Project 7

# Stochastic modeling of pollutant transport in aquifers

Course of *Stochastic Simulation*
Prof. Fabio Nobile

Matteo Calafà
SCIPER 342454

A.Y. 2021/2022

# 1 Introduction

## 1.1 Goal of the project

This work focuses on the predictions of a pollutant particle trajectory in groundwater flows. Four different methods will be illustrated and compared, outlining the criteria to adopt the best among these depending on the kind of problem to be simulated and the results that are pursued.
The domain of interest is an infinite plane region with the exception of a ball in the centre. We will then define $D = \mathbb{R}^2 \setminus B(0, R)$. This ball of radius $R$ represents an obstacle to the water flow, in particular a well from which water is extracted with rate $Q$. Considering this domain, the fluid flow $\mathbf{u}(x, y)$ can be obtained through the resolution of the Darcy equation:

$$\begin{cases} \nabla \cdot \mathbf{u} = 0 & \text{in } D \\ \mathbf{u} = -k\nabla p & \text{in } D \\ \mathbf{u} \cdot n = \frac{Q}{2\pi R} & \text{on } \partial B \\ \text{B.C.} & \text{as } |x| \to \infty \end{cases}$$

However, to simplify the forthcoming calculations with an analytical solution and not a numerical approximation, we can suppose that, if the flow far from the obstacle is $\mathbf{u}^{\text{steady}}(x, y) = (1, 0)$, then a good approximation of $\mathbf{u}$ is:

$$\mathbf{u}(x, y) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + Q\nabla \left( \frac{1}{2\pi} \log \left( \sqrt{x^2 + y^2} \right) \right) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \frac{Q}{2\pi \left( x^2 + y^2 \right)} \begin{bmatrix} x \\ y \end{bmatrix} \tag{1}$$

The formula in Equation 1 will be the definition of the water flow $\mathbf{u}(x, y)$ throughout the work report.
As already anticipated, the real objective of this work is the prediction of a pollutant particle trajectory. More precisely, if the starting position $(X_0, Y_0) \in D$ at $t = 0$ is set, the goal is to predict the probability that $\tau := \inf(t \geq 0 : (X(t), Y(t) \in B))$ (i.e. the first passage time of the particle through the well) is less or equal than a fixed time horizon $T$.
In such setting, the particle trajectory could be described by the following stochastic differential system:

$$\begin{cases} dX(t) = u_1(X(t), Y(t))dt + \sigma dW_1(t) & 0 \leq t \leq T \\ dY(t) = u_2(X(t), Y(t))dt + \sigma dW_2(t) & 0 \leq t \leq T \\ X(0) = X_0, \quad Y(0) = Y_0 \end{cases} \tag{2}$$

where $W_1(t), W_2(t)$ are two different Wiener processes.

## 1.2 Standard Monte Carlo to simulate from the discretized SDE

Aiming to simulate the trajectory instead of resolving it from Equation 2, the first and simple idea is the discretization through a Euler-Maruyama scheme:

$$\begin{cases} X_{k+1} = X_k + u_1(X_k, Y_k)\Delta t + \sigma\sqrt{\Delta t} Z_k & Z_k \sim \mathcal{N}(0, 1) \\ Y_{k+1} = Y_k + u_2(X_k, Y_k)\Delta t + \sigma\sqrt{\Delta t} Z'_k & Z'_k \sim \mathcal{N}(0, 1) \end{cases} \tag{3}$$

where $\Delta t$ is the time step of the $[0, T]$ partition, $(X_0, Y_0)$ are given and $Z_k, Z'_k$ are independent $\forall k = 1 \ldots N$.
A first strategy to predict $\mathbb{P}(\tau \leq T)$ is now straightforward: after the simulation of $N_{iter}$ discretized trajectories, the probability can be estimated through the ratio of the number of trajectories that entered the well over the total number of simulations. This is a standard Monte Carlo method that, most of the times, returns reasonable predictions, however:

1. Because of the slow variance convergence rate of the standard Monte Carlo method ($= O(N_{iter}^{-\frac{1}{2}})$), it is usually needed to have a very numerous sample to obtain precise estimations with low variance. This obviously requires more iterations and then more computational effort. A variance reduction improvement is then sought and will be presented in section 3.

2. Another deficiency of the standard Monte Carlo is the inaccuracy to predict events that are rare or too frequent. For instance, if the true probability is far less than $1/N_{iter}$, then the Monte Carlo estimation is prone to return the 0 value. A modification for rare events will be presented in section 4.

## 2 The Feynmac-Kac formula and the Finite Element Method alternative

### 2.1 The Feynman-Kac equivalence

The same problem presented in the previous section could also be solved with a deterministic approach. Indeed, the following equivalence holds:

**Proposition 1.** *The entrance probability $\mathbb{P}(\tau \leq T)$ (given the starting point $\mathbf{X}(0)$) is equal to $\varphi(\mathbf{X}(0), 0)$, where $\varphi$ is the solution of:*

$$\begin{cases} \varphi_t + \mathcal{L}\varphi = 0 & \textit{in } D \times [0,T] \\ \varphi = 1 & \textit{on } \partial B \times [0,T] \\ \varphi(\mathbf{x},t) \to 0 & \textit{as } |\mathbf{x}| \to \infty \\ \varphi(\mathbf{x},T) = 0 & \textit{in } D \end{cases}$$

*and $\mathcal{L}v := (\mathbf{u} \cdot \nabla)v + \frac{1}{2}(\sigma^2 \Delta v)$ is an operator defined on $C^{2,1}(D \times (0,T))$.*

Then, the same predictions can also be obtained through the numerical resolution of such system. These results will be useful for us to confirm and compare our stochastic predictions. In general, however, this method should not be considered as an alternative to the stochastic methods because it lacks of generality. Even if this method avoids sampling error overheads, the deterministic equivalence holds only for this specific problem. Moreover, it is also demanding since it requires the solution to a partial differential equation when the real objective is only its evaluation in a point.

### 2.2 Weak formulation of the deterministic problem

In any case, in order to get numerically these deterministic reference results, we now present the weak formulation to apply in a finite element library such as `fenics`.
First of all, notice that the previous backward problem can be transformed in a forward formulation. If $\tilde{\varphi} : D \times [0,T] \to \mathbb{R}, \quad \tilde{\varphi}(\mathbf{x},t) := \varphi(\mathbf{x}, T-t)$, then:

$$\mathbb{P}(\tau \leq T | \mathbf{X}(0)) = \tilde{\varphi}(\mathbf{X}(0), T) \quad \text{s.t.} \quad \begin{cases} \tilde{\varphi}_t - \mathcal{L}\tilde{\varphi} = 0 & \text{in } D \times [0,T] \\ \tilde{\varphi} = 1 & \text{on } \partial B \times [0,T] \\ \tilde{\varphi}(\mathbf{x},t) \to 0 & \text{as } |\mathbf{x}| \to \infty \\ \tilde{\varphi}(\mathbf{x},0) = 0 & \text{in } D \end{cases}$$

Another preliminary consideration is the fact that an infinite domain is computationally impracticable. For this reason, we now define $D$ as a big ball with a central hole in $B$. If the same boundary conditions (originally at the infinite points) are here applied on the external circumference and if the big ball radius is sufficiently large, we expect to approximate in an accurate way the infinite domain problem.
Since non-homogeneous Dirichlet boundary conditions are usually imposed afterwards in numerical libraries as `fenics`, we now consider the homogeneous problem. Consider now $\mathcal{T}_h$ as the mesh discretization over $D$. The finite element space of the piece-wise first order polynomials over D with homogeneous boundary conditions is then:

$$V_h = \{f \in C^0(D) : \; f|_{\partial D} = 0, \quad f|_{t_h} \in \mathbb{P}^1(t_h) \, \forall t_h \in \mathcal{T}_h\}$$

So, the homogeneous problem consists in finding $\varphi(\cdot,t) \in V_h$ s.t. $\varphi(\cdot,0)=0$ and

$$\int_D \varphi_t v - \int_D \mathcal{L}\varphi = 0 \quad \Leftrightarrow \quad \int_D \varphi_t v - \int_D (\mathbf{u}\cdot\nabla)\varphi\, v - \frac{1}{2}\sigma^2 \int_D \Delta\varphi\, v = 0$$
$$\overset{\text{Gauss-Green}}{\Leftrightarrow} \quad \int_D \varphi_t v - \int_D (\mathbf{u}\cdot\nabla)\varphi\, v + \frac{1}{2}\sigma^2 \int_D \nabla\varphi\cdot\nabla v = 0 \qquad \forall v \in V_h$$

To have a fully discretized formulation, only the temporal derivative discretization misses. To do this, a first-order implicit Euler has been chosen and the formulation results:

$$\begin{gathered}\text{Find } \{\varphi_n\}_{n=0\ldots N} \subset V_h \text{ s.t. } \varphi_0 = 0 \text{ and } \forall n = 1\ldots N-1: \\ \int_D \varphi_n v - \int_D \varphi_{n-1} v - \Delta t \int_D (\mathbf{u}\cdot\nabla)\varphi_n\, v + \frac{\Delta t}{2}\sigma^2 \int_D \nabla\varphi_n \cdot \nabla v = 0\end{gathered} \tag{4}$$

Equation 4 finally shows the explicit temporal step and will justify the definition of the weak problem in the `fenics` implementation (subsection 7.2, line 170).

## 3 Importance Sampling as variance reduction technique

### 3.1 General aspects of the Importance Sampling method

As anticipated in Section 1.2, we aim to adopt a variance reduction technique to improve the already presented standard Monte Carlo. The proposed technique is the so-called *Importance Sampling* and, before showing how it can be exploited in our problem, we give here the basic and general concepts.
If we consider a standard Monte Carlo estimation from a continuous random variable $X$ that admits a density function $f:\mathbb{R}\to\mathbb{R}$, the idea is to sample from another distribution $g$ called *importance distribution.* We indeed observe that:

$$\mathbb{E}_f[\psi] = \int_{\mathbb{R}} \psi(x) f(x)\,dx = \int_{\mathbb{R}} \psi(x)\frac{f(x)}{g(x)} g(x)\,dx = \mathbb{E}_g\left[\psi\frac{f}{g}\right] \qquad \forall f \ll g$$

This trivial result is instead showing that we can really sample from $g$ as long as we correct the sample function with the distribution fraction. If $g$ is chosen in such a way that the estimator variance is less than the previous one, the Importance Sampling turns out to be a variance reduction technique for the Monte Carlo method.
The main reason why this method has been proposed instead of other techniques is mainly due to the suitability of the Importance Sampling also in the presence of Markov processes. If, indeed, we consider a discrete-time continuous-space Markov chain $\{X_n\}_{n=1,2,\ldots}$ with initial distribution $p_0$ and transition kernel $P$ that admits density function $p$, i.e.:

$$P(x,A) = \mathbb{P}(X_{n+1}\in A | X_n = x) = \int_A p(x,y)\,dy \quad \forall n \in \mathbb{N} \quad \forall A \in \mathcal{B}(\mathbb{R})$$

one can prove that sampling a function of the first N variables from Markov$(p_0,P)$ is equivalent to sample from another process Markov$(q_0,Q)$ as long as $q$ again dominates p and the following correction term (that previously was simply $f/g$) is added:

$$w(X_0,\ldots,X_N) = \frac{p_0(X_0)}{q_0(X_0)} \prod_{j=1}^{N} \frac{p(X_{j-1},X_j)}{q(X_{j-1},X_j)}$$

This result is actually quite trivial since it only exploits the Markov property. Also in the case of a stopping time $\tau \in \mathbb{N}$ (as in our case) that is finite almost surely, one can prove the equivalent result:

$$\begin{aligned} \mathbb{E}_p[\psi_\tau(X_0,\dots,X_\tau)] &= \mathbb{E}_q[\psi_\tau(X_0,\dots,X_\tau)w(X_0,\dots,X_\tau)] \\ \text{where} \quad w(X_0,\dots,X_\tau) &= \frac{p_0(X_0)}{q_0(X_0)}\prod_{j=1}^{\tau}\frac{p(X_{j-1},X_j)}{q(X_{j-1},X_j)} \end{aligned} \tag{5}$$

## 3.2 Importance Sampling for the pollutant transport simulation

Considering again the Euler-Maruyama scheme in Equation 3, we could observe that the original $X$ and $Y$ stochastic processes are reformulated as discrete-time continuous-space Markov chains. Every $X_{k+1}, Y_{k+1}$ depends only on the values of the previous step, indeed:

$$\begin{cases} X_{k+1}|X_k,Y_k \sim N(\mu_x,\sigma^2\Delta t) & \mu_x = X_k + u_2(X_k,Y_k)\Delta t \\ Y_{k+1}|X_k,Y_k \sim N(\mu_y,\sigma^2\Delta t) & \mu_y = Y_k + u_2(X_k,Y_k)\Delta t \end{cases}$$

As discussed in the previous section, we could then try to implement a Important Sampling algorithm. The proposed importance distribution idea is based on the shift of the means of these normal Markov kernels, i.e:

$$\begin{cases} X_{k+1} = X_k + (u_1(X_k,Y_k)+c_x)\Delta t + \sigma\sqrt{\Delta t}Z_k \\ Y_{k+1} = Y_k + (u_2(X_k,Y_k)+c_y)\Delta_t + \sigma\sqrt{\Delta t}Z'_k \end{cases} \qquad c_x, c_y \in \mathbb{R} \tag{6}$$

From another point of view, this is equivalent to change the distribution of the random variation $\Delta W_k = \sigma\sqrt{\Delta t}Z_k \sim N(0,\sigma^2\Delta t)$ to $\Delta W_k = c_x\Delta t + \sigma\sqrt{\Delta t}Z_k \sim N(c_x\Delta t, \sigma^2\Delta t)$. At this point, notice that the function $\psi_\tau(X_0,Y_0,X_1,Y_1\dots X_\tau,Y_\tau)$ actually depends only on these Brownian increments since all the other components are fixed parameters. Then, it is enough to compute the probability density functions only of these random variables that are:

$$p(X_{k+1}|X_k,Y_k) = \tilde{p}(\Delta W_k) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2\Delta t}}e^{-\frac{(\Delta W_k)^2}{2\sigma^2\Delta t}} & \text{for the original distribution} \\ \frac{1}{\sqrt{2\pi\sigma^2\Delta t}}e^{-\frac{(\Delta W_k - c_x\Delta t)^2}{2\sigma^2\Delta t}} & \text{for the IS distribution} \end{cases}$$

The same for the $Y$ case. Finally, the correction term $w(X_{0:\tau}, Y_{0:\tau})$ turns out to be:

$$\begin{aligned} w(X_{0:\tau},Y_{0:\tau}) &\overset{(5)}{=} \prod_{k=0}^{\tau-1}\frac{p(X_{k+1},Y_{k+1}|X_k,Y_k)}{p_{IS}(X_{k+1},Y_{k+1}|X_k,Y_k)} \overset{\perp\!\!\!\perp}{=} \prod_{k=0}^{\tau-1}\frac{p(X_{k+1}|X_k,Y_k)p(Y_{k+1}|X_k,Y_k)}{p_{IS}(X_{k+1}|X_k,Y_k)p_{IS}(Y_{k+1}|X_k,Y_k)} \\ &= \prod_{k=0}^{\tau-1}\underbrace{\exp\left\{\frac{1}{2\sigma^2\Delta t}\left[(\Delta W_k - c_x\Delta t)^2 + (\Delta W'_k - c_y\Delta t)^2 - (\Delta W_k)^2 - (\Delta W'_k)^2\right]\right\}}_{:=w_k(\Delta W_k,\Delta W'_k)} \end{aligned} \tag{7}$$

Equation 7 will justify the Importance Sampling implementation (subsection 7.4, line 37). It is important to notice that, instead of calculating $w$ directly from the entire Markov chain, it has been preferred to compute it step by step from the $w_k$ components in order to avoid memory or floating issues.
For what concerns the choice for the best values of $c_x, c_y$, we opted for experimental tests that returned the approximate optimal solutions (`best_c` function in subsection 7.4) and will be shown in subsection 5.3.
To conclude this section, it is important to explain the reason of the choice of the importance distribution. The most general and simple choices usually consist to shift or scale the original distribution that, in this case, is normal. The shifting transformation is the one proposed that turned out to be successful. On the other hand, the scaling transformation (i.e., for normal distributions, the correction of the variance), causes some issues when the temporal step $\Delta t$ gets smaller. If, indeed, we reply the computation of $w$ as in Equation 7 but with different variances (e.g. $\sigma$ and $\sigma'$), we get a term that diverges as $\Delta t \to 0$. This is the reason why only the shifting transformation is preferred, avoiding to transform the variance too.

## 4 The splitting method

### 4.1 A solution to simulate rare events

We now wander whether it is possible to estimate the entrance probability in the case its true value is very small or, in other terms, we deal with a rare event. This is a very common scenario: for instance, in our problem, this happens every time the starting point $(X_0, Y_0)$ is chosen sufficiently distant from the well.
As already introduced in subsection 1.2, Monte Carlo method is not suitable to predict probabilities that are far less that $1/N_{iter}$ (unless $N_{iter}$ is increased a lot, but this is often not feasible). With Importance Sampling, the estimation can be certainly improved since the rare event, under the importance distribution, could become a "less" rare event. However, we usually do not expect this to be a significant improvement and a method ad hoc for rare events must be adopted.
The proposed strategy is the so-called *Splitting Method* ([1]). The basic idea is to split the full trajectory of the particle into multiple paths that belong to different subdomains. In this way, the probability estimation is obtained through a product of conditional probabilities that are, in general, not small as the one for the full path and can then be estimated with the previous methods.
Consider now $C_0 \supset C_1 \supset \cdots \supset C_L$ nested subdomains and assume the particle is initially located in a point $\mathbf{X}_0$ of $C_0$. We aim to compute the probability that the particle enters $C_N$ before the time horizon $T$. The Splitting Method exploits the following equivalence:

$$\mathbb{P}_{\mathbf{X}_0}(\mathbf{X} \overset{T}{\in} C_L) = \mathbb{P}_{\mathbf{X}_0}(\mathbf{X} \overset{T}{\in} C_L | \mathbf{X} \overset{T}{\in} C_{L-1}) \dots \mathbb{P}_{\mathbf{X}_0}(\mathbf{X} \overset{T}{\in} C_2 | \mathbf{X} \overset{T}{\in} C_1) \cdot \mathbb{P}_{\mathbf{X}_0}(\mathbf{X} \overset{T}{\in} C_1)$$

where the symbol $\overset{T}{\in}$ means the entrance before time $T$ or, equivalently, that the first time passage $\tau_i = \inf_{t \geq 0}\{\mathbf{X}(t) \in C_i\}$ is less or equal than $T$. The idea is then to compute these probabilities separately and then to multiply them. However, except for the first level, it is not known a priori from the formula which starting point to use and how many simulations to run for each level. We opted for the *Fixed Effort Splitting* algorithm described in [3] where the starting points are uniformly sampled from the entrance points of the previous layer in such a way that their cardinality is constant ($= N_{iter}$) for each level. Since not all the trajectories enter the following sub-domain, this last proposition implies that the sampling has to be done with repetition.

### 4.2 The Splitting Method for distant particle trajectories

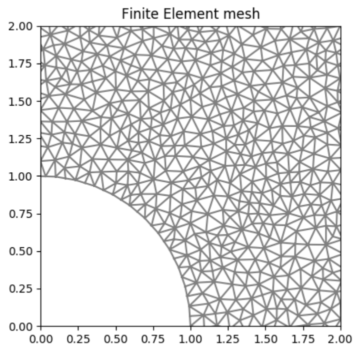
As already stated, we aim to adopt the Fixed Effort Splitting for which the number of simulations is $N_{iter}$ for each level. For what concerns the definitions of the subdomains, because of the round shape of the domain, it is natural to define them as nested balls with constant radius increments. The number of levels is instead chosen as $L \approx -\ln(p)/2$ with $p$ as the true probability. This choice should give the lowest variance for the estimator ([3]).

## 5 Results

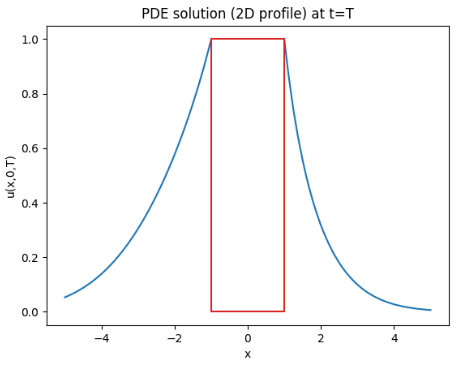
The following parameters have been set for all the simulations: $\sigma = 2$, $Q = 1$, $R = 1$.

### 5.1 Finite Element Reference values

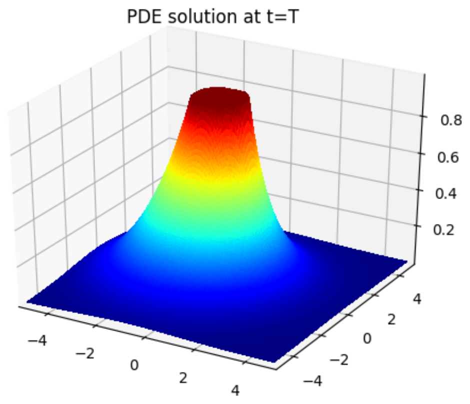
We opted to firstly show the outcomes related to section 2 in order to have some reference results that well approximate the exact values and can be used to evaluate the precision of the stochastic estimations. As already introduced, `fenics` library has been used to compute the finite element solution. In Figure 1, the main visual results of the Finite Element solution are shown.



(a) Finite Element mesh detail. On the bottom-left, a quarter of the well region.



(b) 2D profile of the solution at $t = T, y = 0$. The red square is the profile of the well, notice that the solution respects the boundary conditions on it.



(c) Solution at $t = T$. Notice the effect of the diffusive term. However, the solution turns out to be slightly asymmetrical because of the convective term.

Figure 1: Details of the Finite Element solution. Number of mesh nodes = 710'767, number of temporal steps = 800, radius of the domain = 40 R = 40

After having checked the regularity of the numerical solution, the fulfilment of the boundary conditions and the independence from the mesh, we decided to study how much the probability estimations depend on the main discretization parameters. These are the mesh size, the number of temporal steps, the domain extension and their results are shown respectively in Table 1, Table 2, Table 3.
First of all, we observe that prediction values are consistent with the distance of the starting point from the well. Then, we observe that the predictions are, as well as the solution, almost independent from the discretization parameters. Only the probability from (7.0,7.0), because of its very small value, is prone to relatively higher perturbations. However, we can deduce these probabilities with a high level of accuracy. The reference solutions are then, respectively: 50.8%, 0.98%, 6.26%, $5 \cdot 10^{-7}$.

| $\mathbf{X}_0$ | 178129 nodes | 355619 nodes | 534234 nodes | 710767 nodes |
|---|---|---|---|---|
| (1.2,1.1) | 0.508816167601 | 0.508013732378 | 0.508331147046 | 0.508474982723 |
| (3.0,4.0) | 0.00968527172491 | 0.0097400740273 | 0.00976409737301 | 0.00976687266362 |
| (2.5,2.5) | 0.0622652923923 | 0.062502467559 | 0.062522222027 | 0.0625908059134 |
| (7.0,7.0) | 4.13821908708e-07 | 4.51422147941e-07 | 4.64810669551e-07 | 4.70482174401e-07 |

Table 1: PDE resolution varying the mesh-size (temporal steps=800, domain radius =40R)

| $\mathbf{X}_0$ | 200 steps | 400 steps | 600 steps | 800 steps |
|---|---|---|---|---|
| (1.2,1.1) | 0.508256548529 | 0.508402251884 | 0.508450748054 | 0.508474982723 |
| (3.0,4.0) | 0.00978398016881 | 0.00977257745361 | 0.00976877449247 | 0.00976687266362 |
| (2.5,2.5) | 0.0624920502402 | 0.0625578098727 | 0.0625797986255 | 0.0625908059134 |
| (7.0,7.0) | 5.4747816346e-07 | 4.9558521308e-07 | 4.78787066572e-07 | 4.70482174401e-07 |

Table 2: PDE resolution varying the number of time steps (mesh-size=600, domain radius=40R)

| $\mathbf{X}_0$ | 20 R | 30 R | 40 R |
|---|---|---|---|
| (1.2,1.1) | 0.508408630847 | 0.508564939908 | 0.508474982723 |
| (3.0,4.0) | 0.00978949087774 | 0.00978907182132 | 0.00976687266362 |
| (2.5,2.5) | 0.0626261832305 | 0.0625999443181 | 0.0625908059134 |
| (7.0,7.0) | 4.86692252104e-07 | 4.80475060866e-07 | 4.70482174401e-07 |

Table 3: PDE resolution varying the radius of the domain (mesh-size=600, temporal steps=800)

## 5.2 Standard Monte Carlo results

In the standard Monte Carlo scheme, instead, two main sources of errors need to be considered: the first one is the usual sampling error of the Monte Carlo estimation that has a completely random nature. The second one is the discretization in the Euler-Maruyama scheme that is a numerical and not stochastic error. Then, again, we decided to compare results from different values of Monte Carlo iterations and $\Delta t$ discretization steps that are shown in Table 4, Table 5. The C.I. term indicates the semi-amplitude of the 99% asymptotic confidence interval for the Monte Carlo estimations.

| | $5 \cdot 10^2$ | | $5 \cdot 10^3$ | | $5 \cdot 10^4$ | |
|---|---|---|---|---|---|---|
| $\mathbf{X}_0$ | Prediction | C.I. | Prediction | C.I. | Prediction | C.I. |
| (1.2,1.1) | 0.5100 | 0.0576 | 0.5052 | 0.0182 | 0.5041 | 0.0058 |
| (3.0,4.0) | 0.0064 | 0.0089 | 0.0104 | 0.0037 | 0.0097 | 0.0011 |
| (2.5,2.5) | 0.0612 | 0.0273 | 0.0654 | 0.0090 | 0.0064 | 0.0028 |

Table 4: Monte Carlo estimations varying the number of iterations (time-step = $10^{-4}$)

| | $10^{-2}$ | | $10^{-3}$ | | $10^{-4}$ | |
|---|---|---|---|---|---|---|
| $\mathbf{X}_0$ | Prediction | C.I. | Prediction | C.I. | Prediction | C.I. |
| (1.2,1.1) | 0.44342 | 0.00571 | 0.48638 | 0.00576 | 0.50412 | 0.00575 |
| (3.0,4.0) | 0.00798 | 0.00100 | 0.00844 | 0.00110 | 0.00966 | 0.00105 |
| (2.5,2.5) | 0.05412 | 0.00257 | 0.05896 | 0.00270 | 0.06248 | 0.00274 |

Table 5: Monte Carlo estimations varying the time discretization ($5 \cdot 10^4$ Monte Carlo iterations)

First of all, results clearly resemble the ones obtained in subsection 5.1 and get closer to these values as the discretization parameters improve. The confidence intervals in Table 4 clearly reduce at every improvement

step (more precisely, the rate is $\sqrt{10}$ since the iterations are raised 10 times at each step). On the contrary, the confidence intervals shown in Table 5 are almost constant. We stress again that Euler-Maruyama error is indeed numerical and it is not directly related to sampling errors and variances. This can also be observed noticing that the convergence is monotone (the prediction increases as $\Delta t$ decreases) instead of having an asymptotic normal error. At this point, it is natural to wander the order of this numerical error.
A convergence study has been executed to this purpose paying attention to reduce the random influences on the results. First of all, it is needed to let the sampling error negligible with respect to the one that will be estimated. This can be done increasing a lot the number of Monte Carlo iterations and letting a very coarse trajectory discretization. Another improvement could be to let the same Brownian increments to be shared among simulations with different $\Delta t$. The proposed idea is to run multiple Monte Carlo simulations with $\Delta t$ following the powers of 2. The first simulation will do a Euler-Maruyama increment at every step, the second one at alternating steps, the third one only 1 time out of 4 and so on. In this way, we optimize the computation and many Brownian increments are shared. Thus, trajectories with smaller $\Delta t$ can be perfectly seen as refinements of the trajectories with bigger $\Delta t$. Errors are computed with respect to the solutions from subsection 5.1 and the same $\Delta t$-convergence scheme is repeated 5 times to show the magnitude of possible random influences. These results are finally plotted in Figure 2.

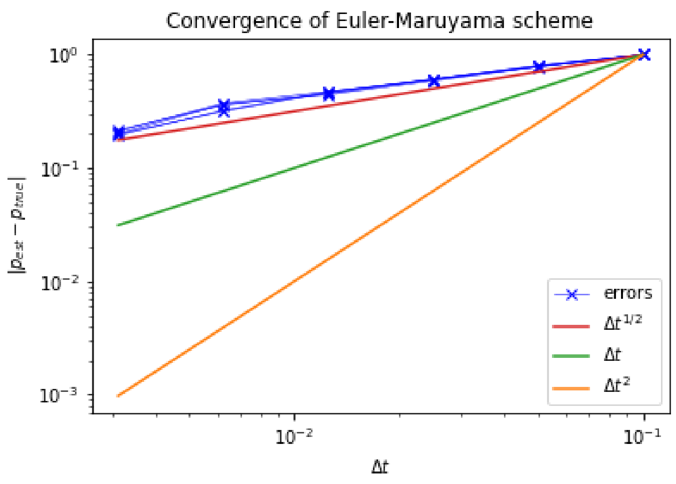
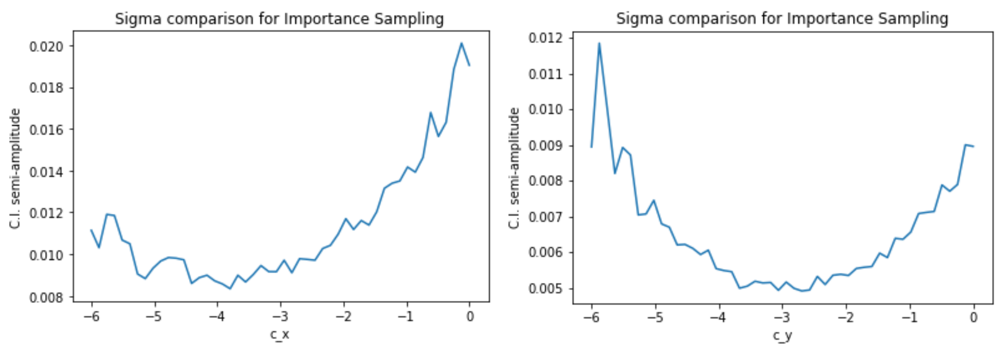


Figure 2: Convergence of Euler-Maruyama estimations ($\mathbf{X}_0 = (2.5, 2, 5), N_{iter} = 5 \cdot 10^5$, 5 recurrences)

First, the number of Monte Carlo iterations is confirmed to be sufficiently high to let the sampling error negligible, indeed the five convergence trends are almost overlapped. From the plot, it is clear that the Euler-Maruyama prediction converges with an 1/2 order. This result differs from the theoretical weak error that is well-known in literature and equal to 1 (see, for instance, Theorem 14.5.1 in [2]). Under a more careful analysis, however, one can notice that this problem does not satisfy the regularity assumptions: the process $\{X_n\}_{n=0\ldots\tau}$ is regular (it has indeed constant or regular coefficients) but it is stopped if it enters the well. So, if $\bar{n}$ is the entrance time, it means that $\{X_n\}_{n\geq\bar{n}}$ keeps constant and then it is evident the discontinuity with the previous process coefficients. Moreover, the probability function $p(X_\tau)$ to be estimated is an indicator function and then it is discontinuous as well (while it is required to be $C^4(D)$, see again the assumptions of $g$ in Theorem 14.5.1 from [2]). It is then clear that the first order can not be guaranteed from the theory and the lack of regularity is presumably the reason of the 1/2 order.

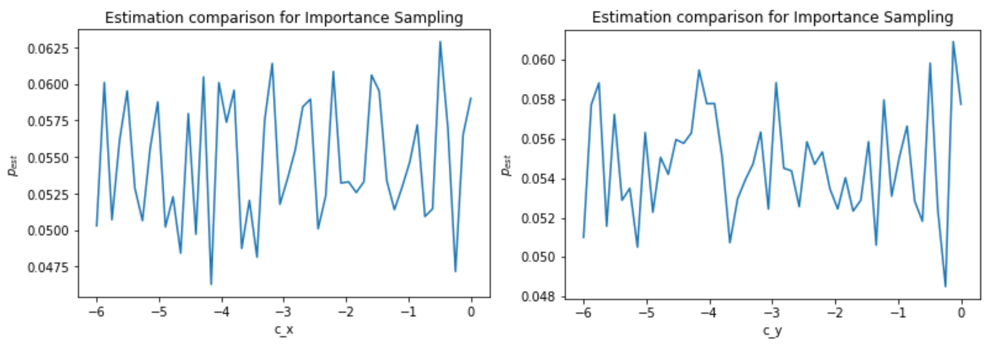
## 5.3 Importance Sampling results

The theoretical motivation of this method has already been discussed in section 3. Here, we aim to experimentally achieve the optimal values for $c_x, c_y$ and show their benefits from the results. The simple strategy that has been adopted consists in executing multiple runs with different $c_x$ keeping fixed $c_y$ and vice versa. A variance comparison is then shown in Figure 3. Here, setting $\mathbf{X}_0 = (2.5, 2.5)$, two passages have been executed: first of all $c_y$ was set to zero and the experiment proved that $c_x = -4$ was the optimal solution (Figure 3a). Then, fixing $c_x = -4$, $c_y = -3$ reveals to be the other solution (Figure 3b). Another passage could be added to conversely confirm that $c_x = -4$ is the optimal solution when $c_y = -3$. Even it is not guaranteed to be the global optimum, the fact that this solution provides a lower variance than $c_x, c_y = 0$ confirms that the Importance Sampling really improves the standard Monte Carlo. To additionally confirm that results are not affected from these parameters choice, Figure 4 shows that the estimators are indeed independent. Results obtained with this choice of parameters are instead shown in Table 6, Table 7.



(a) $c_x$ varies, $c_y = 0$ (b) $c_y$ varies, $c_x = -4$

Figure 3: Comparison of confidence interval semi-amplitudes with different $c_x$, $c_y$ $(\mathbf{X}_0 = (2.5, 2, 5), N_{iter} = 1000, \Delta t = 0.005)$



(a) $c_x$ varies, $c_y = 0$ (b) $c_y$ varies, $c_x = -4$

Figure 4: Comparison of Importance Sampling estimations with different $c_x$, $c_y$ $(\mathbf{X}_0 = (2.5, 2, 5), N_{iter} = 1000, \Delta t = 0.005)$

|  | $5 \cdot 10^2$ |  | $5 \cdot 10^3$ |  | $5 \cdot 10^4$ |  |
|---|---|---|---|---|---|---|
| $\mathbf{X}_0$ | Prediction | C.I. | Prediction | C.I. | Prediction | C.I. |
| (2.5,2.5) | 0.0617 | 0.0078 | 0.0605 | 0.0023 | 0.0620 | 0.00076 |

Table 6: Importance estimations varying the number of iterations (time-step = $10^{-4}$)

|  | $10^{-2}$ |  | $10^{-3}$ |  | $10^{-4}$ |  |
|---|---|---|---|---|---|---|
| $\mathbf{X}_0$ | Prediction | C.I. | Prediction | C.I. | Prediction | C.I. |
| (2.5,2.5) | 0.0523 | 0.00069 | 0.0594 | 0.00074 | 0.0614 | 0.00076 |

Table 7: Importance estimations varying the time discretization ($5 \cdot 10^4$ Monte Carlo iterations)

As for the Monte Carlo simulations, confidence intervals follow the rate $\sqrt{10}$ in Table 6 and keep constant in Table 7. Also the predictions are consistent with the true values. The new outcome is the lower value for the confidence intervals (or, equivalently, the variances). If compared to Table 4, Table 5 results, the confidence intervals turn out to be reduced almost 4 times. With very few additional calculations, the Importance Sampling method is so capable to clearly improve the standard Monte Carlo performance.

## 5.4 Splitting Method results

The last proposed strategy is the Splitting Method for rare events. In this section, we aim to compute the entrance probability when the starting point is $\mathbf{X}_0 = (7.0, 7.0)$. From subsection 5.1, the reference solution of such probability has been obtained even if not accurately as for the other starting points. Despite this, we can consider the exact solution as of the order of $5 \cdot 10^{-7}$, i.e. a very small value. Even if Monte Carlo method runs for $5 \cdot 10^4$ iterations (the highest value considered so far), the probability that the estimation is the exact zero is $\approx (1 - 5 \cdot 10^{-7})^{(5 \cdot 10^4)} \approx 97.5\%$. The Splitting Method then needs to be used. As already anticipated, the *Fixed Effort Splitting* strategy from [3] has been implemented. With such choice, the number of paths is constant for every level and has been set to $10^4$. In addition, the number of levels has been computed with the optimal choice $L = -\ln(p_{true})/2$ from [3] and, considering the previous approximation of $p_{true}$, it resulted that the best choice is $L = 7$. Because of the choice of constant radius increments, the definitions of the subdomains is now straightforward and results from 4 runs are shown in Table 8.

| $\mathbf{X}_0$ | Run 1 | Run 2 | Run 3 | Run 4 | Mean |
|---|---|---|---|---|---|
| (7.0,7.0) | 4.39580e-07 | 4.76504e-07 | 3.90226e-07 | 5.67980e-07 | 4.68573e-07 |

Table 8: Splitting method simulations ($10^4$ Monte Carlo iterations, time-step = $5 \cdot 10^{-4}$, 7 levels)

Despite a precise comparison can not be done because of the lack of an accurate reference solution, results are clearly confirming the $5 \cdot 10^{-7}$ prevision and the method together with the parameters choices turn out to be correct and consistent. For what concerns the discretization parameters, one can indeed observe that the number of total iterations and the time-step are almost the same as the ones in the previous simulations that revealed to be accurate enough for the standard methods.

## 6 Conclusion

In this work, different methods have been presented to solve the probability estimation problem and their results have been shown. First, the Finite Element method applied to the Feynman-Kac equivalent problem has been introduced to find numerically the reference results. Then, 3 variants of the Monte Carlo method have been presented and discussed showing that the standard Monte Carlo, because of its simplicity, often needs to be improved. Then, variance reduction and rare event method results have clearly shown the benefits with respect to the reference solutions. Further extensions to this work could certainly include the choice of different variance reduction techniques and other Splitting Method strategies.

## References

[1] Marnix Joseph Johann Garvels. “The splitting method in rare event simulation”. PhD thesis. University of Twente, 2000.

[2] E. Kloeden Peter and Eckhard Platen. *Numerical Solution of Stochastic Differential Equations*. Berlin: Springer, 1992. ISBN: 978-3-662-12616-5. DOI: `https://doi.org/10.1007/978-3-662-12616-5`.

[3] P. Kroese Dirk, Thomas Taimre, and I. Botev Zdravko. *Handbook of Monte Carlo Methods*. New York: John Wiley & Sons, 2011. ISBN: 9781118014967. DOI: `https://doi.org/10.1002/9781118014967`.

## 7 Appendix: Python codes

The code is organized in the following way: each of the 4 presented methods (Finite Element, standard Monte Carlo, Importance Sampling and Splitting Method) corresponds to a Python script that is separated from the others. Each script contains various functions that are defined to be used in other functions or to be directly called in the main. The only exception is the script `parameters.py` that has been created to define all the main parameters since they are often shared in different scripts and their assigned values are in such way unequivocal.

### 7.1 `parameters.py`

```python
import numpy as np
import scipy.stats as st

####### MAIN PARAMETERS FILE ########

X0_MC = np.array([[1.2, 1.1],[3.0,4.0],[2.5,2.5]])  # starting points to be used in standard Monte Carlo method
X0_FE = np.array([[1.2, 1.1],[3.0,4.0],[2.5,2.5],[7.0,7.0]]) # starting points to be used in Finite Element method
X0_IS = np.array([2.5,2.5]) # starting point to be used in Importance Sampling method
X0_SM = np.array([7.0,7.0]) # starting point to be used in Splitting Method
sigma = 2 # diffusion of the porous media
Q = 1 # extraction mass rate
R = 1 # well radius
T = 1 # time horizon
u = lambda X: np.array([1.,0.]) + (Q/(2*np.pi*(np.power(X[0],2) + np.power(X[1],2))))*X # water flow function
poly_order = 1 # by default, Finite Element method is used with first order polynomials
Za = st.norm.ppf(1 - 0.01 / 2) # multiplicative constant for confidence intervals
```

### 7.2 `finite_element.py`

```python
import fenics as f
import dolfin as d
import mshr as m
import numpy as np
import parameters
from tqdm import tqdm
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.pyplot as plt


def print_message(idx):

    #### A function for multiple printing messages ####

    if (idx==0):
        print ('\n ----- FINITE ELEMENT METHOD ----- \n')
    elif (idx==1):
        print("\r{0}".format("Mesh generation:                              started ---> "), end='')
    elif (idx==2):
        print("completed")
    elif (idx==3):
        print("\r{0}".format("Functions and BC definition:                  started ---> "), end='')
    elif (idx==4):
        print("\n\n\r{0}".format("Post-processing:                              started ---> "), end='')


def post_processing(u,mesh):

    #### Post-processing: 3D plot, 2D profile plot, mesh plot ####
    #
    # INPUT:
    # u = Finite Element solution
    # mesh = Finite Element mesh


    print_message(4)
    n_points = 200

    #------ 3D solution plot -------

    x = np.linspace(-5*parameters.R,5*parameters.R,n_points)
    y = np.linspace(-5*parameters.R,5*parameters.R,n_points)

    x,y = np.meshgrid(x,y) # creation of the 3D plot grid
    x = x.ravel()
    y = y.ravel()
    z = np.zeros(x.shape)
    inner_list = [] # the list of grid points inside the well region
    for i in range (len(x)):
        if (x[i]**2 + y[i]**2 >= parameters.R**2): # i.e. if the point is outside the well region
            P = f.Point(x[i],y[i])
            z[i] = u(P) # z is the evaluation of u(x,y)
        else:
            inner_list.append(i)

    x = np.delete(x,inner_list) # we delete every coordinate corresponding to points inside the
        well
    y = np.delete(y,inner_list)
    z = np.delete(z,inner_list)

    plt.figure()
    ax = plt.axes(projection='3d')
    ax.plot_trisurf(x, y, z, cmap=cm.jet, linewidth=0, antialiased=False)
    ax.set_title('PDE␣solution␣at␣t=T')
    plt.xlim([-5*parameters.R,5*parameters.R])
    plt.ylim([-5*parameters.R,5*parameters.R])
    plt.savefig("./figures/u_3d")

    #------------------------------

    #------ 2D solution plot -------

    xl = np.linspace(-5*parameters.R,-parameters.R,n_points) # i.e. x points at the left of the
        well
    xr = np.linspace(parameters.R,5*parameters.R,n_points) # i.e. x points at the right of the
        well
    zl = np.zeros(xl.shape)
    zr = np.zeros(xr.shape)
    for i in range (len(xl)):
        P = f.Point(xl[i],0) # every z is the evaluation of u(x,y=0), discerning left from right
        zl[i] = u(P)
        P = f.Point(xr[i],0)
        zr[i] = u(P)

    q = np.array([[-parameters.R, -parameters.R, parameters.R, parameters.R, -parameters.R], [0,
        1, 1, 0, 0]]) # the array that will plot the square well
    plt.figure()
    ax = plt.axes()
    plt.plot(xl,zl, 'tab:blue')
    plt.plot(xr,zr, 'tab:blue')
    plt.plot(q[0,:],q[1,:], 'tab:red')
    plt.xlabel('x')
    plt.ylabel('u(x,0,T)')
    ax.set_title('PDE␣solution␣(2D␣profile)␣at␣t=T');
    plt.savefig("./figures/u_2d")
    #------------------------------

    #--------- Mesh plot ---------
    plt.figure()
    d.plot(mesh,title="Finite␣Element␣mesh")
    plt.xlim([0,2*parameters.R])
    plt.ylim([0,2*parameters.R])
    plt.savefig("./figures/mesh")
    #----------------------------------------
    print_message(2)




def probability_print (u):

    #### To be used to print on screen the probability estimations ####
    #
    # INPUT:
    # u = Finite Element solution

    for i in range(parameters.X0_FE.shape[0]):
        X0 = parameters.X0_FE[i,:]
        print ('X0=('+str(X0[0]) +','+ str(X0[1]) +'):␣', u(X0))






def pde_resolution(mesh_size,num_steps,max_length):

    #### Finite Element main function ####
    #
    # INPUT:
    # mesh_size = number of mesh elements along one domain diagonal (it is not the total number
        of elements)
    # num_steps = number of time steps for the temporal discretization
    # max_length = radius of the domain, should be high enough to approximate the infinite domain

    print_message(0)


    #-- preliminaries --
    f.set_log_level(f.LogLevel.ERROR) # to print only error messages (set 13 to print full run
        information)
    T = parameters.T
    dt = T / num_steps # discretization time step length

    #-- mesh generation --
    print_message(1)
    D = m.Circle(f.Point(0,0),max_length)
    B = m.Circle(f.Point(0,0),parameters.R)# i.e. the well
    domain = D  - B # the full domain is the big ball without the well
    mesh = m.generate_mesh(domain,mesh_size)
    print_message(2)


    print_message(3)
    #-- definition of the functional space --
    V = f.FunctionSpace(mesh, 'P', parameters.poly_order) # by default, first order polynomials

    #-- boundary conditions --
    infinite_distance = 'on_boundary␣&&␣pow(x[0],2)+pow(x[1],2)␣>␣2*pow(' + str(parameters.R) +'
        ,2)'
    cylinder = 'on_boundary␣&&␣pow(x[0],2)+pow(x[1],2)␣<␣2*pow(' + str(parameters.R) + ',2)'
    bc_infinite = f.DirichletBC(V,f.Constant(0),infinite_distance) # homogeneous dirichlet on the
        infinite distance
    bc_cylinder = f.DirichletBC(V,f.Constant(1),cylinder) # non-homogeneous dirichlet on the well
        border
    bc = [bc_infinite,bc_cylinder]

    #-- test functions --
```

```python
    u_n = f.Expression('0',degree=1) # it is used to return the solution at the previous temporal step in the time discretization, initialized at 0 because of the initial conditions
    u_n = f.interpolate(u_n,V)
    u = f.TrialFunction(V) # the FE solution
    v = f.TestFunction(V) # the FE test function

    #-- definition of expression needed in variational form --
    U = f.Expression(('1. + Q/(2*pi*(pow(x[0],2) + pow(x[1],2))) * x[0]', 'Q/(2*pi*(pow(x[0],2) + pow(x[1],2))) * x[1]'), degree=1, Q=parameters.Q, pi=np.pi) # the water flow function

    #-- definition of variational problem --
    F = u*v*f.dx - u_n*v*f.dx - dt*f.dot(U,f.grad(u))*v*f.dx + 0.5* parameters.sigma**2*dt*f.dot(f.grad(u),f.grad(v))*f.dx # weak formulation, see report for the derivation
    a, L = f.lhs(F), f.rhs(F)
    print_message(2)



    print ('\n\nSpace polynomial order: ', parameters.poly_order, '    Number of mesh nodes: ', mesh.num_vertices(),'\n\n')


    #-- main calculation steps --
    print ("FE system solving ("+ str(num_steps) +" temporal steps):")
    u = f.Function(V)
    t = 0
    for n in tqdm(range (num_steps)):

        t += dt # time increment
        f.solve(a==L,u,bc,solver_parameters={'linear_solver':'mumps'}) # F.E. system solving
        u_n.assign(u) # assign u to u_n for the next step

    print ('\n FINITE ELEMENT solving completed!\n')

    return u,mesh



def independence_study(mesh_size, num_steps, max_length, ind_list, ind_type):

    #### To be used to compare multiple solutions with different parameters (chosen from mesh_size, num_steps, max_length) ####
    #
    # INPUT:
    # mesh_size = number of mesh elements along one domain diagonal (it is not the total number of elements)
    # num_steps = number of time steps for the temporal discretization
    # max_length = radius of the domain, should be high enough to approximate the infinite domain
    # ind_list = list of parameter values
    # ind_type = 'grid' or 'time' or 'max-distance', the kind of parameter that corresponds to ind_list (i.e. the parameter the user wants to vary)

    # -------- 2D plot --------
    n_points = 200
    x =np.linspace(parameters.R,5*parameters.R,n_points) # the comparison plot is made on the 2D profile of the solution at the right of the well
    z = np.zeros(x.shape)
    plt.figure()
    ax = plt.axes()

    for i,q in enumerate(ind_list):

        print ('SIMULATION ', i+1,'/',len(ind_list))
        if (ind_type == 'grid'):
            u,mesh = pde_resolution(q, num_steps, max_length)
        elif (ind_type == 'time'):
            u,mesh = pde_resolution(mesh_size, q, max_length)
        elif (ind_type == 'max-distance'):
            u,mesh = pde_resolution(mesh_size, num_steps, q)

        probability_print(u)

        for j in range (len(x)):
            P = f.Point(x[j],0)
            z[j] = u(P)

        if (ind_type == 'grid'):
            plt.plot(x,z, label=str(mesh.num_vertices())+' nodes')
        elif (ind_type == 'time'):
            plt.plot(x,z, label=str(q)+' time steps')
        elif (ind_type == 'max-distance'):
            plt.plot(x,z, label=str(q/parameters.R)+' R')

        if (i==0):
            u_first = u
        elif(i==len(ind_list)-1):
            u_last = u

    plt.xlabel('x')
    plt.ylabel('u(x,0,T)')
    plt.legend()
    ax.set_title('FE '+ind_type+'-independence (t=T)');
    plt.savefig("./figures/u_"+ind_type+"_independence")


    #-------- err_inf --------
    x = np.linspace(-5*parameters.R,5*parameters.R,n_points)
    y = np.linspace(-5*parameters.R,5*parameters.R,n_points)
    x,y = np.meshgrid(x,y)
    x = x.ravel()
    y = y.ravel()
    z_first = np.zeros(x.shape)
    z_last = np.zeros(x.shape)
    for i in range (len(x)):
        if (x[i]**2 + y[i]**2 >= parameters.R**2):
            P = f.Point(x[i],y[i])
            z_first[i] = u_first(P)
            z_last[i] = u_last(P)

    print ('\nErr_inf between first and last: ',np.linalg.norm(z_first-z_last, ord=np.inf),'\n')



def save_result (u,mesh):
    #### To save locally results from Finite Element Method ####
    #
    # INPUT:
    # u = Finite Element solution
    # mesh = Finite Element mesh

    mesh_file = d.File("./files/mesh.xml")
    mesh_file << mesh
    u_file = d.HDF5File(d.MPI.comm_world,"./files/f.h5","w")
    u_file.write(u,"/f")
    u_file.close()



def load_result():
    #### To load from local memory results there were saved with save_result.py ####
    #
    # OUTPUT:
    # u = Finite Element solution
    # mesh = Finite Element mesh

    mesh = d.Mesh('./files/mesh.xml')
    V = f.FunctionSpace(mesh, 'P', parameters.poly_order)
    u = f.Function(V)
    u_file = d.HDF5File(d.MPI.comm_world,"./files/f.h5","r")
    u_file.read(u,"/f")
    u_file.close()
    return u,mesh
```

### 7.3 standard_monte_carlo.py

```
import numpy as np
import parameters
from numpy.random import normal
from tqdm import tqdm
import matplotlib.pyplot as plt



def euler_maruyama_step(X,t,dt,W):

    #### Euler-Maruyama main step ####
    #
    # INPUT:
    # X = current point of the trajectory
    # t = current time of the state
    # dt = euler-maruyama discretization step
    # W = Brownian increments
    #
    # OUTPUT:
    # X = new point of the trajectory
    # t = new time of the state
    # entrance = boolean that states if the particle has entered the well or not

    X = X + parameters.u(X)*dt + parameters.sigma*np.sqrt(dt)*W # Euler-Maruyama increment
        formula
    t = t + 1
    entrance = False

    if (np.linalg.norm(X) < parameters.R): # i.e., if the particle has entered the well
        entrance = True

    return X,t,entrance


def euler_maruyama(X0,dt):

    #### Euler-Maruyama scheme ####
    #
    # INPUT:
    # X0 = initial point of the trajectory
    # dt = euler-maruyama discretization step
    #
    # OUTPUT:
    # entrance = boolean that states if the particle has entered the well or not

    T = parameters.T
    X = X0
    entrance = False
    t = 0

    while (t*dt<T and entrance == False):
        W = normal(size=2)
        X,t,entrance = euler_maruyama_step(X, t, dt, W)

    return entrance




def standard_monte_carlo(n_iter,dt):
```

```python

    #### Standard Monte Carlo main function ####
    #
    # INPUT:
    # n_iter = cardinality of the Monte Carlo sample
    # dt = euler-maruyama discretization step
    #
    # OUTPUT:
    # result_list = list of MC estimations corresponding to each starting point

    print ('\n -----  MONTE CARLO METHOD  ----- \n')

    result_list = []
    for i in range(parameters.X0_MC.shape[0]): # loop over all the starting points
        X0 = parameters.X0_MC[i,:]
        results = np.zeros(n_iter)
        for i in tqdm(range(n_iter), desc='X0=('+str(X0[0]) +','+ str(X0[1]) +')'):
            results[i] = euler_maruyama(X0,dt)

        print ('X0=('+str(X0[0]) +','+ str(X0[1]) +'): ', results.mean() , '+-', parameters.Za*results.std()/np.sqrt(n_iter), end='\n')
        result_list.append(results.mean())
    return result_list



def multiple_predictions(n_iter,dt,ind_list,ind_type):

    #### To be used to compare results varying n_iter or dt ####
    #
    # INPUT:
    # n_iter = cardinality of the Monte Carlo sample
    # dt = euler-maruyama discretization step
    # ind_list = list of n_iter or dt (w.r.t. the parameter the user wants to vary)
    # ind_type = 'iter' or 'time', the parameter the user wants to vary


    results = np.zeros([parameters.X0_MC.shape[0],len(ind_type)])
    for i,q in enumerate(ind_list):

        print ('\nSIMULATION ', i+1,'/',len(ind_list))
        if (ind_type == 'iter'):
            results[:,i] = standard_monte_carlo(q, dt)
        elif (ind_type == 'time'):
            results[:,i] = standard_monte_carlo(n_iter,q)



def convergence_study_dt(n_iter, dt_ref, n_halfs):

    #### To be used to study the convergence order of the Euler-Maruyama w.r.t dt ####
    #
    # INPUT:
    # n_iter = cardinality of the Monte Carlo sample
    # dt_ref = smallest dt to be adopted
    # n_halfs = number of times for which dt_ref is divided by 2


    halfs = np.array(range(n_halfs)) # a vector for the halvings
    results = np.zeros([n_halfs])
    X0 = parameters.X0_IS # for this study, we only use the starting point X0 = (2.5, 2.5)
    dt = dt_ref / pow(2,halfs) # vector with the discretization steps
    true_value = 0.0626 # almost exact value obtained from Finite Element method
    T = parameters.T

    for i in tqdm(range(n_iter)):
        X = np.tile(X0, (n_halfs,1))
        t = np.zeros([n_halfs])
        entrance = np.zeros([n_halfs])
        t_ref = 0 # the refence time for all the trajectories (it coincides with the time of the
            finest trajectory)
        counter = True
        while (counter == True): # one single Monte Carlo iteration is stopped when all the
            trajectories are stopped
            counter = False
            for j in range(n_halfs):
                W = normal(size=2) # important aspect: the Brownian increments are the same
                if (t_ref % 2**halfs[n_halfs - j - 1] == 0 and t[j]*dt[j]<T and entrance[j] ==
                    False): # i.e. the standard conditions + the condition related to the number
                    of halvings
                    X[j,:],t[j],entrance[j] = euler_maruyama_step(X[j,:],t[j],dt[j],W)
                    counter = True
            t_ref = t_ref + 1
        results = results + entrance # we sum the result of each MC iteration to compute later
            the mean
    results = results/n_iter # it is now the vector with the MC estimations for all the halvings
    plt.figure()
    plt.loglog(dt, abs(results-true_value), label='error')
    err0 = abs(results[0]-true_value)
    plt.loglog(dt,err0/pow(dt[0],0.5) * pow(dt,0.5),label='$\Delta_t^{1/2}$')
    plt.loglog(dt,err0/dt[0]*dt,label='$\Delta_t$')
    plt.loglog(dt,err0/pow(dt[0],2) * pow(dt, 2),label='$\Delta_t^2$')
    plt.xlabel('$\Delta_t$')
    plt.ylabel('$|p_{est}-_p_{true}|$')
    plt.title('Convergence_of_Euler-Maruyama_scheme')
    plt.legend()
    plt.savefig("./figures/euler-maruyama-convergence")
```

## 7.4 importance_sampling.py

```
import numpy as np
import parameters
import scipy.stats as st
from numpy.random import normal
import matplotlib.pyplot as plt
from tqdm import tqdm



def euler_maruyama_IS(X0,dt,cx,cy):

    #### Variant of the homonymous standard_monte_carlo.py function ####
    #
    # INPUT:
    # X0 = starting point of the trajectory
    # dt = euler-maruyama discretization step
    # cx,cy = Importance Sampling parameters for the importance distribution shifting
    #
    # OUTPUT:
    # entrance*w = Importance Sampling single sample

    sigma = parameters.sigma
    u = parameters.u
    T = parameters.T
    X = X0
    R = parameters.R
    entrance = False # boolean that states if the particle entered the well or not
    t = 0
    w = 1.0 # the correction term is a product of many terms, we then initialize it to 1

    while (t*dt<T and entrance == False): # the simulation is stopped only if it goes beyond the
        time horizon or it enters the well

        DW = np.array([sigma*np.sqrt(dt)*normal() + cx*dt, sigma*np.sqrt(dt)*normal() + cy*dt]) #
            2D vecotr of the Brownian increments
        X = X + u(X)*dt + DW # main Euler-Maruyama step
        t = t + 1

        w = w * np.exp( (pow(DW[0]-cx*dt,2) + pow(DW[1]-cy*dt,2) - pow(DW[0],2) - pow(DW[1],2)) /  (2*sigma**2*dt) ) # main step for the w calculation, see report for the explanation

        if (np.linalg.norm(X) < R): # i.e. if the particle is inside the well region
            entrance = True

    return entrance*w


def importance_sampling(n_iter,dt,cx,cy):

    #### Important sampling main function ####
    #
    # INPUT:
    # n_iter = cardinality of the Monte Carlo sample
    # dt = euler-maruyama discretization step
    # cx,cy = Importance Sampling parameters for the importance distribution shifting
    #
    # OUTPUT:
    # results.mean() = Importance Sampling estimation

    print ('\n␣-----␣IMPORTANCE␣SAMPLING␣RESOLUTION␣-----␣\n')

    X0 = parameters.X0_IS
    results = np.zeros(n_iter) # the vector with the probability samples
    for i in tqdm(range(n_iter), desc='X0=('+str(X0[0]) +','+ str(X0[1]) +')'):
        results[i] = euler_maruyama_IS(X0, dt, cx, cy)

    print ('X0=('+str(X0[1]) +','+ str(X0[1]) +'):␣', results.mean() , '+-', parameters.Za*results.std()/np.sqrt(n_iter), end='\n')

    return results.mean()



def best_c(n_iter,dt,c_list,c_fixed,c_dir):

    #### To be used to compare the variances varying cx or cy parameter ####
    #
    # INPUT:
    # n_iter = cardinality of the Monte Carlo sample
    # dt = euler-maruyama discretization step
    # c_list = list of cx (or cy) parameters (Importance Sampling parameter for the importance distribution shifting)
    # c_fixed = the remaining parameter for the importance distribution shifting
    # c_dir = 'x' or 'y', the parameter that varies in the list (respectively, cx or cy)
    #
    # OUTPUT:
    # sigma_list = list of confidence interval semi-amplitude, each one corresponding to one cx value

    sigma_list = [] # list of confidence interval semi-amplitudes, each one corresponding to one cx value
    mean_list = [] # list of IS estimations, each one corresponding to one cx value
    X0 = parameters.X0_IS

    for c in tqdm(c_list):
        results = np.zeros(n_iter)
        for i in range(n_iter):
            if (c_dir == 'x'):
                results[i] = euler_maruyama_IS(X0, dt, c, c_fixed)
            elif (c_dir == 'y'):
                results[i] = euler_maruyama_IS(X0, dt, c_fixed, c)

        mean_list.append(results.mean())
        sigma_list.append(parameters.Za*results.std()/np.sqrt(n_iter))

    plt.figure()
    ax = plt.axes()
    ax.set_title('Estimation comparison for Importance Sampling')
    plt.plot(c_list,mean_list)
    plt.xlabel('c_'+ c_dir)
    plt.ylabel('$p_{est}$')
    plt.savefig("./figures/IS_means")

    plt.figure()
    ax = plt.axes()
    ax.set_title('Sigma comparison for Importance Sampling')
    plt.plot(c_list,sigma_list)
    plt.xlabel('c_' + c_dir)
    plt.ylabel('C.I. semi-amplitude')
    plt.savefig("./figures/IS_sigmas")

    return sigma_list
```

## 7.5 splitting_method.py

```
import numpy as np
import parameters
from numpy.random import normal
from tqdm import tqdm
import random
import standard_monte_carlo.euler_maruyama_step as euler_maruyama_step

def euler_maruyama(X0,dt,R):

    #### Variant of the homonymous standard_monte_carlo.py function ####
    #
    # INPUT:
    # X0 = starting point of the trajectory
    # dt = euler-maruyama discretization step
    # R = radius that corresponds to the entrance region (here, for the splitting method,
        flexible)
    #
    # OUTPUT:
    # entrance = boolean that states if the particle has entered the well or not
    # np.append(X,t) = 3D vector that contains the location and time of the arrival state

    T = parameters.T
    X = X0[0:2]
    entrance = False
    t = X0[2]

    while (t*dt<T and entrance == False):

        W = normal(size=2)
        X,t,entrance = euler_maruyama_step(X, t, dt, W)

    return entrance, np.append(X,t)


def splitting_method(n_iter,dt,n_levels):

    #### Splitting method main function ####
    #
    # INPUT:
    # n_iter = cardinality of Monte Carlo samples
    # dt = euler-maruyama discretization step
    # n_levels = number of splitting method levels (optimal choice = -ln(true_value)/2 )
    #
    # OUTPUT:
    # p = estimated probability

    print ('\n ------ SPLITTING METHOD ------ \n')
    p = 1.0
    X0 = np.append(parameters.X0_SM, 0) # 3D vector: x,y,t
    starting_points = np.tile(X0, (n_iter,1))  # n_iter copies of X0
    dR = (np.linalg.norm(X0)-parameters.R)/(n_levels) # radius constant increment
    R = np.linalg.norm(X0) - dR # initial radius

    for l in tqdm(range(n_levels)): # loop over all levels
        p_l = np.zeros(n_iter) # initialization of the probabiliy estimations at level l
        starting_points_next = [] # the list that will be used in the next level as starting points
        for X_idx, X_l in enumerate(starting_points):
            entrance, X_final = euler_maruyama(X_l, dt, R)
            if (entrance==True):
                p_l[X_idx] = 1
                starting_points_next.append(X_final) # every state that reaches the next level is added to the starting point list
        p = p * p_l.mean() # main splitting method passage: product of conditional probabilities
        R = R - dR # update the radius for the new level
        print(" Level probability = ", p_l.mean())
        starting_points = random.choices(starting_points_next,k=n_iter) # to sample with repetitions

    print ('X0 = ('+str(X0[0]) +','+ str(X0[1]) +'): ', p)

    return p
```