ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

PSI

On Monte Carlo methods for the Poisson equation

Bachelor Thesis

Department of Physics

ETH Zurich

written by

Caspar Schucan

supervised by

Dr. A. Adelmann (ETH)

scientific advisers

Dr. Mohsen Sadr

June 30, 2025

Abstract

Parabolic Partial Differential Equations (PDE) and, in particular, the Poisson equation frequently arise in various fields, including plasma physics. The Feynman-Kac formula expresses the relationship between the value of a solution to parabolic PDEs and the expectation of a random variable, enabling Monte Carlo methods for pointwise evaluation. This project compares multiple approaches to using the Feynman-Kac formula for the Poisson equation. Depending on the problem, we observe that a modified Walk-on-Sphere algorithm is 100 times more computationally efficient than the naive Euler-Maruyama approach. To further improve performance by reducing variance, we expand the Walk-on-Sphere method to a multilevel formulation. Our results find a speedup for the multilevel formulation that increases logarithmically with the problem size and grows to 3.5x in our experiments.

Contents

Chapter 1

Introduction

The Feynman-Kac formula [1], discovered in 1949 by its namesakes Mark Kac and Richard Feynman, connects stochastic differential equations to parabolic Partial Differential Equations (PDE). Its formulation of point solutions of such a PDE, as expected values, makes it very attractive for using Monte Carlo methods. The Walk-on-Spheres (WoS) algorithm, suggested shortly after the introduction of the Feynman-Kac formula for the solution of the Laplace equation with Dirichlet boundary conditions by Müller 1956 [2], remains one of the prevailing methods for generating the random walks, which are needed to apply the Feynman-Kac formula. The WoS method works by always taking the biggest step possible, which cannot leave the domain, in a random direction, hence jumping from sphere to sphere. Tweaking this general principle allows the WoS method to be applied to many different problems. Mikhailov 1992 [3], Hwang et al. 2001 [4] and Bossy et al 2010 [5] use it to develop an algorithm for the Poisson-Helmholtz equation. Delaurentis and Romero 1990 [6] and Hwang et al 2002 [7] develop a WoS method for the general Poisson equation. Research remains active around the WoS algorithm. Kyprianou et al. 2018 [8] broaden the WoS method to solve fractional Laplace equations, and Nam et al. 2024 [9] suggest using Walk-on-Spheres to help neural networks solve the Poisson equation. The WoS solution to the Poisson equation, like any Monte Carlo method, is subject to noise and, therefore, a natural candidate for optimisation through variance reduction techniques.

Multilevel Monte Carlo Methods, first introduced by Giles 2008 [10], are a widely used variance reduction method for traditional Monte Carlo algorithms. By sampling at different discretisation levels and cleverly correlating these samples, we effectively balance cost and accuracy. Pauli et al. 2013 [11] uses a Multilevel formulation of WoS for the Laplace equation. They prove that this variation improves the asymptotic WoS convergence for the Laplace equation.

In this work, we extend MLMC to the WoS method of solving the Poisson equation proposed by Delaurentis and Romero [6] and show its efficiency in n-dimensional phase space.

In chapter 2, we will introduce the Feynman-Kac formula and apply it to the Poisson equation. We will continue in chapter 3, describing the WoS and Euler-Maruyama [12] algorithms, the latter being a more naive approach to generating random walks. Furthermore, we will explore the MLMC technique in more detail and discuss its convergence. We will examine the results of a simple Python implementation restricted to two dimensions in chapter 4 and a dimension-agnostic C++ implementation in chapter 5. Finally, in chapter 6, we will provide our conclusions.

Chapter 2

Background

2.1 Wiener process

The Wiener process [12], sometimes called Brownian motion, is a specific stochastic process $W = \{W_t, t \geq 0\}$ where each increment is normally distributed. Formally, we note all the conditions as

$$W = \{W(t); t \geq 0\}, \text{ where } W(0) = 0,\ \mathbb{E}[W(t)] = 0,\ \text{Var}(W_t - W_s) = t - s. \tag{2.1}$$

2.2 The Feynman-Kac formula

The Feynman-Kac formula gives a probabilistic solution to parabolic PDEs. The general form of such a PDE is:

$$\begin{aligned} \mathcal{L}[\phi] &= \Delta\phi(\mathbf{x}) + \sum_{i=1}^{n} b_i(\mathbf{x}) \frac{\partial}{\partial x_i} \phi(\mathbf{x_i}) + c(\mathbf{x})\phi(\mathbf{x}) + g(\mathbf{x}) = 0, && \mathbf{x} \in \Omega \subseteq \mathbb{R}^n \\ \phi(\mathbf{x}) &= f(\mathbf{x}), && \mathbf{x} \in \partial\Omega \end{aligned}$$

To describe the Feynman-Kac formula [1] fully, we must define some notations when discussing random walks. We will denote X_t as the Brownian motion at time t starting at $\mathbf{x_0}$. Further, we will use the letter τ to denote the first exit time, meaning the smallest $\tau \in \mathbb{R}^+$ such that $X_\tau \in \partial\Omega$. We will call X_τ the first exit point consistent with the above. Finally, we have all the tools to write down the full Feynman-Kac formula:

$$\phi(\mathbf{x}_0) = \mathbb{E}\left[f(X_\tau) \exp(\int_0^\tau c(X_s)ds) + \int_0^\tau g(X_t) \exp(\int_0^t c(X_s)ds)dt \right].$$

For this project, we only consider the Poisson equation $-\Delta\phi(\mathbf{x}) = g(\mathbf{x})$. Applying the Feynman-Kac formula to this specific PDE, we get

$$\phi(\mathbf{x}_0) = \mathbb{E}\left[f(X_\tau) + \int_0^\tau g(X_t)dt \right]. \tag{2.2}$$

Chapter 3

Method

We investigated two methods, namely the Euler-Maruyama and the Walk-on-Spheres algorithms. Both generate random walks to which we can apply the Poisson-specific Feynman-Kac formula (2.2). Both of these methods are well-suited to variance reduction using Multilevel Monte Carlo.

3.1 Multilevel Monte Carlo

Multilevel Monte Carlo (MLMC) is a variance reduction technique that can be applied to many different Monte Carlo methods. First suggested by Giles in [10], it uses different discretisation "levels" to optimise the total work required to reach a desired variance. A discretisation parameter characterises these different levels, which we will call δ. The interpretation of this parameter varies for different methods to generate random walks. At the level $l \in \mathbb{N}$, $\delta_l = \eta^{-l}\delta_0$ where δ_0 is the discretisation parameter at the 0th level. Further, Y will denote the Feynman-Kac integral (2.2) over one instance of a random walk.

$$Y = f(X_\tau) + \int_0^\tau g(X_t)dt.$$

And hence Y_{δ_l} will be the random variable where X_t is simulated with discretisation parameter δ_l. Using this, we can write the MLMC-estimator as:

$$\mathbb{E}[Y_{\delta_L}] = \mathbb{E}[Y_{\delta_0}] + \sum_{l=1}^{L} \mathbb{E}[Y_{\delta_l} - Y_{\delta_{l-1}}]. \tag{3.1}$$

In the following, we will use that

$$Y_l = \begin{cases} Y_{\delta_l}, & l = 0 \\ Y_{\delta_l} - Y_{\delta_{l-1}}, & l > 0 \end{cases}$$

Which turns the previous expression (3.1) into:

$$\mathbb{E}[Y_{\delta_L}] = \mathbb{E}[Y_0] + \sum_{i=1}^{L} \mathbb{E}[Y_l].$$

For this MLMC estimator to have merit in terms of performance and correctness, we need $\lim_{l\to\infty} Y_{\delta_l} = Y$ and $\mathrm{Var}(Y_l) \leq \mathrm{Var}(Y_0)$ for all $l > 0$.

When given a target error ε to reach, F.Müller et al. [13] show that there exists an optimal number of samples per level M_l, minimising the total cost. The expression (3.2) assumes that we know the cost and variance of sampling at each level.

$$M_l = \frac{1}{\varepsilon^2}\sqrt{\frac{\mathrm{Var}(Y_l)}{w_l}}\sum_{k=0}^{L}\sqrt{\mathrm{Var}(Y_k)w_k} \tag{3.2}$$

where w_l denotes the expected cost of taking a sample at level l. This equation also takes for granted that L is big enough that the target error ε can be reached and is not smaller than the discretisation error inherent in Y_{δ_L}. In practice, the variance and cost of samples at each level are rarely known analytically, and neither is the exact discretisation error of a Y_{δ_l} nor the best number of levels L to use. Therefore, some warmup samples are typically used to estimate variance and cost per level. Usually, 3 levels are used to start meaning $L = 2$ since this is the smallest L that allows for a reasonable convergence estimate. Then Y_L, the difference between the two finest levels, is used to estimate the discretisation error. If needed more levels can be added later.

The general procedure can be summarised as such

1. Generate warmup samples for new level(s)
2. Estimate optimal number of samples per level
3. Take more samples at each level where there are not enough from previous iterations
4. Estimate current discretisation error
5. Stop if the estimated error is smaller than ε otherwise add a level and repeat

3.1.1 Error convergence

In the following, we will present a proof from S.Pauli et al. [11], which is applied to the convergence of an MLMC method to solve Laplace's equation. We noticed that the prerequisites of the proof also apply to our method.

Prerequisites

To prove a convergence rate of in $\mathcal{O}(\frac{1}{\sqrt{W}})$, we will establish some prerequisite results.

Optimal number of samples per level for an MLMC method Eq. (3.2)

Refinement of the discretisation parameter δ_l:

$$\delta_l = \eta^{-l}\delta_0, \text{ for some } \eta > 1. \tag{3.3}$$

Scaling of work per level here, which is a proven result for the walk-on-spheres (WoS) algorithm 2 in 2-dimensional domains or n-dimensional domains with smooth boundaries found in [14]:

$$w_l \in \mathcal{O}(\log(\delta^{-1})) = \mathcal{O}(l) \tag{3.4}$$

Scaling of variance sampling the finer levels.

$$\mathrm{Var}(Y_l) \in \mathcal{O}(\delta_l^{2s}) = \mathcal{O}(\eta^{-2sl}), \text{ for some } 0 < s < \frac{1}{2}. \tag{3.5}$$

[15] proves that the WoS for Poisson equations converge linearly with its discretisation parameter δ for sufficiently smooth problems. All numerical experiments we did confirmed the behaviour of the WoS algorithm consistent with both (3.5) and the result from [15].

Derivation

Using the expressions 3.3, 3.4 and 3.5 plugging into (3.2) we get:

$$\begin{aligned} M_l &\leq C \cdot \frac{1}{\varepsilon^2}\sqrt{\frac{\delta_l^{2s}}{l}}\sum_{k=0}^{L}\sqrt{\eta^{-2sk}\cdot k} \\ &\leq C_1 \cdot \frac{1}{\varepsilon^2}\sqrt{\frac{\delta_l^{2s}}{l}}\sum_{k=0}^{\infty}\sqrt{\eta^{-2sk}\cdot k} \\ &\leq C_1 \cdot \frac{1}{\varepsilon^2}\sqrt{\frac{\delta_l^{2s}}{l}}C_2 = C_1 \cdot C_2 \cdot \frac{1}{\varepsilon^2}\sqrt{\frac{\delta_l^{2s}}{l}} \end{aligned}$$

And using that $W_{tot} = \sum_{l=0}^{L} w_l M_l$:

$$\begin{aligned} W_{tot} &\leq \sum_{l=0}^{L} \underbrace{C_3 \cdot l}_{\geq w_l} \cdot \underbrace{C_1 \cdot C_2 \cdot \frac{1}{\varepsilon^2}\sqrt{\frac{\delta_l^{2s}}{l}}}_{\geq M_l} \\ &= C\frac{1}{\varepsilon^2}\sum_{l=0}^{L}\sqrt{\delta_l^{2s}\cdot l} \qquad (C = C_1 \cdot C_2 \cdot C_3) \\ &\leq C \cdot C_2 \cdot \frac{1}{\varepsilon^2} \end{aligned}$$

Finally, we can conclude that

$$W_{tot} = \mathcal{O}(\varepsilon^{-2}).$$

For the WoS Alg. 2 we can use the result from [15] which shows that the error is proportional to δ. This lets us conclude the desired Monte Carlo convergence rate in work of $\frac{1}{\sqrt{W}}$ for MLMC with WoS.

3.2 Euler-Maruyama

The Euler Maruyama scheme [12] is one of the simplest ways to simulate Statistical Differential Equations of the form

$$dX_t = a(X_t, t)dt + b(X_t, t)dW_t. \tag{3.6}$$

Brownian motion, the situation we care about, is the special case $a \equiv 0$ and $b \equiv 1$.

The procedure works very simply by discretising some timestep Δt and estimating

$$X_{t+\Delta t} \approx X_t + (W_{t+\Delta t} - W_t).$$

We know from the definition of a Wiener process in Sec. 2.1 that this difference in the stochastic process is normally distributed:

$$(W_{t+\Delta t} - W_t) \sim \mathcal{N}(0, \Delta t).$$

Since each timestep is equally sized, we can integrate over the random walk by evaluating the right-hand side $g(x)$ at each timestep and summing them up weighted by the time step size. The error we would have to analyse is twofold. Firstly, the quadrature error is caused by only evaluating at the step points. Secondly, there is the random walk discretisation error, where we do not truly follow a random walk but rather a sequence of random jumps. We will forego detailed analysis here, though we believe the convergence of our rudimentary quadrature rule should be fast enough compared to the other source of error. Gathering all the above, we can generate single samples using Alg. 1.

Algorithm 1 A Euler Maruyama based algorithm for Feynman-Kac for Poisson

procedure EULER-MARUYAMA($\mathbf{x}, g, f$) ▷ where $\Delta u = -g$ in Ω and $u = f$ on $\partial\Omega$
Require: $\mathbf{x} \in \Omega \subseteq \mathbb{R}^n$
  $X \leftarrow \mathbf{x}$
  $u \leftarrow 0$
  while isInsideDomain(X) **do**
    $u \leftarrow u + g(X) \cdot \Delta t$
    $X_{offset} \leftarrow \mathcal{N}^n(0, \sqrt{\Delta t})$ ▷ sampling normal distribution with $\sigma = \sqrt{\Delta t}$
    $X \leftarrow X + X_{offset}$
  end while
  $X_\tau \leftarrow$ findClosestPointOnBoundary(X)
  $u \leftarrow u + f(X_\tau)$
end procedure

3.2.1 Correlated Euler-Maruyama

To use Multilevel-Monte Carlo, we need a way of sampling $Y_l = Y_{\delta_l} - Y_{\delta_{l-1}}$ where the discretisation parameter in the Euler-Maruyama algorithm is the size of the timestep $\delta = \Delta t$. To do this we take an Euler maruyama sample at the finer level and after every η steps where $\eta = \frac{\delta_{l-1}}{\delta_l}$ we use the sum of the previous η offsets at the fine level as the offset at the coarse level. See figure Fig. 3.1 for a visual illustration.

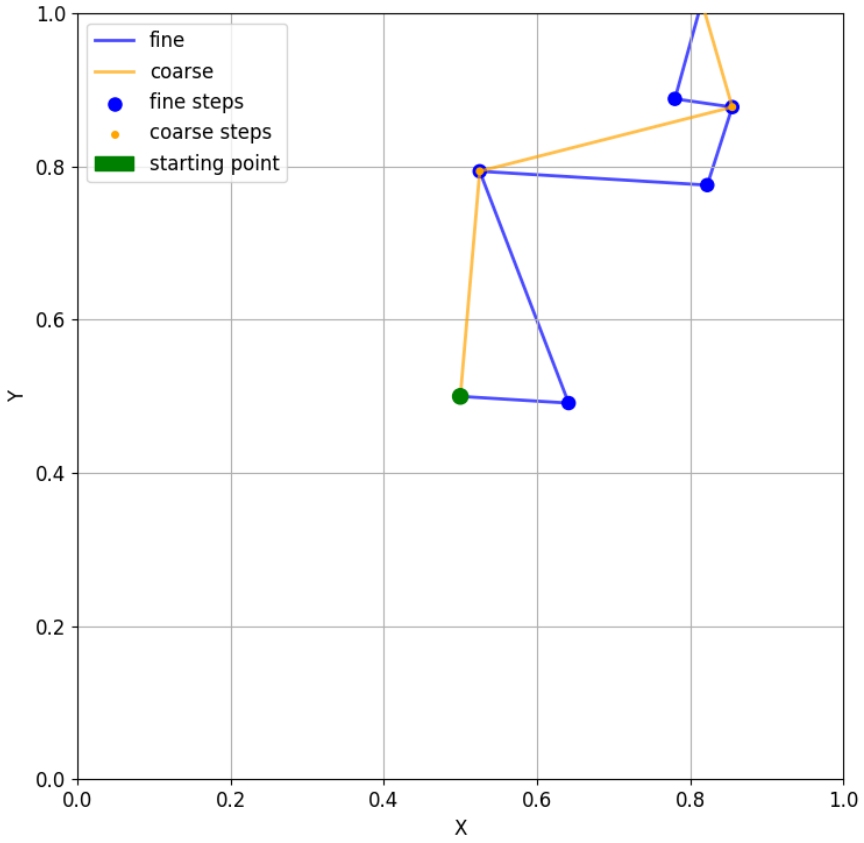


Figure 3.1: A correlated sample of the Euler-Maruyama scheme starting at $(\frac{1}{2}, \frac{1}{2})$ with $\eta = 2$

3.3 The Walk-on-Spheres algorithm

The Walk-on-Spheres (WoS) algorithm is a more sophisticated method proposed by J. Delaurentis and L. Romero in [6] for applying the Feynman-Kac formula to the Poisson equation. Delaurentis and Romers [6] show theoretical convergence for this algorithm, which is why we focus on describing the process and the necessary extensions to make it fully dimension-independent.

The algorithm makes use of Green's function, which allows us to express the solution to the Poisson equation $\Delta\phi = -g$ as

$$\phi(\mathbf{x}_0) = \int_{B(d,\mathbf{x_0})} g(\mathbf{x})G(x, x_0)dx + \frac{q}{|\partial B(d, \mathbf{x_0})|}\int_{\partial B(d,\mathbf{x_0})} \phi(\mathbf{x})d\mathbf{x} \tag{3.7}$$

Where G denotes Green's function, which on a Ball has a closed form for any dimension [16] equation 41. We use $B(0, d)$ for the ball with radius d and its centre at the origin. This closed form can be expressed easily with the dimension as a parameter for dimensions higher than 3. The 2-dimensional form, however, cannot be written down in the same manner. For this reason, the following discussion will be split into these two cases. To use WoS as suggested in [6], we need the integral of Green's function over the Ball,

$$a_n = \int_{B(0,d)} G(\mathbf{x}, 0)d\mathbf{x}, \quad \mathbf{x} \in \mathbb{R}, \tag{3.8}$$

as an expression dependent on the dimension. Additionally, we need a normalised green density on the Ball, which we can write as

$$\rho(\mathbf{x}) = \frac{G(\mathbf{x}, 0)}{a_n}. \tag{3.9}$$

The results will be presented in spherical coordinates. Using that coordinate system allows us to get independent distributions for angles and radius, which we can sample individually. Finally, since we use rejection sampling to sample these distributions, we must determine each independent distribution's maximum. The derivation of these independent densities, courtesy of Dr. Sadr, and the derivation of the normalisation factors follow.

3.3.1 2-dimensions

In two dimensions the needed normalization constant as described in [17] is $a_2 = \frac{d^2}{4}$ and Green's density is:

$$\rho(r, \theta) = \frac{2}{d^2\pi} r \log\left(\frac{d}{r}\right), \quad r \in (0, d), \theta \in (0, 2\pi).$$

And hence the split densities

$$\rho_r = \frac{4}{d^2} r \log\left(\frac{d}{r}\right), \qquad r \in (0, d)$$
$$\rho_\theta = \frac{1}{2\pi}, \qquad \theta \in (0, 2\pi)$$

The maximum of the radial density can be found analytically by finding all critical points inside the relevant region, which turns out to be exactly one. Evaluating at that point gives us:

$$\max_{r\in(0,d)} \rho_r = \rho_r\left(\frac{d}{e}\right) = \frac{4}{de},$$

where e is Euler's number.

3.3.2 n-dimensions n >2

Green's function on the ball $B(0,d)$ in more than 2 dimensions can be written as

$$G(\mathbf{x},0) = \frac{1}{n(n-2)\alpha(n)}\left[\frac{1}{|\mathbf{x}|^{n-2}} - \frac{1}{|\frac{d}{|\mathbf{x}|}\mathbf{x}|^{n-2}}\right] = \frac{1}{n(n-2)\alpha(n)}\left[\frac{1}{|\mathbf{x}|^{n-2}} - \frac{1}{d^{n-2}}\right].$$

The first step in finding closed forms of the necessary quantities is transforming the coordinates into a spherical coordinate system. As described in [18], $|x| = r$ and the determinant of the Jacobian is $|\det J| = r^{n-1}\sin^{n-2}(\phi_1)\cdots\sin(\phi_{n-1})$. Since there are no angular contributions, the integral to calculate a_n looks like

$$a_n = \int_{B(0,d)} \frac{1}{n(n-2)\alpha(n)}\left[\frac{1}{r^{n-2}} - \frac{1}{d^{n-2}}\right] r^{n-1}\sin^{n-2}(\phi_1)\cdots\sin(\phi_{n-1})d\Omega.$$

First, we will examine the integrand since this is the unnormalized total density we seek.

$$\rho(r,\phi_1,\ldots,\phi_{n-1}) = \frac{1}{a_n n(n-2)\alpha(n)}\left[\frac{1}{r^{n-2}} - \frac{1}{d^{n-2}}\right] r^{n-1}\sin^{n-2}(\phi_1)\cdots\sin(\phi_{n-2}),$$

where $\alpha(n)$ denotes the area of the unit Ball in $\mathbb{R}^n$. As we can see, the marginal densities are easily separable (except for normalisation). Hence, we can write:

$$\rho_r = \frac{1}{Z_r}(d^{n-2} - r^{n-2})r, \qquad r \in (0,d) \tag{3.10}$$

$$\rho_{\phi_i} = \frac{1}{Z_{\phi_i}}\sin^{n-i-1}(\phi_i), \qquad i \in \{1,\ldots,n-2\}, \theta \in (0,\pi) \tag{3.11}$$

$$\rho_{\phi_{n-1}} = \frac{1}{2\pi}, \qquad \phi_{n-1} \in (0,2\pi) \tag{3.12}$$

where

$$Z_r = \int_0^d (d^{n-2}r - r^{n-1})dr = \frac{d^n}{2} - \frac{d^n}{n} = \frac{d^n(n-2)}{2n}$$

and per [19]

$$Z_{\phi_i} = \int_0^\pi \sin^{n-i-1}(\phi)d\phi = B\left(\frac{1}{2}, \frac{n-i}{2}\right),$$

where $B(\cdot,\cdot)$ is the beta function as defined in [19] page 258.

To find a_n, we take a different perspective on the integral.

$$\int_{B(0,d)} G(\mathbf{x},0)\,d\mathbf{x} = \frac{1}{n(n-2)\alpha(n)}\int_{B(0,d)}\left(\frac{1}{|\mathbf{x}|^{n-2}} - \frac{1}{d^{n-2}}\right)d\mathbf{x} \tag{3.13}$$

$$= \frac{1}{n(n-2)\alpha(n)}\left(\int_{B(0,d)}\frac{1}{|\mathbf{x}|^{n-2}}d\mathbf{x} - \int_{B(0,d)}\frac{1}{d^{n-2}}d\mathbf{x}\right) \tag{3.14}$$

The first integral is

$$\int_{B(0,d)} \frac{1}{|\mathbf{x}|^{n-2}} d\mathbf{x} = \int_0^d \int_{\partial B(0,r)} \frac{1}{r^{n-2}} ds\, dr \tag{3.15}$$

$$= \int_0^d \frac{1}{r^{n-2}} \beta(n) r^{n-1} dr \qquad \beta(n) \text{ being the surface area of the unit n-ball} \tag{3.16}$$

$$= n\alpha(n) \int_0^d r\, dr = n\alpha(n) \frac{d^2}{2} \qquad \alpha(n) = n\beta(n) \text{ per [15]} \tag{3.17}$$

The second integral is

$$\int_{B(0,d)} \frac{1}{d^{n-2}} d\mathbf{x} = \frac{1}{d^{n-2}} \int_{B(0,d)} d\mathbf{x} = \frac{1}{d^{n-2}} \alpha(n) d^n = \alpha(n) d^2. \tag{3.18}$$

Plugging in, we get:

$$\int_{B(0,d)} G(\mathbf{x}, 0)\, d\mathbf{x} = \frac{1}{n(n-2)\alpha(n)} \left(n\alpha(n) \frac{d^2}{2} - \alpha(n) d^2 \right) \quad \text{per } Eq.\ (3.17), Eq.\ (3.18) \tag{3.19}$$

$$= \frac{1}{n(n-2)} \left(\frac{n}{2} - 1 \right) d^2 \tag{3.20}$$

$$= \frac{1}{n(n-2)} \frac{n-2}{2} d^2 = \frac{d^2}{2n} = a_n. \tag{3.21}$$

To find the maxima of the marginal densities, we follow the standard strategy of taking the derivative, finding extremal points, and plugging back in.

The maxima for the radius density and the non-uniform angular densities are:

$$\max_{r \in (0,d)} \rho_r = \frac{2n}{d(n-1)\sqrt[n-2]{n-1}} \tag{3.22}$$

$$\max_{\phi_i \in (0,\pi)} \rho_{\phi_i} = B\left(\frac{1}{2}, \frac{n-i}{2} \right)^{-1} \qquad i \in \{1, \ldots, n-2\} \tag{3.23}$$

3.3.3 Simple Walk-on-Spheres

The WoS algorithm uses the result that the distribution of the first exit point of random walks from a sphere starting at its centre is uniform. This way, when we jump from a point $\mathbf{x}_0$ onto a sphere with $\mathbf{x}_0$ as its centre in a uniformly random direction, we arrive at each point of this sphere as often as an actual Brownian motion would. Taking the biggest sphere possible that is entirely contained in the domain, WoS jumps onto the surface of that sphere for each step in the algorithm. WoS needs a different stopping criterion than Euler-Maruyama since we can never leave the domain with this method. To get one, we define a small region along the boundary with thickness $\delta > 0$. We will call this region the boundary region. If our jumps ever get us into this region, meaning when the distance to the boundary gets smaller than δ, we consider the boundary hit and stop. This process can be seen in Fig. 3.2. To account for the right-hand side contribution, a single sample distributed according to Green's density is taken inside the current sphere and used to estimate the integral over Green's function over the sphere. Intuitively, this works because each sphere is traversed often enough that this flawed estimation will converge

when we average over samples. A more rigorous discussion of why this method works can be found in [6]. We can express a single sample of this algorithm mathematically as:

$$Y = f(X_\tau) + \sum_{k=1}^{N} a_n(d_k) g(Z_k)$$

Where X_τ is the closest point on the boundary to where we stop, d_k is the distance to the boundary after the $k-1$th jump, and Z_k is a random variable distributed according to Green's density inside the region after the $k-1$th step. Due to other notation clashes, we have abandoned the notational convention of [6].

This theoretical base finally allows us to write an algorithm generating a single WoS sample in Alg. 2.

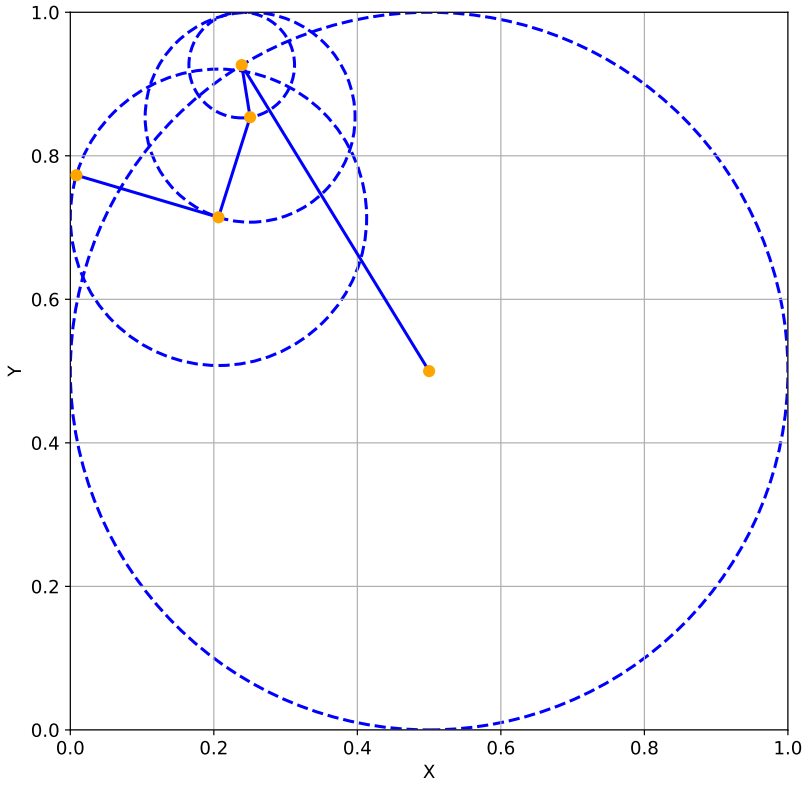


Figure 3.2: one sample from the Python implementation of WoS starting at $(\frac{1}{2}, \frac{1}{2})$ with visualised spheres

Algorithm 2 Walk-on-Spheres for the poisson equation

procedure WALK-ON-SPHERES($\mathbf{x}, \delta, g, f$) ▷ where $\Delta u = -g$ in Ω and $u = f$ on $\partial\Omega$
Require: $\mathbf{x} \in \Omega \subseteq \mathbb{R}^n$
  $X \leftarrow \mathbf{x}$
  $u \leftarrow 0$
  while distanceToBoundary$(X) > \delta$ **do**
    $d \leftarrow$ distanceToBoundary(X)
    $X_{offset} \leftarrow$ sampleSphereSurface(d)
    $X_{next} \leftarrow X + X_{offset}$
    $Z \leftarrow X +$ sampleGreenDensity(d)
    $u \leftarrow u + a_n \cdot g(Z)$ ▷ where $a_n = \int_{B(0,d)} G(0, y)dy$
    $X \leftarrow X + X_{next}$
  end while
  $X_\tau \leftarrow$ findClosestPointOnBoundary(X)
  $u \leftarrow u + f(X_\tau)$
end procedure

3.3.4 Correlated Walk-on-Spheres

To apply multilevel, we need a version of WoS Alg. 2 that samples the difference between two levels. The discretisation parameter we change in the case of WoS is the strategically labelled boundary layer thickness δ. So to sample the difference between two levels $Y_l = Y_{\delta_l} - Y_{\delta_{l-1}}$ directly, we assume they take the same path but apply their stopping criteria separately. Since we also use the same Green distributed samples for Y_{δ_l} and $Y_{\delta_{l-1}}$ to account for the right-hand side, they will entirely cancel each other out most of the time. The single exception being, when the stopping criterion with boundary layer thickness δ_{l-1} has been applied, but δ_l has not come into play yet. This allows us to ignore the right-hand side for most of the walk and generate these correlated samples as in Alg. 3

Algorithm 3 Correlated Walk-on-Spheres for the Poisson equation

procedure WALK-ON-SPHERES CORRELATED($\mathbf{x}, \delta_l, \delta_{l-1}, g, f$) ▷ where $\Delta u = -g$ in Ω and $u = f$ on $\partial\Omega$
Require: $\mathbf{x} \in \Omega \subseteq \mathbb{R}^n$
  $X \leftarrow \mathbf{x}$
  $u \leftarrow 0$
  coarseWalkIn $\leftarrow$ True ▷ coarseWalkIn tracks when $Y_{\delta_{l-1}}$ completes
  while distanceToBoundary(X) $> \delta_l$ **do**
    if distanceToBoundary(X) $> \delta_{l-1}$ **then**
      coarseWalkIn $\leftarrow$ False
      $X_\tau \leftarrow$ projectToBoundary(X)
      $u \leftarrow u - f(X_\tau)$ ▷ account for boundary term of $Y_{\delta_{l-1}}$
    end if
    $d \leftarrow$ distanceToBoundary(X)
    $X_{offset} \leftarrow$ sampleSphereSurface(d)
    $X_{next} \leftarrow X + X_{offset}$
    if ¬coarseWalkIn **then**
      $Y \leftarrow X +$ sampleGreenDensity(d)
      $u \leftarrow u + a_n \cdot g(Y)$ ▷ where $a_n = \int_{B(0,d)} G(0, y)dy$
    end if
    $X \leftarrow X + X_{next}$
  end while
  $X_\tau \leftarrow$ findClosestPointOnBoundary(X)
  $u \leftarrow u + f(X_\tau)$
end procedure

3.3.5 MLMC speedup

As shown in [11], the convergence rate in terms of work done for pure WoS for Laplace is $\mathcal{O}(\frac{\log(W)}{\sqrt{W}})$ for the rather regular domains and function we will look at here. This result is found for any WoS method where the total error comprises the typical statistical error in the number of samples M and a linear error in the boundary thickness δ.

$$e_{tot} = \mathcal{O}(\frac{1}{\sqrt{M}}) + \mathcal{O}(\delta)$$

[15] shows that WoS for Poisson also follows this rule. Hence, pure WoS for Poisson should also converge in $\mathcal{O}(\frac{\log(W)}{\sqrt{W}})$. Comparing this with the convergence rate of MLMC from Sec. 3.1.1 allows us to calculate an expected speedup for a given error.

$$e_{WoS} = \mathcal{O}\left(\frac{\log(W)}{\sqrt{W}}\right), \quad e_{MLMC} = \mathcal{O}\left(\frac{1}{\sqrt{W}}\right)$$
$$\implies S(W) = \frac{e_{WoS}}{e_{MLMC}} = \mathcal{O}(\log(W))$$

3.3.6 Random number generation

As one would expect, Green's density is not a standard distribution, so we had to implement it from scratch. We used a rejection sampling approach.

Rejection sampling of a distribution with probability density function $\rho : [a, b] \to [0, m]$ works as follows:

1. draw two random numbers $x \sim$ `Uniform`[a, b], $y \sim$ `Uniform`[0,m]
2. check whether $\rho(x) \leq y$ and if yes return x otherwise go back to 1.

We chose rejection sampling for two reasons. Firstly and most importantly, it is easy to implement. Secondly, Green's density is quite well suited for a rejection sampling method. A good rejection method is one where not too many samples are rejected. The probability that a sample gets accepted is $\int_a^b \rho(x)dx \cdot ((b-a)m)^{-1}$. Since ρ is a probability density function, its integral will evaluate to 1. The acceptance probability that we are looking for is therefore, $\rho_{acc} = ((b-a)m)^{-1}$. To maximise this ρ_{acc}, we want to choose the smallest possible m since a and b will usually be given by the problem. Rejection sampling produces the correct result if $\max_{x\in[a,b]} \rho \leq m$. From which we can conclude that the optimal choice for m is the maximum of ρ.

Chapter 4

Python

4.1 Implementation

The Python implementation can be found on GitHub feynman-kac-python-prototype. I used the standard libraries Numpy [20], Pandas [21] [22] and Matplotlib [23] for math, data manipulation and visualisation, respectively. Multiprocessing and Jupyter allowed for faster runs and interactive development. To recreate any results, consult the —h option of the main.py executable or look at the README, which should contain all the information needed to run the code. I did not include command-line options to switch between the Euler-Maruyama and WoS implementations of random walk generation; this has to be done by modifying the source code in the function `generate_mlmc_data`.

This Repository has two primary purposes. Firstly, we used it to compare the Euler-Maruyama and the WoS algorithms. Secondly, we implemented more test functions since the restriction to two dimensions makes this a lot easier.

We restricted testing to the unit square $[0, 1]^2$ for ease of implementation.

4.1.1 Test functions

Starting from analytical functions with different characteristics, we calculated the right-hand side that would make any particular function the solution to a Poisson equation. This allows us to test many different right-hand sides quickly, but restricts us to functions that are reasonably simple to differentiate. In the following, we will list those functions, give each a name so that we can refer back easily and give a reason for why it was chosen.

- To isolate the right-hand side contribution, a function with homogeneous boundary conditions:
$$\phi(x, y) = \sin(\pi x) \sin(\pi y). \tag{4.1}$$
We will call this test case **Sine**.

- A function where the boundary terms dominate:
$$\phi(x, y) = \cos(\pi x) \cos(\pi y). \tag{4.2}$$
We will call this test case **Cosine**.

- A function where the boundary is significant, that additionally is non-negative:

$$\phi(x, y) = 4 \cdot (\cos(\pi x) \cos(\pi y))^2. \tag{4.3}$$

  We will call this test case **Cosine^2**.

- A function with a small peak:

$$\phi(x, y) = \exp\left(-\left[(x - 0.5)^2 + (y - 0.5)^2\right] \cdot \mu\right), \tag{4.4}$$

  where $\mu = 100$ unless explicitly stated otherwise. We will call this test case **Gaussian**.

- A function without critical points inside the domain:

$$\phi(x, y) = \exp(x + y). \tag{4.5}$$

  We will call this test case **Exponential**

- A Polynomial function because it grows comparatively slowly.

$$\phi(x, y) = x^2 + y^2. \tag{4.6}$$

  We will call this test case **Polynomial**.

4.2 Results

4.2.1 Comparative results

To compare Euler-Maruyama and WoS, we measured the number of random numbers drawn to solve a point with MLMC at a given accuracy. This is a good cost indicator since it is directly proportional to the needed floating-point operations. We chose not to differentiate between sampling different distributions, assuming a dedicated developer could optimise them to be similarly fast. In this case, we solved to a low accuracy of $\varepsilon = 0.01$, since the Python implementation is not highly optimised. For each Test function, we ran MLMC with the two random walk generating algorithms 20 times and averaged the needed work. The same starting discretisation parameter for level 0 was used for both algorithms, namely $\delta_0 = 0.01$. This was chosen because it turned out to be a good choice for Euler-Maruyama, giving that algorithm the best chance of performing well. Further, the ratio of discretisation between levels was set to $\eta = 16$ again because it improved performance for the Euler-Maruyama scheme. Nonetheless, WoS vastly outperforms Euler-Maruyama. Depending on the test case, it requires up to 100 times fewer samples, as can be seen in Fig. 4.1 and in Tab. 4.1. Only in the single exception of the Gaussian test case can Euler-Maruyama compete with WoS. With its tiny peak at the centre of the domain, it is a worst-case scenario for the mechanics of WoS. All of the information is densely packed in the exact area where WoS samples the least densely, since its stepsize is the largest in the centre of the domain. Whereas Euler-Maruyama samples everywhere with the same density, so it should have less trouble with this distribution. Even in this scenario, WoS only needs around 57% of the random samples that Euler-Maruyama needs.

Test function	Euler-Maruyama	Walk on spheres
Sine	7912876	73830
Cosine	1300193	61854
Cosine^2	24982924	582319
Exponential	16223619	260174
Gaussian	9238784	5658725
Polynomial	1169221	43530

Table 4.1: Comparing the amount of random numbers needed for each random walk generating algorithm, where $\varepsilon = \delta_0 = 0.01, \eta = 16$ and $\mathbf{x}_0 = (0.5, 0.5)$ averaged over 20 runs

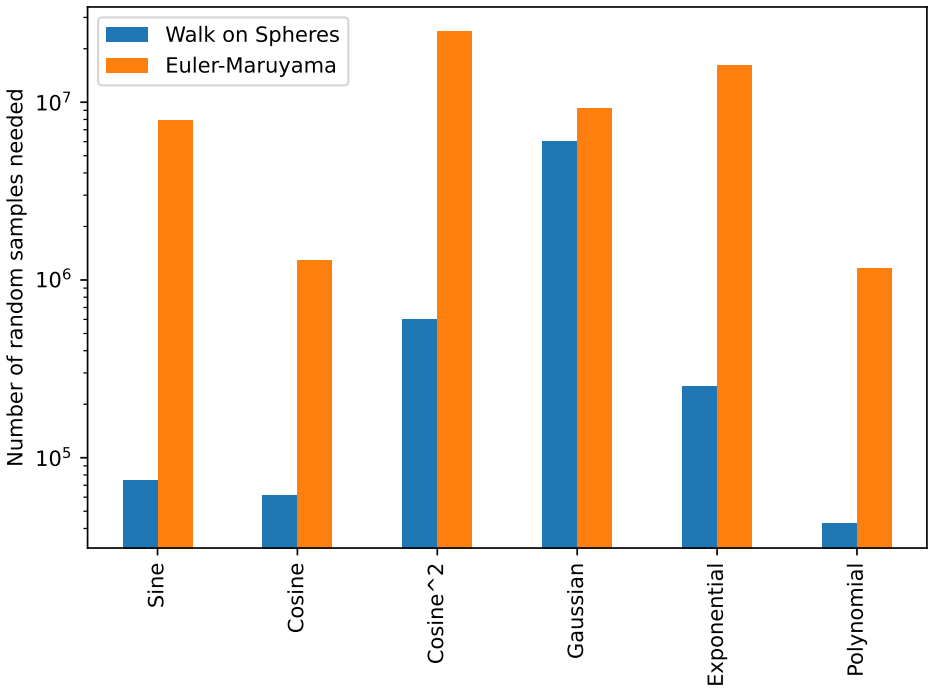


Figure 4.1: Plotting the data from Tab. 4.1

4.2.2 WoS results

Having found a clear winner for the choice of random walk algorithm, we will look more closely at how it behaves.

Pure WoS

A pure WoS simulation without MLMC should have two sources of error; statistical noise and a discretisation error. By fixing the number of random walks and checking the error and amount of random numbers sampled, we should be able to determine both the convergence rate of the discretisation error of WoS and the scaling of work that needs to be done to compute a single WoS sample for a given boundary layer thickness. We sampled $N = 10^6$ random walks 20 times for each δ. With this additional information, we calculated the standard deviation of the results for each of the 20 iterations, which should give us a decent estimate of the statistical noise. As we can see from Fig. 4.2, the average measured absolute error over the iterations decreases with δ until around the estimated statistical noise. Initially, with δ, the discretisation error dominates, so we see the error decrease, but the total expected error does not get smaller than the statistical noise. Therefore, we can estimate the rate of convergence of the discretisation error with the slope of the curve in Fig. 4.2. As we can see, this slope follows $\mathcal{O}(\delta)$. This example specifically shows the convergence particularly well, but the pattern holds for all other test functions, as can be seen in the plots in Appendix A.1

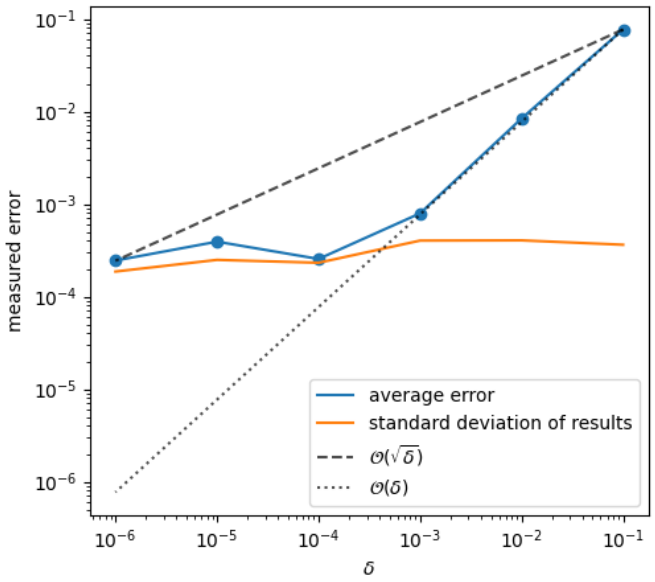


Figure 4.2: Error of pure WoS sampling at different δ with $N = 10^6$ samples averaged over 20 runs for the test function Sine 4.1

We see a couple of exceptions that do not fit the picture norm of this convergence, for example, in Fig. 4.3. We would need many more samples to see any significant information about the discretisation error, since statistical noise immediately makes up most of the error. This test resembles a singular function at the point $(0.5, 0.5)$. Hence, nearly all information is contained within the sphere of the first step of WoS. For the same run parameters of 10^6 samples done 20 times, we can test a less severe Gaussian, say $\mu = 10$ in Eq. (4.4). And find that

since more of the information is contained closer to the boundary, the discretisation error now dominates for large δ. Still compared to Fig. 4.2 the initial error for big δ is small in 4.4.

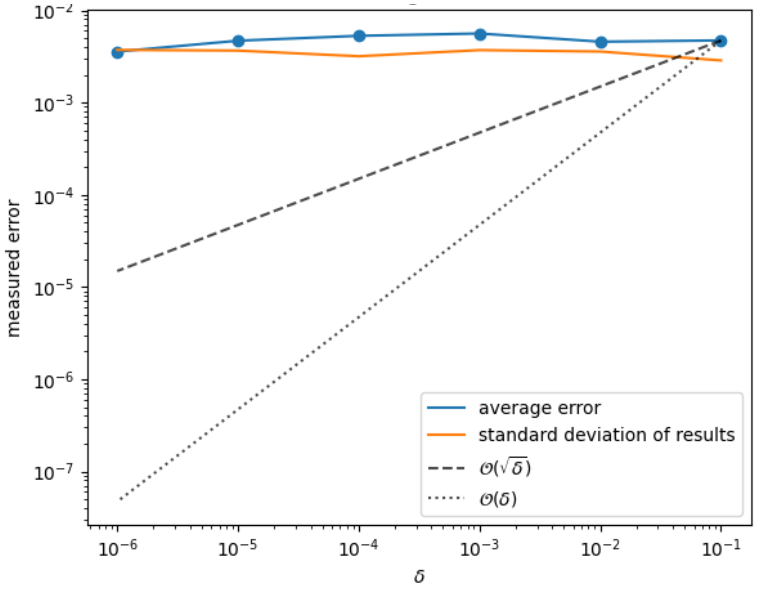


Figure 4.3: Behaviour of pure WoS sampling at different δ with $N = 10^6$ samples averaged over 20 runs for the test function Gaussian 4.4

Since we recorded the cost, we can also check how the number of samples changes as we decrease δ. This can be seen in Fig. 4.5.a The perfectly straight line in the plot with only one logarithmic axis demonstrates that the work grows in $\mathcal{O}(|\log(\delta)|)$. This confirms the bound found in [14] for WoS in 2 dimensions.

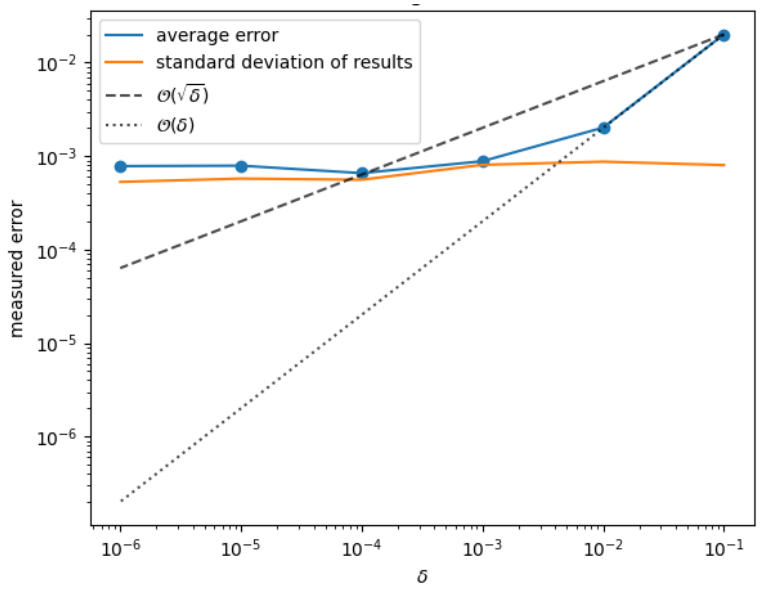


Figure 4.4: Behaviour of pure WoS sampling at different δ with $N = 10^6$ samples averaged over 20 runs for a less peaky Gaussian (4.4) with $\mu = 10$.

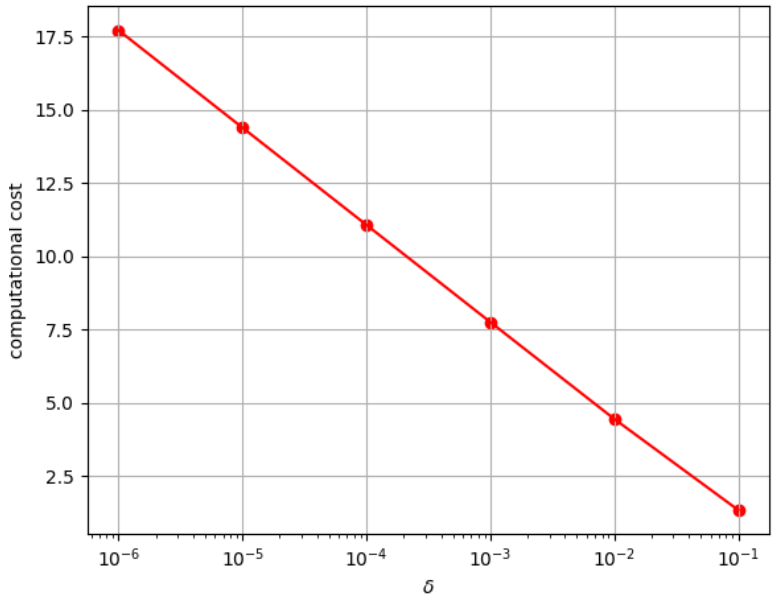


Figure 4.5: Behaviour of pure WoS sampling at different δ with $N = 10^6$ samples averaged over 20 runs for the test function Sine 4.1 averaged over 20 runs

MLMC WoS

Further, we investigated the performance of WoS combined with MLMC for the different test functions. We took the same hyperparameter configuration $\delta = 10^{-3}, \varepsilon = 0.01, \eta = 16$ and 10 runs for every test function on every position of the grid $\{0.1, 0.2, \ldots, 0.9\}^2$, averaging the results.

In Fig. 4.6 we can see a complete qualitative overview of the results. We note the following tendencies:

1. We can generally see that the cost tends to be highest there, where the change in the function is biggest. The most extreme example being the Gaussian 4.4 test-case.
2. Most differences in cost are not big enough to reflect in execution time. The machine's resource allocation was more important than the characteristics of the function.
3. The places where MLMC-WoS has the most trouble are sharp extremal points, as we can see from the Cosine^2 4.3 with a minimum in the centre and the Gaussian 4.4 with a maximum at that exact location.

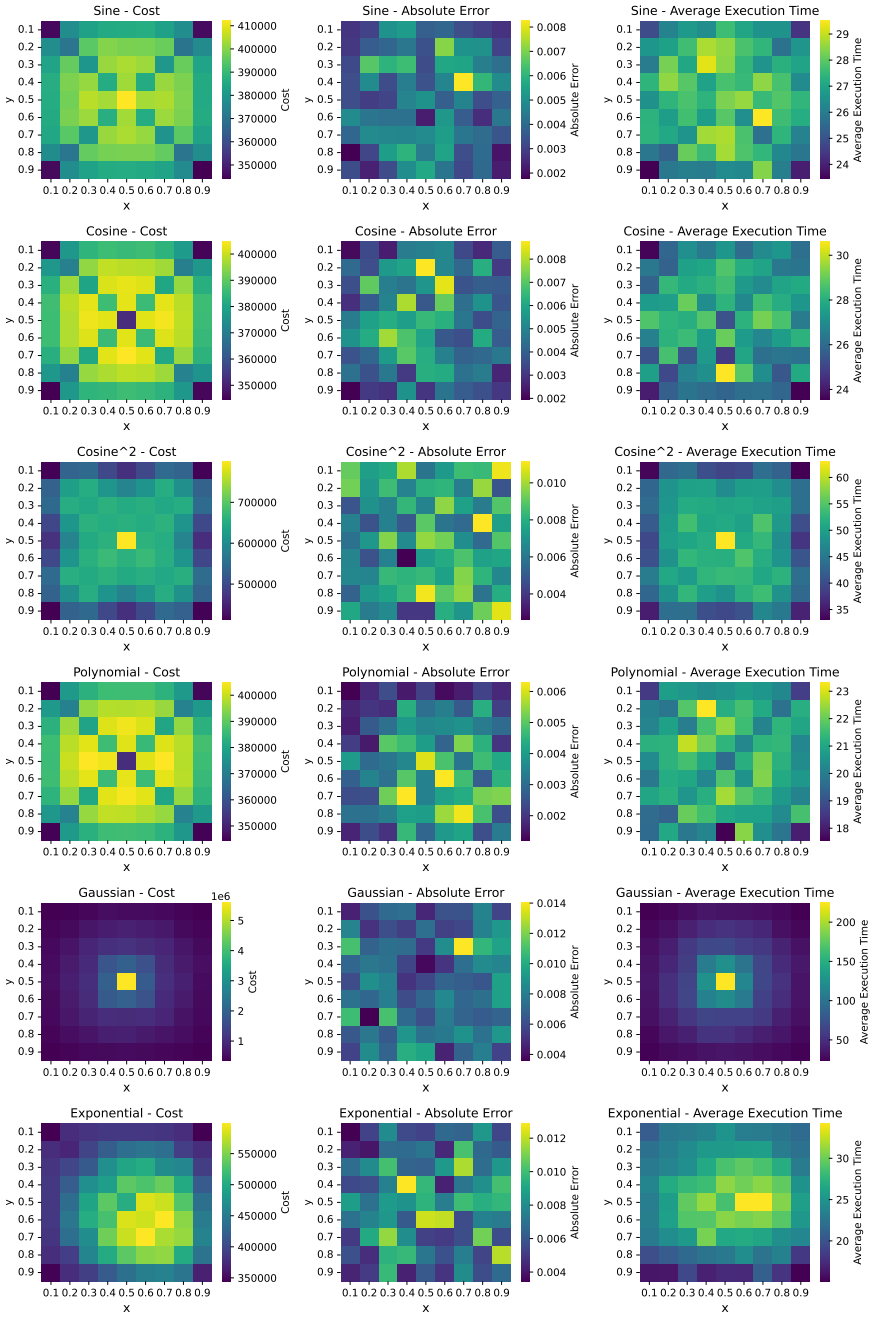


Figure 4.6: Heatmaps of MLMC-WoS for all test functions at $\varepsilon = 10^{-2}, \delta_0 = 10^{-3}, \eta = 16$ where each function is sampled 10 times at 0.1 intervals in each direction

In Fig. 4.7 we see an overview of these average costs comparing the different test functions. It further supports our conclusions 1 and 3, which state that functions containing sharp extrema or large(r) derivatives cost significantly more. Additionally, we get a sense of scale for these differences. The factor between the cheapest and most expensive outliers is barely 10. Since the convergence rate of MLMC-WoS is $\mathcal{O}(\frac{1}{\sqrt{W}})$, the bonus in accuracy if both are run to be equally expensive will be less than a factor of 4. This factor decreases again if we ignore the single biggest outlier at the peak of the sharp Gaussian distribution. We will discuss the Gaussian test case further in Sec. 5.2

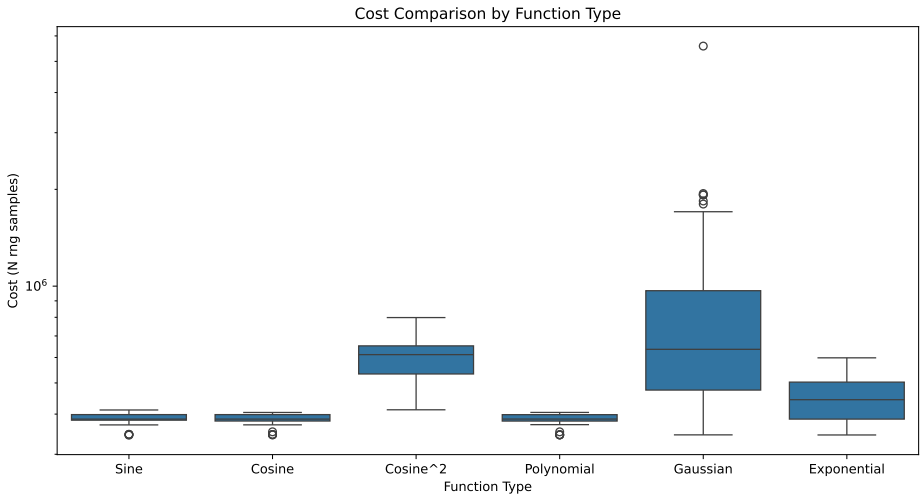


Figure 4.7: Comparing the costs of the different test functions where each box contains the average costs for each sampled point from the same data as Fig. 4.6

4.3 Conclusions

To take with us into the more performance-oriented C++ implementation, we know that WoS Alg. 2 outperforms Euler-Maruyama Alg. 1 by far. We know that the combination of MLMC and WoS performs consistently except for small extrema, which gives us a place to consider using the higher accuracy afforded by more performant code. Further, we are missing a comparison between pure WoS and MLMC-WoS, which we have left for the more performant implementation in Sec. 5.2.6.

Chapter 5

C++

5.1 Implementation

The implementation in C++ can be found on GitHub at ippl. Instead of being a standalone piece of code, this is a fork of the IPPL [24] physics library onto which we added another Poisson solver called the Feynman-Kac solver. It is a fully functional Poisson solver for single-rank programs. The tests/benchmarks presented in the results are written in the test/ directory of IPPL and are activated with the `-DIPPL_ENABLE_TESTS=ON` flag. More information on compilation can be found in the Repository README.

5.1.1 Solver structure

The C++ code, focusing more on performance, implements only WoS Alg. 2 together with MLMC. In contrast to the Python implementation, it does not restrict itself to 2 dimensions but instead is designed to work for systems of any dimension greater than or equal to 2 The dimension is taken as a template argument. However, at the time of writing, due to limitations within the Kokkos library [25] [26], it only works up to dimension six. Updates that allow higher dimensions are upcoming. We use Kokkos for the algorithm's parallelisation since it allows for an easy switch between CPU and GPU parallelisation.

Again, at the time of writing, IPPL uses rectangular meshes. This made the choice easy to stick to the unit hypercube $[0,1]^n, n \geq 2$ for testing. Testing was done on 2 to 5 dimensions.

5.1.2 Test functions

Due to IPPL's limitations, which only support constant Dirichlet boundary conditions, we were limited in our choice of test functions. A further difficulty of creating test cases for this implementation is comparing dimensions. Any function will change slightly when extended to higher dimensions. To make their performance over the dimensions somewhat comparable and working with the limitations of IPPL, we chose to extend the Sine 4.1 and Gaussian 4.4 test cases. We extended them as follows:

- **Sine**:

$$\phi(\mathbf{x}) = \prod_{i=1}^{n} \sin(\pi x_i). \tag{5.1}$$

- **Gaussian**:

$$\phi(\mathbf{x}) = \exp\left[-\sum_{i=1}^{n}\left(x_i - \frac{1}{2}\right)^2\right]^{100}. \tag{5.2}$$

The Gaussian test case is not entirely constant on the boundary; however, the maximal value on the boundary is $e^{-25} \approx 1.4 \cdot 10^{-11}$ regardless of which dimension we are in. We can ignore this safely, because the error introduced by this is much smaller than the accuracy to which we solve.

5.1.3 Test environment

All results presented in this chapter were run on the Merlin 6 cluster at PSI. Specifically on the Gwendolen partition on a single A100 GPU. We used the following libraries:

- GCC 12.3.0 *
- libfabric 1.18 *
- openmpi 4.1.5 *
- cmake 3.25.2 *
- ucx 1.14.1 *
- fftw 3.3.10 *
- cuda 12.1.1
- Kokkos 4.5.00 (loaded directly by IPPL)
- Heffte (default commit downloaded by IPPL)

The libraries marked with * are only needed to ensure smooth compilation and are not called at runtime.

Since the benchmarks were not run, reserving an entire GPU, performance measurements in terms of execution time were avoided. Since this Feynman-Kac solver has very different strengths from all the other Poisson solvers in IPPL, their direct comparison is less important than otherwise.

5.1.4 Test structure

The presented results do not reflect the solver's performance for standard use of Poisson solvers in IPPL. To isolate the characteristics of the MLMC-WoS algorithm, the test functions were inserted procedurally into the class. For this reason, we do not mention information about the mesh.

To compile the test we used the cmake command:

```
cmake .. -DCMAKE_BUILD_TYPE=Release \
    -DIPPL_PLATFORMS=CUDA \
    -DKokkos_ARCH_AMPERE80=ON \
    -DIPPL_ENABLE_FFT=ON \
    -DIPPL_ENABLE_SOLVERS=ON \
    -DIPPL_ENABLE_ALPINE=OFF \
    -DIPPL_ENABLE_TESTS=ON \
    -DCMAKE_CXX_STANDARD=20
```

Then we can compile the correct target with `make TestFeynmanKac`. The source code features more detailed instruction on how to run the different tests. We implemented three test cases:

- The **MLMCspeedup** test compares MLMC WoS and pure WoS for different dimensions
- The **CGComparison** test compares MLMC WoS with the previously implemented CG solver.
- The **TestConvergence** test lets the user record a chosen number of samples from pure WoS at a set δ.

All results presented in Sec. 5.2 stem from running these three tests for different input parameters. The raw data from these results is stored in the Python git repository feynman-kac-python in the folder cpp_results. This folder also features a script which can filter out data or the run parameters from the raw output files.

5.2 Results

5.2.1 WoS cost

In the convergence proof in Sec. 3.1.1 we show the expected convergence of MLMC-WoS of $\mathcal{O}(\frac{1}{\sqrt{W}})$ assuming that the expected cost for a single WoS sample is in $\mathcal{O}(\log(|\delta|))$. However, for Dimensions higher than three [14] only proves this result for domains with smooth boundaries. The hypercube is not quite smooth but still very regular. In our test cases this logarithmic growth of cost per sample holds empirically as can be seen in 5.1.

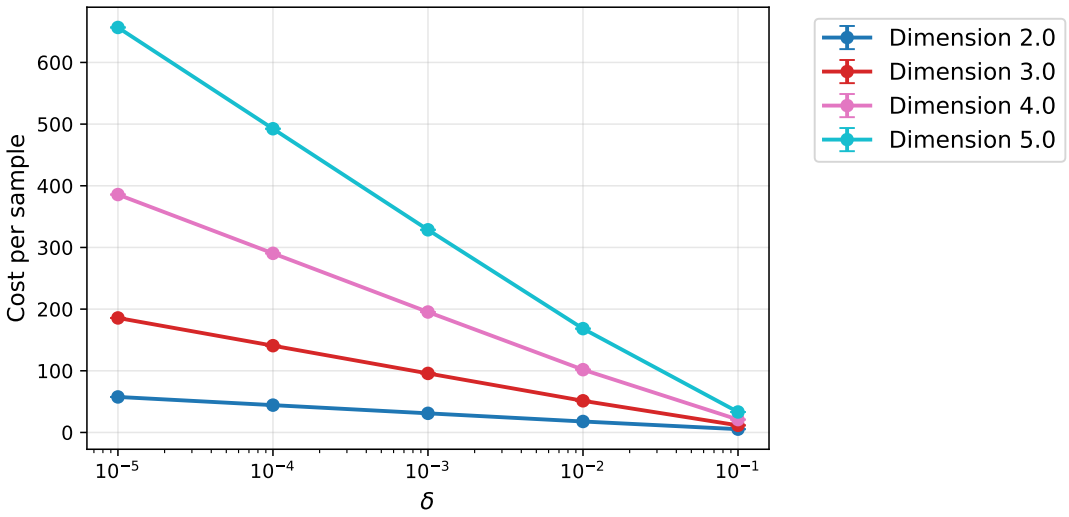


Figure 5.1: Progression of how many random numbers are generated for a single WoS sample starting at the centre of the hypercube. The result is averaged over $2 \cdot 10^7$ WoS samples where δ denotes the boundary layer thickness. A straight line indicates the expected logarithmic relationship between δ and cost.

5.2.2 Convergence

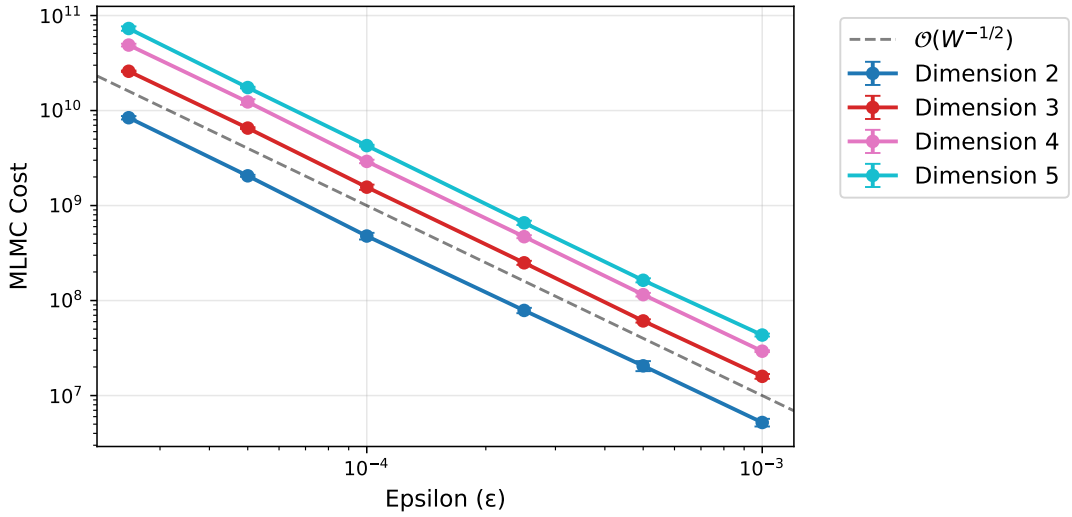
First, we check that the previous convergence result holds. To show it across dimensions, we will analyse the work done to solve Poisson's equation for a given position $\mathbf{x}_0$ across different target accuracies ε and dimensions.

In Fig. 5.2(a) and Fig. 5.2(b), we can see that the results seem to be converging at the expected rate of $\mathcal{O}(\frac{1}{\sqrt{W}})$. We tested some at random to ensure that the centre of the coordinate system was not a special case and that we would also converge for other positions. To create a plot similar to the ones at the centre of the coordinate system, we need to examine in more detail which positions we run for the different dimensions in the example. The procedure we used was the following:

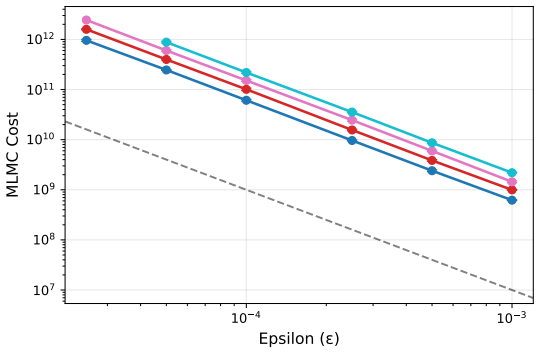
1. Generate uniformly random $\mathbf{x} \in [0, 1]^5$
2. In dimension n take the first n coordinates of $\mathbf{x}$ as the starting position we will call $\mathbf{y}_n$

$$\mathbf{y}_n = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

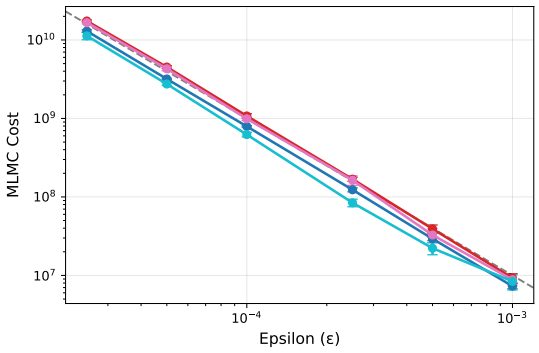
In 5.2(c) we can see such an example. While still following the expected rate, the cost does not increase when the dimension increases. This happens because we are not sampling at comparable positions anymore. Nonetheless, the convergence behaviour remains consistent.



(a) Progression of work versus target error for the test case Sine (5.1) at the centre of the unit hypersphere



(b) The test case Gaussian (5.2)



(c) The test case Sine (5.1) at a random position

Figure 5.2: Comparing the convergence of MLMC-WoS for right-hand side functions and positions with an initial boundary layer thickness $\delta_0 = 10^{-2}$, at level 0 and a ratio between levels of $\eta = 16$. The random position is $(0.715658, 0.908702, 0.797225, 0.210759, 0.818127)$ where dimension n takes the first n coordinates of this position. We can see that the cost varies with the test case. Additionally, the correlation between cost and dimension gets lost at the random position, indicating that the underlying function is more important than the dimension. Cost measures how many random numbers were generated.

5.2.3 Comparison with CG solvers

IPPL implements a variety of different Poisson solvers. We chose to compare our Feynman-Kac solver with the Conjugate gradient [27] solver, which uses a Finite difference discretisation combined with the iterative conjugate gradient method to solve the resulting sparse linear system of equations. The comparison of these methods is difficult to do directly. The Feynman-Kac solver is not designed to compete when solving a PDE on each vertex of a grid. Further, the schemes optimise for different errors. Whereas the Feynman-Kac algorithm will have a consistent absolute error at every point it solves, the CG solver reaches a certain L^2-error but does not optimise for each location separately. For this reason, we gave up the notion of a 1-to-1 comparison of the two algorithms. Instead, we show that the time it takes for a CG solver to solve to a given relative L^2 error over different dimensions versus the time it takes the Feynman-Kac solver to finish at a single point with a given target error ε. This way in Fig. 5.3 we can see the potential strength of a Feynman-Kac-based Poisson solver. It has little meaning that the absolute times for the Feynman-Kac solver are lower. Much more relevantly, its runtime grows less steeply than that of the CG solver. As exponential growth in a log plot would form a straight line, we can conclude that the runtime of the Feynman-Kac solver increases more slowly than this, and the CG solver might even be growing faster, though it is difficult to tell from only three data points.

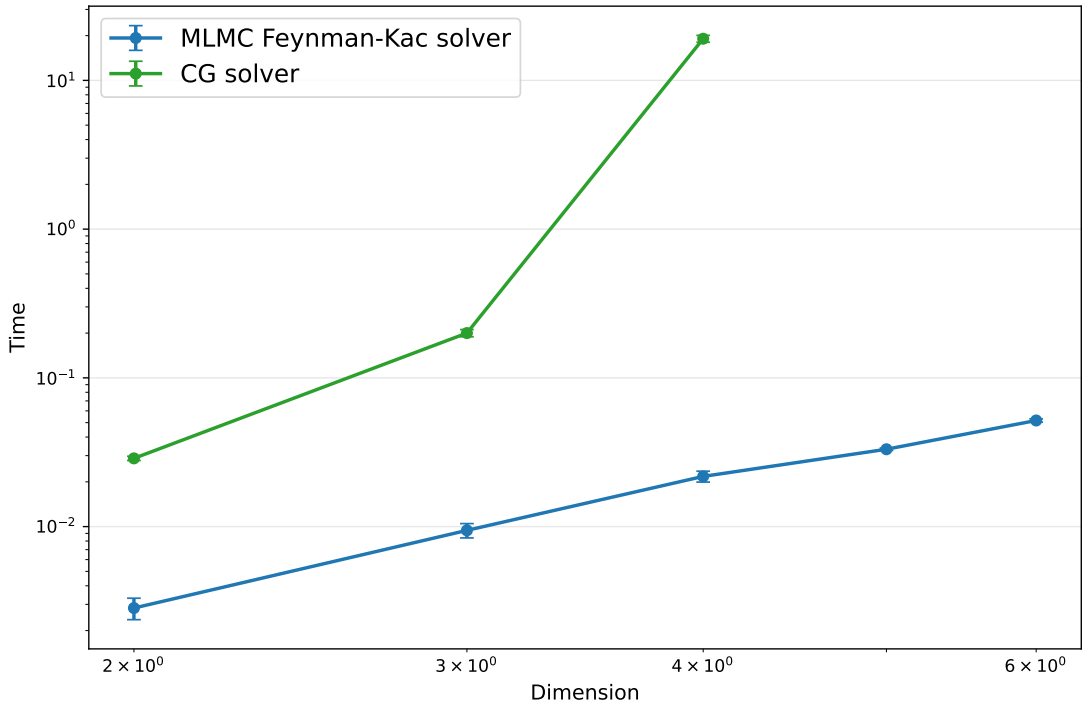


Figure 5.3: Comparing the times that the CG solver takes to solve on a grid with meshwidth $\frac{1}{128}$ corresponding to a relative L^2 error of approximately 0.018 with the time it takes for the Feynman-Kac solver to solve at the centre of the hypercube with $\varepsilon = 10^{-3}, \delta_0 = 10^{-2}, \eta = 16$ where the exact solution for both solvers is the Sine 5.1 test case. The execution time of the MLMC Feynman-Kac grows subexponentially, while the CG solver grows at least exponentially.

5.2.4 Ratio between levels

A rather little discussed factor in MLMC-WoS is the ratio of boundary thicknesses between levels, which we call η. We tested several different ratios from 2 to 16. The results can be seen in Fig. 5.4. $\eta = 2$ is the only choice that performs worse over all dimensions, and even its performance is only around 20% worse than the best value at each dimension, either at $\eta = 8$ or $\eta = 16$. While it is more difficult to make generalisations since no theoretical results relate η to MLMC performance, we did not observe any significant effect in test runs.

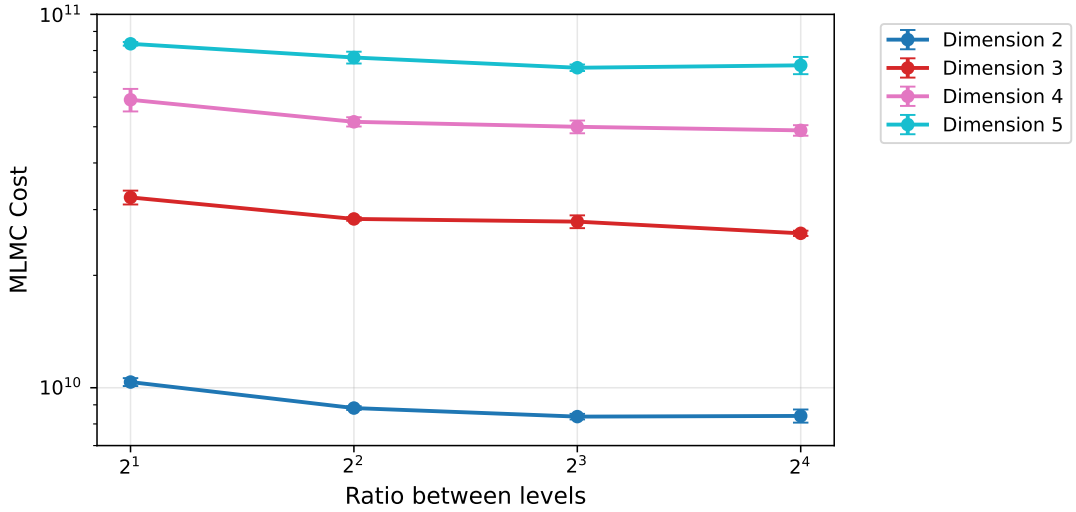
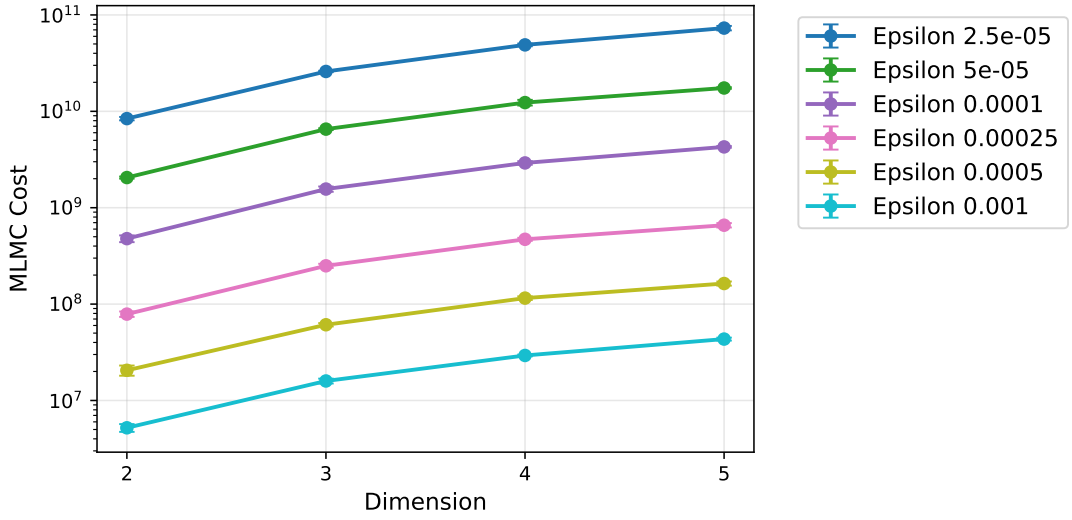


Figure 5.4: Comparing the cost for different η at different dimensions for the test case Sine 5.1 at the centre of the hypercube where $\delta_0 = 10^{-2}, \varepsilon = 2.5 \cdot 10^{-5}$

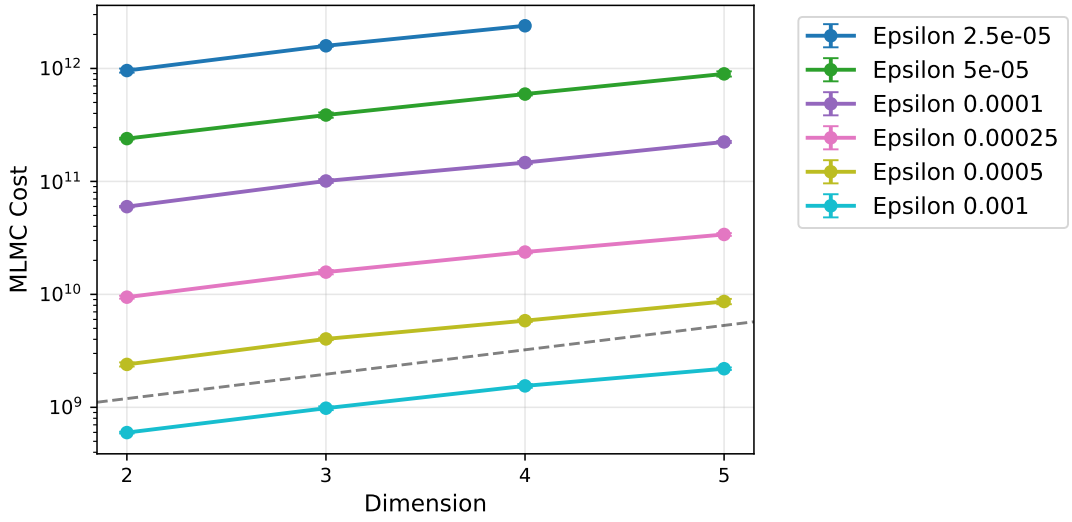
5.2.5 Cost increase per dimension

In most cases, one of the biggest arguments for using Monte Carlo schemes is how its computational cost scales in higher dimensions. Methods like FEM, which solve an entire mesh at a time, become more expensive exponentially as the dimension increases. Therefore, observing whether and how fast the cost increases in higher dimensions is relevant. The simplest way to compare this is to compare the test function over the different dimensions at the centre of the hypercube, where their respective characteristics remain consistent when adding more dimensions. In Fig. 5.5(a) and Fig. 5.5(b) we can see the results from these tests. While in Fig. 5.5(a), as one would expect considering Fig. 5.3, the growth is quite clearly non-exponential, the situation in Fig. 5.5(b) is not as visible. Nonetheless, the factor by which the work increases when adding another dimension decreases slightly as we add dimensions. We tried to illustrate this by adding the expected progression of the work at $\varepsilon = 0.001$ if the growth was perfectly exponential.

From just these two test cases, we can already conclude that the behaviour of MLMC-WoS over different dimensions depends strongly on the specific right-hand side. A five-dimensional case where the solution value is small and not changing rapidly might converge much quicker than a three-dimensional case, following something akin to our Gaussian test case. If we look in Fig. 5.2(a) at the dimension 5 datapoint at $\varepsilon = 2.5 \cdot 10^{-5}$ and compare it to 2 dimensions in Fig. 5.2(b), the Gaussian test case in 2 dimensions is around 10 times as expensive as Sine in 5 dimensions.



(a) Progression of work vs dimension for the test case Sine (5.1)



(b) Progression of work vs dimension for the test case Gaussian (5.2)

Figure 5.5: Comparing the work progression per dimension for both test functions at the centre of the hypersphere. Runs are made for different target errors, Epsilon, from 10^{-3} to $2.5 \cdot 10^{-5}$. The growth is non-exponential since this would produce a straight line. In the subfigure 5.5(b), the dashed line has the same slope as the cost between dimensions 2 and 3.

5.2.6 Speedup

Finally, it is time to discuss the relative performance of pure WoS compared to MLMC-WoS. It is not all that easy to get an accurate estimate of how thin the boundary layer has to be to reach a particular target error optimally. However, MLMC gives us a way to estimate it. If our scheme of adding levels to the point of just converging works, then the discretisation error of the final level should be close to the target error of MLMC. For this reason, we use the boundary thickness of the last level to estimate the correct δ to compare pure WoS with MLMC-WoS. We take several samples to help us estimate the variance and cost per sample, then extrapolate the cost of solving such that the statistical noise is as big as the target error. This way we can compute a speedup in cost of $\frac{W_{WoS}}{W_{MLMC}}$, so saying how many times we can execute MLMC-WoS to a given accuracy for the precise computational cost as running pure WoS once to the same accuracy.

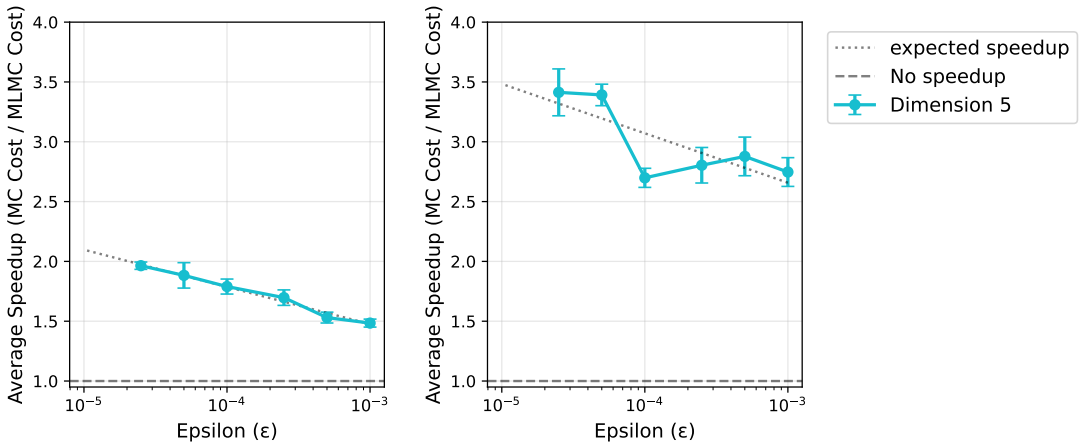
MLMC's estimate of the ideal δ is only precise to a factor of η since it can only suggest boundary layers corresponding to one of its levels.

Since the work per sample only increases logarithmically when decreasing δ, the error of the pure WoS cost estimate is bounded by a constant if MLMC does not underestimate δ due to, for example, a too small δ_0 boundary at the 0-th level.

It is therefore best to use $\eta = 2$ to estimate speedup. In Fig. 5.6(a) we can see the progression of this speedup as the target error decreases. The speedup is only small, but it is consistently increasing. This increase is consistent with what is expected from Sec. 3.3.5. We can see the weaknesses of this speedup estimate when we increase the ratio between levels, as seen in Fig. 5.6(b). We can see a sudden jump in the estimated speedup exactly when the decreased target error requires one more level in the MLMC-WoS at $\varepsilon = 10^{-4}$. After that, the speedup remains constant or decreases slightly for the next couple of runs, where the number of levels of MLMC-WoS remains constant.

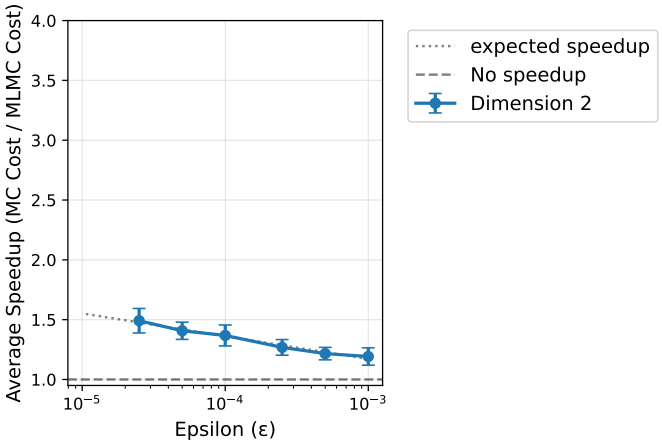
Comparing Fig. 5.6(a) with Fig. 5.6(c), we can also see that the speedup grows when increasing dimension and not only for decreasing ε.

This speedup does not seem all that significant, given that for Monte Carlo, the convergence rate means the bonus in accuracy for the same work is the square root of the speedup. However, this analysis ignores MLMC-Wos's most crucial advantage over pure WoS: its robustness to non-optimal hyperparameters. To use a pure WoS solution to the Poisson equation, one must determine a suitably small δ. Then either risk not reaching the wanted accuracy and having to run again, or choosing a δ that is too small, incurring more cost than necessary. MLMC-WoS is much more stable. The algorithm finds the needed level of discretisation on its own and is often stable to suboptimal hyperparameters, as seen in Sec. 5.2.4 for the parameter η.



(a) Speedup progression at $\eta = 2$ with fitted speedup $S(\varepsilon) \approx 0.14 \log(|\varepsilon|) + 0.54$.

(b) Speedup progression at $\eta = 16$ with fitted speedup $S(\varepsilon) \approx 0.18 \log(|\varepsilon|) + 1.42$.



(c) Speedup progression at $\eta = 2$ with fitted speedup $S(\varepsilon) \approx 0.08 \log(|\varepsilon|) + 0.60$.

Figure 5.6: Comparing the speedup at different dimensions and ratios between levels η. All runs are based on the Sine (5.1) test function and solve at the centre of the hypersphere with a boundary layer thickness at level 0 of $\delta_0 = 10^{-2}$. The speedup is greater for simulations with higher workloads because of the dimension or a smaller target error ε. Additionally, the speedup in Fig. 5.6(b) is unstable because the speedup estimate is less accurate due to the big step between levels. A jump in speedup is visible in Fig. 5.6(b) at $\varepsilon = 10^{-4}$, where one more level is needed for MLMC.

Chapter 6

Conclusion and Outlook

In this work, we have combined and extended the works of S. Pauli et al in [11], and J. Delaurentis and L. Romero in [6] to form a solver to the Poisson equation based on WoS solution to the Feynman-Kac formula, leveraging the power of Multilevel Monte Carlo to improve performance. We have shown that the improved convergence rate for an MLMC method shown in [11] holds for solutions to the Poisson equation. Convergence in experiments was consistent over different right-hand sides and dimensions matching the theoretical result. Even though the convergence rate of MLMC-WoS is faster than rate of pure WoS, the speedups we observed did not exceed four. Nonetheless, the speedup was biggest for computationally expensive experiments.

MLMC-WoS by solving point-wise cannot compete with traditional methods for the Poisson equation when solving on an entire mesh. To develop the method further, it is therefore important to find applications which suit it. Such applications will require a high-dimensional space and a right-hand side that can be evaluated quickly while incurring manageable memory costs.

We believe that this work paves the way for more sophisticated MLMC and WoS based methods solving PDEs. We see the possibility of using it for particle simulations where the Poisson equation is solved only at the exact particle positions and the right-hand side is estimated using either a tree-based non-uniform mesh, or a Kernel density estimation. Hybridized methods that use MLMC-WoS in a few points to regularise a domain, such that a traditional solver can be applied after, could be an interesting point of research. Extending the method to a more general parabolic PDE might allow for more applications. The Black-Scholes equation from finance is an example of a different problem that allows for Feynman-Kac-like formulas [28] and could therefore make for another application of MLMC-WoS.

Chapter 7

Acknowledgements

First and foremost, I want to thank my direct supervisor Dr. Mohsen Sadr. Without his tireless help, this work could not exist. He was invaluable not only through his great scientific advice and experience but also his responsiveness, his tactful feedback and enthusiasm for the topic. I never felt the need to hold back any question and conversations always made for stimulating discussions.

I want to thank the entire AMAS group at PSI. The welcoming atmosphere around their group made me feel right at home. In particular I want to name Sonali Mayani and Ryan Ammann for their ready help with any issues on the technical side.

Last but not least I want to thank Dr. Andreas Adelmann for his help in making the project possible in the first place and his encouraging presence throughout.

Bibliography

[1] M. Kac, "On distributions of certain wiener functionals," *Transactions of the American Mathematical Society*, vol. 65, no. 1, pp. 1–13, 1949.

[2] M. E. Muller, "Some continuous monte carlo methods for the dirichlet problem," *The Annals of Mathematical Statistics*, vol. 27, no. 3, pp. 569–589, 1956.

[3] G. Mikhailov, "'Walk on spheres' algorithms for solving helmholtz'equation," *Russian Journal of Numerical Analysis and Mathematical Modelling*, 1992.

[4] C.-O. Hwang and M. Mascagni, "Efficient modified "walk on spheres" algorithm for the linearized poisson–bolzmann equation," *Applied Physics Letters*, vol. 78, no. 6, pp. 787–789, 2001.

[5] M. Bossy, N. Champagnat, S. Maire, and D. Talay, "Probabilistic interpretation and random walk on spheres algorithms for the poisson-boltzmann equationin molecular dynamics," *ESAIM: Mathematical Modelling and Numerical Analysis*, vol. 44, no. 5, pp. 997–1048, 2010.

[6] J. Delaurentis and L. Romero, "A monte carlo method for poisson's equation," *Journal of Computational Physics*, vol. 90, no. 1, pp. 123–140, 1990.

[7] C.-O. Hwang, M. Mascagni, and J. A. Given, "A feynman–kac path-integral implementation for poisson's equation using an h-conditioned green's function," *Mathematics and computers in simulation*, vol. 62, no. 3-6, pp. 347–355, 2003.

[8] A. E. Kyprianou, A. Osojnik, and T. Shardlow, "Unbiased "walk-on-spheres" monte carlo methods for the fractional laplacian," *IMA Journal of Numerical Analysis*, vol. 38, no. 3, pp. 1550–1578, 2018.

[9] H. C. Nam, J. Berner, and A. Anandkumar, "Solving poisson equations using neural walk-on-spheres," *arXiv preprint arXiv:2406.03494*, 2024.

[10] M. B. Giles, "Multilevel monte carlo path simulation," *Operations research*, vol. 56, no. 3, pp. 607–617, 2008.

[11] S. Pauli, R. N. Gantner, P. Arbenz, and A. Adelmann, "Multilevel monte carlo for the feynman–kac formula for the laplace equation," *BIT Numerical Mathematics*, vol. 55, no. 4, pp. 1125–1143, 2015.

[12] P. E. Kloeden and E. Platen, *Numerical Solution of Stochastic Differential Equations*. Springer, 1992.

[13] F. Müller, P. Jenny, and D. W. Meyer, "Multilevel monte carlo for two phase flow and buckley–leverett transport in random heterogeneous porous media," *Journal of Computational Physics*, vol. 250, pp. 685–702, 2013.

[14] I. Binder and M. Braverman, "The complexity of simulating brownian motion," in *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 58–67, SIAM, 2009.

[15] M. Mascagni and C.-O. Hwang, "e-shell error analysis for "walk on spheres" algorithms," *Mathematics and Computers in Simulation*, vol. 63, no. 2, pp. 93–104, 2003.

[16] L. C. Evans, *Partial differential equations*, vol. 19. American Mathematical Society, 2022.

[17] E. C. Zachmanoglou and D. W. Thoe, *Introduction to partial differential equations with applications*. Courier Corporation, 1986.

[18] L. Blumenson, "A derivation of n-dimensional spherical coordinates," *The American Mathematical Monthly*, vol. 67, no. 2, pp. 63–66, 1960.

[19] M. Abramowitz and I. A. Stegun, *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*, vol. 55. Courier Corporation, 1965.

[20] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[21] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020.

[22] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference* (Stéfan van der Walt and Jarrod Millman, eds.), pp. 56 – 61, 2010.

[23] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[24] S. Muralikrishnan, M. Frey, A. Vinciguerra, M. Ligotino, A. J. Cerfon, M. Stoyanov, R. Gayatri, and A. Adelmann, "Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures," in *Proceedings of the 2024 SIAM Conference on Parallel Processing for Scientific Computing (PP)*, pp. 26–38, SIAM, 2024.

[25] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[26] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[27] J. R. Shewchuk *et al.*, "An introduction to the conjugate gradient method without the agonizing pain," 1994.

[28] S. Janson and J. Tysk, "Feynman–kac formulas for black–scholes–type operators," *Bulletin of the London Mathematical Society*, vol. 38, no. 2, pp. 269–282, 2006.

Appendix A

Additional plots

A.1 WoS discretisation error

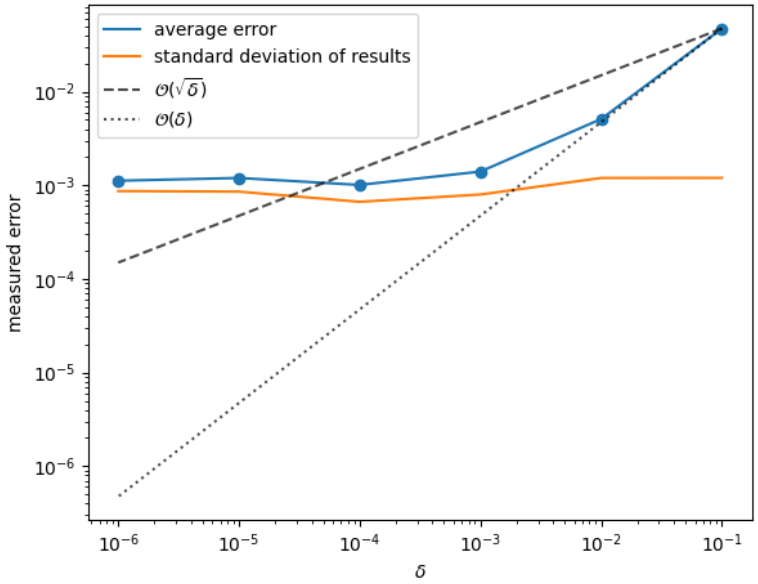


Figure A.1: Behaviour of pure WoS sampling at different δ with $N = 10^6$ samples averaged over 20 runs for the test function Exponential 4.5 averaged over 20 runs

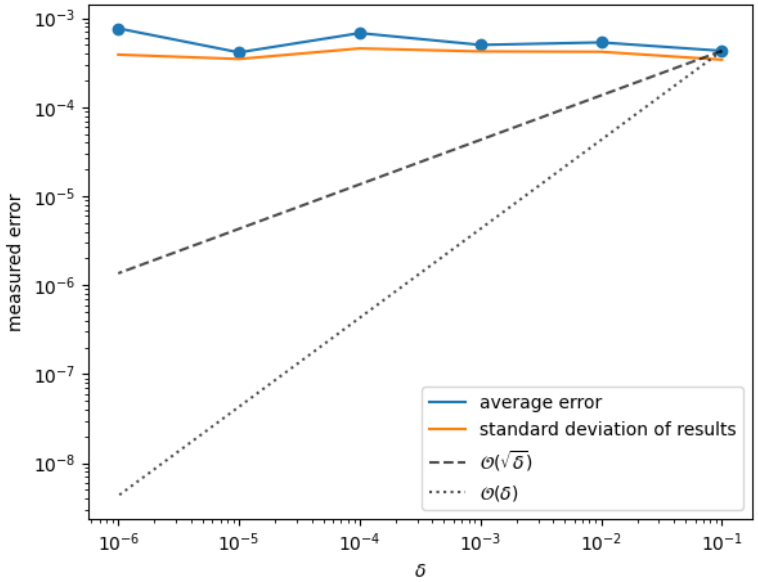


Figure A.2: Behaviour of pure WoS sampling at different δ with $N = 10^6$ samples averaged over 20 runs for the test function Cosine 4.2 averaged over 20 runs

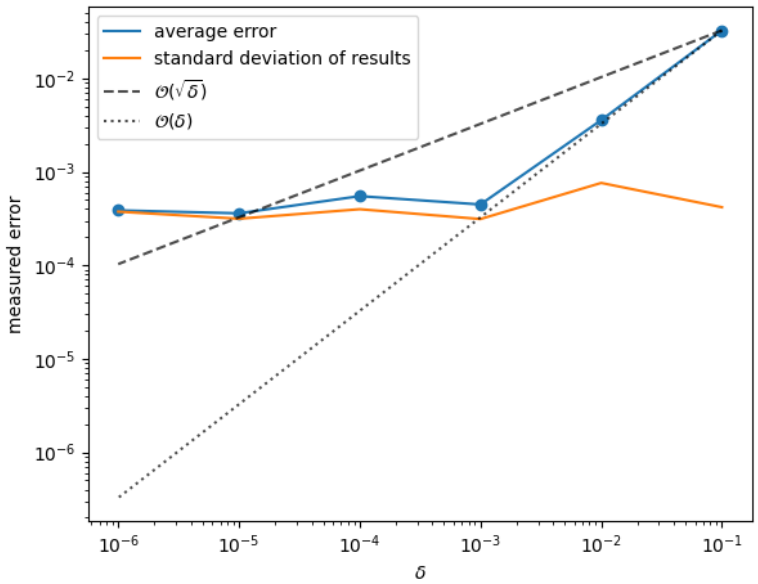


Figure A.3: Behaviour of pure WoS sampling at different δ with $N = 10^6$ samples averaged over 20 runs for the test function Polynomial 4.6 averaged over 20 runs

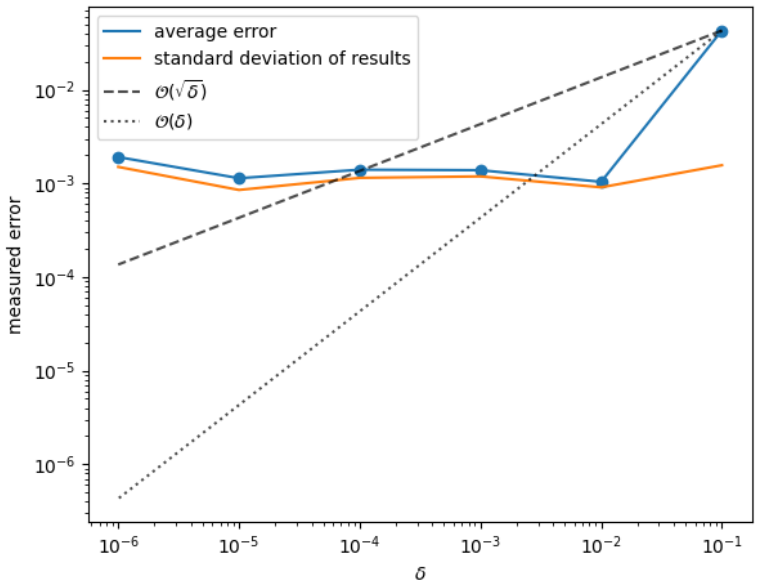


Figure A.4: Behaviour of pure WoS sampling at different δ with $N = 10^6$ samples averaged over 20 runs for the test function Cosine^2 4.3 averaged over 20 runs

A.2 MLMC-WoS convergence

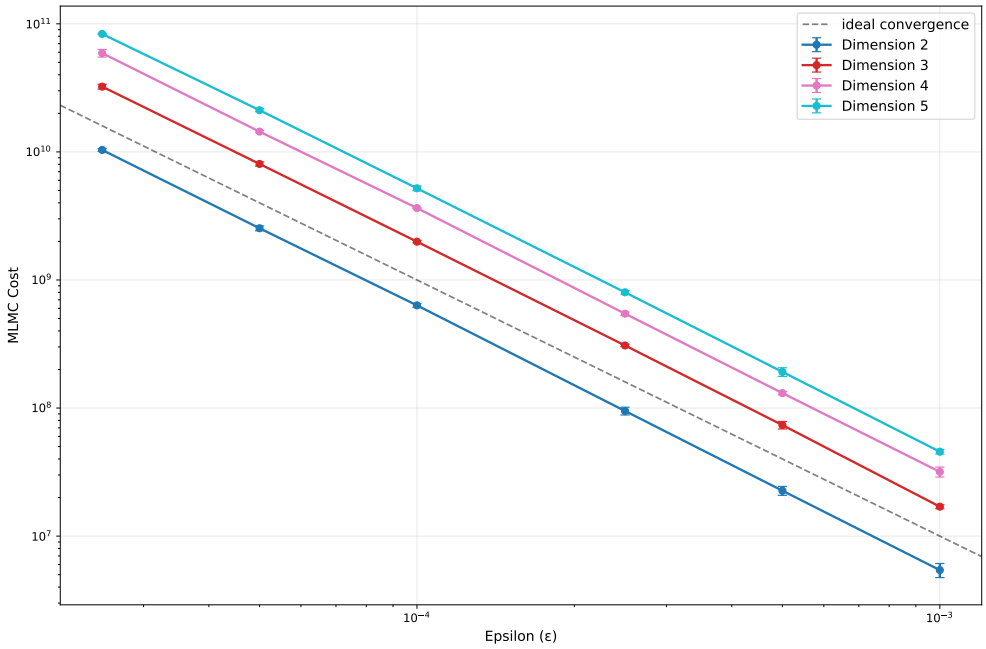


Figure A.5: Progression of work versus target error for the test case Sine 5.1 at the centre of the hypercube where $\delta_0 = 10^{-2}, \eta = 2$

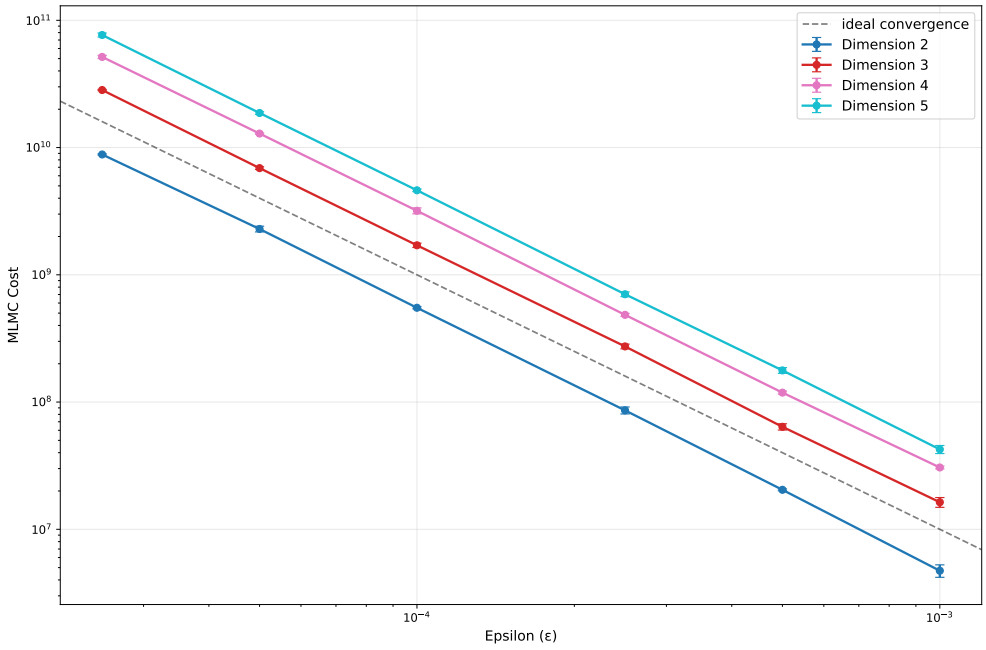


Figure A.6: Progression of work versus target error for the test case Sine 5.1 at the centre of the hypercube where $\delta_0 = 10^{-2}, \eta = 4$

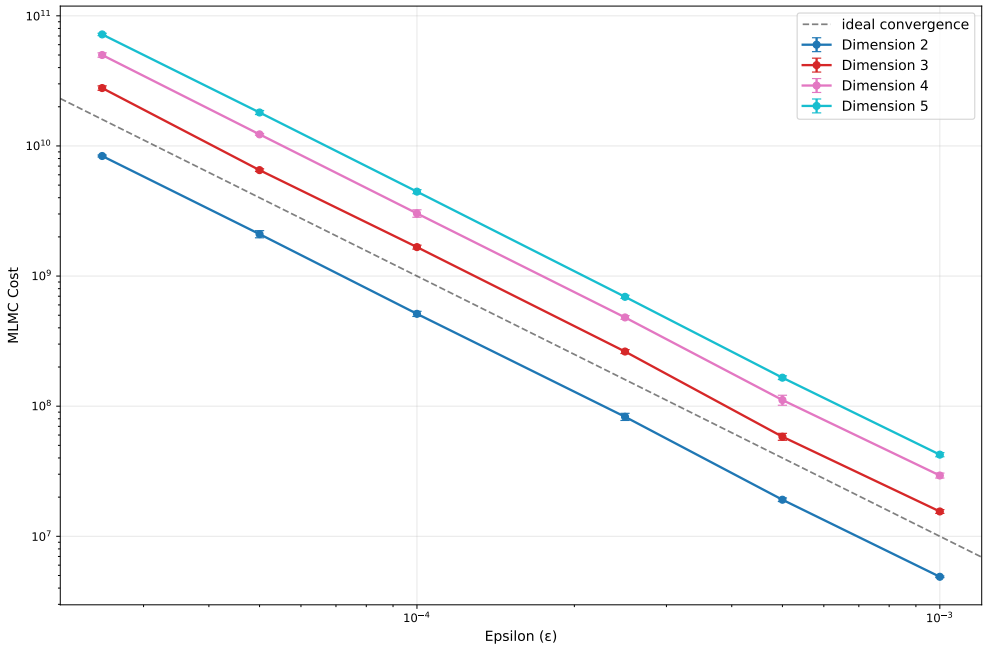


Figure A.7: Progression of work versus target error for the test case Sine (5.1) at the centre of the hypercube where $\delta_0 = 10^{-2}, \eta = 8$

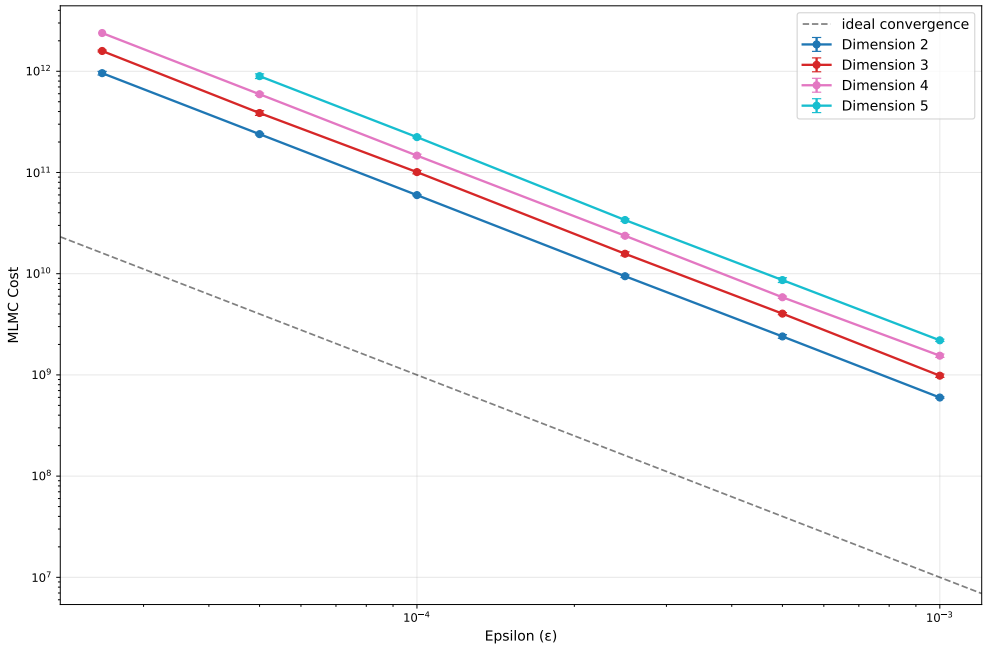


Figure A.8: Progression of work versus target error for the test case Gaussian (5.2) at the centre of the hypercube where $\delta_0 = 10^{-2}, \eta = 2$

A.3 MLMC-Wos speedups

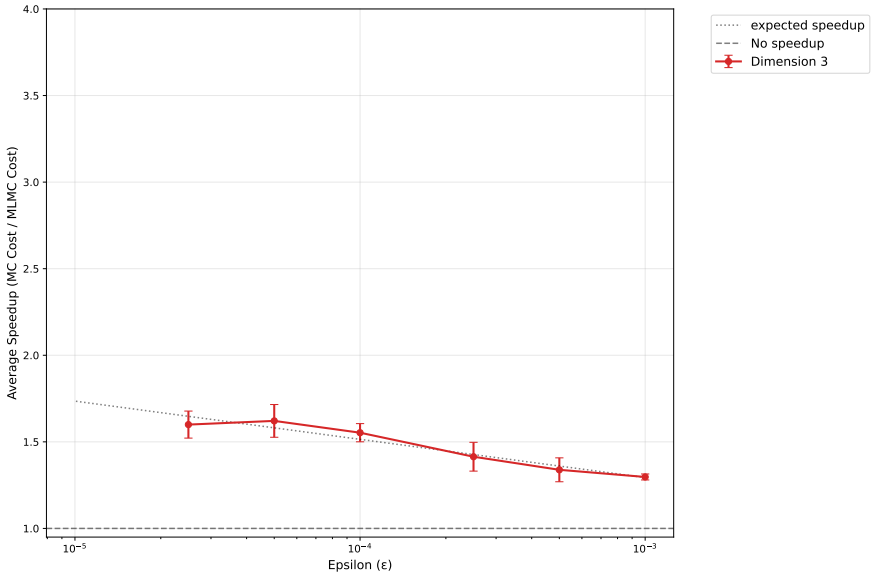


Figure A.9: The same run parameters as Fig. 5.6(a) in three dimensions

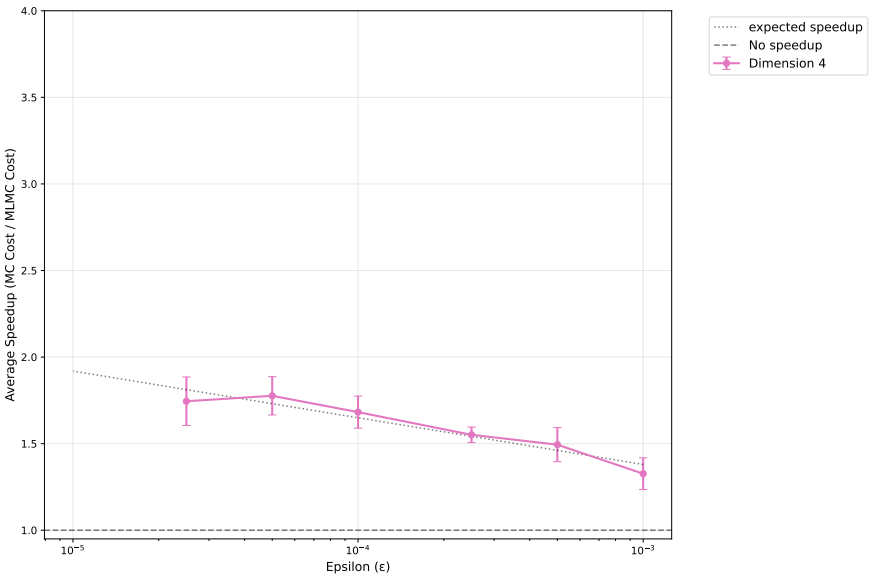


Figure A.10: The same run parameters as Fig. 5.6(a) in four dimensions

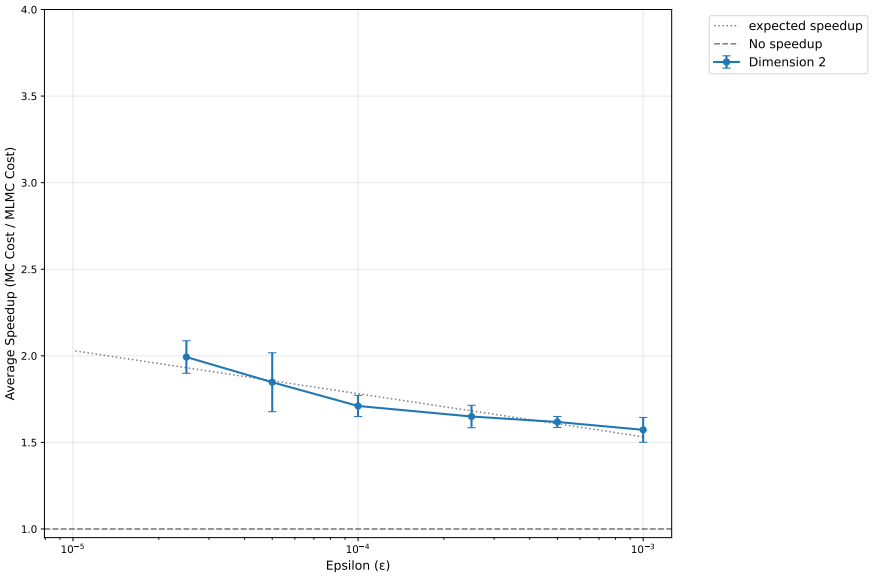


Figure A.11: The same run parameters as Fig. 5.6(c) but with $\eta = 4$

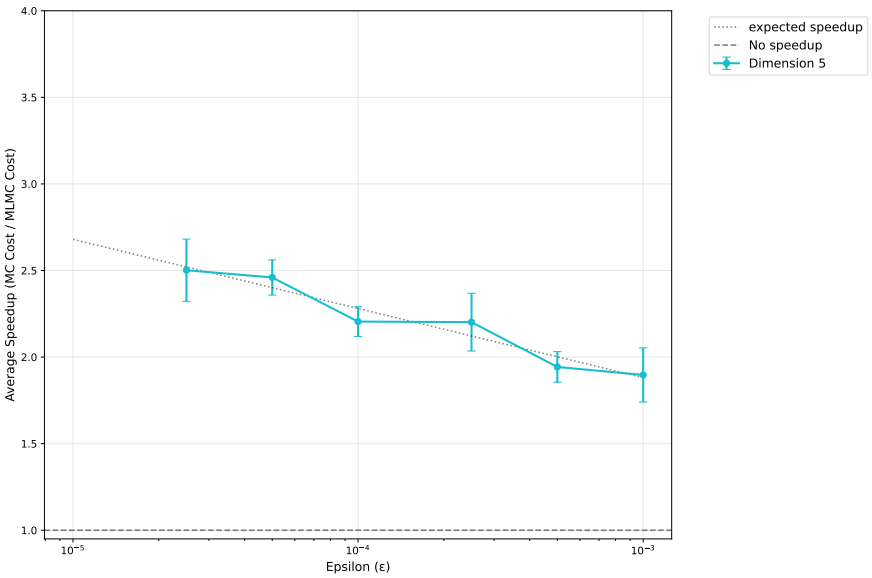


Figure A.12: The same run parameters as Fig. 5.6(a) but with $\eta = 4$

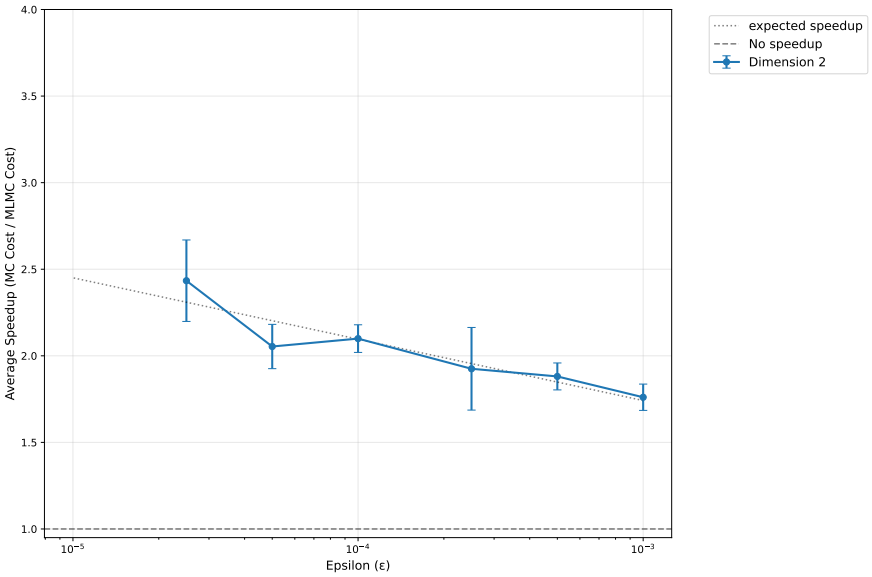


Figure A.13: The same run parameters as Fig. 5.6(c) but with $\eta = 8$

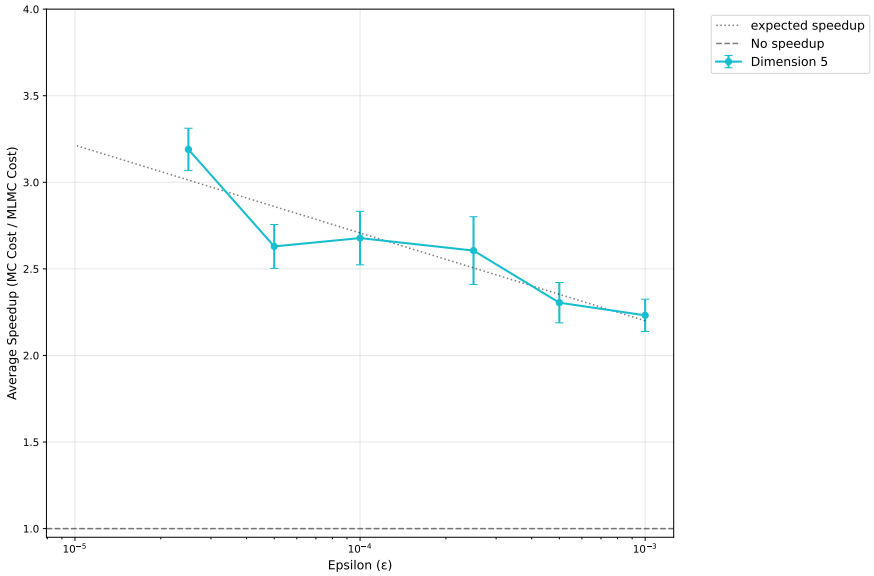


Figure A.14: The same run parameters as Fig. 5.6(a) but with $\eta = 8$

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten schriftlichen Arbeit. Eine der folgenden zwei Optionen ist **in Absprache mit der verantwortlichen Betreuungsperson** verbindlich auszuwählen:

☐ Ich erkläre hiermit, dass ich die vorliegende Arbeit eigenverantwortlich verfasst habe, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge der Betreuungsperson. Es wurden keine Technologien der generativen künstlichen Intelligenz[1] verwendet.

☒ Ich erkläre hiermit, dass ich die vorliegende Arbeit eigenverantwortlich verfasst habe. Dabei habe ich nur die erlaubten Hilfsmittel verwendet, darunter sprachliche und inhaltliche Korrekturvorschläge der Betreuungsperson sowie Technologien der generativen künstlichen Intelligenz. Deren Einsatz und Kennzeichnung ist mit der Betreuungsperson abgesprochen.

Titel der Arbeit:

On Monte Carlo methods for the Poisson equation

Verfasst von:
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Schucan	Caspar

Ich bestätige mit meiner Unterschrift:
- Ich habe mich an die Regeln des «Zitierleitfadens» gehalten.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu und vollständig dokumentiert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Eigenständigkeit überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
23.06.2025	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie grundsätzlich gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

[1] Für weitere Informationen konsultieren Sie bitte die Webseiten der ETH Zürich, bspw. https://ethz.ch/de/die-eth-zuerich/lehre/ai-in-education.html und https://library.ethz.ch/forschen-und-publizieren/Wissenschaftliches-Schreiben-an-der-ETH-Zuerich.html (Änderungen vorbehalten).