlo simulation
- We estimate the integral I of a function $f(x)$ on the interval $[a, b]$ by estimating I :

$$I \approx \frac{b-a}{N} \sum_{i=1}^{N_s} \frac{1}{n_i} \sum_{j=1}^{n_i} f(x_{ij}^{(i)})$$

N_s : number of sub-intervals
n_i : number of random points in the subinterval i
$x_{ij}^{(i)}$: j-th random point in subinterval i

- **Monte Carlo method**
  Let n be the number of random points generated in the domain $[0,1]^d$. We can estimate the integral of the function $f(u)$ using the following formula:

$$I \approx \frac{1}{n}\sum_{i=1}^{n} f(u_i)$$

  where u_i is the i-th random point in the domain $[0,1]^d$.
- **Quasi Monte Carlo method**:
  The generation of random numbers is replaced by a deterministic sequence of points such as the Halton sequence or the quasi-random sequence of Sobol.

Estimation by Monte Carlo methods
Principle

```
def monte_carlo_integration(d,Ns):
    samples = np.random.uniform(0, 1, size=(Ns,d))
    values = f(samples)
    return np.mean(values)
```

Figure: Code for MC

```
def quasi_monte_carlo1(d, Ns):
    samples = stats.qmc.Sobol(d).random(Ns)
    values = f(samples)
    return np.mean(values)
```

Figure: Code for QMC with Sobol

- **Generation according to Sobol**
  Method based on a family of sequences of binary numbers constructed from irreducible polynomials of high degree.
  Polynomials used to generate sequences of binary numbers which are then converted into reals via inverse transformation.
- **Generation according to Halton**:
  Choice of a numeric base b and generation of a sequence of numbers x_i which are fractions in base b
- **In practice**:
  Sobol is more efficient for large dimensions
  Halton easier to generate

- **Mean Squared Error :** mean of the squares of the differences between the estimated values and the true values

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(\hat{I}_i - I)^2$$

- **Confidence intervals**: intervals of probable values for the value of the integral.

Estimation by Monte Carlo methods

Comparison Between MC and QMC Sobol

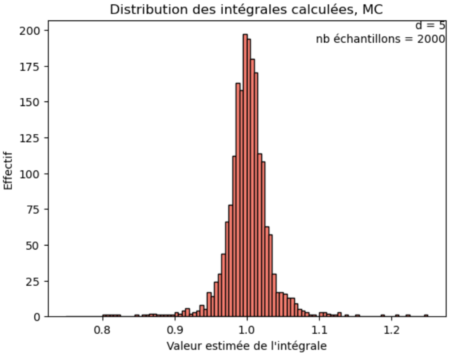


Figure: Distribution MC

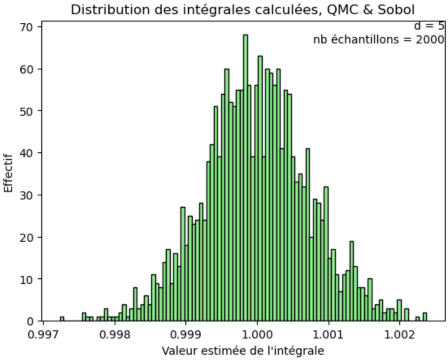


Figure: Distribution QMC Sobol

For d = 5, nb samples = 2000 and nb estimates = 1000:
MSE.MC = 0.00018 and MSE.QMC = 1.8955e-07
CI.MC = (0.97438, 1.0497) and CI.QMC = (0.9617, 1.0384)

Estimation by Monte Carlo methods

Comparison between MC and QMC Sobol

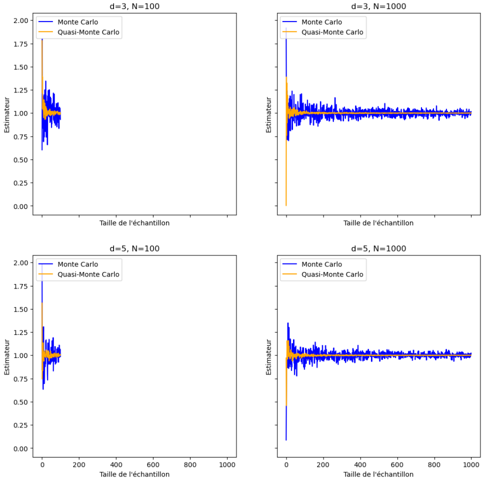


Figure: Comparison of convergences between MC and QMC for different parameter values

d	nb_echantillons	nb_estimations	MSE_MC	MSE_QMC
1	10000	1000	0.000324	2.2554e-09
2	10000	1000	0.000612	4.4849e-09
3	10000	1000	0.000971	6.1972e-09
4	10000	1000	0.001279	9.1086e-09

Figure: Comparison between MC and QMC Sobol for different values of d

Estimation by Monte Carlo methods

Comparison between Sobol and Halton for QMC

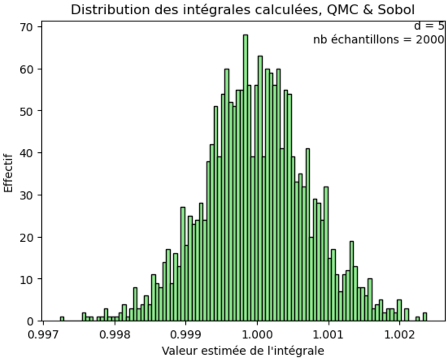


Figure: Distribution QMC Sobol

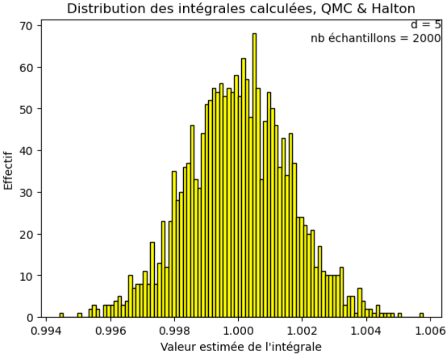


Figure: Distribution QMC Halton

For d = 5, nb samples = 2000 and nb estimates = 2000:
MSE.S = 1.8728e-07 and MSE.H = 7.1403e-07
CI.QMCS = (0.9862, 1.0139) and CI.QMCH = (0.9860, 1.0137)

Estimation by Monte Carlo methods

Comparison between Sobol and Halton for QMC : numerical errors

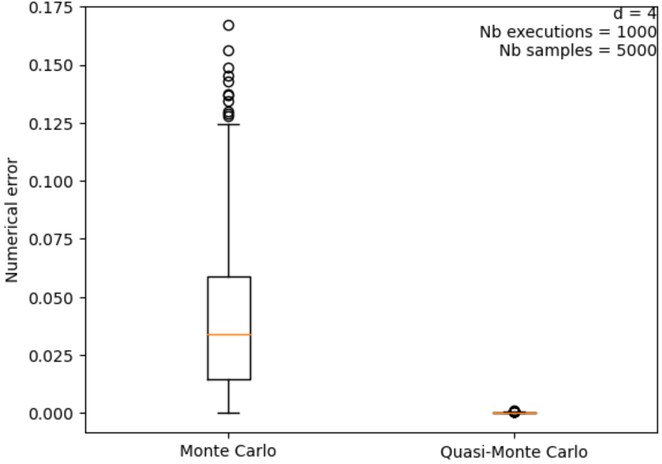


Figure: Numerical errors for MC and QMC

- Article by Nicolas Chopin and Mathieu Gerber.
- Proposition of 2 unbiased estimators of the integral of the function f over $[0,1]^s$ depending on a regularity parameter $r \in \mathbb{N}$
- Based on cubic stratification
- Here, the goal is to implement the Haber estimators of order 1 and 2:

$$I_{1,k}(f) := \frac{1}{k^s} \sum_{c \in C_k} f(c + U_c), \quad U_c \sim U\left(-\frac{1}{2k}, \frac{1}{2k}\right)^s$$

$$\text{et } I_{2,k}(f) := \frac{1}{k^s} \sum_{c \in C_k} g_c(Uc),\ U_c \sim U\left[-\frac{1}{2k}, \frac{1}{2k}\right]^s$$

with
$C_k = C_{0,k}$ and
$C_{m,k} = \left\{\frac{2j_1+1}{2k}, \ldots, \frac{2j_s+1}{2k}\right\} \mid (j_1, \ldots, j_s) \in \{-m, \ldots, k+m-1^s\}$ and
$g_c(u) := \frac{f(c+u)+f(c-u)}{2}$, where $n = 2k$

Haber estimators

Haber of order 1

```
def haber_ordre1(N):
    n = k**s
    estimates = []
    for _ in range(N):
        Uc = [random.uniform(-1/(2*k), 1/(2*k)) for i in range(s)]
        I = 0
        for c in C(k, s):
            I += f(tuple(ci + ui for ci, ui in zip(c, Uc)))
        I /= n
        estimates.append(I)
    return estimates
```

Figure: Our code for Haber 1

Haber estimators

Haber of order 2

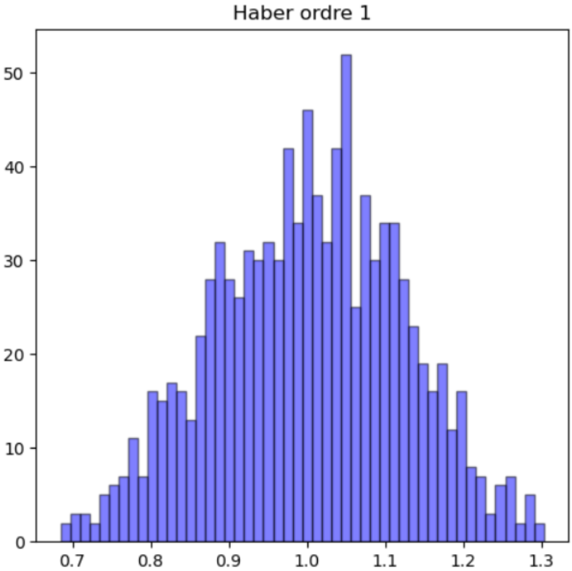
```
def gc(u, c, f):
    g = (f(tuple(ci + ui for ci, ui in zip(c, u))) + f(tuple(ci - ui for ci, ui in zip(c, u)))) / 2
    return g

def haber_ordre2( N):
    n = 2*k**s
    estimates = []
    for _ in range(N):
        Uc = [random.uniform(-1/(2*k), 1/(2*k)) for i in range(s)]
        I = 0
        for c in C(k, s):
            I += gc(Uc, c, f)
        I /= n
        estimates.append(I)
    return estimates
```

Figure: Our code for Haber 2

Haber estimators

Result for Haber of order 1



Haber estimators

Results for Haber of order 1 and 2

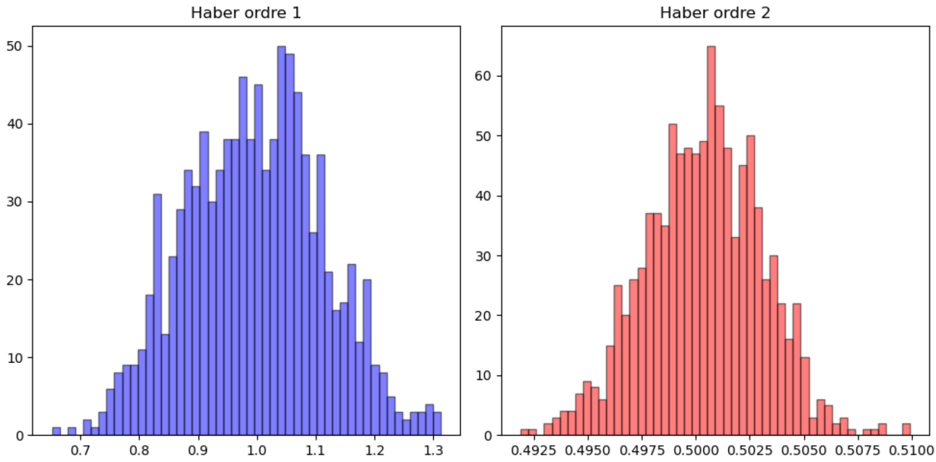


Figure: Distribution of estimators for s = 4, k = 5, N = 1000

Haber estimators

Comparison in terms of speed and MSE

```
L'algorithme monte_carlo_integration a pris 0.0009996891021728516 secondes pour s'exécuter.
L'algorithme quasi_monte_carlo a pris 0.0020079612731933594 secondes pour s'exécuter.
L'algorithme haber_ordre1 a pris 55.08616590499878 secondes pour s'exécuter.
```

Figure: Time for d = 10, s=4, k=5, number of samples = 10000

```
L'algorithme monte_carlo_integration a pris 0.0019993782043457031 secondes pour s'exécuter.
L'algorithme quasi_monte_carlo a pris 0.0030021667480468750 secondes pour s'exécuter.
L'algorithme haber_ordre1 a pris 0.4043452739715576 secondes pour s'exécuter.
```

Figure: Time for d = 4, s = 2, k = 2 and number of samples = 10000

```
'MSE_MC = 0.006731834267663948 et MSE_QMC = 9.053440901287969e-07 et MSE_haber1 = 7.284517252308317e-05'
```

Figure: MSE for d = 4, s = 2, k = 2 and number of samples = 1000

Approach by Importance Sampling

Principle

- Importance sampling is a numerical method for estimating the integral of a function $h(u)$ over the interval $[0,1]^d$:

$$\int_{[0,1]^d} h(u)du$$

- Use a probability distribution different from the uniform distribution over $[0,1]^d$ to generate samples, and weight each sample according to the probability of being chosen from this distribution.
- Here, generation of the samples $u_1, u_2, \dots, u_N$ from a probability distribution $p(u)$ different from the uniform distribution, weighting of each sample according to the ratio between the function $h(u)$ and the weighting function $g(u)$.
- The estimator of the integral is then given by:

$$\hat{I} = \frac{1}{N} \sum_{i=1}^{N} \frac{h(u_i)}{g(u_i)}$$

Approach by Importance Sampling

Our function

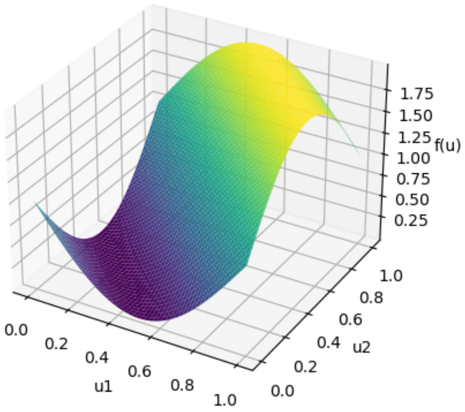


Figure: Studied function in multidimensionnal view

We chose $g(u)$ such as that when

$||\mathrm{u}||_1 = \sum_{i=1}^{n} |u_i|$

is extreme the weight is more important in the distribution.

Hence, when we have important values that strongly affects the value of the integral, we are sure that they are taken into account.

Once we are sure that they are taken into account, we downweight them

```python
def g(u, n, value1, value2):
    norm1 = np.sum(np.abs(u))
    if norm1 > 3 * d / 4 or norm1 < d / 4:
        return value1
    else:
        return value2
```

Figure: Our g function to create weights

```
The best estimated value of the integral is 1.0045025285020868
The optimal value1 for function g is 0.0
The optimal value2 for function g is 1.0
```

Figure: Estimated best values

Bibliography

- Higher-order stochastic integration through cubic stratification, Nicolas Chopin and Mathieu Gerber, 5th October 2022.
- Quasi-Monte Carlo Methods in Python, Pamphile T. Roy, Art B. Owen, Maximilian Balandat and Matt Haberland.
- Monte Carlo Statistical Methods, C.P. Robert, Springer.
- "Importance sampling", Lectures on probability theory and mathematical statistics, Taboga, Marco (2021).
- `https://towardsdatascience.com/importance-sampling-introduction-e76b2c32e744`.
- `https://towardsdatascience.com/python-powered-monte-carlo-simulations-fc3c71b5b83f`.

Bibliography

The link to our Git : `https://github.com/marinablaz/ProjetMonteCarlo`