QMC integration of non-smooth functions: application to pricing exotic options

Stochastic Simulations - Mini-project

Huguelet Fanny, Durussel Shad

January 12, 2020

1 Introduction

This project aims at investigating the integration of functions by the mean of stochastic methods. More specifically, Monte Carlo methods can be used to approximate a multiple integral and quasi-Monte Carlo is a popular method for high-dimensional problems. However, in this project, we will face non-smooth functions that compromise the guarantees on quasi-Monte Carlo. The task of this project is to propose a trick that allows us to counter the problems of kicks or jumps, present in the usual payoff functions.

2 Application to option pricing

The given problem of the project is to estimate the price of an Asian option with maturity $T > 0$ based on the stock price S which is the solution of the following stochastic differential equation.

$$\begin{cases} dS = rSdt + \sigma S dw_t \\ S(0) = S_0 \end{cases} \tag{1}$$

The solution of this SDE can be expressed by:

$$S_t = S_0 exp\left(\left(r - \frac{\sigma^2}{2}\right)t + \sigma w_t\right) \tag{2}$$

With w_t being a standard Wiener process, S_0 is the initial value of the underlying asset, K is the strike price, r is the interest rate and σ is the volatility. We introduce the discretization $t_i = i\Delta t$ with $\Delta t = T/m$ for $m \in \mathbb{N}$. For pricing Asian option, we are given two different payoff functions Ψ_i, $i = 1, 2$.

$$\Psi_1(w_{t_1}, ..., w_{t_m}) = \left(\frac{1}{m}\sum_{i=1}^{m} S_{t_i}(w_{t_i}) - K\right)_+ = \left(\frac{1}{m}\sum_{i=1}^{m} S_{t_i}(w_{t_i}) - K\right)\mathbb{I}_{\left\{\frac{1}{m}\sum_{i=1}^{m} S_{t_i}(w_{t_i}) - K\right\}}$$

$$\Psi_2(w_{t_1}, ..., w_{t_m}) = \mathbb{I}_{\left\{\frac{1}{m}\sum_{i=1}^{m} S_{t_i}(w_{t_i}) - K\right\}}$$

The goal here is to compute the value $V_i = e^{-rT}\mathbb{E}[\Psi_i(w_{t_1}, ..., w_{t_m})]$, of these options for $i = 1, 2$. This gives us the following integration problem:

$$V_i = \frac{e^{-rT}}{(2\pi)^{m/2}\sqrt{\det(C)}} \int_{\mathbb{R}^m} \Psi_i(w) e^{-\frac{1}{2}w^\top C^{-1} w} dw \tag{3}$$

3 Method

3.1 Rewriting the integral

In order to evaluate S_t, we need to sample w_t. As w_t is a Wiener process, we can sample it using Gaussian increments.

$$w_0 = 0$$
$$w_{t_i} = w_{t_{i-1}} + Y_i \qquad\qquad Y_i \sim \mathcal{N}(0, t_i - t_{i-1})$$

With the time discretization $t_i = i\Delta t$ and $\Delta t = T/m$. As suggested in the handout, we will first rewrite integral (3) in terms of independent standard normal random variables of dimension 1 and then in terms of independent uniform random variable. We can express w in terms of z_i, $i = 1, ..., m$ as follow.

$$z_i = \frac{w_{t_i} - w_{t_{i-1}}}{\sqrt{t_i - t_{i-1}}} \qquad\qquad z_i \sim \mathcal{N}(0,1)$$

The stock price S_{t_i} becomes then :

$$S_{t_i} = S_0 exp\left(\left(r - \frac{\sigma^2}{2}\right)t_i + \sigma\left(w_{t_{i-1}} + \sqrt{t_i - t_{i-1}}z_i\right)\right)$$

In order to rewrite (3), we describe w as a linear system.

$$w = Az, \qquad z = (z_1, ..., z_m)^\top$$

$$A_{ij} = \begin{cases} \sqrt{t_j - t_{j-1}} & j \le i \\ 0 & i < j \end{cases}$$

Note that A is then a factorization of C since $AA^\top = C$. Thus $\det(A) = \sqrt{\det(C)}$ and the value V_i becomes:

$$V_i = \frac{e^{-rT}}{(2\pi)^{m/2}\sqrt{\det(C)}}\int_{\mathbb{R}^m}\Psi_i(w)e^{-\frac{1}{2}w^\top C^{-1}w}dw \;=\; \frac{e^{-rT}}{(2\pi)^{m/2}}\int_{\mathbb{R}^m}\Psi_i(Az)e^{-\frac{1}{2}z^\top z}dz \qquad (4)$$

The next step to recast the integral into an hypercube is to express z in terms of uniform random variables in $[0,1]$. By inversion sampling:

$$z_i = \Phi^{-1}(u_i) \qquad\qquad u_i \sim \mathcal{U}(0,1)$$

Φ is the cumulative distribution function of $\mathcal{N}(0,1)$. Thus, the stock price becomes:

$$S_{t_i} = S_0 exp\left(\left(r - \frac{\sigma^2}{2}\right)t_i + \sigma\left(w_{t_{i-1}} + \sqrt{t_i - t_{i-1}}\Phi^{-1}(U_i)\right)\right) \qquad (5)$$

We can now explicitly write $\Phi^{-1}(u_i)$ and its derivative to help rewrite integral (3).

$$\Phi^{-1}(u_i) = \sqrt{2}\,\mathrm{erf}^{-1}(2u_i - 1)$$
$$\frac{d\Phi^{-1}(u_i)}{du_i} = \sqrt{2\pi}e^{\left(\mathrm{erf}^{-1}(2u_i-1)\right)^2} = \sqrt{2\pi}e^{\frac{1}{2}\left(\Phi^{-1}(u_i)\right)^2} = \sqrt{2\pi}e^{\frac{1}{2}z_i^2}$$

with $\mathrm{erf}(x) = \frac{1}{\sqrt{\pi}}\int_{-x}^{x} e^{-t^2}dt$.

Finally, let $z = \phi^{-1}(u) = \left(\Phi^{-1}(u_1), ..., \Phi^{-1}(u_m)\right)$ with $u = (u_1, ..., u_m)$. The value V_i becomes:

$$V_i = \frac{e^{-rT}}{(2\pi)^{m/2}}\int_{\mathbb{R}^m}\Psi_i(Az)e^{-\frac{1}{2}z^\top z}dz \;=\; e^{-rT}\int_{[0,1]^m}\Psi_i(A\phi^{-1}(u))du \qquad (6)$$

3.2 Estimate without pre-integration

First, we use a crude Monte Carlo (CMC) and a randomized Quasi-Monte Carlo (QMC), without the pre-integration trick. We generate samples $(u^{(1)}, ..., u^{(N)})$ over the m-dimensional unit cube $[0,1]^m$, either randomly inside the hypercube for CMC or with low discrepancy for QMC. The estimation of the integral is as usual:

$$\int_{[0,1]^m} \Psi_i(u)du \approx \frac{1}{N}\sum_{j=1}^{N} \Psi_i(u^{(j)}) =: \hat{\mu}_N \tag{7}$$

Note that to have the final price of the Asian option, we need to multiply the final output, which approximate the integral, by e^{-rT}. With $T > 0$, the maturity and r, the interest rate.

For the Crude Monte Carlos method, the rate of convergence is $O(N^{-1/2})$ and is independent of the dimension m of the problem.

For the randomized QMC estimator, we sampled the hypercube $[0,1]^m$ using a Sobol sequence. We used the code given for lab 9 of the course, based on [2]. This method allows to sample a sequence in high dimension, hence it is adapted to our case. Randomized QMC is estimated by averaging over K randomized shifts (we chose K = 20). This Sobol sequence is a low discrepancy sequence, i.e. it achieves the bound $D_N^*(P) = O\left(\frac{(\log N)^m}{N}\right)$. The QMC algorithm can outfit an asymptotic confidence interval:

$$I_\alpha = \left[\hat{\mu}^{QMC} - c_{1-\alpha/2}\frac{\hat{\sigma}^{QMC}}{\sqrt{K}},\ \hat{\mu}^{QMC} + c_{1-\alpha/2}\frac{\hat{\sigma}^{QMC}}{\sqrt{K}}\right].$$

Thus we will plot the error estimation for CMC and QMC approximations, express as follow:

$$\text{error}^{CMC} = \frac{\hat{\sigma}^{CMC}}{\sqrt{N}}, \qquad \text{error}^{QMC} = \frac{\hat{\sigma}^{QMC}}{\sqrt{K}}, \tag{8}$$

with $\hat{\sigma}^{CMC}$ and $\hat{\sigma}^{QMC}$ the estimated standard deviation of the output data.

3.3 Estimate with pre-integration

First, we generate (Q)MC points over $[0,1]^{m-1}$ and we need to choose a direction $j \in \{1, ..., m\}$ to perform the pre-integration:

$$p(z_{-j}) = \int_{\psi(z_{-j})}^{+\infty} \Psi_i(A(z_j, z_{-j})) \cdot \rho_j(z_j)dx_j \tag{9}$$

with the same notation as in the handout: $z = (z_j, z_{-j})$ are random variable $\mathcal{N}(0,1)$. As above, A is such that $Az = w$ and ρ_j is the normal standard density function.

We can write $\Psi_i(w) = \theta_i(w)\mathbb{I}_{\{\phi(w)\}}$. The lower bound for the integral (9) is then given by finding the value of z_j such that $\phi(w) = 0$. This is

$$\frac{1}{m}\sum_{i=1}^{m} S_{t_i}(z_i) - K = 0 \tag{10}$$

with $S_{t_i}(z_i)$ define as following where A_i is the i^{th} row of A.

$$S_{t_i}(z_i) = S_0 exp\left(\left(r - \frac{\sigma^2}{2}\right)t_i + \sigma A_i z\right)$$

3.3.1 Initial attempt

First, we try to use $j = m$, the last element. The idea behind is that for different values of z_m we only need to recompute the last value of the mean: S_{t_m}. We don't need to recompute all the different path $S_0, ..., S_{t_{m-1}}$ since they are all identical. The lower bound for the integral (9) is then given explicitly by the following.

$$z_m^* = \frac{1}{\sigma \Delta t} \left(\ln(S_{t_m}^*/S_0) - (r - \sigma^2/2)T \right) - \frac{w_{t_{j-1}}}{\Delta t},$$

with,

$$S_{t_m}^* = mK - \sum_{i=1}^{m-1} S_{t_i}(z_i)$$

But with this choice for the direction the results weren't really significant. This is because the value at index m isn't the one with the greatest contribution in the payoff function. So the results are not false and efficient in computing-time, but the problem of jumping and kicking persists.

3.3.2 Implemented algorithm

So, as in the paper [1], we choose to use an SVD algorithm to decompose the matrix of covariance of the Wiener process C. Thus $C = USV$ as usual and we use $A = US^{1/2}$ in the integral with normally distributed increment (6).

With this decomposition, the first contribution, i.e. the direction $j = 1$, is the biggest one. This is since SVD sorts the eigenvalues by order of magnitude. Thus the pre-integration (9) is done with respect to x_1.

We use the newton algorithm to find the root of equation (10) and a scipy algorithm to compute the integral (9). Moreover, we use the matrix A from the SVD decomposition, in the implementation of the function to integrate.

The CMC and QMC algorithms remain the same for the $m - 1$ remaining points.

3.4 Additional variance reduction technique

The course offers us several choices of variance reduction techniques. Some of these techniques tend to better cover the variability of the random variables involved (stratification, Latin Hypercube sampling). As the previous parts use the quasi-Monte Carlo algorithm, which aims at the same thing, we chose to use another approach here. That let us with antithetic variables, importance sampling, and control variate. In fact, we can explicitly compute the mean of $\frac{1}{m}\sum_{i=1}^{m} S_{t_i}$ which is higher than K in our case. This makes importance sampling unnecessary in our case. As the mean is known, we choose to use control variate. Taking the notation from the course, we want to compute the mean of $Z_i = \Psi_i(w)$.

As a control variate, we can choose: $Y = \frac{1}{m}\sum_{i=1}^{m} S_{t_i}$ with expectation:

$$\mathbb{E}[Y] = \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} S_{t_i}\right] = \frac{S_0}{m}\sum_{i=1}^{m} e^{rt_i} \tag{11}$$

The algorithm is identical as in the course and the error approximation is the same as CMC but with σ^{CV} the standard deviation of the output data for the control variate algorithm.

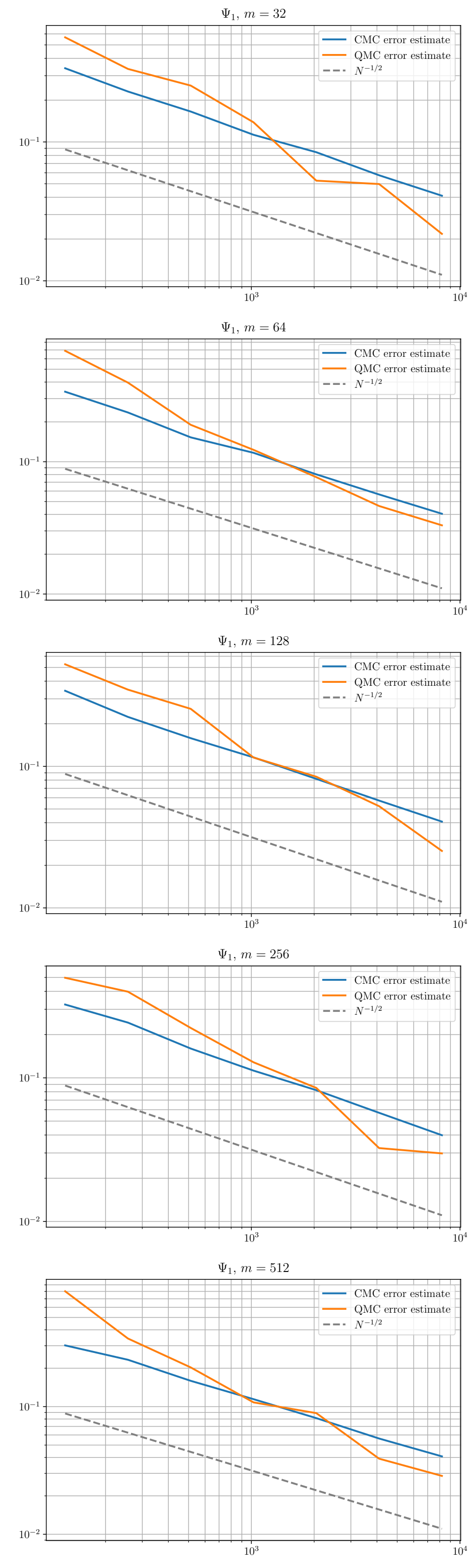

Figure 1: Error estimate with respect of N for Ψ_1. Each graph represents another dimension m.

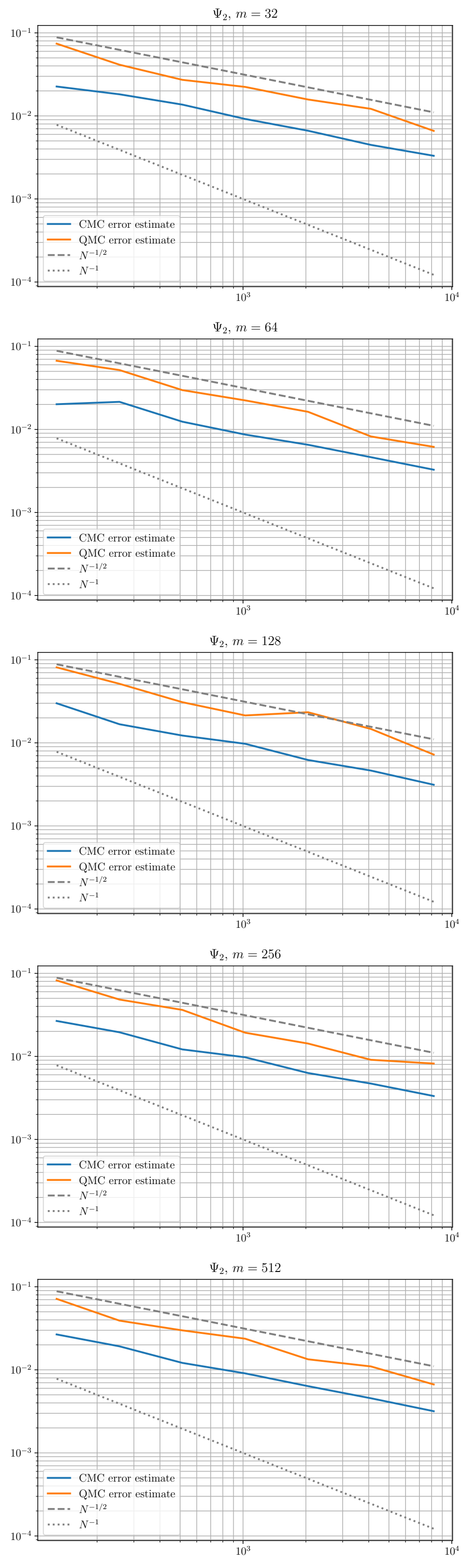

Figure 2: Error estimate with respect of N for Ψ_2. Each graph represents another dimension m.

4 Results

So we run our algorithms with the following parameters: $K = 100$, $S_0 = 100$, $r = 0.2$, $\sigma = 0.1$ and $T = 1$. For the time discretization we take $m = 2^{5,\cdots,9}$ and we use $N = 2^{7,\cdots,13}$ samples for CMC and QMC for both part 1 and 2.

The code for the three parts is in appendix. In the code, we need to change the *types* variable to 1 or 2, in order to have the two different payoffs functions Ψ_1 or Ψ_2.

4.1 Part 1. Estimate without pre-integration

The results are shown in figure 1 for Ψ_1 and figure 2 for Ψ_2. We observe that the Crude Monte Carlos method seems to follow the convergence rate of $O(N^{-1/2})$, as expected, for both payoff functions.

For the randomized quasi-Monte Carlos method, we observe, first for Ψ_1, a good improvement of the convergence rate, which is around $O(N^{-0.7})$ as we will see later. But for Ψ_2, we don't see any really significant improvement. This may be due to the fact that, in this second case, the payoff function is discontinuous. Whereas in the first case the payoff function only had kicks and no jumps. A summary of the orders of convergence will be given later to compare all the different used methods.

The plots in figures 1 and 2 are produced with the same code as for the following part, but simply without the estimates for pre-intingration and control variate.

4.2 Part 2. Estimate with pre-integration

As we can see in the code in the appendix, we implement the pre-integration trick. Note that the CMC and QMC function are similar to before, with the only difference that they generate randomized points over only $[0, 1]^{m-1}$.

The results are shown in figures 3 and 4 for the two payoff functions. In both cases, one can observe the clear improvement that is brought by the pre-integration trick. The error estimates have been clearly diminished. But moreover, the order of convergence (i.e. the slope) of the QMC method with pre-integration is slightly better.

All the orders of convergence are approximate (with interpolation of the error estimates) and synthesized in figures 5 and 6. For the first payoff, we can clearly see the improvement of the order brought first with the QMC method and then with the pre-integration trick. For the second payoff, which is discontinuous, it can be seen that the most significant improvement of the order occurs in the case of QMC with pre-integration.

In both cases, the CMC order of convergence is already around $O(N^{-1/2})$, so the pre-integration trick for CMC can't improve the order (i.e. the slope on the plot figure 1 or 2) anymore. We note that the QMC error estimate with pre-integration tends to achieve the bounds of $O\left(\frac{(\log N)^{2(m-1)}}{N}\right)$ of the randomized QMC algorithms for smooths functions. The exponent $2(m-1)$ in the previous expression may be an explanation of why the slope of the error function (see figure 5 and 6) tend to grow in high dimension. There is just slight noise in the results when m is smaller.

At right in figures 3 and 4, we have plotted the computational time of each method. All methods are in $O(N)$, but it's important to note that the methods with the pre-integration step take much longer. This is due to the two time-consuming algorithms "Newton" and "Quad" (for integration) in addition to a full re-computation of $\Phi_i(Az)$ for each iteration of the two cited methods. Note that time naturally increases as the dimension m increases.

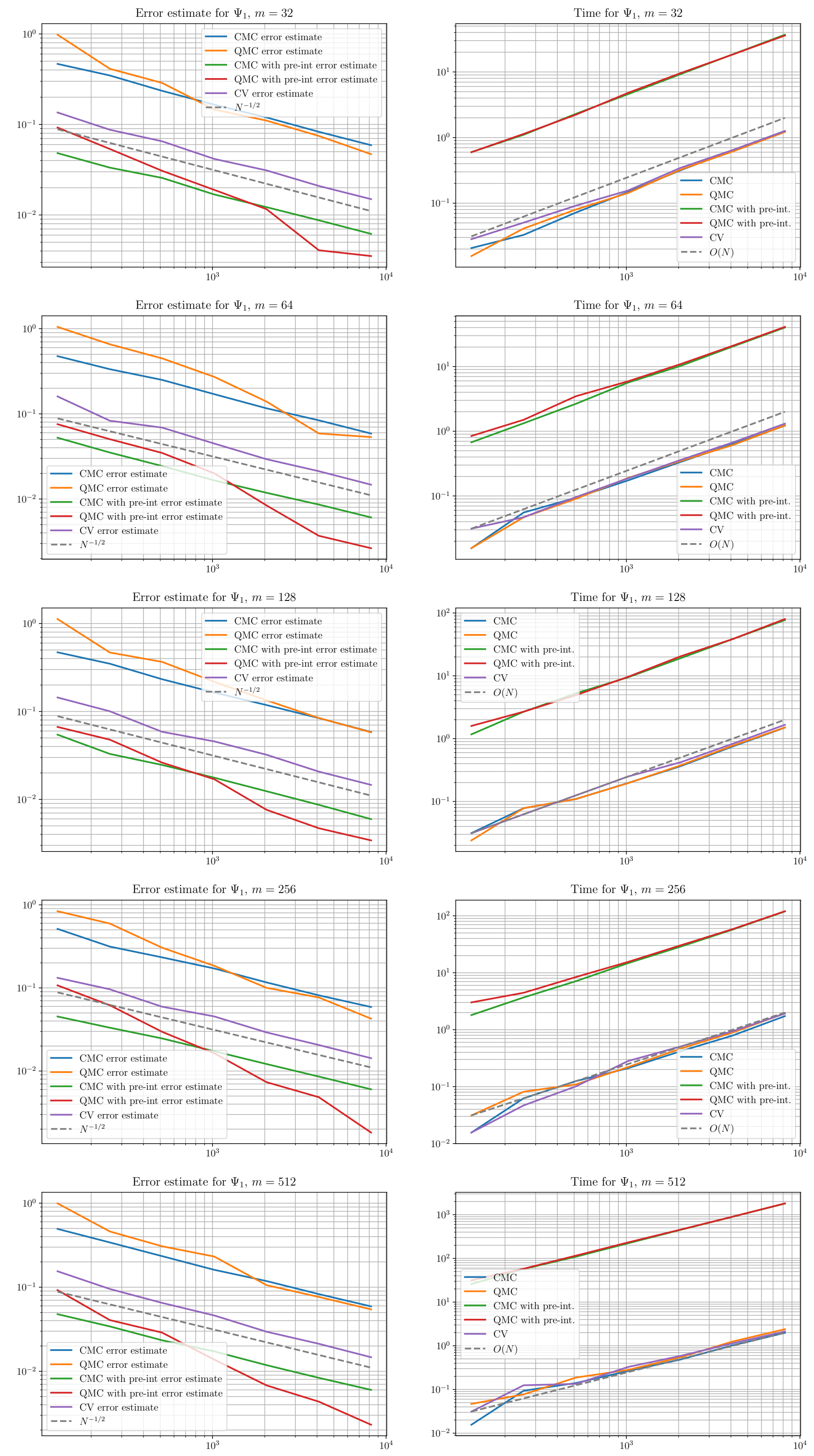

Figure 3: Synthesized of all the methods for Ψ_1. At left the error estimate with respect of N, at right the computational time.

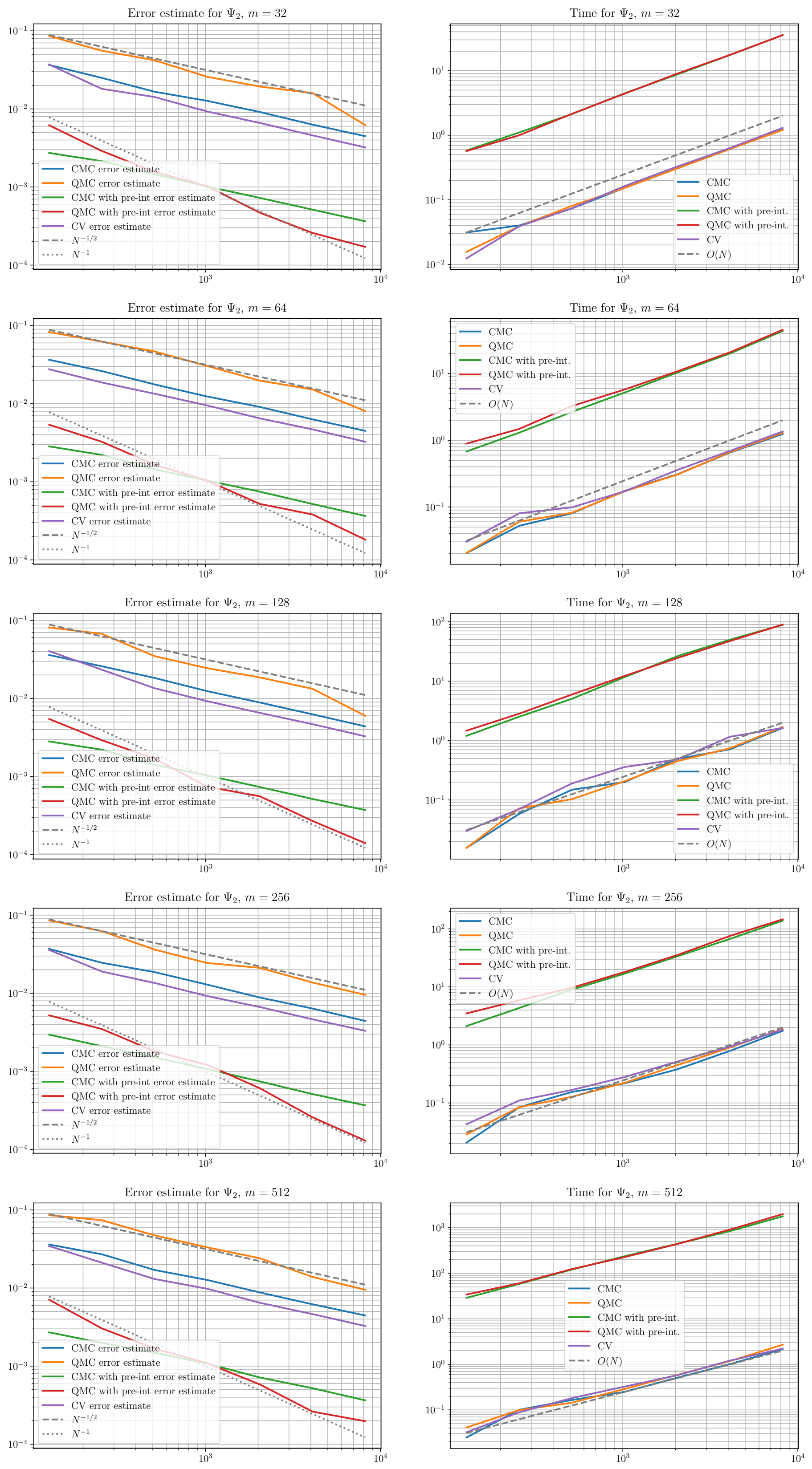

Figure 4: Synthesized of all the methods for Ψ_2. At left the error estimate with respect of N, at right the computational time.

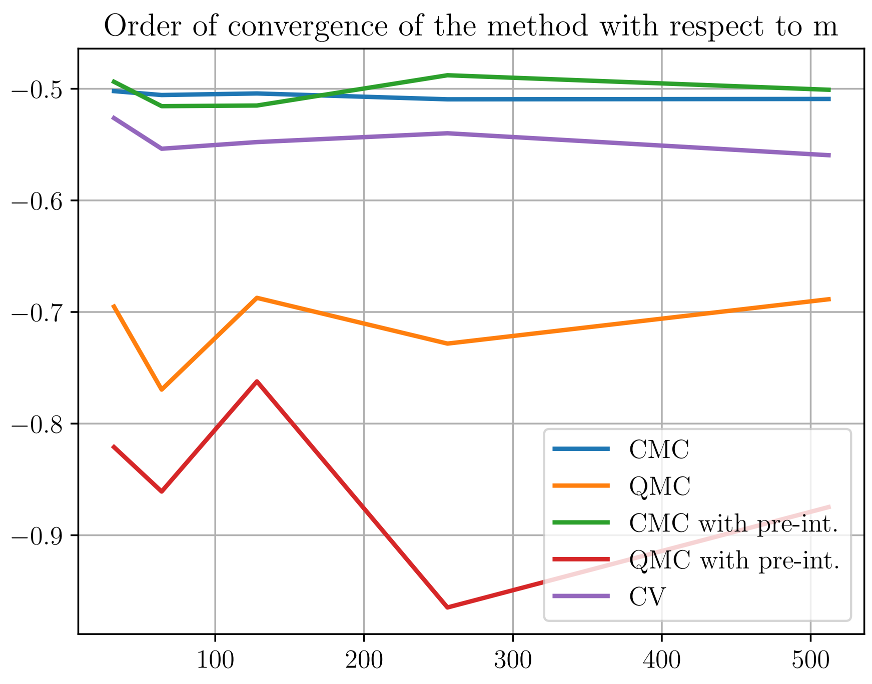

Figure 5: Order of convergence for Ψ_1 of each method. The values on the x-axis are m and on the y-axis, it's the slope or the order of convergence: $O(N^{\text{order}})$.

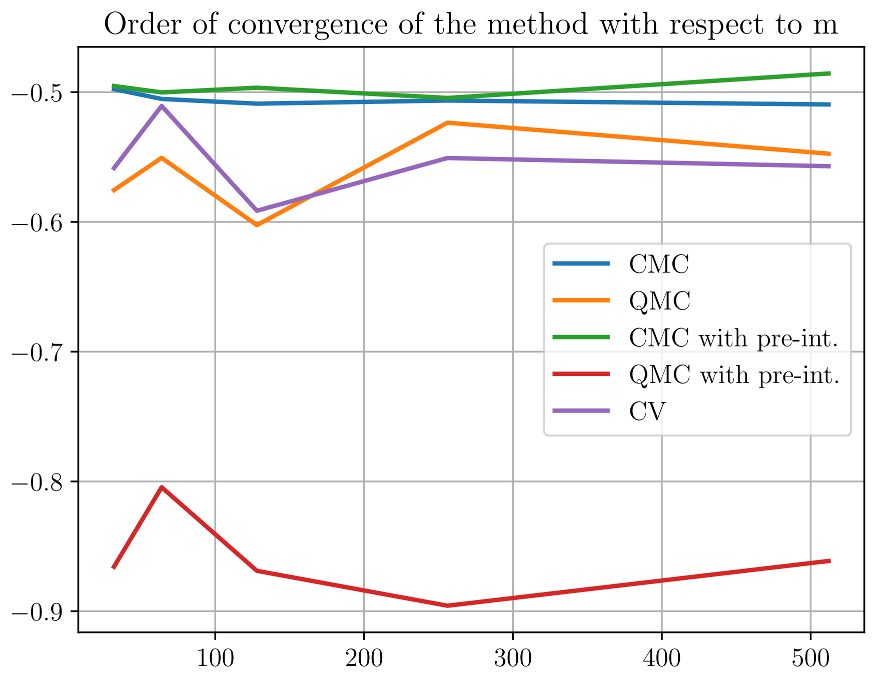

Figure 6: Order of convergence for Ψ_2 of each method. The values on the x-axis are m and on the y-axis, it's the slope or the order of convergence: $O(N^{\text{order}})$.

4.3 Part 3. Additional variance reduction technique

As we can see in figure 3, the estimated error for control variate is better than the one of CMC. This difference is a little less visible for Ψ_2 in figure 4. We also note that, although the results are better, the asymptotic order of convergence is the same than CMC. To see why, let us rewrite the variance achieved by the control variate algorithm.

$$\operatorname{Var}\left(\tilde{Z}_{\alpha_{opt}}\right) = Var(Z)\left(1 - \operatorname{Corr}(Z,Y)^2\right) \tag{12}$$

Here $\tilde{Z}_\alpha = Z + \alpha\,(Y - \mathbb{E}[Y])$. It is evident from this expression that the variance reduction doesn't depend on the number of sample. Thus the order of convergence doesn't change. The variance reduction is linked with the correlation between $Y = \sum_{i=1}^{m} S_{t_i}$ and Z. Since Ψ_2 is an indicator function, we indeed expect that the correlation between Y and $Z = \Psi_2$ will be smaller than the correlation between Y and $Z = \Psi_1$. This is coherent with our result as we achieve a better variance reduction with Ψ_1 than with Ψ_2.

5 Conclusion

We can then conclude that, in general, the pre-integration trick allows us to greatly improve the CMC and QMC error, in the case of a non-smooth function. In addition, the order of convergence is clearly improving in the case of the QMC method.

In order to improve this project, one could imagine trying to improve the computation time of the two methods with pre-integration. To do so, it would be possible to use another method to find the decomposition A of the matrix C. The purpose is to have less calculation in each re-computation of S in the evaluation of the function Ψ_i in the functions Newton (for the root-finding routing) and Quad (for integration).

Outside the framework of this project, it would be interesting to use a different model than Black Scholes for the stock price, in the purpose to compute efficiency option price. For example, the Cox–Ingersoll–Ross (CIR) model, or Heston model which one allows us to use stochastic volatility which represents better the reality. But those models are more complicated to implement and do not necessarily have an explicit form for the stock price, as needed in this project.

References

[1] Andreas Griewank, Frances Y. Kuo, Hernan Leövey, and Ian H. Sloan. High dimensional integration of kinks and jumps-smoothing by preintegration. *Journal of Computational and Applied Mathematics*, 344:259–274, 2018.

[2] Stephen Joe and Frances Y. Kuo. Constructing sobol sequences with better two-dimensional projections. *SIAM Journal on Scientific Computing*, 30(5):2635–2654, 2008.

A Appendice

Python final code for all 3 parts and that generated plots on figures 3, 4, 5 and 6:

```python
import numpy as np
from numpy import matlib
import scipy.stats as st
import matplotlib.pyplot as plt

#file to generate Sobol low discrepancy sequance (admits dimensions up to 21201)
import sobol_new as sn
from scipy.integrate import quad
from scipy.optimize import newton
from scipy.optimize import root_scalar
import time
import sys

# Ploting parameters
from matplotlib import rc
rc('font',**{'family':'serif','serif':['Computer Modern Roman'],
     'size' : '12'})
rc('text', usetex=True)
rc('lines', linewidth=2)
plt.rcParams['axes.facecolor']='w'
import matplotlib
matplotlib.rcParams['text.latex.preamble'] = [
    r'\usepackage{amsmath}',
    r'\usepackage{amssymb}']


########################################
# PART 1 and 3 without pre-integration #
########################################

# Payoff funtion
def genPsi(type, xi, t, r, sigma, S0, K):
    dt = np.diff(t)
    W = np.cumsum(np.sqrt(dt)*xi)
    S = S0*np.exp((r - sigma**2/2)*t[1:] + sigma*W)
    if type == 1:
        val = (np.abs(np.mean(S) - K) + (np.mean(S) - K))/2
    elif type == 2:
        val = (np.mean(S) - K) > 0
    # return value of the payoff function and the mean of S (for CV)
    return val, np.mean(S)

# Common part of CMC, QMC or CV
def evaluate(type, x, r=0.1, sigma=0.1, T=1, S0=100, K=100):
    d = x.shape[0]
    M = x.shape[1]
    val = np.zeros(M)
    S = np.zeros(M)
    t = np.linspace(0, T, d+1)
    for j in range(M):
        xi = st.norm.ppf(x[:,j])
        val[j], S[j] = genPsi(type, xi, t, r, sigma, S0, K)
    return val, S

# Crude Monte Carlos
def CMC(type, d, M):
    x = np.random.random((d, M))
    data, _ = evaluate(type, x)
    est = np.mean(data)
    err_est = np.std(data)/np.sqrt(M)
    return est, err_est

```

```
# Randomized Quasi-Monte Carlos
def QMC(type, d, N, K):
    x = sn.generate_points(N, d, 0) # sobol sequence generator
    x = x.T
    data = np.zeros(K)
    for i in range(K):
        dat, _ = evaluate(type, np.mod(x + matlib.repmat(np.random.random(
            size=(int(d),1)), 1, int(N)),1))
        data[i] = np.mean(dat)
    est = np.mean(data)
    err_est = 3*np.std(data)/np.sqrt(K)
    return est, err_est

# Control Variate for part 3
def CV(type, d, N, N_bar, r=0.1, S0=100, T=1):
    # pilot run
    x1 = np.random.random((d, N_bar))
    data1, S2 = evaluate(type, x1)
    t = np.linspace(0,T,d+1)[1:]
    mean = S0/d*np.sum(np.exp(r*t))
    C = np.cov(data1,S2)
    a_opt = -C[0,1]/C[1,1]
    # Monte Carlo
    x1 = np.random.random((d, N))
    data1, S2 = evaluate(type, x1)
    Z_tilde = data1 + a_opt*(S2-mean)
    est = np.mean(Z_tilde)
    err_est = np.sqrt(np.var(Z_tilde)/N)
    return est, err_est


###########################################
# PART 2 CMC and QMC with pre-integration #
###########################################

# Integration w.r.t the first variable x1: the choosen direction is j = 1
def p(type, xi, t, r, sigma, S0, K, A):
    dt = np.diff(t)
    # function to find the root:
    fun = lambda x: np.mean(S0*np.exp((r - sigma**2/2)*t[1:] \
        + sigma*A@np.concatenate(([x],xi)).T)) - K
    # find the root of fun for the lower bound of the integration:
    value = newton(fun,0)

    # definition of the function to integrate:
    if type == 1:
        f = lambda x: np.maximum(np.mean(S0*np.exp((r - sigma**2/2)*t[1:] \
            + sigma*A@np.concatenate(([x],xi)).T)) - K, 0)*np.exp(-0.5*x**2) \
            /np.sqrt(2*np.pi)
    elif type == 2:
        f = lambda x: ((np.mean(S0*np.exp((r - sigma**2/2)*t[1:] \
            + sigma*A@np.concatenate(([x],xi)).T))- K) > 0)*np.exp(-0.5*x**2) \
            /np.sqrt(2*np.pi)
    # integration:
    val, _ = quad(f, value, np.inf)
    return val


# Common part of CMC and QMC
def pre_int_evaluate(type, x, r=0.1, sigma=0.1, T=1, S0=100, K=100):
    d = x.shape[0]
    M = x.shape[1]
    val = np.zeros(M)
    t = np.linspace(0, T, d+2)
    d = t.shape[0]
    C = np.zeros((d-1,d-1))
    for i in range(1,d):
        for j in range(1,d):
            C[i-1,j-1] = min(t[i],t[j]) # C is symmetric
```

```python
    U, s, Vh = np.linalg.svd(C) # SVD decomposition of C
    A = -U*np.sqrt(s) # then A@A.T = C. The decompostion is not unique
    # we take the minus since we want an increasing function after in p.

    for j in range(M):
        xi = st.norm.ppf(x[:,j])
        val[j] = p(type, xi, t, r, sigma, S0, K, A)
    return val

# CMC usual algorithm
def pre_int_CMC(type, d, M):
    x = np.random.random((d-1, M))
    data = pre_int_evaluate(type, x)
    est = np.mean(data)
    err_est = np.std(data)/np.sqrt(M)
    return est, err_est

# QMC usual algorithm
def pre_int_QMC(type, d, N, K):
    x = sn.generate_points(N, d-1, 0) # sobol sequence generator
    x = x.T
    data = np.zeros(K)
    for i in range(K):
        dat = pre_int_evaluate(type, np.mod(x + matlib.repmat(np.random.random(
        size=(int(d-1),1)), 1, int(N)),1))
        data[i] = np.mean(dat)
    est = np.mean(data)
    err_est = 3*np.std(data)/np.sqrt(K)
    return est, err_est




###################
# Computing part: #
###################

Mlist = 2**np.arange(5,10)
Nlist = 2**np.arange(7,14)

nM = np.size(Mlist)
nN = np.size(Nlist)

# mean value of the return to check the sanity of the results
cmc_mean, qmc_mean          = np.zeros(nM), np.zeros(nM)
pre_cmc_mean, pre_qmc_mean  = np.zeros(nM), np.zeros(nM)
cv_mean = np.zeros(nM)

# time of each methods
times_cmc, times_qmc        = np.zeros(nN), np.zeros(nN)
times_pre_cmc,times_pre_qmc = np.zeros(nN), np.zeros(nN)
times_cv = np.zeros(nN)

# estimated error of each methods
cmc_est, cmc_err_est = np.zeros(nN), np.zeros(nN)
qmc_est, qmc_err_est = np.zeros(nN), np.zeros(nN)
cmc_est_pre, cmc_err_est_pre = np.zeros(nN), np.zeros(nN)
qmc_est_pre, qmc_err_est_pre = np.zeros(nN), np.zeros(nN)
cv_est, cv_err_est = np.zeros(nN), np.zeros(nN)

# final order of convergence for each dimentions
order_cmc = np.zeros(nM)
order_qmc = np.zeros(nM)
order_cmc_pre = np.zeros(nM)
order_qmc_pre = np.zeros(nM)
order_cv = np.zeros(nM)


# figures
```

```python
fig, axes = plt.subplots(nrows = 5, ncols = 2, figsize = (16,30))
axes = axes.flatten()

# type od payoff function Psi:
types = 1 # change to 1 or 2

KK = 20 # for QMC algorithm
for j in range(nM):
    print("Computation for m = " +str(Mlist[j]))
    for i in range(nN):
        sys.stdout.write(" n = " +str(Nlist[i]) +"\r")
        sys.stdout.flush()

        # All methods:
        start = time.time()
        cmc_est[i], cmc_err_est[i] = CMC(types, Mlist[j], Nlist[i])
        time1 = time.time()
        times_cmc[i] = time1-start

        qmc_est[i], qmc_err_est[i] = QMC(types, Mlist[j], Nlist[i]/KK, KK)
        time2 = time.time()
        times_qmc[i] = time2-time1

        cmc_est_pre[i], cmc_err_est_pre[i] = pre_int_CMC(types, Mlist[j],
            Nlist[i])
        time3 = time.time()
        times_pre_cmc[i] = time3-time2

        qmc_est_pre[i], qmc_err_est_pre[i] = pre_int_QMC(types, Mlist[j],
            Nlist[i]/KK, KK)
        time4 = time.time()
        times_pre_qmc[i] = time4-time3

        cv_est[i], cv_err_est[i] = CV(types, Mlist[j], Nlist[i],
            int(Nlist[i]/2**4))
        time5 = time.time()
        times_cv[i] = time5 - time4

    # compute approximation of convergence rate
    coeff = np.polyfit(np.log(Nlist), np.log(cmc_err_est),deg=1)
    order_cmc[j] = coeff[0]
    coeff = np.polyfit(np.log(Nlist), np.log(qmc_err_est),deg=1)
    order_qmc[j] = coeff[0]
    coeff = np.polyfit(np.log(Nlist), np.log(cmc_err_est_pre),deg=1)
    order_cmc_pre[j] = coeff[0]
    coeff = np.polyfit(np.log(Nlist), np.log(qmc_err_est_pre),deg=1)
    order_qmc_pre[j] = coeff[0]
    coeff = np.polyfit(np.log(Nlist), np.log(cv_err_est),deg=1)
    order_cv[j] = coeff[0]

    # save final price results
    cmc_mean[j], qmc_mean[j] = np.mean(cmc_est), np.mean(qmc_est)
    pre_cmc_mean[j], pre_qmc_mean[j] = np.mean(cmc_est_pre),np.mean(qmc_est_pre)
    cv_mean[j] = np.mean(cv_est)

    # save error results in a file
    cmc = np.append('cmc_err_est_pre',cmc_err_est_pre)
    qmc = np.append('qmc_err_est_pre',qmc_err_est_pre)
    cv = np.append('cv_err_est', cv_err_est)
    fileName = 'results/final/error_Psi'+str(types)+'_' + str(Mlist[j]) + '.csv'
    np.savetxt(fileName, [p for p in zip(cmc, qmc, cv)], delimiter=';',fmt='%s')

    # plot error estimate
    ax = axes[int(2*j)]
    ax.loglog(Nlist, cmc_err_est, '-', label = 'CMC error estimate')
    ax.loglog(Nlist, qmc_err_est, '-', label = 'QMC error estimate')
    ax.loglog(Nlist, cmc_err_est_pre, '-', label = 'CMC pre-int error estimate')
    ax.loglog(Nlist, qmc_err_est_pre, '-', label = 'QMC pre-int error estimate')
    ax.loglog(Nlist, cv_err_est, '-', label = 'CV error estimate')
    ax.loglog(Nlist, Nlist**-0.5, '--', label = r'$N^{-1/2}$',color='gray')
    if types == 2:
        ax.loglog(Nlist, Nlist**-1.0, ':',  label = r'$N^{-1}$',color='gray')
    ax.set_title('Error estimate for '+r'$\Psi_'+str(types) \
                +'$, $m='+str(Mlist[j])+'$')
    ax.grid(True,which='both')
    ax.legend()

    # plot time
    ax = axes[int(2*j+1)]
    ax.loglog(Nlist, times_cmc, '-', label = 'CMC')
    ax.loglog(Nlist, times_qmc, '-', label = 'QMC')
    ax.loglog(Nlist, times_pre_cmc, '-', label = 'CMC with pre-int.')
    ax.loglog(Nlist, times_pre_qmc, '-', label = 'QMC with pre-int.')
    ax.loglog(Nlist, times_cv, '-', label = 'CV')
    ax.loglog(Nlist, Nlist/2**12, '--', label = r'$O(N)$',color='gray')
    ax.set_title('Time for '+r'$\Psi_'+str(types)+'$, $m='+str(Mlist[j])+'$')
    ax.grid(True,which='both')
    ax.legend()

plt.savefig('./figures/final_error_Psi_'+str(types)+'.pdf', format='pdf',
    bbox_inches='tight')
plt.show()

# plot the order of convergence of each methods with respect to the dimention
# of the problem m:
plt.figure()
plt.plot(Mlist, order_cmc, label = 'CMC')
plt.plot(Mlist, order_qmc, label = 'QMC')
plt.plot(Mlist, order_cmc_pre, label = 'CMC with pre-int.')
plt.plot(Mlist, order_qmc_pre, label = 'QMC with pre-int.')
plt.plot(Mlist, order_cv, label = 'CV')
plt.title('Order of convergence of the method with respect to m')
plt.grid(True,which='both')
plt.legend()
plt.savefig('./figures/final_order_of_convergence_Psi_'+str(types)+'.pdf',
    format='pdf', bbox_inches='tight')
plt.show()


# Sanity check:
# final price for each m:
r, T = 0.1, 1
price_cmc = np.exp(-r*T) * cmc_mean
price_qmc = np.exp(-r*T) * qmc_mean
price_pre_cmc = np.exp(-r*T) * pre_cmc_mean
price_pre_qmc = np.exp(-r*T) * pre_qmc_mean
price_cv = np.exp(-r*T) * cv_mean

print('Price for m = 32, 64, 128, 256, 512')
print(price_cmc)
print(price_qmc)
print(price_pre_cmc)
print(price_pre_qmc)
print(price_cv)
```