École Polytechnique Fédérale de Lausanne

Pollutant Transport Rare Events

MATH-414: Stochastic Simulation

Student: Giulia Mescolini

19th January, 2022

1 THE PROBLEM

This project proposes a stochastic approach to pollutant transport modelling in groundwater.
The particular phenomenon that we are analyzing is set in an infinite 2D aquifer region where a drinking well B occupies the ball of radius R centered in the origin, denoted as $B(0, R)$.
Hence, the domain we are focusing on is $D = \mathbb{R}^2 \setminus B(0, R)$. In this area, we study the contamination of the well by some particles of pollutant, drifted by a fluid with velocity $\mathbf{u}$ whose motion is governed by *Darcy Equations*:

$$\begin{cases} \nabla \cdot \mathbf{u} = 0 & \text{in } D \\ \mathbf{u} = -k\nabla p & \text{in } D \\ \mathbf{u} \cdot n = \frac{Q}{2\pi R} & \text{on } \partial B \\ \text{+ boundary conditions at infinity} \end{cases} \tag{1}$$

where p is the pressure of the fluid and k is the permeability of the porous medium (the soil, in this case), which has a diffusive effect on particles.
Given a starting point $(X_0, Y_0) \in D$, the trajectories of the particles are described by the following system of stochastic differential equations:

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

where σ represents the diffusion related to the porous medium and $W_1(t), W_2(t)$ are two independent standard Brownian motions, i.e. $\{W_i(t), t \geq 0, W_i(0) = 0\}$, where $i = 1, 2$ and:

- $\forall 0 \leq t_1 < t_2 \leq t_3 < t_4 \quad W_i(t_2) - W_i(t_1)$ is independent of $W_i(t_4) - W_i(t_3)$;
- $\forall 0 \leq t_1 < t_2 \quad W_i(t_2) - W_i(t_1) \sim \mathcal{N}(0, t_2 - t_1)$;
- each path $W_i(t)$ is continuous.

We are interested in the probability that a specific particle has to pollute the well within a certain time horizon T, and in our model the contamination happens if the particles reach the well B.
We introduce a random variable indicating the first passage time of the particle through the well:

$$\tau = \inf\{t \geq 0 : (X(t), Y(t)) \in B\}$$

Hence, our goal is to estimate the probability that the particle, starting from a position $(X_0, Y_0) \in D$, reaches B before time T: $\mu := \mathbb{P}(\tau \leq T | (X(0), Y(0)) = (X_0, Y_0))$.

DATA

- Time limit: $T = 1$
- Diffusion coefficient: $\sigma = 2$
- Mass rate: $Q = 1$
- Radius of the well: $R = 1$
- Drift velocity arising from a perturbation of $\mathbf{u}^{\text{steady}} = (1, 0)^T$ due to the presence of the well:

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

2 Stochastic simulation of the SDE

2.1 Method

In order to simulate Equation 2, we first needed to discretize it in time.
After having introduced a temporal grid with M subintervals of $[0,T]$ of equal width Δt, we used the *Euler-Maruyama* method:

$$\begin{cases} X_{k+1} = X_k + u_1(X_k, Y_k)\Delta t + \sigma\sqrt{\Delta t}Z_k, & Z_k \sim \mathcal{N}(0,1) \\ Y_{k+1} = Y_k + u_2(X_k, Y_k)\Delta t + \sigma\sqrt{\Delta t}Z'_k, & Z'_k \sim \mathcal{N}(0,1) \\ X_0,\, Y_0 \text{ given, } Z_k \text{ and } Z'_k \text{ independent} \end{cases} \tag{3}$$

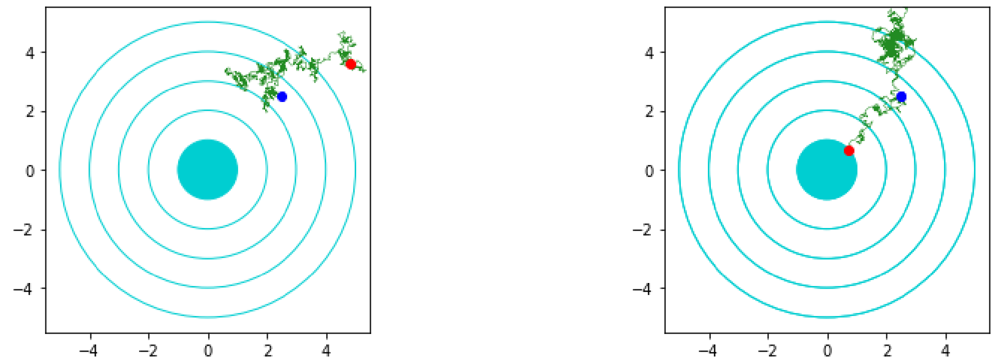
In choosing the parameter Δt, we respected the constraint $\Delta t < R^2$, which ensured that steps are not too wide to jump over the well. In our case, it is trivial to respect this condition as $\Delta t \leq T = 1 = R^2$, so any time step smaller than 1 could in principle be fine. At the same time, we stress the fact that the resolution of the time grid has an impact on the accuracy of the approximation. Hence, we made experiments with different values of Δt and compared the performances in order to study the order of the discretization in time, which will be discussed in subsection 2.3.
As the quantity we want to estimate is $\mu = \mathbb{P}_{(X_0,Y_0)}(\tau \leq T)$[1], we can first recognize the fact that we are working in the typical Monte Carlo setting:

$$\psi(\tau) = I_{\{\tau \leq T | (X_0,Y_0)\}} = \tilde{\psi}(Z_1, Z'_1, ...)$$

Then, $\mu = \mathbb{E}[\psi(\tau)] = \mathbb{E}[\tilde{\psi}(Z_1, Z'_1, ...)]$.
We remark that the two assumptions of the Monte Carlo method hold:

- we do not know the probability distribution of the random variable $\psi(\tau)$
- we can generate iid replicas of $\psi(\tau)$

Therefore, we first implemented a *Monte Carlo estimator* $\hat{\mu}$ based on independent simulations of the paths.
In our case, we generated iid paths $\{(X_k, Y_k)\}_k$, up to the limit time or to the time of impact with the well (if the particle reached the well within the limit time), and some examples can be seen in Figure 1.



(A) Particle not polluting the well.

(B) Particle polluting the well.

FIGURE 1
Two possible trajectories for a particle starting from $(2.5, 2.5)$.

Then, by averaging the number of experiments in which the particle hit the well over the total number of simulations, we got an estimator for μ.

[1] We switch to this notation to increase readability: the subscript (X_0, Y_0) represents the conditioning on the starting position.

The number N of the simulations to perform was chosen with a *two stages Monte Carlo method*:

Algorithm 1 Two stages Monte Carlo

```
1: Fix the number of replicas for the pilot run, the desired tolerance and the level of confidence:
   $\bar{N} = 1000$, $tol$ (depending on the starting position, see subsection 2.2), $\alpha = 0.05$.
2: Perform a pilot run with $\bar{N}$ replicas $(\psi^{(1)}, ..., \psi^{(\bar{N})})$ and compute:
   $\hat{\mu}_{\bar{N}} = \frac{1}{\bar{N}} \sum_{i=1}^{\bar{N}} \psi^{(i)} \quad \hat{\sigma}^2_{\bar{N}} = \frac{1}{\bar{N}-1} \sum_{i=1}^{\bar{N}} (\psi^{(i)} - \hat{\mu}_{\bar{N}})^2$
3: Fix
   $N = \frac{c^2_{1-\alpha/2} \hat{\sigma}^2_{\bar{N}}}{tol^2}$
   being $c_{1-\alpha/2}$ the quantile of order $\alpha/2$ of a standard normal.
4: Generate a run with $N$ replicas $(\psi^{(1)}, ..., \psi^{(N)})$and output $\hat{\mu}_N, \hat{\sigma}^2_N$
5: if $\hat{\sigma}^2_N > \hat{\sigma}^2_{\bar{N}}$ then
6:     Set $\bar{N} = N$ and go back to 2
7: else
8:     Output $\hat{\mu}_N$ with its confidence interval[2]$\hat{I}_{\alpha,N} = \left[\hat{\mu}_N \pm c_{1-\alpha/2} \frac{\hat{\sigma}_N}{\sqrt{N}}\right]$
9: end if
```

2.2 RESULTS

We report in Table 2 the estimates of μ for different starting positions and simulations performed with $\Delta t = 10^{-2}$. For each starting position, we reported in Table 1 the number N of replicas that should be performed, determined in each case with algorithm 1.
Note that the tolerance level required varies with the starting position; the intuition behind this is that the target probability has a different order of magnitude depending on how close is the starting point to the well (which is actually confirmed by the numerical solution in section 3), hence the tolerance set should change, in order to state with a given confidence that the first significant digits are right and should be more restrictive as the probability gets smaller.

Unfortunately, for the starting point $(3.0, 4.0)$ the N required is not feasible. Hence, we decided to limit ourselves to $N = 50.000$, even if it leaded to a confidence interval wider than our desired tolerance, as it will be reported in Table 2.

In addition to the three assigned starting points, we computed the probability for two other ones, which are shown in Figure 2:

- point $(-1.2, 1.1)$, which is on the left of the well, at the same distance as point $(1.2, 1.1)$. We expect that in this case the probability of reaching the well is higher, as the drift moves it into the well.
  In this case, a tolerance of 1×10^{-2} leads to $N = 36592$.
- point $(7, 7)$, which is extremely far from the the well; we expect that in this case the probability of contamination is nearly 0, and we will deal with this starting point in section 5, when a new technique for the simulation of rare events is implemented.

Note that, for the furthest point from the well, the Monte Carlo technique blows up, since we are employing a N that is too small, and the output estimated probability is 0 even for the largest N that can be considered given the computational power of a laptop (we tried with $N = 50.000$). We will consider this point in the framework of rare events simulation, with the *splitting method*.

[2] The confidence interval is obtained exploiting the *asymptotic normality* of the MC estimator, and the fact that $\frac{\sigma}{\hat{\sigma}_N} \to 1$ a.s.

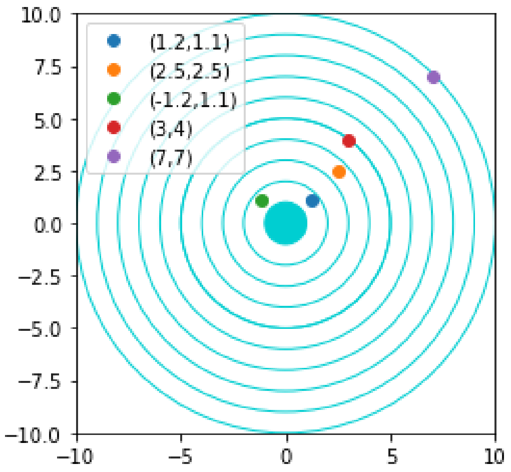


FIGURE 2
Starting points (X_0, Y_0) considered.

(X_0, Y_0)	N	Tolerance
$X_0 = 1.2, Y_0 = 1.1$	38005	1×10^{-2}
$X_0 = 2.5, Y_0 = 2.5$	31430	5×10^{-3}
$X_0 = 3.0, Y_0 = 4.0$	9173358	1×10^{-4}

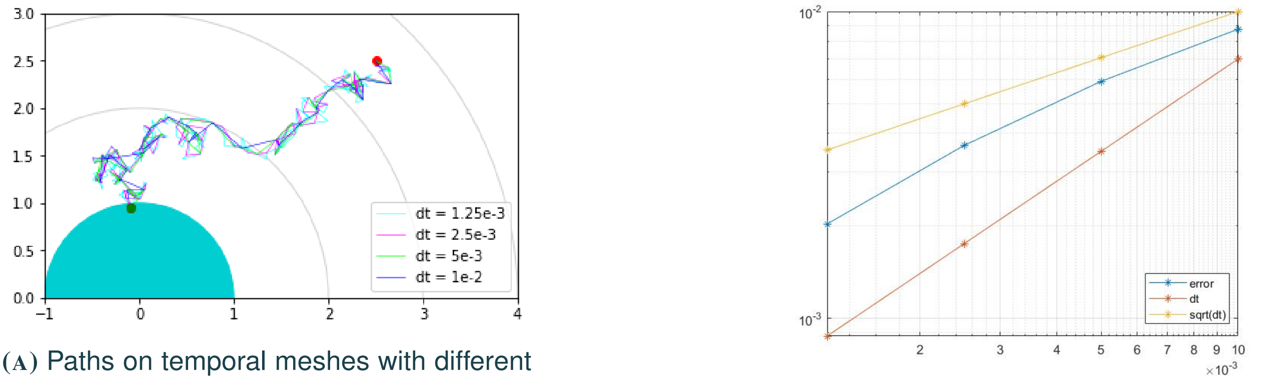
TABLE 1
Number of replicas N to perform, according to algorithm 1.

(X_0, Y_0)	$\hat{\mu}_N$	$\hat{\sigma}^2_N$	α - Confidence Interval
$X_0 = 1.2, Y_0 = 1.1$	0.4421	0.2467	$[0.4421 \pm 0.004993]$
$X_0 = 2.5, Y_0 = 2.5$	0.0529	0.0501	$[0.0529 \pm 0.002476]$
$X_0 = 3.0, Y_0 = 4.0$	0.0082	0.0081	$[0.0082 \pm 0.000789]$
$X_0 = -1.2, Y_0 = 1.1$	0.6195	0.2357	$[0.6195 \pm 0.004975]$
$X_0 = 7.0, Y_0 = 7.0$	0.0000	0.0000	-

TABLE 2
Monte Carlo estimates of μ for $\Delta t = 10^{-2}$

2.3 ERROR ESTIMATE WITH RESPECT TO Δt

Together with N, Δt is another crucial parameter for the quality of the simulation.
We analyzed its impact by running simulations *on the same Brownian paths* for $\Delta t = 10^{-2}, 5 \times 10^{-3}, 2.5 \times 10^{-3}, 1.25 \times 10^{-3}$ keeping N fixed ($N = 10.000$ to get a sustainable computational time). From Figure 3a we can notice the different resolution of the trajectories.



(A) Paths on temporal meshes with different resolutions.

(B) Order of convergence in logarithmic scale.

FIGURE 3
Study of the impact of time resolution.

The estimated probabilities and variances are reported in Table 3 and Table 4:

(X_0, Y_0)	$\Delta t = 10^{-2}$	$\Delta t = 5 \times 10^{-3}$	$\Delta t = 2.5 \times 10^{-3}$	$\Delta t = 1.25 \times 10^{-3}$
$X_0 = 1.2, Y_0 = 1.1$	0.4407	0.4606	0.4775	0.4870
$X_0 = 2.5, Y_0 = 2.5$	0.0548	0.0570	0.0592	0.0612
$X_0 = 3.0, Y_0 = 4.0$	0.0086	0.0092	0.0095	0.0098

TABLE 3
Values of $\hat{\mu}$ for varying Δt.

(X_0, Y_0)	$\Delta t = 10^{-2}$	$\Delta t = 5 \times 10^{-3}$	$\Delta t = 2.5 \times 10^{-3}$	$\Delta t = 1.25 \times 10^{-3}$
$X_0 = 1.2, Y_0 = 1.1$	0.2465	0.2485	0.2495	0.2499
$X_0 = 2.5, Y_0 = 2.5$	0.0518	0.0538	0.0557	0.0575
$X_0 = 3.0, Y_0 = 4.0$	0.0085	0.0091	0.0094	0.0097

TABLE 4
Values of $\hat{\sigma}^2$ for varying Δt.

We note that the values of $\hat{\mu}$ are increasing, and, as it will be shown in section 3, they get closer to the numerical deterministic solution; variances increase, too.
Using the numerical solution as reference for the calculation of the error, we performed a study of the order of convergence of our method with respect to Δt, reported in Figure 3b, for the case of the starting point $(2.5, 2.5)$. In order to reduce the influence of the seed, for this study we performed a simulation with $N = 100.000$, lasting ~ 70 min. Our empirical results showed an order of 1 of the method with respect to the time discretization.

3 NUMERICAL SOLUTION

There exists a deterministic approach to compute the target probability, too; indeed, the probability of hitting the well by time T is given by:

$$\mathbb{P}(\tau \leq T | \mathbf{X}(0) = (X_0, Y_0)) = \phi(\mathbf{X}(0), 0)$$

where $\phi(\mathbf{x}, t) \in \mathcal{C}^{(2,1)}(D \times (0, T))$ solves the backward parabolic PDE arising from the *Feynman-Kac* formula:

$$\begin{cases} \phi_t + (\mathbf{u} \cdot \nabla)\phi + \frac{1}{2}(\sigma^2 \Delta \phi) = 0 & \text{in } D \times [0, T] \\ \phi = 1 & \text{on } \partial B \times [0, T] \\ \phi(\mathbf{x}, t) \to 0 & \text{as } |\mathbf{x}| \to \infty \\ \phi(\mathbf{x}, T) = 0 & \text{in } D \end{cases} \tag{4}$$

In Figure 5 we report the results obtained by approximating Equation 4 with `FeNiCS`.
For the discretization in time, we introduced a temporal grid on $[0, T]$, with $N = 100$ subintervals of length $\Delta t = 10^{-2}$. We chose an **implicit** method for the solution of the system: denoting by $\phi^n(\mathbf{x})$ the solution at each time instant $t_n = n\Delta t$, we approximated the time derivative with

$$\phi_t(\mathbf{x}) \approx \frac{\phi^{n+1}(\mathbf{x}) - \phi^n(\mathbf{x})}{\Delta t}$$

and solved, for $t_n = t_{N-1}, ..., t_0$ the semi-discrete problem[3]

$$\begin{cases} \phi^{n+1} - \phi^n + \Delta t(\mathbf{u} \cdot \nabla)\phi^n + \frac{1}{2}\Delta t(\sigma^2 \Delta \phi^n) = 0 & \text{in } D \\ \phi^N = 0 & \text{in } D \\ + \text{ spatial boundary conditions of Equation 4} \end{cases} \tag{5}$$

[3] By simplicity of notation we omit the dependence on the space variable $\mathbf{x}$, but note that at this stage the problem is not yet discretized in space.

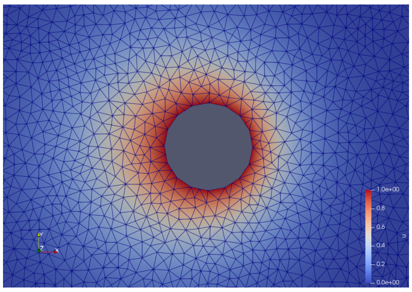


FIGURE 4
Zoom close to the well of the mesh, with the solution plotted.

(X_0, Y_0)	μ_{num}	$\hat{\mu}_N$
$X_0 = 1.2, Y_0 = 1.1$	0.5063	0.4421
$X_0 = 2.5, Y_0 = 2.5$	0.0615	0.0529
$X_0 = 3.0, Y_0 = 4.0$	0.0095	0.0082
$X_0 = -1.2, Y_0 = 1.1$	0.6754	0.6195
$X_0 = 7.0, Y_0 = 7.0$	4.5635×10^{-7}	0.0000

TABLE 5
Comparison between numerical solution of the PDE and output of the stochastic simulation.

Note that this reflects an implicit way to solve the system because we are in the framework of backwards PDEs: what is known at each time step is ϕ^{n+1}, and the goal is to determine ϕ^n.
We recall that `FeNiCS` requires the *weak formulation* of the problem, which was obtained by multiplying by a *test function* v belonging to the same space as ϕ and by integrating over the domain; we do not report it here, for the sake of conciseness, but it can be found in Appendix A, and it is obtained right after an integration by parts on the term with the laplacian.
To discretize the problem in space, first of all we restricted our infinite domain to

$$\Omega = B((0,0), R) \setminus B((0,0), 1)$$

with $R = 10$; after making trials with bigger radii, indeed, we found out that at a distance of ≈ 10 from the well the solution is below 10^{-10}. Therefore, only for the solution evaluation at point $(7.0, 7.0)$, we decided to keep $R = 20$ in order to ensure that the target point is far enough from the domain boundary.
We generated a triangular mesh with 100 refinements on the diameter (see Figure 4) and imposed *Dirichlet Boundary Conditions* on the mesh boundaries.

- On the **inner boundary**, $\phi = 1$.
  Note that we are approximating a circle with the edges of triangles; hence, in the implementation, we kept in mind that the tolerance for the imposition of the BC cannot be as low as usual in `FeNiCS` for straight boundaries, but it should be bigger, else the condition is not applied to any point.
- On the **outer boundary**, $\phi = 0$, considering the solution to be 0 after $R = 10$ and cutting the infinite domain.

The Finite Elements chosen for the spatial discretization are the typical $\mathbb{P}_1^C$ *Lagrange* elements and by performing some mesh refinements we checked that the solution remained stable.

4 VARIANCE REDUCTION

In this section, we propose a technique for reducing the variance of the Monte Carlo estimator.
Given the problem structure, it is straightforward to identify negatively correlated paths (see Figure 5a) and the generation of a step of a path and of the one of *antithetic path* has the same computational cost.
Moreover, the *splitting method* proposed in 5 shows some similarities with Importance Sampling (as observed in [1]), hence we chose to investigate here another strategy.
The technique proposed is **Antithetic Variables**; we remark that the random variable $\psi = I_{\{\tau \leq T | (X_0, Y_0)\}}$ is a function of the vector of Gaussian increments $\mathbf{Z} := \{(Z_k, Z'_k)\}_k$, which are *independent* and with *symmetric distribution around the mean*.
Note that the function representing the relationship between ψ and $\mathbf{Z}$ is monotone *non-increasing* in $\mathbf{Z}$, as the probability of hitting the well increases as the steps Z_k, Z'_k decrease (for example, a negative value of Z_k means that X_{k+1} is closer to the well than X_n. Hence, we can construct a Monte Carlo approximation of μ with antithetic variables as the hypotheses of *Proposition 6.1* of [3] hold.

We are focusing on the case $(X_0, Y_0) = (2.5, 2.5)$, therefore we consider $N = 31430$ as done in section 2 (which is even, so it is fine for antithetic variables).
We generated $N/2$ iid replicas of the paths:

$$X_{k+1} = X_k + u_1(X_k, Y_k)\Delta t + \sigma\sqrt{\Delta t} Z_k, \quad Z_k \sim \mathcal{N}(0,1)$$

$$Y_{k+1} = Y_k + u_2(X_k, Y_k)\Delta t + \sigma\sqrt{\Delta t} Z'_k, \quad Z'_k \sim \mathcal{N}(0,1)$$

as well as the *antithetic paths*:

$$\tilde{X}_{k+1} = \tilde{X}_k + u_1(\tilde{X}_k, \tilde{Y}_k)\Delta t + \sigma\sqrt{\Delta t} Z_k, \quad Z_k \sim \mathcal{N}(0,1)$$

$$\tilde{Y}_{k+1} = \tilde{Y}_k + u_2(\tilde{X}_k, \tilde{Y}_k)\Delta t + \sigma\sqrt{\Delta t} Z'_k, \quad Z'_k \sim \mathcal{N}(0,1)$$

In output, we got two vectors $\mathbf{Z}$, $\mathbf{Z}_{av}$, containing 1 at position n if the well is reached in the limit time by the n-th path/antithetic path, 0 else. Then, we averaged them and computed $\hat{\mu}_{av}$, $\hat{\sigma}^2_{av}$:

$$\hat{\mu}_{av} = \frac{1}{N/2}\sum_{n=1}^{N/2} \frac{(\mathbf{Z})_n + (\mathbf{Z}_{av})_n}{2} \qquad \hat{\sigma}^2_{av} = \frac{1}{N/2-1}\sum_{n=1}^{N/2} \left(\frac{(\mathbf{Z})_n + (\mathbf{Z}_{av})_n}{2} - \hat{\mu}_{av}\right)^2$$

We report in Table 6 the results obtained, together with the amount of variance reduction achieved.

Method	$\hat{\mu}$	$\hat{\sigma}^2$	α - Confidence Interval
Crude Monte Carlo	0.0532	0.0503	$[0.0531 \pm 0.00248]$
Antithetic Variables	0.0514	0.0231	$[0.0514 \pm 0.00237]$

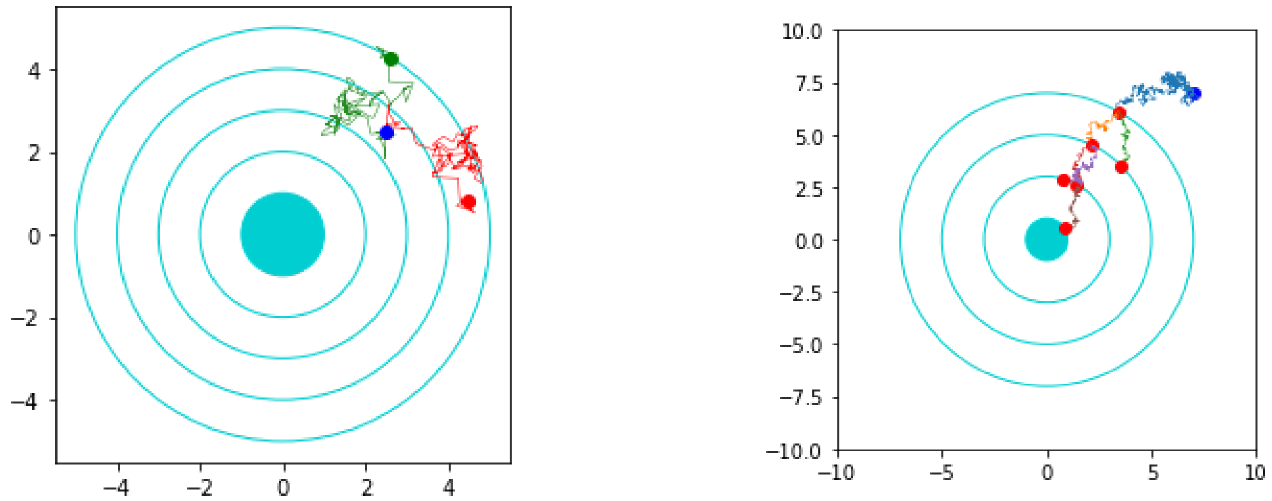
TABLE 6
Comparison between Crude Monte Carlo and Antithetic Variables in terms of output and variance.

The sample covariance between $\mathbf{Z}$ and $\mathbf{Z}_{av}$ is negative and equal to -0.002638, indeed we got a reduction of variance, as it can be seen by the reduction of width of the confidence interval ℓ:

$$\frac{\ell_{MC}}{\ell_{AV}} = 1.04491 > 1$$

5 SPLITTING METHOD

In this section we consider the estimate of the probability of polluting the well for a particle starting very far from it: $(X_0, Y_0) = (7.0, 7.0)$. In this case, as already verified in section 2, the traditional Monte Carlo technique outputs a 0 probability with the sample size affordable for our computational availability, and more sophisticated techniques are needed in order to estimate the probability of this *rare event*.
The reason behind this is discussed in Section 2.2.1 of [1]. Let us introduce a that *rarity parameter* L such that the rare event has a decreasing probability with respect to L (in our case $L = \sqrt{x^2 + y^2}$ is appropriate, and it represents the distance from the origin, which is the center of the well); algorithms may have an **exponential** or **polynomial** complexity with respect to L, and the algorithm will be asymptotically efficient only if it belongs to the **polynomial** class.
The *splitting method*, proposed by Kahn and Harris, falls into this class and aims at generating more occurrences of the rare event. This is obtained with a *divide et impera* technique: the probability of the rare event μ is viewed as the probability of an intersection of a nested sequence of events, so it is computed as a product of conditional probabilities which can be estimated more accurately.
We now present in Algorithm 2 the procedure followed, then we will discuss the choice of the *splitting parameters*.



(A) An example of antithetic paths starting from $(2.5, 2.5)$.

(B) Visual representation of the splitting algorithm (4 thresholds).

FIGURE 5
Antithetic Variables and Splitting Method.

In this case, we obtained a feasible computational time with $\Delta t = 10^{-3}$, hence we decided to keep this finer time resolution.

Algorithm 2 Splitting Algorithm

```
Choose in the domain $D$ a sequence of concentric circles $B = C_m \subset C_{m-1} \subset \cdots \subset C_1$, where each $C_i$ has radius $R_i$, with $R_i > R_{i+1}$.
For each level $C_i$, set $s_i$, the number of new paths to generate from each element of $S_{i-1}$, the set of all valid candidate starting points in the previous level (setting $S_0 = \{(X_0, X_0)\}$).
for all levels $C_i$, with $i = 1, \dots, m-1$ do
    for all valid starting points in the previous level do
        Generate $s_i$ paths.
        if a path hits $C_{i+1}$ before the limit time $T$ [4] then
            Increase a counter $n_i$ and store the hitting coordinates in $S_i$.
        end if
    end for
    Being $\tau_i = \inf\{t \geq 0 \ (X(t), Y(t)) \in C_i\}$, estimate the probability $\mathbb{P}(\tau_i \leq T | \tau_{i-1} \leq T)$ with
        $$\hat{c}_i = \frac{n_i}{|S_{i-1}| s_i}$$
end for
Estimate $\mu$ with the unbiased estimator $\hat{\mu} = \prod_{i=1}^{m} \hat{c}_i$.
```

5.1 CHOICE OF THE PARAMETERS

There are several parameters to be determined with this method, and we will now analyze their optimal choice inspired by [1, 2].

[4] This means that we must keep track of the hitting times, as well.

NUMBER OF STAGES: HOW MANY CIRCULAR CROWNS?

In Section 3.2.1 of [1] the optimal m is derived for the case in which the success probabilities do not depend on the entrance state into that level, and it corresponds to

$$m^* = \frac{-log(\mu)}{2}$$

In general, we need a pilot run to estimate μ, but as we have the solution of the PDE we could exploit that datum as reference value for μ, and get $m^* = 7$.
Note that it may not be too accurate to work under this assumption, because we recall that in our problem there is a position-dependent velocity field; however, we may assume as first approximation that its influence is limited, as the coordinate of starting points are pretty close to each other.
We performed a trial with a smaller number of stages as well, with $m = 4$, but the results got closer to the numerical ones using $m = 7$ (see Table 8).

NUMBER OF TRIALS: WHAT SHOULD s_i BE?

The choice of s_i has a huge impact on the computational effort; its optimal value is $n_i := 1/p_i$, being $p_i = \mathbb{P}(\tau_i \leq T|\tau_{i-1} \leq T)$. This value may be integer or not, and one could both use a randomized n_i as proposed in [2] or simply take the integer part of the fraction $1/p_i$; this was the strategy adopted, as the gain obtained with the first technique is minimal as reported in [1]. Note that, in order to have an estimate of the p_i, first we ran a pilot run with $s_i = 100\ \forall i = 0, ..., m-1$. The result obtained was already of the order of 10^{-7}, hence in line with the PDE result, but thanks to the update of the number of stages we reduced significantly the computational time.
As it can be seen from Table 7, the number of trials adopted is close to the reciprocal of the p_i, and only in the first stage we were forced to keep a high number of trials, else no hitting points were found in some of the following stages, which clearly made the simulation result drown to 0.

p_i	0.08	0.3125	0.16	0.1	0.1375	0.1234	0.0763
s_i	100	4	5	10	10	14	20

TABLE 7
Probabilities of hitting the next stage p_i vs number of trials s_i.

THRESHOLDS: WHICH CIRCULAR CROWNS SHOULD WE CONSIDER?

If p_i are known, there exists a recipe to choose the optimal thresholds, but this is clearly not our case.
In the choice of the thresholds, we must be consider if it is the case when multiple thresholds can be crossed at a time: in this case, indeed, we should refer to Section 3.3.2 of [1].
However, in our case we assumed that circular crowns are far one from each other, and given the entity of the displacements that take place (checked by visual inspection), we did not consider the multiple-crossing hypothesis and chose the circular crowns to be equally spaced.

5.2 VARIANCE

Denoting by R_m the number of hits of the last threshold (corresponding to the well), and as Y_j the number of hits caused by the offspring of first stage path j, we have that

$$\hat{\mu} = \frac{R_m}{\prod_{i=0}^{m-1} s_i} \Rightarrow Var(\hat{\mu}) = \frac{Var(R_m)}{\left(\prod_{i=0}^{m-1} s_i\right)^2} = \frac{Var\left(\sum_{j=1}^{s_0} Y_i\right)}{\left(\prod_{i=0}^{m-1} s_i\right)^2} = \frac{Var(Y_1)}{s_0\left(\prod_{i=1}^{m-1} s_i\right)^2}$$

where the last equality holds since Y_j are clearly independent one from each other. We studied the variance with two methods:

- by performing $N = 100$ (given the high computational effort) repetitions of the splitting algorithm, and then computing the sample variance of R_m.
- by computing the sample variance of Y_1. In this way, the computational time is definitely shorter, and we can see from Table 8 that the order of magnitude of the results is similar.

5.3 RESULTS

The result obtained are in line with the order of magnitude of the numerical solution of the PDE, as shown in Table 8.

	$n_{circles} = 4$	$n_{circles} = 7$
$\hat{\mu}$	2.2007×10^{-7}	5.1786×10^{-7}
$\hat{\sigma}$ (method 1)	5.7395×10^{-7}	8.7876×10^{-7}
$\hat{\sigma}^2$ (method 1)	3.2942×10^{-13}	7.7221×10^{-13}
$\hat{\sigma}$ (method 2)	2.2007×10^{-7}	5.0014×10^{-7}
$\hat{\sigma}^2$ (method 2)	4.8431×10^{-14}	2.5014×10^{-13}

TABLE 8
Estimated probabilities with Fixed Splitting.

	$n_{circles} = 4$	$n_{circles} = 7$
$\hat{\mu}$	1.6320×10^{-7}	4.9474×10^{-7}

TABLE 9
Estimated probabilities with Fixed Effort.

5.4 FIXED EFFORT VS FIXED SPLITTING

What we presented up to now is known as *Fixed Splitting* (FS) method, as we created a fixed number of offspring from each saved state, and we explained how to choose this number optimally.
Note that there exists an alternative: the *Fixed Effort* (FE) method, in which we create at every stage a fixed total number of offspring. We decided to implement this as well, because, as discussed in [1], the FS method is very sensitive to the choice of splitting levels and number of splits and has the risk of *explosion* and *die-out* of paths. Notice that in FE the problem of path explosion is solved, and we got control on the computational effort, while there still can be stages in which we do not obtain any hit. With a fixed effort of $N_{FE} = 1000$ offsprings per stage, we obtained the estimates in Table 9. There are no theoretical clues for the estimate of variance in this case, and it would be necessary to estimate it asymptotically, as stated in [1]. Hence, we decided to omit this aspect.

6 CONCLUSIONS

In this project, we proposed a stochastic approach to pollutant transport modelling. Our goal was to estimate the probability that a particle, starting from a given position, pollutes a drinking well; this value was estimated by solving numerically a backwards PDE with `FeNiCS` and with Monte Carlo techniques.
First of all, we implemented a *two stages Monte Carlo technique* for the simulation of the stochastic differential equation governing the phenomenon, with the *Euler-Maruyama* time discretization; then, we analyzed the impact on the solution of the refinement of the time mesh, and proposed an *Antithetic Variables* technique to reduce the variance of our estimator.
As for the case in which the particle starts very far from the well the Monte Carlo techniques turned out to be not adequate, we implemented two variants of the *Splitting Method* and attempted to tune its parameters following the theoretical results obtained in [1, 2].

The code can be found in the GitHub repo:
`https://github.com/giuliamesc/pollutant_transport`

Appendix A

Numerical Simulation

Core part of the numerical solution of the PDE:

```python
# Domain Generation
domain = mshr.Circle(df.Point(0, 0), R) - mshr.Circle(df.Point(0,0), r)
# Mesh Generation
mesh = mshr.generate_mesh(domain,100)

# Finite Elements Space: P1
V = df.FunctionSpace(mesh,"Lagrange",1)

# BCs
bcs = list()
def my_well(x,on_boundary):
        rad = np.sqrt(np.power(x[0],2)+np.power(x[1],2))
        return on_boundary and (np.abs(rad - r) < tol_int)

def my_out(x,on_boundary):
        rad = np.sqrt(np.power(x[0],2)+np.power(x[1],2))
        return on_boundary and (np.abs(rad - R) < tol_ext)
bcs.append(df.DirichletBC(V, df.Constant(1), my_well))
bcs.append(df.DirichletBC(V, df.Constant(0), my_out))

# Test and Trial Functions in the bilinear form
v = df.TestFunction(V)
phi = df.TrialFunction(V)

# Time discretization
N = int(T/dt)
times = np.linspace(0, T, N)
times = times[::-1]

phi_old = df.interpolate(df.Constant(0),V) # final condition

for i in range(1, len(times)):
    t = times[i]
    # Bilinear form
    F = (- phi*v - df.Constant(dt) * df.Constant(sigma**2/2) * df.inner(df.grad(phi), df.grad(v)) + df.Constant(dt) * (df.inner(df.as_vector((u1,u2)), df.nabla_grad(phi)))*v + phi_old * v ) * df.dx
    # Right-hand side
    a, rhs = df.lhs(F), df.rhs(F)
    # Solver
    df.solve(a == rhs, phi_old, bcs=bcs)

print(np.array(phi_old(X0,Y0)))
```

Appendix B

Two-stages Monte Carlo

Core part of the code needed for the first question of the assignment:

```python
# Function to produce N independent trajectories
def path(N):
    count = 0 # to store how many trajectories hit the well
    Z_mc = np.zeros([N,]) # to store whether each trajectory leads to a hit or not
    for i in range(N):
        # Starting values
        X0 = parameters.X0
        Y0 = parameters.Y0
        # Preparation of vectors where we store coordinates over all times until stop
        X = [X0]
        Y = [Y0]
        stop = False
        it = 0
        while(stop == False):
            # Generating two independent standard normals
            Z1 = np.float(np.random.normal(size = 1))
            Z2 = np.float(np.random.normal(size = 1))
            # Computation of the following position
            new_X = X0 + u1(X0,Y0)*dt + sigma * np.sqrt(dt)*Z1
            new_Y = Y0 + u2(X0,Y0)*dt + sigma * np.sqrt(dt)*Z2
            # Storage of the next position
            X.append(new_X)
            Y.append(new_Y)
            # Checking wheter time limit T is reached
            if (it > int(T/dt)) : stop = True
            # Checking whether the well is hit
            if (killing_boundary(new_X,new_Y)) :
                stop = True # stop the current trajectory
                count = count + 1 # update counter of success
                Z_mc[i] = 1 # register a success for the current trajectory

            it = it + 1
            # Update coordinates
            X0 = new_X
            Y0 = new_Y

    return count/N, X, Y, Z_mc

tol = parameters.tol # tolerance requested
N = 1000 # N for pilot run
dt = 1e-2 # temporal step for discretization
mu_hat, X, Y, Z_mc_hat = path(N) # running N paths
sigma_hat = np.var(Z_mc_hat, ddof = 1) # computing empirical variance
new_N = int(np.ceil(C_alpha**2*sigma_hat/(tol/2)**2)) # computing the new N
```

Generation of paths with different refinements on the same Brownian Paths:

```python
def res_path(N):
    # Time discretizations
    dt_c = 1e-2
    dt_f = dt_c/2
    dt_ff = dt_f/2
    dt_fff = dt_ff/2
    # Counters of hits
    ...

    for i in range(N):
        # Starting values
        ...
        # Boolean variables to stop the procedure
        ...
        # Cleaning brownian paths of coarser paths
        W1_c = 0
        W2_c = 0
        ...
        while(stop_fff == False or stop_ff == False
                or stop_f == False or stop_c == False):
            it = it + 1 # incrementing the iteration counter
            # Generating the Brownian path for the finest path
            W1_fff = np.sqrt(dt_fff)*np.float(np.random.normal(size = 1))
            W2_fff = np.sqrt(dt_fff)*np.float(np.random.normal(size = 1))

            # Accumulation the brownian path for the other two resolutions
            W1_ff = W1_ff + W1_fff
            W2_ff = W2_ff + W2_fff
            ...

            # FINEST PATH
            # Computation of the following position
            new_X_fff = X0_fff + u1(X0_fff,Y0_fff)*dt_fff + sigma * W1_fff
            new_Y_fff = Y0_fff + u2(X0_fff,Y0_fff)*dt_fff + sigma * W2_fff

            # Checking whether the well is hit
            if (killing_boundary(new_X_fff,new_Y_fff)) :
                if stop_fff == False:
                    stop_fff = True
                    count_fff = count_fff + 1
                    Z_mc_fff[i] = 1
            # Updating the position
            X0_fff = new_X_fff
            Y0_fff = new_Y_fff

            # SECOND FINEST PATH: update it every two iterations
            if np.mod(it,2) == 0 :

                # Computation of the following position
                new_X_ff = X0_ff + u1(X0_ff,Y0_ff)*dt_ff + sigma * W1_ff
                new_Y_ff = Y0_ff + u2(X0_ff,Y0_ff)*dt_ff + sigma * W2_ff

                # Updating the position
                X0_ff = new_X_ff
                Y0_ff = new_Y_ff

                W1_ff = 0 # cleaning W_f for the next iteration
                W2_ff = 0 # cleaning W_f for the next iteration
            else : # do not update, just store
                new_X_ff = X0_ff
                new_Y_ff = Y0_ff

            # Checking whether the well is hit
            if (killing_boundary(new_X_ff,new_Y_ff)) :
                if stop_ff == False:
                    stop_ff = True
                    count_ff = count_ff + 1
                    Z_mc_ff[i] = 1

            # MEDIUM-FINE PATH: update it every four iterations
            ...
            # COARSE PATH: update it every eight iterations
            ...
            # Checking wheter time limit T is reached
            if (it > int(T/dt_fff)) :
                stop_fff = True
                stop_ff= True
                stop_f = True
                stop_c = True
```

Appendix C

Variance Reduction

Core part of the code needed for the second question of the assignment:

```
# Defining the trajectories for CMC and Antithetic Variables
def paths(K):
    for i in range(K):

        # Generating two independent standard normals
        Z1 = np.float(np.random.normal(size = 1))
        Z2 = np.float(np.random.normal(size = 1))

        # Storing stop values in order to stop updates when the well is hit
        stop_mc = False
        stop_av = False

        # Updating crude MC
        ...

        # Updating AV with the same random realization
        if stop_av == False:
            u1_old = u1(x_av,y_av)
            u2_old = u2(x_av,y_av)
            # Position update with minus (antithetic path)
            x_av = x_av + u1_old*dt - sigma * np.sqrt(dt)*Z1
            y_av = y_av + u2_old*dt - sigma * np.sqrt(dt)*Z2
            # Position storage
            Xav.append(x_av)
            Yav.append(y_av)
            # Checking wheter the path hits the well
            if (killing_boundary(x_av,y_av)) :
                finished_av = 1
                stop_av = True

    return finished_mc,finished_av,X,Y,Xav,Yav

# Antithetic Variables
for i in range(int(N2)):
    Z_mc2[i],Z_av[i],X,Y,Xav,Yav = paths(K)

Y_av = 0.5*(Z_mc2+Z_av)
mean_av=np.mean(Y_av) # estimate of AV
```

Appendix D

Splitting Method

Core part of the code needed for the third question of the assignment:

```python
def splitting():
    X_start = [X0]
    Y_start = [Y0]
    times = [0] # to store after how many steps we hit the inner ball
    P = np.zeros(len(R)) # vector to store probabilities
    counts = np.zeros(len(R))
    parents = [] # to store parents
    for i in range(len(R)): # for each level
        if task == ’FSestimation’ or task == ’FSvariance’ :
            its = int(iters[i]) # get how many iterations to perform for each valid starting point
        if task == ’FEestimation’:
            its =  int(N_fe[i] / len(X_start))
        den = its * len(X_start) # computing the denominator needed to divide P at the end: number of trials per starting point * number of starting points
        X_new = [] # I will store here the valid ending points found
        Y_new = []
        times_new = []
        parents_new = []
        for j in range(len(X_start)): # for each starting point
            t_old = times[j] # time employed to reach the current starting point
            if i > 0:
                parents_old = parents[j]
            for k in range(its):
                if i == 0 :
                    parents_old = k
                finished,x_new,y_new,temp_t = subpath(K,X_start[j],Y_start[j],R[i]) # generate its time a trajectory starting from my starting point
                t_new = temp_t + t_old
                if finished == 1 and t_new <= K :
                    X_new.append(x_new) # store the new initial position
                    Y_new.append(y_new)
                    times_new.append(t_new)
                    parents_new.append(parents_old)
                    P[i] = P[i] + 1 # update counter
        counts[i] = P[i]
        P[i] = P[i]/den # count the fraction of successes
        X_start = np.copy(X_new) # valid ending points become the new starting points
        Y_start = np.copy(Y_new)
        times = np.copy(times_new)
        parents = np.copy(parents_new)
    return counts[-1], np.prod(P), parents
```

BIBLIOGRAPHY

[1] MJJ. Garvels. *The splitting method in rare event simulation*. PhD thesis, Univerisry of Twente, 2000.

[2] DP. Kroese, T. Taimre, and IB. Zdravko. *Handbook of Monte Carlo Methods*, volume 706. John Wiley and Sons, 2013.

[3] F. Nobile. *Stochastic Simulation. Lecture Notes*. 2021.