Stochastic Simulations: Project 5
QMC integration of non-smooth functions: application to pricing exotic options

Benoît Müller

October 4, 2023

Preparations

To re-cast the integral into a hypercube, we rewrite it first in terms of into uniform random variables in $[0, 1]$. To do this we write the discretized Brownian motion in term of normal variables by Gaussian increments ξ_i:

$$w_{t_i} = w_{t_{i-1}} + \sqrt{t_i - t_{i-1}}\xi_i = \sum_{k=1}^{i} \sqrt{t_k - t_{k-1}}\xi_k \quad \text{for } i \in \{1, \dots, m\}$$

with $t_i = iT/m$ and ξ_i some independent normal standard variables. Now we write ξ_i using uniform variables U_i. For efficiency and because we can impose the dimension to be even, we choose the Box-Müller method[1]:

$$\xi_{2k,2k-1} = \sqrt{-2\log U_{2k}}(\cos, \sin)(2\pi U_{2k-1}) \quad \text{for } k \in \{1, \dots, m/2\}$$

We can write now explicitly $\Psi_i = \theta_i(U)\mathbb{1}_{\phi(U)}$ for a uniform variable U in $\mathbb{R}^m$. We fix m and define the functions related to the transformations: $S_i = S_{t_i}$, $Z_i(U)$ for the Gaussian variables , $W_i(Z)$ for the Brownian motion.

$$\begin{aligned}
\phi(U) &= \frac{1}{m}\sum_{i=1}^{m} S_i\big(W_i(U)\big) - K \\
&= \frac{1}{m}\sum_{i=1}^{m} S_i\big(\sum_{k=1}^{i} \sqrt{t_k - t_{k-1}}Z_k(U)\big) - K \\
&= \frac{1}{m}\sum_{i=1}^{m} S_0 e^{(r-\sigma^2/2)t_i + \sigma\sum_{k=1}^{i}\sqrt{t_k - t_{k-1}}Z_k(U)} - K \qquad (1)\\
&= \frac{1}{m}\sum_{i=1}^{m} S_0 e^{(r-\sigma^2/2)t_i + \sigma\sum_{k=1}^{i}\sqrt{t_k - t_{k-1}}Z_k(U)} - K
\end{aligned}$$

where we have fixed m

Part I

We decide to go for an object-oriented implementation, using classes of objects. First, we will use along the project a upper class called `RandomVariable`, which contain general statistical purpose methods such as the confidence interval. It has a property X for the random sample and N for its length.

We create now a subclass `Payoff` that define the random variable Ψ for fixed parameters passed in properties: m, K, S_0, r, σ, T, and the time steps $(t_i)_i$. The methods will consist of the core of the code. We try to vectorize functions as we can, putting always the dimension m in the last axis of the Numpy arrays.

[1] I'm a Mister Müller too but I promise I don't receive any royalties on the spread of this fancy method.

We define then some methods for the variables $S(w)$, $\theta(w)$[*il faudra changer le nom de theta en phi ici et dans le code*], $\Psi_i(w)$ for $i = 1, 2$, $Z(U)$, $W(Z)$, and they help us to define the two final transformations from U to Ψ_i. From this, we write a random variable sample generator that return the transformation of a uniform random sample.

The Monte Carlo (MQ) method use `rvs` to generate N samples and return the confidence interval. For the Quasi Monte Carlo (QMC) method, we use the module `SoboL_new` given in the Lab session, that generate low discrepancy points sets, so we can try to see if this increase the order of the error. We then transform these points by `transform` and take the average on N of them. To randomize the method and compute an standard error, we repeat the process k times while shifting all points by vector randomly generated with a uniform distribution. We choose k fixed with value 20 so it does not cost too much more time. The precision of the mean was already qualitative with the N sampling, and this is only for error estimation purposes.

We will see the evolution for different values of N, but we haven't vectorized the (Q)MC methods with respect to N, or build update formulas for the mean and variance. This doesn't affect the efficiency of the utilization of the (Q)MC methods but just our study of the error.

We fix the variables $K = S_0 = 100$ and $r = \sigma = 0.1$ for now, and consider the dimension m taking values in $\{2^5, \ldots, 2^{10}\}$ and compute for sample of lengths N in $\{2^7, \ldots, 2^{14}\}$. We set the confidence as $1 - \alpha = 1 - 0.01$. The plots we display show in a first time the evolution of the mean and its interval around in a log-scale for the x-axis which represent N (Figure [1]):

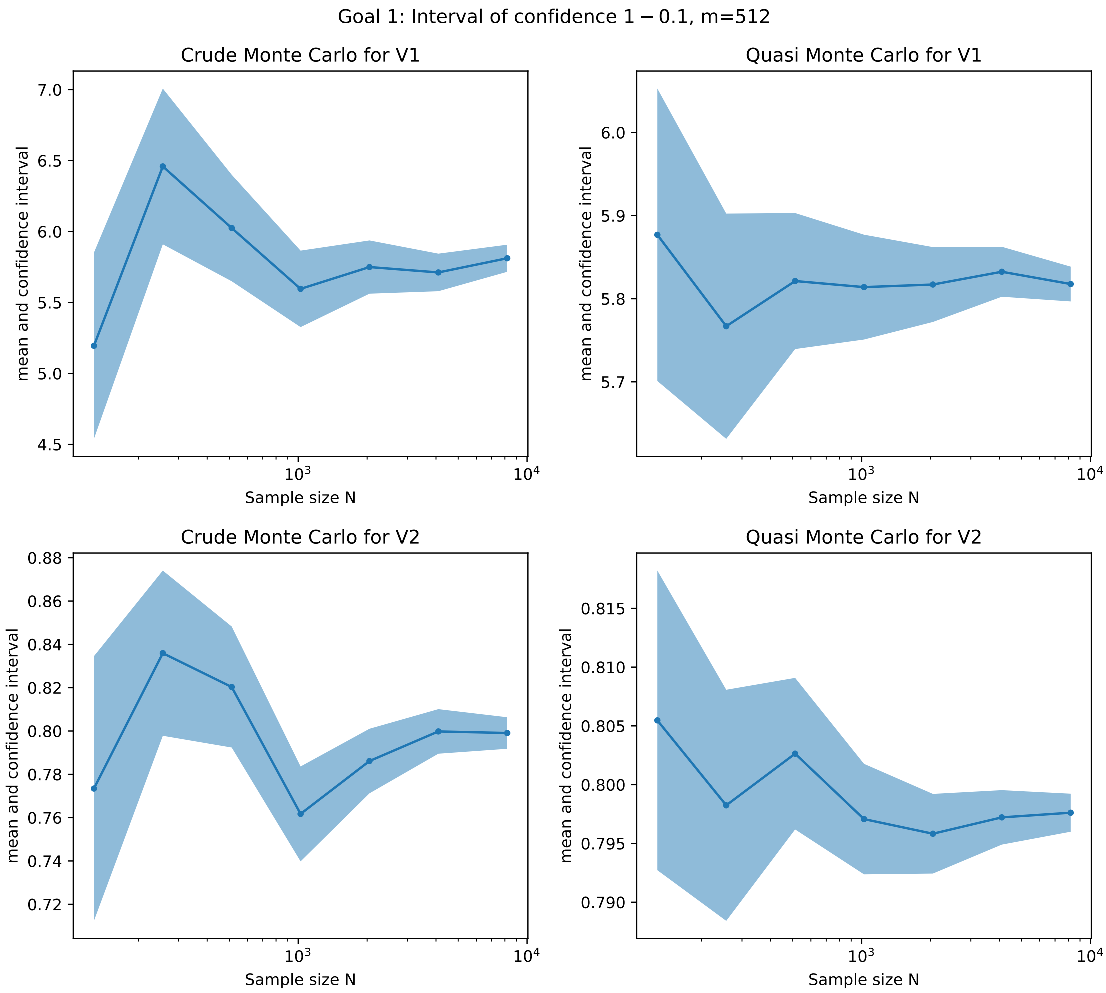

Figure 1: Evolution of the mean and the interval given by (Q)MC with respect to the sample size

We see that both methods seem to converge to the same value. Even if the mean need time to be stabilized, all the intervals always contain the final mean done with the most precision and attest their coherence. The output is:

The computed intervals for m = 512 , N = 8192 and alpha = 0.1 are:
V1: MC: 5.811725874367571 ± 0.09548596396958434
QMC: 5.817669062629391 ± 0.02093917604003062
V2: MC: 0.799072265625 ± 0.007282353659082176
QMC: 0.797613525390625 ± 0.0016172925932021554

In a second time, we display the evolution of the error only, with respect to N in a log-scale for both axis (Figure [2]):

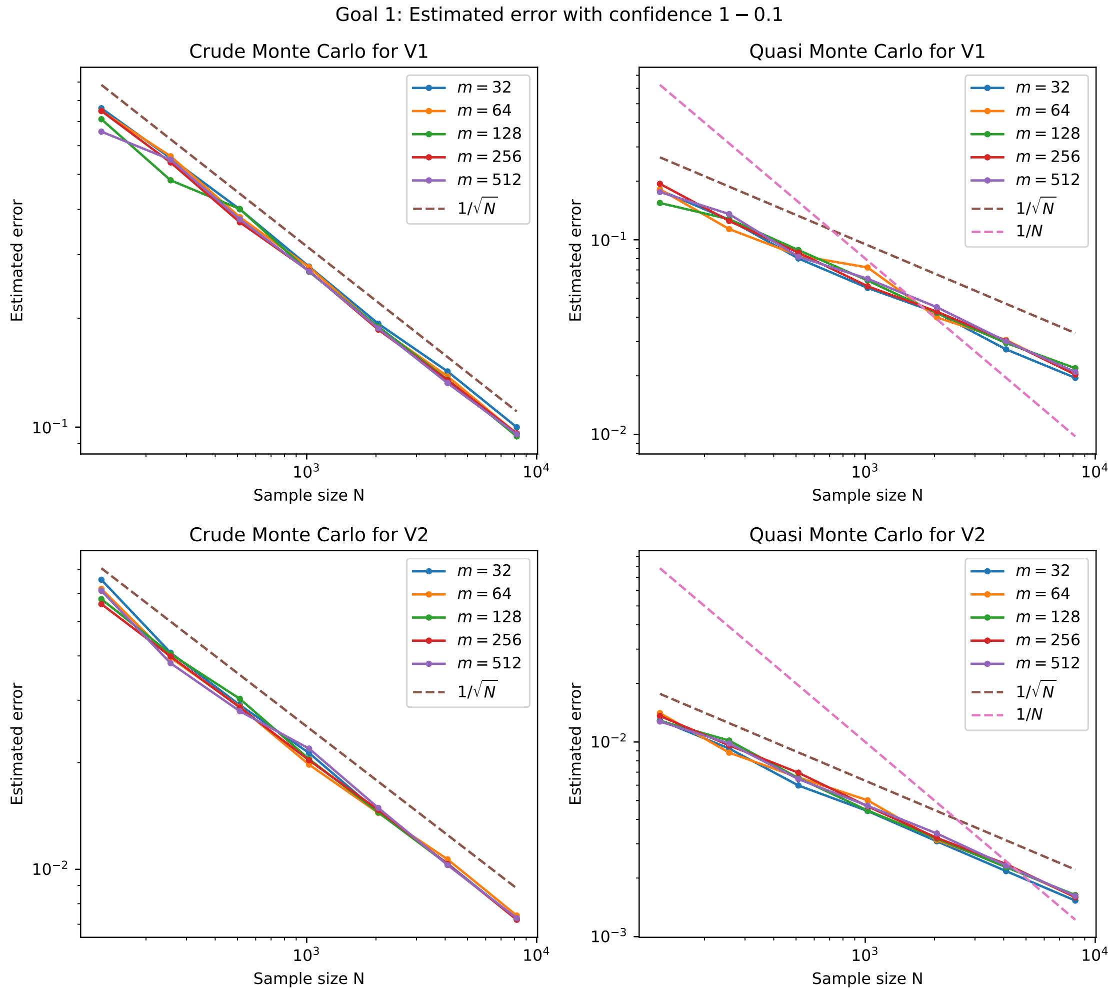

Figure 2: Evolution of the error (standard deviation) given by (Q)MC with respect to the sample size

We see that like the theory predicted, the MC method has a convergence of order $1/\sqrt{N}$. The factor doesn't change as the dimension m gets bigger. This is because the MQ method has the good property to not deteriorate with big dimensions. For the QMC method we see that we do not obtain a better convergence, due to the non-smoothness of the integrant.

Part II

A point that is important in the implementation of the pre-integration trick, is to have a good estimation of the value of the integral. Only likewise we will have a regular integrant for the QMC method and supposedly obtain a better convergence. If we first have a good approximation of the support of $\phi(., x_{-j})$, the integral can be done on a interval where the function is actually smooth, and a quadrature will be precise. The implementation we choose in equation (1) do not have a simple expression with respect to all variables. When the index is even, the variable appear in expressions like $\sqrt{-2\log U_{2k}}$ and when odd in expressions like $(\cos,\sin)(2\pi U_{2k-1})$. The first one is decreasing and the the second is not. Moreover if we take off the first two terms of the sum in W_i we have

$$W_i(U) = \sum_{k=1}^{i} \sqrt{t_k - t_{k-1}} Z_k(U) = \sqrt{t_1}\sqrt{-2\log U_2}(\cos+\sin)(2\pi U_1) + \sum_{k=3}^{i} \sqrt{t_k - t_{k-1}} Z_k(U),$$

where the rest of the sum does not imply a variable with index one or two. We then see that W must be monotone with respect to the second variable. Actually this argument is valid for any variable with even index. Now since $\phi(U) = \frac{1}{m}\sum_{i=1}^{m} S_i\big(W_i(U)\big) - K$ it is suffisient to show that S_i is increasing. We have indeed that $S_i(W_i) = S_0 e^{(r-\sigma^2/2)t_i + \sigma W_i}$ is increasing, assuring that $\phi(U)$ is monotone, as we wanted. The function ϕ has then at most one root in $[0, 1]$.

In our implementation, the variables are in a different permutation for the transformation of the uniform variable into the Gaussian one: instead of alternating the pairs, the vector is cut in two halves that are being transformed into the angle and the radius variable. However, the second variable is a fixed point of this permutation, so we can keep the second variable for the pre-integration. We doesn't plot for all the values of m because the time of computation has increase significantly and directly depends on the number of dimension, and because we know that the rate will be the same for all values. The sample size range is a bit smaller too. Here are the results we got:

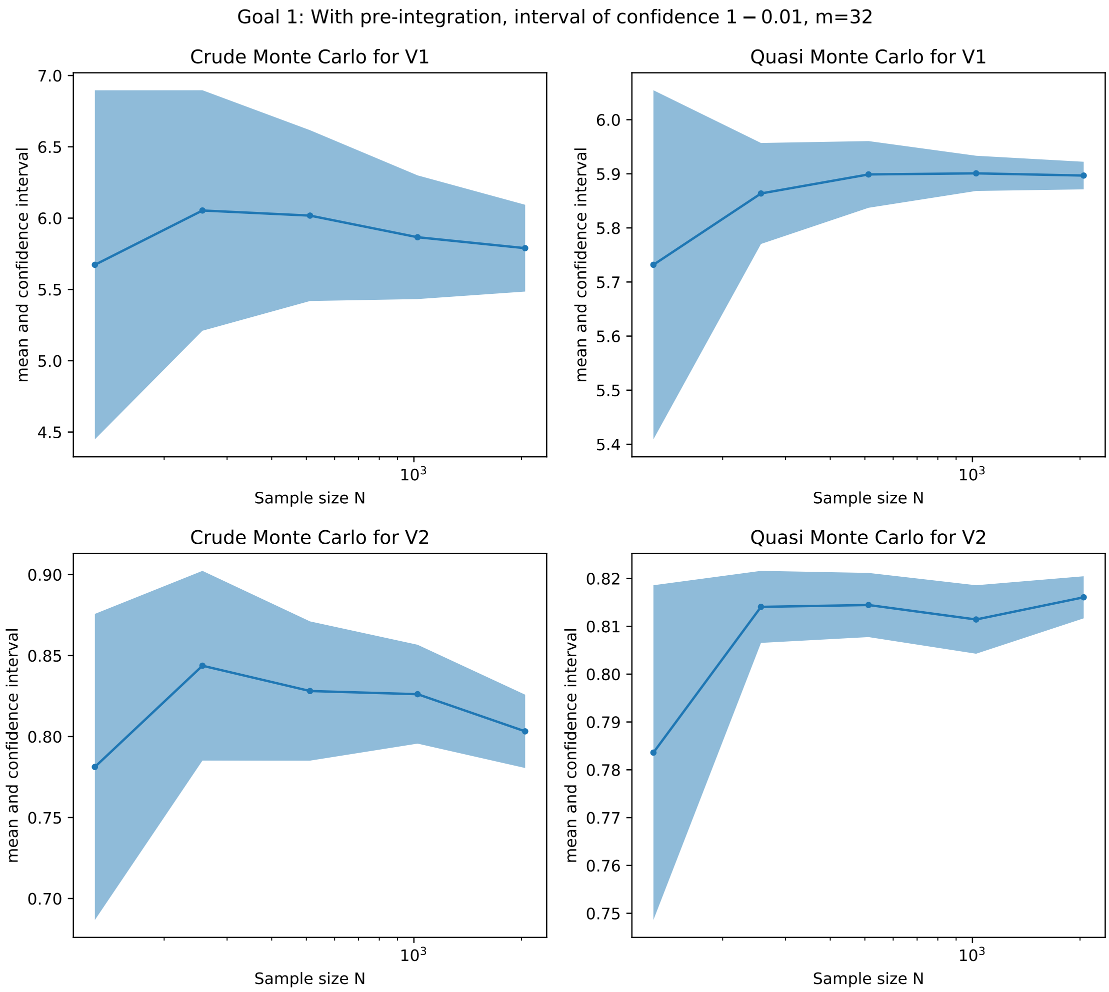

Figure 3: With preintegration: evolution of the mean and the interval given by (Q)MC with respect to the sample size

The output:
The computed intervals for m = 32 , N = 2048 and alpha = 0.01 are:

V1: PIMC: 5.789356690015531 ± 0.304260988271941
PIQMC: 5.896785954787449 ± 0.025505059981263036
V2: PIMC: 0.80322265625 ± 0.022634123444309658
PIQMC: 0.816064453125 ± 0.004397258072902301

The method still converge, and almost to the same value with a coherent evolution of intervals. The error is the following:

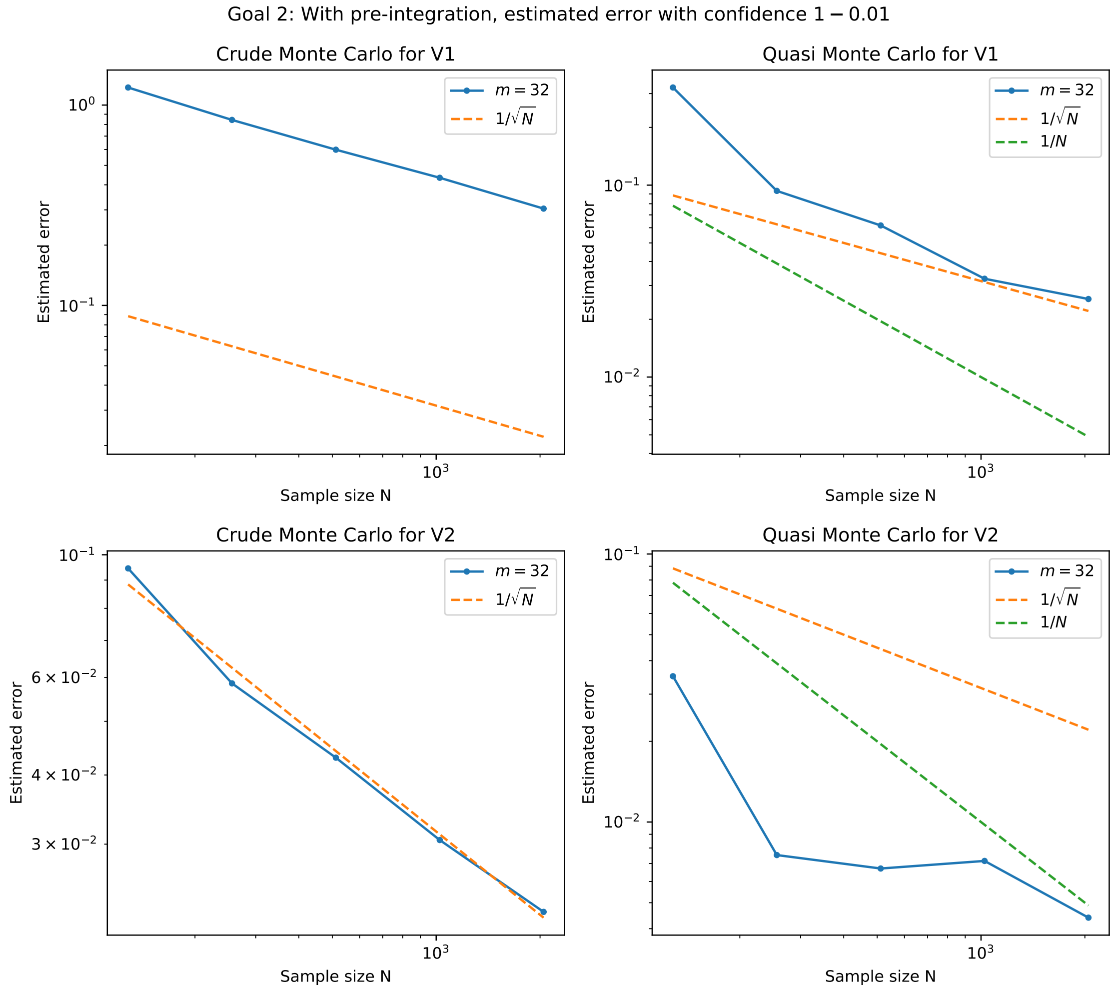

Figure 4: With re-integration: evolution of the error (standard deviation) given by (Q)MC with respect to the sample size

We see a little change, the rate seems to be a bit better, but not significantly enough to say that it is clearly $1/N$ and not $1/\sqrt{N}$ any more.

A posteriori, we realize that if the generation of the random variable needed to start from a uniform variable on the unit square, the pre-integration trick could have done a step less deep. Indeed we could have integrated with respect to one of the Gaussian variable, taking care of multiplying by a weight function (the gaussian pdf) which can be factorized as the gaussian invrements are independents. After that, we would have restate the problem back as a integral on the unit square and finish with the (Q)MC method. In this case the integrant would have been simpler to integrate because more regular, and the root finding step could have been done with the newton method for example. This could explain why the results hasn't the expected rate.

Part III

Based on a previous run made with sample size $\tilde{N}$, the supposed sample size we should use is

$$N = \frac{c_{1-\alpha}^2 \hat{\sigma}_{\tilde{N}}^2}{\text{tol}^2} = \frac{\text{err}^2 \tilde{N}}{\text{tol}^2} = \frac{\text{err}^2 \tilde{N}}{10^{-4} \hat{\mu}_{\tilde{N}}^2}$$

(with the condition that the first run is big enough to be meaningful) and where $\text{tol} = 10^{-2} \hat{\mu}_{\tilde{N}}$.

When $K = 500$, it is really greater than the mean of S, $S_0 e^{rT}$. As a result, most of the the mass of S falls in the region where $/Psi = 0$. Hence a crude Monte Carlo estimator will be very ineffective as only few replicas of S will fall in the "interesting" region $S > K$. The idea would then be to "artificially" push the distribution to the right. This method is the importance sampling. This can be achieved, for instance, by increasing the drift parameter r in the dynamics of S. The new parameter is $\tilde{r}$ and its associated new variable $\tilde{S}$ with likelihood ration in the importance sampling estimator reads by a formula of the course:

$$\exp\Big(\frac{\tilde{r} - r}{\sigma}\big(\frac{T(\tilde{r} - r)}{2\sigma} + \tilde{w_T}\big)\Big)$$

The

A mes_statsp.y

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Oct 24 14:03:15 2021

@author: benoitmuller
Mes fonctions pour les statistiques
"""
import numpy as np
import scipy.stats  as st
from sobol_new import *
import numpy.random as rnd
#import matplotlib.pyplot as plt

def cdf(X,x=None):
    """
    Empirical cdf of X evaluated in x
    """
    n=len(X)
    X=np.sort(X)
    if x is None:
        F=np.arange(1,n+1)/n
        X= np.repeat(X, 2)
        F= np.repeat(F, 2)
        F[0:2*n:2]=F[0:2*n:2]-1/n
        return X,F
    else:
        X=np.reshape(X,(n,1))
        return np.sum(X<=x,0)/n

class RandomVariable:
    """
    Methods for statistics on scalar or vector(iid components) random variable samples
    Use arrays of numpy
    """
    def __init__(self,X=np.array([]),ordered=False):
```

```
        "Initiate a rv with sample X"
        self.X=X
        self.ordered=ordered
    def sort(self):
        "If scalar, sort the sample to increasing order"
        if self.ordered==False:
            self.X.sort()
            self.ordered=True
    def cdf(self,x=None):
        """If scalar, Empirical cdf of X (evaluated in x)
        if x=None  : return locations of jumps and their heigth
        if x!=None : return the images cdf(x) """
        n=len(self.X)
        self.sort()
        if x is None:
            F=np.arange(1,n+1)/n
            self.X= np.repeat(self.X, 2)
            F= np.repeat(F, 2)
            F[0:2*n:2]=F[0:2*n:2]-1/n
            return self.X,F
        else:
            self.X=np.reshape(self.X,(n,1))
            return np.sum(self.X<=x,0)/n
    def add_data(self,X):
        "Allow to add some new data and add it to the sample"
        self.X = np.concatenate((self.X,X),axis=0)
        self.ordered = False
        self.N = np.shape(self.X)[0]
        return self
    def set_data(self,X):
        "Alow to set some new data and change the sample"
        self.X = X
        self.ordered = False
        self.N =np.shape(X)[0]
        return self
    def mean(self):
        "Compute the empirical esperance"
        return np.mean(self.X,axis=0)
    def variance(self,mean=None):
        """Compute the empirical variance of each dimension,
        using mean if already computed """
        if mean==None:
            mean=self.mean()
        return np.sum((mean - self.X)**2,axis=0) / (self.N-1)
    def interval(self,alpha):
        """Compute the 1-alpha confidence interval of the expected value"
        return the mean and the error s.t. I=[mu +- error] """
        mu= self.mean()
        err =   st.norm.ppf(1-alpha/2) * np.sqrt(self.variance(mu)/self.N)
        return mu, err
```

B Payoff.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Dec 23 10:42:21 2021

```

```python
@author: benoitmuller
Class for the random variable simulating the payoff
discribed in the application part.
"""

import numpy as np
import scipy as sp
import scipy.stats  as st
#import matplotlib.pyplot as plt
#import time
import numpy.random as rnd
from mes_stats import RandomVariable
from sobol_new import generate_points

class Payoff(RandomVariable):
    """ Subclass of RandomVariable that simulate the payoff random variable
    discribed in the financial application part.
    """
    def __init__(self,m,K=100,S0=100,r=0.1,sigma=0.1,T=1):
        if m%2==1:
            raise Exception("The dimension m must be even")
        self.m=m
        self.K=K
        self.S0=S0
        self.r=r
        self.sigma=sigma
        self.T=T
        self.t= T/m * np.arange(1,m+1)

    # Define functions of the problem, to reduce to a uniform variable:
    def S(self,w):
        """vectorized well if the dimension for w is in the last axis
        Attention,false formula in the pdf: w isn't multiplied by t."""
        return self.S0*np.exp((self.r - self.sigma**2/2)*self.t
                              + self.sigma*w)
    def phi_w(self,w):
        "vectorized"
        return np.sum(self.S(w),axis=-1)/self.m - self.K
    def Psi1(self,w): #deletable
        "vectorized well if the dimension for w is in the last axe"
        phi= self.phi_w(w)
        phi[phi<0]=0
        return phi
    def Psi2(self,w): #deletable
        "vectorized well if the dimension for w is in the last axe"
        return 1*(self.phi_w(w)>=0)
    def Psi(self,w):
        "vectorized well if the dimension for w is in the last axe"
        phi=self.phi_w(w)
        Psi2 = 1.*(phi>=0)
        return Psi2*phi, Psi2
    def normal(self,U):
        """" Transform uniform rvs into iid standard normal rvs;
        Last dimension must be even!
        Vectorized """
        U[U==0] = 1
        rho = np.sqrt(np.abs(2 * np.log(U[...,:int(self.m/2)])))
        # the argument of np.abs should be always negative,
        # unless some approximation erros
```

```
        theta = 2 * np.pi * U[...,int(self.m/2):]
        Z=np.zeros(np.shape(U))
        Z[...,:int(self.m/2)]  = rho * np.cos(theta)
        Z[...,int(self.m/2):]  = rho * np.sin(theta)
        return Z
    def normal_bis(self,U): #tested: slower. (but advantage of monotonicity)
        return st.norm.ppf(U) # need to import scipy.stats
    def wiener(self,Z):
        " Vectorized (the Z-dimension go through the last axe) "
        W= np.zeros(np.shape(Z))
        W[...,0]=np.sqrt(self.t[0])*Z[...,0]
        for i in range(1,self.m):
            W[...,i] = W[...,i-1] + np.sqrt(self.t[i] - self.t[i-1])*Z[...,i]
        return W
    def transform1(self,U): #deletable
        " Vectorized "
        return self.Psi1(self.wiener(self.normal(U)))
    def transform(self,U):
        " Vectorized "
        return self.Psi(self.wiener(self.normal(U)))
    def rvs1(self,N): #deletable
        return self.transform1(rnd.uniform(size=(N,self.m)))
    def rvs(self,N):
        return self.transform(rnd.uniform(size=(N,self.m)))

    # Define the methods:
    # Question 1:(without preintegration)
    def MC1(self,N,alpha=0.01): #deletable
        self.set_data(self.rvs1(N))
        return self.interval(alpha)
    def MC(self,N,alpha=0.01):
        X1,X2 = self.rvs(N)
        return (self.set_data(X1).interval(alpha),
                self.set_data(X2).interval(alpha))
    def QMC1(self,N,K=20,alpha=0.01): #deletable
        X = generate_points(N,self.m)
        U = rnd.uniform(size=(K,1,self.m))
        X= X[None,:,:]
        points = np.floor(X+U)
        Mu=np.mean(self.transform1(points), axis=0)
        return self.set_data(Mu).interval(alpha)
    def QMC(self,N,alpha=0.01,K=20):
        X = generate_points(N,self.m)
        U = rnd.uniform(size=(K,1,self.m))
        X= X[None,:,:]
        Psi1,Psi2 = self.transform((X+U)%1)
        Mu1,Mu2 = np.mean(Psi1, axis=0), np.mean(Psi2, axis=0)
        return (self.set_data(Mu1).interval(alpha),
                self.set_data(Mu2).interval(alpha))
    # Question 2: (with preintegration)
    def phi(self,x,U,position):
        U = np.insert(U,position,x,axis=-1)
        return self.phi_w(self.wiener(self.normal(U)))
    def psi(self,U,position): #should use ridder or newton method
        "Compute the support of phi over [0,1]"
        fa,fb = ( self.phi(0,U,position=position),
                self.phi(1,U,position=position) ) # border values
        if fa>0 and fb>0: # phi always positive
            return 0,1
```

```
        if fa<0 and fb<0: # phi always negative
            return 1,1
        r = sp.optimize.root_scalar(self.phi,args=(U,position),
                                    bracket=[0,1],method="ridder").root
        if fa<0:
            return r,1.
        else:
            return 0.,r
    def integrate(self,U,position,order):
        integrant=lambda xx: np.array([self.phi(x,U,position)
                                    for x in np.array(xx)])
        a,b = self.psi(U,position)
        return sp.integrate.fixed_quad(integrant,a,b,n=order)[0],b-a
    def PIMC(self,N,alpha=0.01,order=5,position=0):
        U = rnd.uniform(size=(N,self.m-1))
        Mu = np.array([self.integrate(U[n,:],position,order) for n in range(N)])
        Mu1,Mu2= Mu[:,0],Mu[:,1]
        return (self.set_data(Mu1).interval(alpha),
                self.set_data(Mu2).interval(alpha))
    def PIQMC(self,N,alpha=0.01,order=5,position=0,K=20):
        X = generate_points(N,self.m-1)
        U = rnd.uniform(size=(K,1,self.m-1))
        X= X[None,:,:]
        XX=(X+U)%1
        Mu =np.array([[self.integrate(XX[k,n,:],position,order)
                    for n in range(N)] for k in range(K)])
        Mu = np.mean(Mu,axis=1)
        Mu1,Mu2= Mu[:,0],Mu[:,1]
        return (self.set_data(Mu1).interval(alpha),
                self.set_data(Mu2).interval(alpha))
```

C q1.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Dec 20 17:28:31 2021

@author: benoitmuller
----------------------------------
            PROJECT 5
----------------------------------
"""

import numpy as np
#import scipy as sp
#import scipy.stats  as st
import matplotlib.pyplot as plt
#import time
#import numpy.random as rnd
#from mes_stats import RandomVariable
#from sobol_new import generate_points
from Payoff import Payoff

# Fix the seed:
np.random.seed(12345)

# File saving options:
```

```
save_figures = True # Change the value if needed
if (save_figures==True and
    input("Do you really want to save figures"+
          " into files?\n(yes/no): ") == "no"):
    save_figures = False

# Choice of parameters:
alpha=0.1 # 1-confidence
NN=(2**np.arange(7,14)).astype(int) # sample size (7,14)
mm=(2**np.arange(5,10)).astype(int) # dimension (5,10)
MC1 = np.zeros((2,len(NN))) # MC for V1
MC2 = np.zeros((2,len(NN))) # MC for V2
QMC1 = np.zeros((2,len(NN))) # QMC for V1
QMC2 = np.zeros((2,len(NN))) # QMC for V2

plt.figure(figsize=(10, 9)) # figsize=(6,4) by default
plt.suptitle("Goal 1: Estimated error" +
             " with confidence $1-" + str(alpha) + "$")
for m in mm:
    X=Payoff(m)
    for j in range(len(NN)):
        MC1[:,j], MC2[:,j] = X.MC(NN[j],alpha)
        QMC1[:,j], QMC2[:,j] = X.QMC(NN[j],alpha)
    # Plot the errors:
    plt.subplot(221)
    plt.loglog(NN,MC1[1,:],'.-',label='$m=$'+str(m))
    plt.subplot(222)
    plt.loglog(NN,QMC1[1,:],'.-',label='$m=$'+str(m))
    plt.subplot(223)
    plt.loglog(NN,MC2[1,:],'.-',label='$m=$'+str(m))
    plt.subplot(224)
    plt.loglog(NN,QMC2[1,:],'.-',label='$m=$'+str(m))

# Finest result (biggest m and N):
text= ("The computed intervals for m =" + str(m)
       + ", N =" + str(NN[-1]) + "and alpha =" + str(alpha) + " are:\n"
       + "V1:  MC: " + str(MC1[0,-1])  + " ± " + str(MC1[1,-1])  + "\n"
       + "     QMC: " + str(QMC1[0,-1]) + " ± " + str(QMC1[1,-1]) + "\n"
       + "V2:  MC: " + str(MC2[0,-1])  + " ± " + str(MC2[1,-1])  + "\n"
       + "     QMC: " + str(QMC2[0,-1]) + " ± " + str(QMC2[1,-1]))

print("The computed intervals for m =", m,
      ", N =", NN[-1],"and alpha =",alpha,"are:")
print("V1:  MC:",MC1[0,-1],"±",MC1[1,-1])
print("    QMC:",QMC1[0,-1],"±",QMC1[1,-1])
print("V2:  MC:",MC2[0,-1],"±",MC2[1,-1])
print("    QMC:",QMC2[0,-1],"±",QMC2[1,-1])

# Error plots(suite):
plt.subplot(221)
plt.loglog(NN,10*NN**(-0.5),"--",label='$1/\sqrt{N}$')
plt.title("Crude Monte Carlo for V1")
plt.xlabel('Sample size N')
plt.ylabel('Estimated error')
plt.legend()

plt.subplot(222)
plt.loglog(NN,3*NN**(-0.5),"--",label='$1/\sqrt{N}$')
plt.loglog(NN,80/NN,"--",label='$1/N$')
```

```
plt.title("Quasi Monte Carlo for V1")
plt.xlabel('Sample size N')
plt.ylabel('Estimated error')
plt.legend()

plt.subplot(223)
plt.loglog(NN,0.8*NN**(-0.5),"--",label='$1/\sqrt{N}$')
plt.title("Crude Monte Carlo for V2")
plt.xlabel('Sample size N')
plt.ylabel('Estimated error')
plt.legend()

plt.subplot(224)
plt.loglog(NN,0.2*NN**(-0.5),"--",label='$1/\sqrt{N}$')
plt.loglog(NN,10*1/NN,"--",label='$1/N$')
plt.title("Quasi Monte Carlo for V2")
plt.xlabel('Sample size N')
plt.ylabel('Estimated error')
plt.legend()

# Plot saving:
plt.tight_layout()
if save_figures == True:
    plt.savefig('graphics/q1error.pdf')

# The interval plots:
plt.figure(figsize=(10,9))
plt.suptitle("Goal 1: Interval of confidence $1-"
             + str(alpha) + "$, m="+str(mm[-1])) #+text?
plt.subplot(221)
plt.xscale('log')
plt.fill_between(NN,+MC1[0,:]+MC1[1,:],MC1[0,:]-MC1[1,:],alpha=0.5)
plt.plot(NN,MC1[0,:],'.-')
plt.title("Crude Monte Carlo for V1")
plt.xlabel('Sample size N')
plt.ylabel('mean and confidence interval')

plt.subplot(222)
plt.xscale('log')
plt.fill_between(NN,QMC1[0,:]+QMC1[1,:],QMC1[0,:]-QMC1[1,:],alpha=0.5)
plt.plot(NN,QMC1[0,:],'.-')
plt.title("Quasi Monte Carlo for V1")
plt.xlabel('Sample size N')
plt.ylabel('mean and confidence interval')

plt.subplot(223)
plt.xscale('log')
plt.fill_between(NN,+MC2[0,:]+MC2[1,:],MC2[0,:]-MC2[1,:],alpha=0.5)
plt.plot(NN,MC2[0,:],'.-')
plt.title("Crude Monte Carlo for V2")
plt.xlabel('Sample size N')
plt.ylabel('mean and confidence interval')

plt.subplot(224)
plt.xscale('log')
plt.fill_between(NN,QMC2[0,:]+QMC2[1,:],QMC2[0,:]-QMC2[1,:],alpha=0.5)
plt.plot(NN,QMC2[0,:],'.-')
plt.title("Quasi Monte Carlo for V2")
plt.xlabel('Sample size N')
```

```python
plt.ylabel('mean and confidence interval')

plt.tight_layout()
# Plot saving:
if save_figures == True:
    plt.savefig('graphics/q1interval.pdf')
```

D q2.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Dec 25 17:10:21 2021

@author: benoitmuller
question 2.
"""

import numpy as np
#import scipy as sp
#import scipy.stats  as st
import matplotlib.pyplot as plt
#import time
#import numpy.random as rnd
#from mes_stats import RandomVariable
#from sobol_new import generate_points
from Payoff import Payoff

"""paramètres ou ça marche bien:
np.random.seed(54321) (7,14) (5,6)
"""
# Fix the seed:
np.random.seed(12345)

# File saving options:
save_figures = True # Change the value if needed
if (save_figures==True and
    input("Do you really want to save figures"+
          " into files?\n(yes/no): ") == "no"):
    save_figures = False

alpha=0.01
NN=(2**np.arange(7,12)).astype(int) #(7,14)
mm=(2**np.arange(5,6)).astype(int) #(5,10)
MC1 = np.zeros((2,len(NN)))
MC2 = np.zeros((2,len(NN)))
QMC1 = np.zeros((2,len(NN)))
QMC2 = np.zeros((2,len(NN)))

plt.figure(figsize=(10, 9)) # figsize=(6,4) by default
plt.suptitle("Goal 2: With pre-integration, estimated error" +
             " with confidence $1-" + str(alpha) + "$")
for m in mm:
    X=Payoff(m)
    for j in range(len(NN)):
        MC1[:,j], MC2[:,j] = X.PIMC(NN[j],alpha,order=5)
        QMC1[:,j], QMC2[:,j] = X.PIQMC(NN[j],alpha,order=5,K=10)
    plt.subplot(221)
```

```python
    plt.loglog(NN,MC1[1,:],'.-',label='$m=$'+str(m))
    plt.subplot(222)
    plt.loglog(NN,QMC1[1,:],'.-',label='$m=$'+str(m))
    plt.subplot(223)
    plt.loglog(NN,MC2[1,:],'.-',label='$m=$'+str(m))
    plt.subplot(224)
    plt.loglog(NN,QMC2[1,:],'.-',label='$m=$'+str(m))

# Finest result (biggest m and N):
print("The computed intervals for m =", m,
      ", N =", NN[-1],"and alpha =",alpha,"are:")
print("V1:  PIMC:",MC1[0,-1],"±",MC1[1,-1])
print("    PIQMC:",QMC1[0,-1],"±",QMC1[1,-1])
print("V2:  PIMC:",MC2[0,-1],"±",MC2[1,-1])
print("    PIQMC:",QMC2[0,-1],"±",QMC2[1,-1])

# Error plots:
plt.subplot(221)
plt.loglog(NN,NN**(-0.5),"--",label='$1/\sqrt{N}$')
plt.title("Crude Monte Carlo for V1")
plt.xlabel('Sample size N')
plt.ylabel('Estimated error')
plt.legend()

plt.subplot(222)
plt.loglog(NN,NN**(-0.5),"--",label='$1/\sqrt{N}$')
plt.loglog(NN,10/NN,"--",label='$1/N$')
plt.title("Quasi Monte Carlo for V1")
plt.xlabel('Sample size N')
plt.ylabel('Estimated error')
plt.legend()

plt.subplot(223)
plt.loglog(NN,NN**(-0.5),"--",label='$1/\sqrt{N}$')
plt.title("Crude Monte Carlo for V2")
plt.xlabel('Sample size N')
plt.ylabel('Estimated error')
plt.legend()

plt.subplot(224)
plt.loglog(NN,NN**(-0.5),"--",label='$1/\sqrt{N}$')
plt.loglog(NN,10*1/NN,"--",label='$1/N$')
plt.title("Quasi Monte Carlo for V2")
plt.xlabel('Sample size N')
plt.ylabel('Estimated error')
plt.legend()

plt.tight_layout()
if save_figures == True:
    plt.savefig('graphics/q2error.pdf')

# The interval plot:
plt.figure(figsize=(10,9))
plt.suptitle("Goal 1: With pre-integration, interval of confidence $1-"
             + str(alpha) + "$, m="+str(mm[-1]))
plt.subplot(221)
plt.xscale('log')
plt.fill_between(NN,+MC1[0,:]+MC1[1,:],MC1[0,:]-MC1[1,:],alpha=0.5)
plt.plot(NN,MC1[0,:],'.-')
```

```
plt.title("Crude Monte Carlo for V1")
plt.xlabel('Sample size N')
plt.ylabel('mean and confidence interval')

plt.subplot(222)
plt.xscale('log')
plt.fill_between(NN,QMC1[0,:]+QMC1[1,:],QMC1[0,:]-QMC1[1,:],alpha=0.5)
plt.plot(NN,QMC1[0,:],'.-')
plt.title("Quasi Monte Carlo for V1")
plt.xlabel('Sample size N')
plt.ylabel('mean and confidence interval')

plt.subplot(223)
plt.xscale('log')
plt.fill_between(NN,+MC2[0,:]+MC2[1,:],MC2[0,:]-MC2[1,:],alpha=0.5)
plt.plot(NN,MC2[0,:],'.-')
plt.title("Crude Monte Carlo for V2")
plt.xlabel('Sample size N')
plt.ylabel('mean and confidence interval')

plt.subplot(224)
plt.xscale('log')
plt.fill_between(NN,QMC2[0,:]+QMC2[1,:],QMC2[0,:]-QMC2[1,:],alpha=0.5)
plt.plot(NN,QMC2[0,:],'.-')
plt.title("Quasi Monte Carlo for V2")
plt.xlabel('Sample size N')
plt.ylabel('mean and confidence interval')

plt.tight_layout()

if save_figures == True:
    plt.savefig('graphics/q2interval.pdf')
```