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

Matteo Burzio **Luca Civilla** **Francesco Vinciguerra**

January 11, 2026

Abstract

The goal of this project was to estimate the price of Asian call and binary digital Asian options, formulated as high-dimensional integrals. We established baseline convergence rates using Crude Monte Carlo (CMC) and Randomized Quasi-Monte Carlo (RQMC). To restore QMC efficiency for non-smooth integrands, we applied pre-integration smoothing. Finally, we employed importance sampling to achieve variance reduction in the out-of-the-money regime.

Contents

1 Introduction

The task of computing high-dimensional integrals is a problem which arises often in the context of financial mathematics, for example when dealing with evaluating Asian options. Due to the absence of closed form solutions, the standard approach is to use numerical methods which don't suffer from the curse of dimensionality, of which Monte Carlo methods are the standard benchmark. Even if they are capable of obtaining a robust convergence rate of $O(N^{-1/2})$, Crude Monte Carlo methods often are too computationally expensive. For this reason, Quasi-Monte Carlo methods are preferred since they theoretically are capable of reaching the superior convergence rate of $O(N^{-1})$ (up to logarithmic factors).

However, the efficacy of QMC is linked to the smoothness of the integrand, which, in the case of the payoff functions for Asian options, presents either a kink or a jump, breaking the regularity requirements for the improved convergence rate. To tackle this, we consider a smoothing technique which involves analytically integrating along a certain direction (in our implementation the specific one chosen is the primary active direction).

Furthermore, we address the problematic high relative variance of the deep out-of-the-money regime by implementing a variance reduction technique known as Optimal Drift Importance Sampling, which shifts the samples closer to the strike price in order to make exercising the option a less rare event.

2 Theory and Background

2.1 Option pricing framework

Let $T > 0$ and S be the price of a stock given by the solution of the stochastic differential equation

$$\begin{cases} dS_t = rS_t dt + \sigma S_t dW_t, & t \in [0,T] \\ S_0 = s_0, \end{cases} \tag{1}$$

where r is the interest rate, σ is the volatility, s_0 is the initial price and $W = (W_t, t \in [0,T])$ is a standard one-dimensional Wiener process. It can be shown that

$$\forall t \in [0,T], \;\; S_t = s_0 e^{(r-\frac{\sigma^2}{2})t+\sigma W_t}, \tag{2}$$

which allows us to write the stock price S_t as a function of W_t.

Consider an option with maturity T based on the stock price S.

The payoff $\Psi = \Psi((S_t)_{t\in[0,T]})$ is the function determining the cash flow received by the option holder at maturity T, contingent on the state of the underlying asset S.

The value of an option is defined as the discounted expected payoff under the risk-neutral measure, formally

$$V = e^{-rT}\mathbb{E}[\Psi]. \tag{3}$$

2.2 Asian options

An Asian option is a path-dependent exotic derivative whose payoff is determined by the average price of the underlying asset S, rather than the price at maturity alone (as in European options). In particular, we consider arithmetic Asian options, where the average is taken over a discrete partition of $[0, T]$.

Let $d \in \mathbb{N}$, we define the partition of the time interval $[0, T]$ as $t_i := i\Delta t$ for $i = 1, \dots, d$ where $\Delta t := T/d$. The payoffs of Asian options can then be expressed as a function of the random vector $\mathbf{W} = (W_{t_1}, \dots, W_{t_d})$, which is distributed as a multivariate Gaussian with mean zero and covariance matrix $\Sigma_{ij} = \min(t_i, t_j)$.

Let $K > 0$ be the strike price. We will focus on the two following kinds of Asian options:

- **Asian call option**:

$$\Psi_1(\mathbf{W}) := \left(\frac{1}{d}\sum_{i=1}^{d} S_{t_i}(W_{t_i}) - K\right)_+ = \left(\frac{1}{d}\sum_{i=1}^{d} S_{t_i}(W_{t_i}) - K\right)\mathbb{1}_{\left\{\frac{1}{d}\sum_{i=1}^{d} S_{t_i}(W_{t_i}) - K \geq 0\right\}} \tag{4}$$

- **Binary digital Asian option**:

$$\Psi_2(\mathbf{W}) := \mathbb{1}\left\{\frac{1}{d}\sum_{i=1}^{d} S_{t_i}(W_{t_i}) - K \geq 0\right\} \tag{5}$$

The values V_i with $i = 1, 2$ of these Asian options are given by

$$V_i = e^{-rT}\mathbb{E}[\Psi_i(\mathbf{W})] = \frac{e^{-rT}}{(2\pi)^{d/2}\sqrt{\det(C)}}\int_{\mathbb{R}^m} \Psi_i(\mathbf{w}) e^{-\frac{1}{2}\mathbf{w}^T\Sigma^{-1}\mathbf{w}} d\mathbf{w} \tag{6}$$

2.3 Numerical Integration Methods

Since the integral (6) generally lacks a closed-form analytical solution, numerical approximation is necessary. Given the high dimensionality of the problem ($d \gg 1$), deterministic quadrature rules are subject to the curse of dimensionality, making Monte Carlo simulation the preferred methodology.

2.3.1 Crude Monte Carlo (CMC)

The standard Monte Carlo estimator approximates the expected value by the sample mean of N independent and identically distributed (i.i.d.) realizations of $\mathbf{W}$:

$$\hat{V}_{CMC} = \frac{e^{-rT}}{N}\sum_{i=1}^{N} \Psi(\mathbf{W}^{(i)}), \quad \mathbf{W}^{(i)} \underset{\text{i.i.d.}}{\sim} \mathcal{N}_d(\mathbf{0}, \Sigma) \tag{7}$$

By the Strong Law of Large Numbers (SLLN), $\hat{V}_{CMC}$ converges almost surely to the true value V as $N \to \infty$. The convergence rate is governed by the Central Limit Theorem (CLT), which states that the error is asymptotically normal with standard deviation proportional to $\sigma N^{-1/2}$, where $\sigma^2 = \mathrm{Var}[\Psi(\mathbf{W})]$. Thus, the convergence rate is $O(N^{-1/2})$, which is robust to dimension but computationally slow.

2.3.2 Quasi-Monte Carlo (QMC)

To accelerate convergence, we consider Quasi-Monte Carlo methods. Instead of random samples, QMC utilizes a deterministic low-discrepancy sequence $\mathcal{S} = \{\mathbf{X}^{(n)}\}_{n\in\mathbb{N}}$ in the unit hypercube $[0, 1]^d$, designed to fill the space more uniformly than random points. The estimator is given by

$$\hat{V}_{QMC} = \frac{e^{-rT}}{N}\sum_{i=1}^{N} \Psi(A\Phi^{-1}(\mathbf{X}^{(i)})), \tag{8}$$

where Φ^{-1} is the inverse cumulative distribution function of the standard normal applied element-wise, and A is the lower triangular matrix obtained from the Cholesky decomposition of the covariance matrix Σ, satisfying $\Sigma = AA^T$.

The "a priori" error bound for QMC is provided by the Koksma-Hlawka inequality:

$$|V - \hat{V}_{QMC}| \leq \|\Psi\|_{HK} \cdot D^*(\mathcal{S}_N), \tag{9}$$

where $\|\cdot\|_{HK}$ denotes the total variation norm in the sense of Hardy and Krause and D^*_N is the star-discrepancy of the point set. Since low-discrepancy sequences satisfy $D^*(\mathcal{S}_N) = O(N^{-1}(\log N)^d)$, QMC theoretically achieves a convergence rate of $O(N^{-1})$ (up to logarithmic terms), provided the integrand has bounded variation.

2.4 Smoothing by pre-integration

However, the payoff functions for Asian options are non-smooth:

- Ψ_1 has a continuous derivative discontinuity (a "kink") at the strike K.
- Ψ_2 has a jump discontinuity at K.

These singularities result in unbounded variation in high dimensions, causing the convergence rate of standard QMC to degrade to $O(N^{-1/2})$, providing little advantage over CMC.

In order to maintain the favorable convergence rate of QMC, one can implement the following pre-integration trick.

Consider the integral over $\mathbb{R}^d$ with respect to some distribution $\prod_{j=1}^d \rho_j(x_j)$ of an integrand $f : \mathbb{R}^d \to \mathbb{R}$ of the form

$$f(\mathbf{x}) = \theta(\mathbf{x})\mathbb{1}_{\{\phi(\mathbf{x})\geq 0\}},$$

where θ and ϕ are smooth functions. Also, assume that $\frac{\partial \phi}{\partial x_j}(\mathbf{x}) > 0 \ \forall \mathbf{x} \in \mathbb{R}^d$ for some $j = 1, \ldots, d$ and that $\phi(\mathbf{x}) \to \pm\infty$ as $x_j \to \pm\infty$.

The method consists in rewriting the integral in the following way:

$$\int_{\mathbb{R}^d} f(\mathbf{x})\rho(\mathbf{x})\mathrm{d}\mathbf{x} = \int_{\mathbb{R}^{d-1}} \underbrace{\left(\int_{\mathbb{R}} f(x_j, \mathbf{x}_{-j})\rho_j(x_j)\mathrm{d}x_j\right)}_{:= p(\mathbf{x}_{-j})} \rho_{-j}(\mathbf{x}_{-j})\mathrm{d}\mathbf{x}_{-j}. \tag{10}$$

where $\mathbf{x} = (x_j, \mathbf{x}_{-j})$, with $x_{-j} = (x_1, x_2, \ldots, x_{j-1}, x_{j+1}, \ldots, x_d)$, and similarly for ρ_j and ρ_{-j}. The reason for doing this is that the resulting inner function $p(\mathbf{x}_{-j})$ is smooth, i.e., it doesn't have a kink or a jump anymore. Notice that for any fixed $\mathbf{x}_{-j} \in \mathbb{R}^{d-1}$ by the assumptions on the derivatives of ϕ, the function $\mathbf{x}_j \mapsto f(x_j, \mathbf{x}_{-j})$ has a jump at the (unique) point where $\phi(x_j, \mathbf{x}_{-j}) = 0$. It follows from the implicit function theorem that for each $\mathbf{x}_{-j}$, there exists a unique value $x_j^* := \psi(\mathbf{x}_{-j})$ for which $\phi(x_j, \mathbf{x}_{-j}) < 0$ if $x_j < x_j^*$ and $\phi(x_j, \mathbf{x}_{-j}) > 0$ if $x_j > x_j^*$. Thus, given the chosen form of f, we can write

$$p(\mathbf{x}_{-j}) := \int_{\mathbb{R}} f(x_j, \mathbf{x}_{-j})\rho_j(x_j)\mathrm{d}x_j = \int_{\psi(\mathbf{x}_{-j})}^{+\infty} \theta(x_j, \mathbf{x}_{-j})\rho_j(x_j)\mathrm{d}x_j, \tag{11}$$

with both θ and ψ smooth, and the integral

$$\int_{\mathbb{R}^d} f(\mathbf{x})\rho(\mathbf{x})\mathrm{d}x = \int_{\mathbb{R}^{d-1}} p(\mathbf{x}_{-j})\rho_{-j}(\mathbf{x}_{-j})\mathrm{d}\mathbf{x}_{-j} \tag{12}$$

hopefully now has a smooth integrand.

In the specific case of Asian options, first note that both payoffs (4) and (5) satisfy the regularity conditions required to use this technique.

While the pricing integral V_i in (6) is originally defined under a correlated measure with covariance Σ, we can perform a change of variables using the Cholesky decomposition $\Sigma = AA^T$. Let $\mathbf{Z} \sim \mathcal{N}_d(\mathbf{0}, I_d)$ be a vector of independent standard normal variables such that $\mathbf{W} = A\mathbf{Z}$. The integral (6) can be rewritten as

$$V_i = e^{-rT}\mathbb{E}[\tilde{\Psi}_i(\mathbf{Z})] = e^{-rT}\int_{\mathbb{R}^d}\tilde{\Psi}_i(\mathbf{z})\rho(\mathbf{z})\mathrm{d}\mathbf{z}, \tag{13}$$

where the new integrand $\tilde{\Psi}_i(\mathbf{z}) := \Psi_i(A\mathbf{z})$ is now defined over a product distribution (since ρ now is the density of a standard multivariate Gaussian), satisfying the requirement for pre-integration.

2.5 Active subspaces and optimal smoothing direction

Standard coordinate-aligned pre-integration may be suboptimal if the discontinuity is not aligned with one of the canonical axes. To maximize the regularizing effect of pre-integration, it is necessary to identify the direction in the input space along which the payoff functions $\tilde{\Psi}_i$ exhibit the greatest variability (in this specific case, a jump or a kink). A possible choice involves building the path via a Brownian Bridge Construction, selecting the first dimension as the smoothing direction because it captures the largest portion of the total path variance. Instead we choose to follow the protocol proposed in [1] which chooses the direction which maximizes the expected squared directional derivative of the payoff, which corresponds to the solution of the following optimization problem

$$\max_{\|\theta\|=1} \mathbb{E}\left[(\nabla\tilde{\Psi}(\mathbf{Z})^T\theta)^2\right] = \max_{\|\theta\|=1} \theta^T C\theta, \tag{14}$$

where $\mathbf{Z} \sim \mathcal{N}_d(\mathbf{0}, I_d)$ and $C \in \mathbb{R}^{d\times d}$ defined as $C = \mathbb{E}\left[\nabla\tilde{\Psi}(\mathbf{Z})\nabla\tilde{\Psi}(\mathbf{Z})^T\right]$ is the uncentered gradient covariance matrix. For the binary digital Asian option, where the gradient vanishes almost everywhere, we approximate the active subspace using the gradient of the corresponding Asian call payoff as a smooth proxy.

Since C is symmetric and positive semi-definite, it admits the spectral decomposition $C = U\Lambda U^T$, where $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_d)$ contains the eigenvalues sorted in descending order ($\lambda_1 \geq \cdots \geq \lambda_d \geq 0$), and $U = [\mathbf{u}_1, \ldots, \mathbf{u}_d]$ contains the corresponding orthonormal eigenvectors.

The leading eigenvector $\mathbf{u}_1$ defines the primary active direction (i.e., the best direction to perform the pre-integration). In fact, by the Rayleigh-Ritz theorem, the solution of (14) is precisely the leading eigenvector $\mathbf{u}_1$, with the maximum value being the corresponding eigenvalue λ_1. By pre-integrating along $\mathbf{u}_1$, we resolve the dimension responsible for the largest fluctuations in the function value, leaving a residual integration problem over the subspace where the function is comparatively flat.

To implement this, we construct an orthogonal matrix $Q \in \mathbb{R}^{d\times d}$ such that its first column aligns with the active direction $\mathbf{u}_1$. This is achieved efficiently via a Householder reflection. Let $\mathbf{e}_1$ denote the first canonical basis vector of $\mathbb{R}^d$. We define the Householder vector $\mathbf{v} := \mathbf{u}_1 - \|\mathbf{u}_1\|\mathbf{e}_1$ and the resulting matrix as

$$Q = I_d - 2\frac{\mathbf{v}\mathbf{v}^T}{\|\mathbf{v}\|^2}. \tag{15}$$

By construction, Q is symmetric, orthogonal, and satisfies $Q\mathbf{e}_1 = \mathbf{u}_1$. We then define the rotated random variable $\mathbf{Z}' := Q^T\mathbf{Z}$, which remains distributed as $\mathcal{N}_d(\mathbf{0}, I_d)$. The payoff is re-parameterized again as $\tilde{\Psi}_{rotated}(\mathbf{Z}) := \tilde{\Psi}(\mathbf{Z}')$. Since the first component Z_1 captures the variation along the active direction $\mathbf{u}_1$, we apply the pre-integration technique over it:

$$\mathbb{E}[\Psi(\mathbf{W})] = \mathbb{E}[\tilde{\Psi}_{rotated}(\mathbf{Z})] = \int_{\mathbb{R}^{d-1}}\left(\int_{\mathbb{R}}\tilde{\Psi}(z_1, \mathbf{z}_{-1})\rho_1(z_1)\,\mathrm{d}z_1\right)\rho_{-1}(\mathbf{z}_{-1})\,\mathrm{d}\mathbf{z}_{-1}, \tag{16}$$

where ρ_1 denotes the standard normal density and ρ_{-1} the density of a standard normal Gaussian of dimension $d-1$.

2.6 Sample size in the out of the money regime

When estimating the value of an option in the "out-of-the-money" (OTM) regime, the probability of the option expiring with a positive payoff is low. Consequently, the Crude Monte Carlo estimator $\hat{V}_{CMC}$ often yields zero for small sample sizes, and its relative error can be prohibitively large.

To determine the sample size N required to achieve a target relative accuracy ε in the mean squared sense, we consider the Root Mean Squared Error (RMSE) criterion:

$$\frac{\sqrt{\mathbb{E}[(\hat{V}_{CMC} - V)^2]}}{V} \leq \varepsilon. \tag{17}$$

Since the CMC estimator is unbiased, the Mean Squared Error (MSE) is equal to the variance of the estimator, $\mathrm{Var}[\hat{V}_{CMC}] = \frac{1}{N}\mathrm{Var}[\Psi(\mathbf{W})]$. The condition becomes:

$$\frac{\sqrt{\mathrm{Var}[\Psi(\mathbf{W})]}}{\sqrt{N} \cdot V} \leq \varepsilon. \tag{18}$$

Solving for N, we obtain the required sample size in terms of the coefficient of variation of the payoff:

$$N \geq \left\lceil \frac{\mathrm{Var}[\Psi(\mathbf{W})]}{\varepsilon^2 V^2} \right\rceil. \tag{19}$$

Since the true variance $\mathrm{Var}[\Psi(\mathbf{W})]$ and the true option value V are unknown a priori, they must be estimated numerically.

Equation (19) highlights a critical dependency: for a fixed relative accuracy ε, the required computational effort scales with the variance of the payoff. In the deep OTM regime, the coefficient of variation is typically very large, rendering the CMC approach computationally prohibitive as N grows uncontrollably. To achieve the target precision within a reasonable computational budget, increasing the sample size is often not a viable strategy. Instead, we must focus on reducing the numerator in (19). This motivates the implementation of variance reduction techniques.

2.7 Importance Sampling via Optimal Drift (ODIS)

Importance Sampling (IS) is a variance reduction technique that is particularly effective for estimating probabilities of rare events, which is precisely the case for the payoff of deep out-of-the-money (OTM) options. IS introduces a proposal density $q(\mathbf{w})$ to rewrite the value of the option (6) as

$$V_i = e^{-rT} \int_{\mathbb{R}^d} \tilde{\Psi}_i(\mathbf{z}) \frac{\rho(\mathbf{z})}{q(\mathbf{z})} q(\mathbf{z}) \mathrm{d}\mathbf{z} = e^{-rT} \int_{\mathbb{R}^d} \tilde{\Psi}_i(\mathbf{z}) w(\mathbf{z}) q(\mathbf{z}) \mathrm{d}\mathbf{z} = e^{-rT} \mathbb{E}_q \left[\tilde{\Psi}(\mathbf{Z}) w(\mathbf{Z}) \right], \tag{20}$$

where $\mathbf{Z} \sim \mathcal{N}_d(\mathbf{0}, I_d)$, $w = \frac{\rho}{q}$ and the expectation $\mathbb{E}_q$ is taken with respect to the density q.

While the theoretical optimal estimator is achieved by choosing $q^*(\mathbf{z}) \propto |\tilde{\Psi}_i(\mathbf{z})| \rho(\mathbf{z})$, this density is generally intractable as its normalization constant is precisely the integral V_i we wish to compute. Consequently, we restrict our search to a parametric family of distributions that approximates q^* while remaining easy to sample from.

Optimal Drift Importance Sampling (ODIS) employs a mean-shift strategy, restricting q to the family of multivariate Gaussian densities with a shifted mean $\boldsymbol{\mu}$ and identity covariance:

$$q(\mathbf{z}; \boldsymbol{\mu}) = (2\pi)^{-d/2} e^{-\frac{1}{2}\|\mathbf{z} - \boldsymbol{\mu}\|^2}. \tag{21}$$

Under this change of measure, the likelihood ratio simplifies to the exponential of a linear function of $\mathbf{z}$:

$$w(\mathbf{z}; \boldsymbol{\mu}) = \frac{e^{-\frac{1}{2}\|\mathbf{z}\|^2}}{e^{-\frac{1}{2}\|\mathbf{z} - \boldsymbol{\mu}\|^2}} = e^{-\boldsymbol{\mu}^T \mathbf{z} + \frac{1}{2}\|\boldsymbol{\mu}\|^2}. \tag{22}$$

To select the optimal parameter $\boldsymbol{\mu}^*$, we aim to match the mode of the proposal density $q(\mathbf{z};\boldsymbol{\mu})$ to the mode of the optimal zero-variance density $q^*(\mathbf{z})$. This is equivalent to finding the point $\mathbf{z}$ that maximizes the integrand product $\tilde{\Psi}_i(\mathbf{z})\rho(\mathbf{z})$, which corresponds to solving the following optimization problem:

$$\boldsymbol{\mu}^* = \arg\max_{\mathbf{z}\in\mathbb{R}^d} \left(\ln \tilde{\Psi}_i(\mathbf{z}) - \frac{1}{2}\|\mathbf{z}\|^2 \right), \tag{23}$$

where we have taken the logarithm of the product (which doesn't affect the solution since it's an increasing transformation) and ignored the terms not depending on $\mathbf{z}$. In the case of the binary digital Asian option, where the payoff is an indicator function, the optimization objective is modified to maximize the proposal density at the boundary surface defined by the strike price.

2.8 Importance sampling for QMC estimators

The integration of Importance Sampling within a Quasi-Monte Carlo framework requires mapping the deterministic low-discrepancy sequence to the proposal distribution q rather than the nominal distribution ρ. Let $\mathcal{S} = \{\mathbf{X}^{(n)}\}_{n\in\mathbb{N}}$ be a low-discrepancy sequence in the unit hypercube $[0,1]^d$. The RQMC-IS estimator is given by:

$$\hat{V}^{IS}_{RQMC} = \frac{e^{-rT}}{N} \sum_{i=1}^{N} \tilde{\Psi}(G^{-1}(\mathbf{X}^{(i)})) \frac{\rho(G^{-1}(\mathbf{X}^{(i)}))}{q(G^{-1}(\mathbf{X}^{(i)}))}, \tag{24}$$

where $G^{-1} : [0,1]^d \to \mathbb{R}^d$ denotes the inverse CDF applied component-wise associated with the proposal density q.

In the context of ODIS, the proposal density $q(\mathbf{z};\boldsymbol{\mu})$ corresponds to a mean-shifted multivariate Gaussian $\mathcal{N}_d(\boldsymbol{\mu}, I_d)$. The generation of sample points from this distribution using QMC is computationally direct; we simply shift the standard normal inverse mapping:

$$\mathbf{Z}^{(i)} = \Phi^{-1}(A\mathbf{X}^{(i)}) + \boldsymbol{\mu}, \tag{25}$$

where as before Φ^{-1} is the inverse cumulative distribution function of the standard normal applied element-wise, and A is the lower triangular matrix obtained from the Cholesky decomposition of the covariance matrix Σ, satisfying $\Sigma = AA^T$.

Substituting this transformation and the likelihood ratio $w(\mathbf{z};\boldsymbol{\mu}) = e^{-\boldsymbol{\mu}^T\mathbf{z}+\frac{1}{2}\|\boldsymbol{\mu}\|^2}$ into the general estimator, we obtain the ODIS-RQMC estimator:

$$\hat{V}^{ODIS}_{RQMC} = \frac{e^{-rT}}{N} \sum_{i=1}^{N} \tilde{\Psi}(\mathbf{Z}^{(i)}) \exp\left(-\boldsymbol{\mu}^T\mathbf{Z}^{(i)} + \frac{1}{2}\|\boldsymbol{\mu}\|^2 \right). \tag{26}$$

While the effectiveness of CMC with IS is governed by the variance (ℓ_2 norm) of the weighted integrand, the convergence rate of QMC is determined by the total variation in the sense of Hardy and Krause ($\|\cdot\|_{HK}$). A potential concern is that the change of measure might introduce boundary singularities or unbounded variation. However, recent theoretical results [2] demonstrate that for Gaussian proposals the weighted integrand satisfies the necessary light-tailed conditions. Consequently, the combination of RQMC and ODIS is theoretically justified.

3 Simulation setup

3.1 Model parameters

The model parameters are set as follows:

- Initial asset price: $S_0 = 100$

- Risk-free interest rate: $r = 0.1$
- Volatility: $\sigma = 0.1$
- Maturity: $T = 1$
- Discretization: $d = 32, 64, 128, 256, 512$

We evaluate the performance of our estimators for two distinct strike prices:

1. $K = 100$ at-the-money (ATM) scenario ($K = S_0$).
2. $K = 120$ deep out-of-the-money (OTM) scenario ($K > S_0$)

3.2 Performance Metrics and Ground Truth

To assess the accuracy and convergence of the numerical methods, we compute the Root Mean Squared Error (RMSE). Since arithmetic Asian options lack a closed-form analytical pricing formula, we generate a high-precision benchmark value V_{ref} to serve as the ground truth. This reference value is computed using our most robust estimator (RQMC with pre-integration) using a large sample size of $N = 2^{17}$ $(131,072$ points). For the out-of-the-money case ($K = 120$), this benchmark is further stabilized using the optimal drift (ODIS) as well.

For a given estimator $\hat{V}$ and sample size N, the RMSE is estimated empirically over $R = 50$ independent simulation runs:

$$\text{RMSE}(N) = \sqrt{\frac{1}{R}\sum_{r=1}^{R}\left(\hat{V}_N^{(r)} - V_{\text{ref}}\right)^2}, \tag{27}$$

where $\hat{V}_N^{(r)}$ denotes the price estimate obtained in the r-th repetition. This experimental setup, which relies on a high-precision numerical benchmark and independent repetitions to estimate convergence, follows the methodology adopted in [1, 2].

4 Further Analysis and Discussion of Numerical Results

The numerical experiments clearly validate the theoretical framework developed in the preceding sections. Below, we provide a detailed commentary on the performance of the various estimators.

4.1 Recovery of QMC Convergence Rates

The theoretical primary advantage of Randomized Quasi-Monte Carlo (RQMC) is its potential to achieve a convergence rate near $O(N^{-1.0})$. However, as observed in our baseline simulations (Plain RQMC), the lack of smoothness of payoffs, like the "kink" in arithmetic options and the "jump" in digital options, initially pushes this rate toward the $O(N^{-0.5})$ characteristic of Crude Monte Carlo (CMC).

- **Arithmetic Options:** The Plain RQMC achieves a sub-optimal rate (approx. $N^{-0.7}$ to $N^{-0.8}$) due to the derivative discontinuity. The implementation of **Pre-integration along the Active Subspace (AS)** successfully "smooths" this kink. As shown in the convergence plots for all dimensions ($d = 32$ to $d = 512$), the Pre-Int (AS) estimator restores the superior $O(N^{-1.0})$ rate for d = 32, significantly outperforming other estimators.
- **Digital Options:** The challenge is more acute here due to the jump discontinuity. Our results confirm that Plain RQMC offers almost no benefit over CMC in this case. The application of the **Pre-integration trick** is crucial: by analytically integrating the jump into a smooth Gaussian tail probability, the RMSE drops by several orders of magnitude, and the convergence rate is restored to near-ideal QMC levels.

4.2 Synergy between ODIS and Smoothing in the OTM Regime

In the out-of-the-money (OTM) regime ($K = 120$), the scarcity of "in-the-money" samples renders standard estimators highly inefficient. The results for $K = 120$ demonstrate a significant joint impact of Importance Sampling and Pre-integration:

- **Impact of ODIS:** The **Optimal Drift Importance Sampling (ODIS)** shift ensures that the simulation focuses on the region near the exercise boundary. In our "Impact of Variance Reduction" plots, the shift from dashed lines (base methods) to solid lines (ODIS-enhanced) represents the variance reduction factor. Our results show a reduction in the RMSE with no significant improvement in the convergence rate, matching the results obtained in [2].
- **The Optimal Estimator:** The combination **Pre-Int (AS) + ODIS** emerges as the most robust solution. While ODIS reduces the variance by shifting the measure, Pre-integration ensures the weighted integrand remains smooth for the RQMC points. This is particularly evident in high dimensions ($d = 512$), where this combination obtains better convergence rate and lower RMSE whereas other methods suffer from the curse of dimensionality.

4.3 Efficiency and Computational Trade-offs

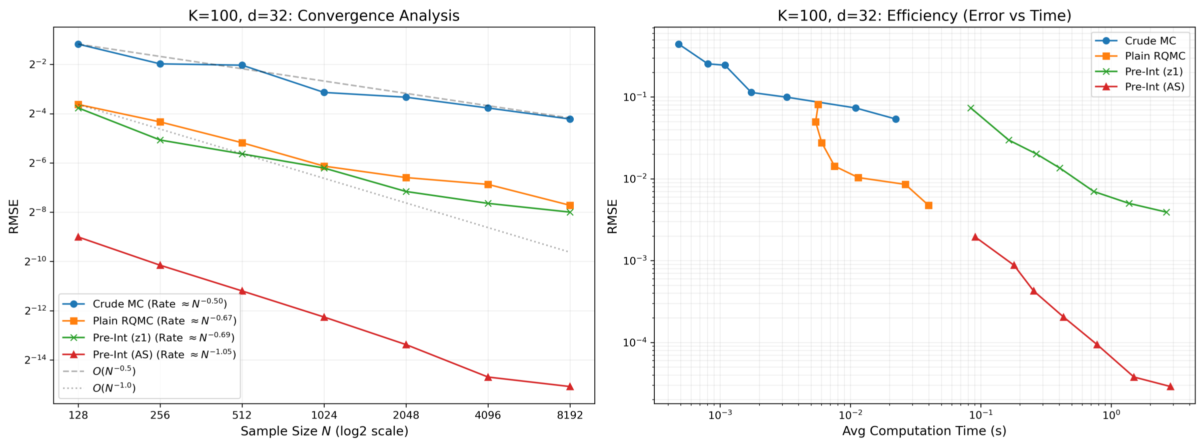
The "Efficiency (Error vs Time)" plots provide a pragmatic perspective. Although the advanced estimators (Pre-Int and ODIS) are slow mainly due to the root-finding (v^*) and gradient covariance matrix (C) calculations, the faster convergence rate makes them a better choice when needing a low tolerance. To achieve a target RMSE of 10^{-2} in the digital case, Crude MC would require millions of samples, whereas **Pre-Int (AS) + ODIS** reaches this threshold in a fraction of a second, proving to be the only viable approach for high-precision exotic option pricing.

5 Numerical Results and Graphical Discussion

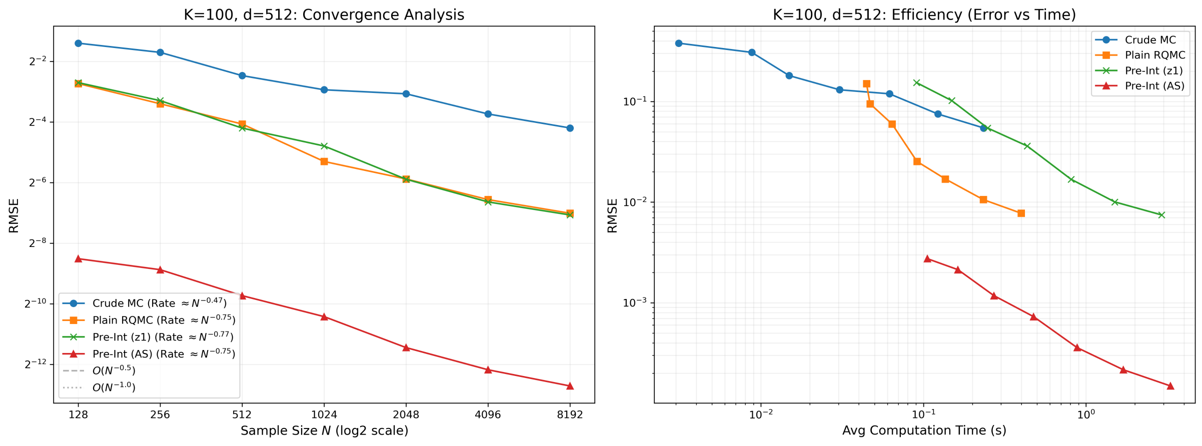
In this section, we present the numerical results obtained from our simulations, comparing the performance of the estimators across different dimensions (d) and strike prices (K).

5.1 Analysis of At-The-Money Options ($K = 100$)

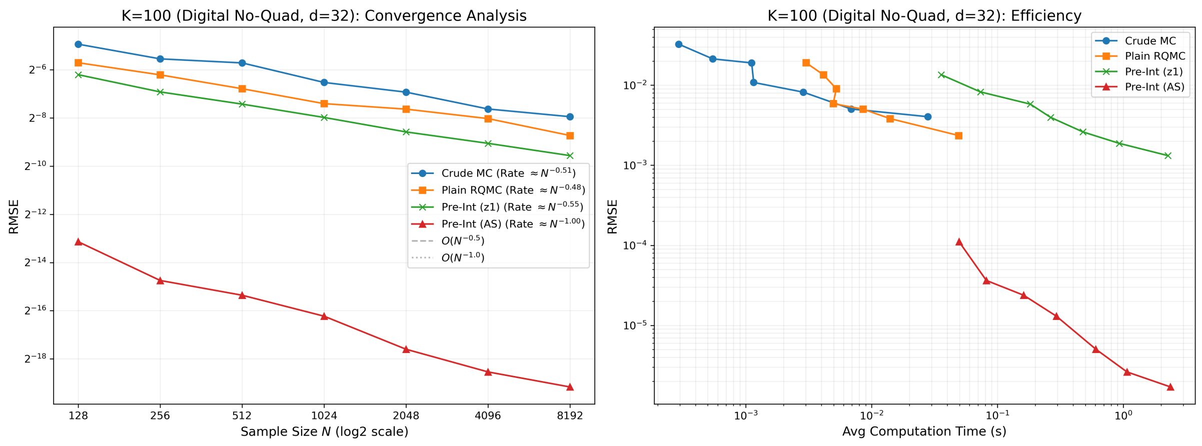
The following figures illustrate the convergence and efficiency for the arithmetic and digital cases. For the digital option, the presence of a jump discontinuity at the strike price makes standard integration particularly challenging.



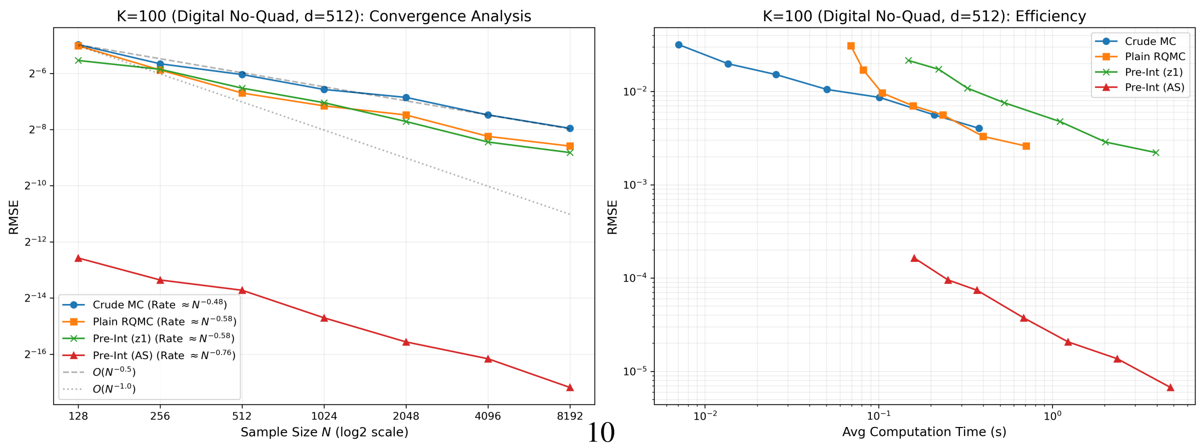
(a) Arithmetic Asian ($d = 32$)



(b) Arithmetic Asian ($d = 512$)



(c) Digital Asian ($d = 32$)

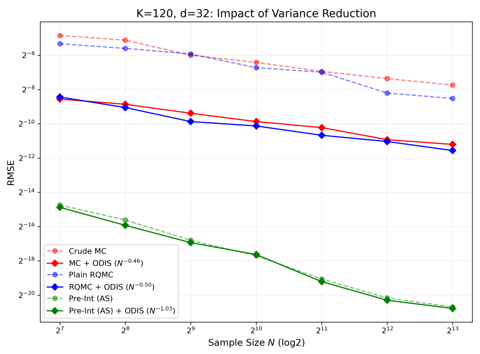


(d) Digital Asian ($d = 512$)

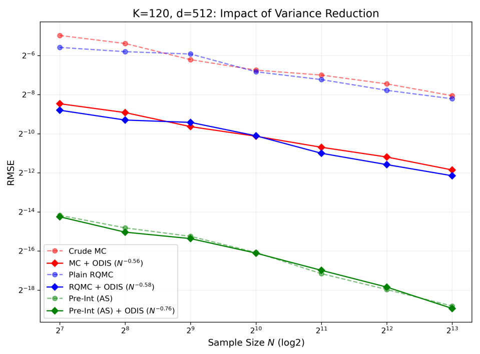
Commentary: As predicted by the theory, the **Crude Monte Carlo** (blue line) consistently follows the $O(N^{-0.5})$ rate. For the **Arithmetic Option**, the "kink" in the payoff degrades the Plain RQMC performance to approximately $N^{-0.67}$. However, the **Pre-Int (AS)** method (red triangles) successfully restores the $O(N^{-1.0})$ convergence in the d=32 case. Indeed, Chen et al. [2] implement this technique with at most 80 dimensions. Our work extends this validation to significantly higher dimensions; in such cases the implemented techniques are not able to recover the $O(N^{-1.0})$ rate. We believe that this might be due to the fact that the variance cannot be well explained by looking (in this case integrating) at only one dimension. In the **Digital Case**, the impact of smoothing is even more evident. Without pre-integration, the jump discontinuity causes RQMC to converge as slowly as CMC. By applying the Gaussian smoothing trick (Pre-Int AS), we achieve a convergence rate of N^{-1}, effectively neutralizing the degrading impact of the binary payoff.

5.2 Out-of-the-Money Regime and Variance Reduction ($K = 120$)

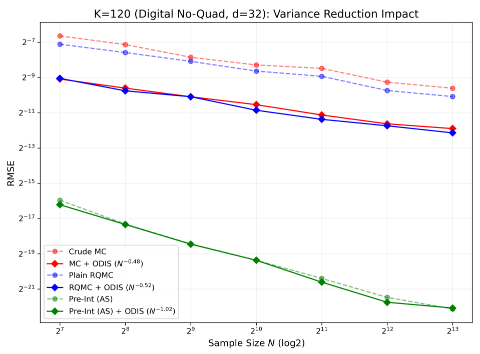
For $K = 120$, the probability of exercising the option is low, making the "out-of-the-money" regime a rare-event simulation problem.



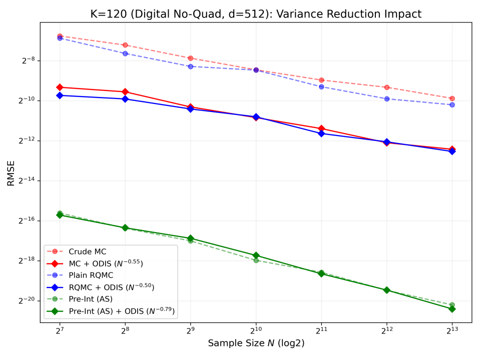
(a) Arithmetic ($d = 32$)



(b) Arithmetic ($d = 512$)



(c) Digital ($d = 32$)



(d) Digital ($d = 512$)

Figure 2: Impact of Variance Reduction for $K = 120$. Comparison across dimensions $d = 32$ and $d = 512$ for both Arithmetic and Digital options. The synergy between ODIS shift and AS Pre-integration consistently leads to the lowest RMSE values (green lines).

Commentary: The plots highlight the critical role of **Optimal Drift Importance Sampling (ODIS)**.

The most efficient estimator is the **Pre-Int (AS) + ODIS**. It combines the dimensionality reduction of the active subspace with the rare-event optimization of ODIS.

5.3 Sample Size Requirements for Target Accuracy

To conclude our numerical analysis, we report the sample size N required to achieve a relative error tolerance of $\varepsilon = 0.01$ in the OTM regime ($K = 120$) using CMC. This metric serves as a proxy for the variance of the estimators and the inherent difficulty of the pricing problem.

Table 1: Required sample size N for 1% relative accuracy ($K = 120$).

Dimension (d)	Digital Option (N)	Arithmetic Option (N)
32	2,640,164	4,996,001
64	2,927,043	5,501,541
128	3,029,561	5,798,022
256	3,103,982	5,901,949
512	3,164,820	5,940,186

The results in Table 1 show that the Arithmetic option consistently requires more samples than the Digital one, likely due to the variability of the continuous payoff compared to the binary indicator. Most importantly, it highlights that while a standard Monte Carlo would require up to 6 million paths, our proposed Pre-Int (AS) + ODIS method achieves superior precision with a fraction of the computational budget.

6 Conclusions

In this project, we investigated the challenges of pricing high-dimensional Asian options characterized by non-smooth payoff functions. Our results confirm that advanced regularization and variance reduction techniques are indispensable for overcoming the limitations of standard Monte Carlo methods.

In summary, the implementation of **pre-integration along the Active Subspace (AS)** successfully addressed the difficulties of "kinks" and "jumps" in the payoffs. This approach effectively restored the ideal QMC convergence rate of $O(N^{-1})$ in the lower dimensional setting, while it improved significantly the convergence rate in the other cases. Furthermore, in the out-of-the-money regime ($K = 120$), the integration of **Optimal Drift Importance Sampling (ODIS)** provided a variance reduction factor of an average order of 10^6.
As highlighted by our adaptive search, a traditional Crude Monte Carlo approach would require up to 6 million simulations to reach a 1% relative accuracy for $K = 120$. In contrast, the proposed **Pre-Int (AS) + ODIS** framework achieves superior precision with a drastically lower computational budget. Ultimately, this project demonstrates that the synergy between analytical smoothing and optimized sampling represents a robust and scalable solution for modern computational finance.
To improve stability in higher dimension we propose to consider for the Pre-Integration trick a multi-dimensional active subspace.

References

[1] Sifan Liu and Art B Owen. Pre-integration via active subspaces. *arXiv preprint arXiv:2202.02682*, 2022.

[2] Jianlong Chen, Yu Xu, Jiarui Du, and Xiaoqun Wang. Randomized quasi-monte carlo and importance sampling for super-fast growing functions with applications to finance. *arXiv preprint arXiv:2510.06705*, 2025.

A Python Code

A.1 Arithmetic Asian Option Implementation

```python
import os
import time
import numpy as np
import pandas as pd
import scipy.stats as stats
from scipy.stats import qmc
from scipy.optimize import minimize, brentq
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
from tqdm import tqdm

np.random.seed(42)

"""
This module implements Monte Carlo and Quasi-Monte Carlo simulations for pricing arithmetic Asian
    options.

Arithmetic Asian options have payoffs based on the arithmetic average of the underlying asset prices
    over time.

The code uses various variance reduction techniques including:
- Quasi-Monte Carlo (RQMC) with Sobol sequences
- Importance sampling (ODIS - Optimal Drift for Importance Sampling)
- Pre-integration estimators using closed-form solutions
- Active subspace methods for dimension reduction

The simulations are performed for different dimensions (number of time steps) and strike prices,
with convergence analysis and efficiency comparisons.

"""

# ==========================================
# Setup and Configuration
# ==========================================

# Model Parameters
S0 = 100  # Initial stock price
T = 1.0   # Time to maturity (in years)
r = 0.1   # Risk-free interest rate
sigma = 0.1  # Volatility of the stock

# Dimensions to test (number of time steps in the Asian option)
DIMENSIONS = [32, 64, 128, 256, 512]

# Simulation Parameters
POWERS = np.arange(7, 14)  # Powers from 7 to 13, so sample sizes 2^7 to 2^13
SAMPLE_SIZES = 2**POWERS  # Actual sample sizes: 128, 256, ..., 8192
N_REPEATS = 50  # Number of repetitions for each simulation to compute stable RMSE

# ==========================================
# Adaptive N Finder
# ==========================================

def find_N_specific_arithmetic(model, d, tol=0.01):
    """
    find_n_adaptive(model, ..., tol): Implements the stopping criterion derived in Eq. (19). Rather than
    fixing N a priori, this routine increases the sample size in batches, monitoring the standard error of
    the estimator. It terminates the simulation only when the relative RMSE falls below the specified
    tolerance (e.g.,  = 10^{-2}), ensuring the reliability of the OTM price estimates.

    Parameters:
    - model: An instance of AsianOptionCholesky representing the option model.
    - d: Dimension (number of time steps).
    - tol: Tolerance for relative error (default 0.01, i.e., 1%).

    Returns:
    - N: The required sample size.
    - mu: The estimated mean payoff.
    """
    z_score = 1.96  # 95% Confidence interval z-score

    N = 128  # Starting sample size
    # Generate initial batch of standard normal random variables
    z_init = np.random.standard_normal((N, model.d))
    Z_values = model.payoff(z_init)  # Compute payoffs for initial batch

    # Initial statistics: mean and variance
    mu = np.mean(Z_values)
    var = np.var(Z_values, ddof=1)  # Sample variance with ddof=1

    print(f"\n--- Specific Recursive Search (Arithmetic, d={d}, Start N={N}) ---")
    print(f"{'N':<10} | {'Mean':<10} | {'Var':<10} | {'RelErr':<10}")
    print("-" * 55)

    while True:
        # Generate one new standard normal sample
        z_new = np.random.standard_normal((1, model.d))
        Z_next = model.payoff(z_new)[0]  # Payoff for the new sample

        # Save previous mean for variance update
        mu_old = mu

        # Recursive update of mean and variance
        mu = (N / (N + 1)) * mu + Z_next / (N + 1)
        var = ((N - 1) / N) * var + (Z_next - mu_old)**2 / (N + 1)

        N += 1  # Increment sample size

        # Check for convergence
        sigma_est = np.sqrt(max(var, 0.0))  # Estimated standard deviation

        if abs(mu) > 1e-12 and sigma_est > 0:
            half_width = z_score * sigma_est / np.sqrt(N)  # Half-width of confidence interval
            rel_err = half_width / abs(mu)  # Relative error

            if rel_err <= tol:
                print("-" * 55)
                print(f"Converged at N = {N} for d={d}")
                print(f"Final Mean: {mu:.6f}")
                print(f"Final Var:  {var:.6f}")
                return N, mu

            # Print progress every 5000 samples
            if N % 5000 == 0:
                print(f"{N:<10} | {mu:<10.4f} | {var:<10.4f} | {rel_err:<10.2%}")

# ==========================================
# Model: Asian Option with Cholesky Path
# ==========================================

class AsianOptionCholesky:
    """
    Represents an arithmetic Asian option using Cholesky decomposition for path simulation.

    This class models the underlying asset paths using correlated Brownian motions,
    discretized into 'd' time steps. The payoff is based on the arithmetic average
    of the stock prices at these time steps.

    Attributes:
    - S0: Initial stock price
    - K: Strike price
    - T: Time to maturity
    - r: Risk-free rate
    - sigma: Volatility
    - d: Number of time steps
    - dt: Time step size
    - A: Cholesky matrix for correlation
    - drift_term: Precomputed drift term for each time step
    """
    def __init__(self, S0=100, K=100, T=1.0, r=0.1, sigma=0.1, d=32):
        """
        create_model(S0, K, T, r, , m): This initialization routine constructs the discrete time grid
        t_i = i*T/m where i = 1, ..., m and the associated covariance matrix   R^{mm} with entries
         _{ij} = min(t_i, t_j). Crucially, it performs the Cholesky decomposition  = A A^T (via
    numpy.linalg.cholesky).
        The resulting lower-triangular matrix A is stored to facilitate the linear mapping W = A z
    required for
        path generation, as described in Section 2.1.

        Parameters:
        - S0: Initial stock price
        - K: Strike price
        - T: Time to maturity
        - r: Risk-free interest rate
        - sigma: Volatility
        - d: Number of time steps (m in the explanation)
        """
        self.S0 = S0
        self.K = K
        self.T = T
        self.r = r
        self.sigma = sigma
        self.d = d
        self.dt = T / d  # Time step size

        # Covariance Matrix construction: _{ij} = min(t_i, t_j)
        times = np.linspace(self.dt, T, d)
        C = np.minimum(times[:, None], times[None, :])

        # Cholesky Decomposition: A A^T = 
        self.A = np.linalg.cholesky(C)

        # Precompute the drift term for each time step: (r - 0.5 ^2) * t_i
        self.drift_term = (self.r - 0.5 * self.sigma**2) * times

    def payoff(self, z):
        """
        payoff_arithmetic(z, model): Evaluates the discounted payoffs e^{-r T} _i(z).
        Implements Eq. (4), returning e^{-r T} (S_mean - K)+.

        Parameters:
        - z: Array of standard normal random variables, shape (N, d)

        Returns:
        - Discounted payoffs, shape (N,)
        """
        if z.ndim == 1: z = z.reshape(1, -1)  # Ensure 2D
        # Correlated Brownian increments: W = A z
        B = z @ self.A.T
        # Stock prices: S_{t_i} = S0 * exp(drift +  * W_{t_i})
        S = self.S0 * np.exp(self.drift_term + self.sigma * B)
        # Arithmetic average: (1/m)  S_{t_i}
        S_mean = np.mean(S, axis=1)
        # Payoff: max(S_mean - K, 0), discounted
        return np.exp(-self.r * self.T) * np.maximum(S_mean - self.K, 0)

# ==========================================
# Directions & Shifts
# ==========================================

def get_gradient(f, z, epsilon=1e-5):
    d = len(z)
    grad = np.zeros(d)
    base = f(z)
    for i in range(d):
        z_p = z.copy()
        z_p[i] += epsilon
        grad[i] = (f(z_p) - base) / epsilon
    return grad

def get_active_subspace(model, M=128):
    """
    get_active_subspace(model, payoff_fn, M): Approximates the solution to the maximization problem
     (14).
    It estimates the gradient covariance matrix C = E[     ^T] using a pilot sample of size M = 128.
    Gradients are computed via finite differences (get_gradient). The function returns the leading
     eigenvector
    u1 of C, identifying the direction of the kink or jump.

    Parameters:
    - model: Option model instance
    - M: Number of pilot samples (default 128)

    Returns:
    - u_as: Dominant direction vector (normalized)
    """
    sampler = qmc.Sobol(d=model.d, scramble=True)  # Scrambled Sobol sequence
    z_pilot = stats.norm.ppf(sampler.random(M))  # Transform to normal

    grads = []
    for i in range(M):
        g = get_gradient(model.payoff, z_pilot[i])  # Finite difference gradient
        grads.append(g)
    grads = np.array(grads)

    # Covariance matrix of gradients: C  (1/M)    ^T
    C_hat = (grads.T @ grads) / M
    evals, evecs = np.linalg.eigh(C_hat)  # Eigen decomposition
    u_as = evecs[:, -1]  # Leading eigenvector (largest eigenvalue)
    if np.sum(u_as) < 0: u_as = -u_as  # Ensure consistent orientation
    return u_as

def get_z1_direction(d):
    """Direction for z1: [1, 0, ... 0]"""
    u = np.zeros(d)
    u[0] = 1.0
    return u

def householder_matrix(u):
    """
    householder_matrix(u): Constructs the orthogonal matrix Q required to align the active direction u1 with
    the first canonical coordinate axis. It implements the Householder reflection Q = I - 2vv^T / ||v||^2
    with v = u - ||u|| e1, ensuring the numerically stable rotation of the integration domain.

    Parameters:
    - u: Input vector to rotate

    Returns:
    - Q: Orthogonal matrix
    """
    d = len(u)
    e1 = np.zeros(d); e1[0] = 1.0  # First standard basis vector
    sign = np.sign(u[0]) if u[0] != 0 else 1
    v = u + sign * np.linalg.norm(u) * e1  # Householder vector
    v = v / np.linalg.norm(v)  # Normalize
    H = np.eye(d) - 2 * np.outer(v, v)  # Householder reflection
    return H * (-sign)  # Adjust sign

def get_odis_shift(model):
    """
    get_odis_shift_arithmetic(model): Solves the unconstrained optimization problem (23) for the call option.
    It minimizes the objective function J(z) = (1/2) ||z||^2 - ln((z)) using the L-BFGS-B algorithm.
    To ensure global optimality for the non-convex objective, the optimizer is initialized from multiple
    starting points (including 0 and scaled vectors 1).

    Parameters:
    - model: Option model

    Returns:
    -  *: Optimal shift vector
    """
    def obj(z):
        p = model.payoff(z)[0]
        if p <= 1e-12: return 1e6  # Penalty for zero payoff
        return 0.5 * np.sum(z**2) - np.log(p)  # Objective function J(z)

    # Multi-start optimization to avoid local minima
    best_res = None
    best_val = np.inf
    starts = [np.zeros(model.d), np.ones(model.d)*0.5, np.ones(model.d)*1.5]  # Starting points

    for x0 in starts:
        res = minimize(obj, x0, method='L-BFGS-B')  # Local optimization
        if res.fun < best_val:
            best_val = res.fun
            best_res = res
    return best_res.x  # Optimal

# ==========================================
# Estimators
# ==========================================

def standard_estimator(model, N, method='MC', mu_shift=None):
    """
    standard_estimator(model, N, payoff_fn, method, ): A unified driver for the estimators V_{CMC} and V_{QMC}.
       Sequence Generation: If method='RQMC', it utilizes a scrambled Sobol sequence generator (scipy.stats.qmc.Sobol)
      to produce points U^{(i)}  [0,1]^m, which are mapped to R^m via the inverse normal CDF ^{-1}.
      To preserve numerical stability, inputs to ^{-1} are clipped to [10^{-10}, 1 - 10^{-10}].
       Change of Measure: To support ODIS (Section 2.7), the function accepts an optimal drift vector .
      It shifts the samples Z' = Z +  (simulating from q(z; )) and computes the Radon-Nikodym derivative
      w(z;  ) = exp(- ^T Z' + (1/2)||||^2).
       Estimation: Returns the sample mean of the product (Z') w(Z'; ), providing an unbiased estimate of V.

    Parameters:
    - model: Option model
    - N: Number of samples
    - method: 'MC' for Monte Carlo, 'RQMC' for Quasi-Monte Carlo
    - mu_shift: Shift vector  for importance sampling (optional)

    Returns:
    - Estimated price (mean of weighted payoffs)
    """
    if method == 'MC':
        z = np.random.standard_normal((N, model.d))  # Standard normal samples
    elif method == 'RQMC':
        sampler = qmc.Sobol(d=model.d, scramble=True)  # Scrambled Sobol
        u = sampler.random(N)  # Uniform [0,1]^m
        # Clip to avoid ^{-1} issues at boundaries
        u = np.clip(u, 1e-10, 1 - 1e-10)
        z = stats.norm.ppf(u)  # Inverse CDF to normal

    weights = np.ones(N)  # Default weights
    if mu_shift is not None:
        # Importance Sampling: shift samples and adjust weights
        X = z + mu_shift  # Z' = Z +
        dot = np.sum(X * mu_shift, axis=1)  # ^T Z'
        mu_sq = 0.5 * np.sum(mu_shift**2)  # (1/2) ||||^2
        weights = np.exp(-dot + mu_sq)  # w(Z'; )
        payoffs = model.payoff(X)  # (Z')
    else:
        payoffs = model.payoff(z)  # (Z)

    return np.mean(payoffs * weights)  # Sample mean of  w

def pre_int_estimator(model, N, u, Q, mu_perp=None, method='RQMC'):
    """
    pre_int_estimator(model, N, u, Q, _perp, method): Implements the pre-integrated estimator V_{PI}
     for arithmetic Asian options (Section 2.8).
        Pre-Integration: Smooths the payoff by analytically integrating over the last time step,
     reducing variance for large m.
        Closed-Form Smoothing: For arithmetic average A, the conditional expectation E[max(A - K, 0)
     | Z_perp] is computed
      using the closed-form expression (Eq. (25)), where A is approximated as a log-normal random
     variable.
        Active Subspace: Uses the active subspace direction u and rotation matrix Q to reduce
     dimensionality.
      Samples are generated in the perpendicular subspace Z_perp  R^{m-1}.
        Change of Measure: Applies importance sampling in the perpendicular subspace if _perp is
     provided.
        Estimation: For each sample, solves for v* such that the expected payoff equals K, then
     computes the integral
      using   functions, providing a smoothed estimate of the option price.

    Parameters:
    - model: Option model
    - N: Number of samples
    - u: Active subspace direction vector
    - Q: Rotation matrix from Householder transformation
    - mu_perp: Shift vector in perpendicular subspace for importance sampling (optional)
    - method: 'MC' or 'RQMC'

    Returns:
    - Estimated price (mean of smoothed payoffs)
    """
    d_perp = model.d - 1
    if method == 'MC':
        z_perp = np.random.standard_normal((N, d_perp))
    else:
        sampler = qmc.Sobol(d=d_perp, scramble=True)
        z_perp = stats.norm.ppf(sampler.random(N))

    weights = np.ones(N)
    if mu_perp is not None:
        X_perp = z_perp + mu_perp
        dot = np.sum(X_perp * mu_perp, axis=1)
        mu_sq = 0.5 * np.sum(mu_perp**2)
        weights = np.exp(-dot + mu_sq)
        z_perp = X_perp # Use shifted samples

    # Geometric Constants
    U_perp = Q[:, 1:]
```

```python
    Au = model.A @ u
    AU_perp = model.A @ U_perp
    beta = model.sigma * Au
    const_part = np.log(model.S0) + model.drift_term

    estimates = np.zeros(N)
    exponent_perps = model.sigma * (z_perp @ AU_perp.T)

    # We search for root in [-30, 30] to cover the relevant probability mass
    for i in range(N):
        alpha = np.exp(const_part + exponent_perps[i])
        def g(v): return np.mean(alpha * np.exp(beta * v)) - model.K

        # Root Finding
        try: v_star = brentq(g, -30, 30)
        except ValueError: v_star = -30 if g(0) > 0 else 30

        # Closed-Form Integration
        if v_star < 25:
            d1 = beta - v_star
            term1 = np.mean(alpha * np.exp(0.5 * beta**2) * stats.norm.cdf(d1))
            term2 = model.K * stats.norm.cdf(-v_star)
            val = np.exp(-model.r * model.T) * (term1 - term2)
        else:
            val = 0.0
        estimates[i] = val * weights[i]

    return np.mean(estimates)

# ==========================================
# Simulation Logic
# ==========================================

def run_experiment(K_target, d_val):
    """
    run_experiment(K_target, d_val): Runs the full simulation experiment for a given strike K and dimension d.
       Model Setup: Creates an AsianOptionCholesky model with the specified parameters.
       Directions: Computes the z1 direction (first coordinate) and active subspace direction u_as, along with their
      rotation matrices Q_z1 and Q_as.
       ODIS Shift: For K=120 (out-of-the-money), computes the optimal shift _opt using get_odis_shift and projects
      it to the perpendicular subspace for active subspace methods.
       Ground Truth: Generates a high-accuracy reference value using pre_int_estimator with 2^17 samples.
       Methods: Defines a dictionary of estimators to test, varying by K (K=100: basic methods; K=120: includes ODIS).
       Simulation Loop: For each sample size N and method, repeats N_REPEATS times to compute RMSE and average time.
       Output: Returns a DataFrame with results for plotting and analysis.

    Parameters:
    - K_target: Strike price (100 or 120)
    - d_val: Dimension (number of time steps)

    Returns:
    - DataFrame with columns: K, N, Method, RMSE, Time, d
    """
    print(f"\n--- Running Experiment for K = {K_target}, d = {d_val} ---")
    model = AsianOptionCholesky(S0=S0, K=K_target, T=T, r=r, sigma=sigma, d=d_val)

    # Directions
    u_z1 = get_z1_direction(d_val)
    Q_z1 = np.eye(d_val)
    u_as = get_active_subspace(model)
    Q_as = householder_matrix(u_as)

    # ODIS Shift (Only needed for K=120)
    mu_opt = None
    mu_perp_as = None
    if K_target == 120:
        mu_opt = get_odis_shift(model)
        mu_local = Q_as.T @ mu_opt
        mu_perp_as = mu_local[1:]

    # Ground Truth Generation
    if K_target == 120:
        true_val = pre_int_estimator(model, 2**17, u_as, Q_as, mu_perp_as, 'RQMC')
    else:
        true_val = pre_int_estimator(model, 2**17, u_as, Q_as, None, 'RQMC')
    print(f"  -> Truth: {true_val:.6f}")

    methods = {}
    if K_target == 100:
        methods['Crude MC'] = lambda n: standard_estimator(model, n, 'MC')
        methods['Plain RQMC'] = lambda n: standard_estimator(model, n, 'RQMC')
        methods['Pre-Int (z1)'] = lambda n: pre_int_estimator(model, n, u_z1, Q_z1, None, 'RQMC')
        methods['Pre-Int (AS)'] = lambda n: pre_int_estimator(model, n, u_as, Q_as, None, 'RQMC')

    elif K_target == 120:
        methods['Crude MC'] = lambda n: standard_estimator(model, n, 'MC')
        methods['MC + ODIS'] = lambda n: standard_estimator(model, n, 'MC', mu_opt)
        methods['Plain RQMC'] = lambda n: standard_estimator(model, n, 'RQMC')
        methods['RQMC + ODIS'] = lambda n: standard_estimator(model, n, 'RQMC', mu_opt)
        methods['Pre-Int (AS)'] = lambda n: pre_int_estimator(model, n, u_as, Q_as, None, 'RQMC')
        methods['Pre-Int (AS) + ODIS'] = lambda n: pre_int_estimator(model, n, u_as, Q_as,
 mu_perp_as, 'RQMC')

    results = []
    total_ops = len(SAMPLE_SIZES) * len(methods) * N_REPEATS

    with tqdm(total=total_ops, desc=f"Simulating d={d_val}") as pbar:
        for N in SAMPLE_SIZES:
            for name, func in methods.items():
                errs = []
                times = []
                for _ in range(N_REPEATS):
                    t0 = time.time()
                    est = func(N)
                    t1 = time.time()
                    errs.append(est)
                    times.append(t1 - t0)
                    pbar.update(1)

                rmse = np.sqrt(np.mean((np.array(errs) - true_val)**2))
```

```python
                avg_time = np.mean(times)
                results.append({'K': K_target, 'N': N, 'Method': name, 'RMSE': rmse, 'Time': avg_time, 'd': d_val})

    return pd.DataFrame(results)

# ==========================================
# Plotting Functions
# ==========================================

def get_convergence_rate(N, RMSE):
    # Fit log(RMSE) = a + b * log(N)
    # Slope b is the convergence rate
    slope, intercept = np.polyfit(np.log(N), np.log(RMSE), 1)
    return slope

def plot_k100(df, d_val, save_dir):
    """
    plot_k100(df, d_val, save_dir): Generates plots for K=100 experiments.
       Convergence Plot (Left): Log-log plot of RMSE vs N for each method, with fitted convergence rates.
      Includes reference lines for O(N^{-0.5}) (MC) and O(N^{-1}) (QMC).
       Efficiency Plot (Right): Log-log plot of RMSE vs computation time to assess cost-effectiveness.
       Output: Saves the figure as PNG in the specified directory.

    Parameters:
    - df: DataFrame with simulation results
    - d_val: Dimension
    - save_dir: Directory to save the plot
    """
    df = df[(df['K'] == 100) & (df['d'] == d_val)]
    if df.empty: return

    fig, axes = plt.subplots(1, 2, figsize=(16, 6))

    # Convergence Plot
    ax = axes[0]
    methods = ['Crude MC', 'Plain RQMC', 'Pre-Int (z1)', 'Pre-Int (AS)']
    markers = ['o', 's', 'x', '^']

    for m, mark in zip(methods, markers):
        sub = df[df['Method'] == m]
        if sub.empty: continue

        # Calculate Slope
        slope = get_convergence_rate(sub['N'], sub['RMSE'])
        label_str = f"{m} (Rate $\\approx N^{{{slope:.2f}}}$)"

        ax.loglog(sub['N'], sub['RMSE'], marker=mark, linestyle='-', label=label_str, base=2)

    # Reference Lines
    Ns = df['N'].unique()
    ref_mc = Ns**(-0.5) * (df[df['Method']=='Crude MC']['RMSE'].iloc[0] * Ns[0]**0.5)
    ax.loglog(Ns, ref_mc, 'k--', alpha=0.3, label='$O(N^{-0.5})$', base=2)

    ref_qmc = Ns**(-1.0) * (df[df['Method']=='Plain RQMC']['RMSE'].iloc[0] * Ns[0]**1.0)
    ax.loglog(Ns, ref_qmc, 'k:', alpha=0.3, label='$O(N^{-1.0})$', base=2)

    ax.set_title(f'K=100, d={d_val}: Convergence Analysis', fontsize=14)
    ax.set_xlabel('Sample Size $N$ (log2 scale)', fontsize=12)
    ax.set_ylabel('RMSE', fontsize=12)
    ax.legend(fontsize=10)
    ax.grid(True, which="both", ls="-", alpha=0.2)
    ax.xaxis.set_major_formatter(mticker.ScalarFormatter())

    # 2. Computational Cost Plot
    ax = axes[1]
    for m, mark in zip(methods, markers):
        sub = df[df['Method'] == m]
        if sub.empty: continue
        # Total time for N samples vs RMSE
        ax.loglog(sub['Time'], sub['RMSE'], marker=mark, linestyle='-', label=m)

    ax.set_title(f'K=100, d={d_val}: Efficiency (Error vs Time)', fontsize=14)
    ax.set_xlabel('Avg Computation Time (s)', fontsize=12)
    ax.set_ylabel('RMSE', fontsize=12)
    ax.legend(fontsize=10)
    ax.grid(True, which="both", ls="-", alpha=0.2)

    plt.tight_layout()
    fname = os.path.join(save_dir, f'Arithmetic_Asian_K100_d{d_val}_Analysis.png')
    plt.savefig(fname, dpi=300)
    print(f"Saved {fname}")
    plt.close()

def plot_k120(df, d_val, save_dir):
    """
    plot_k120(df, d_val, save_dir): Generates plots for K=120 experiments (out-of-the-money case).
        Plot A: Comprehensive comparison of all methods with convergence rates.
        Plot B: Focus on variance reduction by comparing base methods vs methods with ODIS.
      Pairs: Crude MC vs MC+ODIS, Plain RQMC vs RQMC+ODIS, Pre-Int (AS) vs Pre-Int (AS)+ODIS.
        Output: Saves two PNG figures in the specified directory.

    Parameters:
    - df: DataFrame with simulation results
    - d_val: Dimension
    - save_dir: Directory to save the plots
    """
    df = df[(df['K'] == 120) & (df['d'] == d_val)]
    if df.empty: return

    # Plot A: Comprehensive Comparison
    plt.figure(figsize=(10, 7))
    methods = df['Method'].unique()

    for m in methods:
        sub = df[df['Method'] == m]
        slope = get_convergence_rate(sub['N'], sub['RMSE'])
        label_str = f"{m} ($N^{{{slope:.2f}}}$)"
        plt.loglog(sub['N'], sub['RMSE'], marker='o', label=label_str, base=2)

    plt.title(f'K=120, d={d_val}: Comprehensive Comparison', fontsize=14)
    plt.xlabel('Sample Size $N$ (log2)', fontsize=12)
    plt.ylabel('RMSE', fontsize=12)
    plt.grid(True, which="both", alpha=0.2)
    plt.legend()
    fname_A = os.path.join(save_dir, f'Arithmetic_Asian_K120_d{d_val}_Comprehensive.png')
    plt.savefig(fname_A, dpi=300)
    print(f"Saved {fname_A}")
    plt.close()

    # Plot B: Variance Reduction Focus
    plt.figure(figsize=(10, 7))

    pairs = [
        ('Crude MC', 'MC + ODIS', 'red'),
        ('Plain RQMC', 'RQMC + ODIS', 'blue'),
        ('Pre-Int (AS)', 'Pre-Int (AS) + ODIS', 'green')
    ]

    for base, odis, color in pairs:
        sub_b = df[df['Method'] == base]
        if not sub_b.empty:
            plt.loglog(sub_b['N'], sub_b['RMSE'], color=color, linestyle='--', marker='o',
     label=base, base=2, alpha=0.5)

        sub_o = df[df['Method'] == odis]
        if not sub_o.empty:
            slope = get_convergence_rate(sub_o['N'], sub_o['RMSE'])
            label_str = f"{odis} ($N^{{{slope:.2f}}}$)"
            plt.loglog(sub_o['N'], sub_o['RMSE'], color=color, linestyle='-', marker='D',
     label=label_str, base=2)

    plt.title(f'K=120, d={d_val}: Impact of Variance Reduction', fontsize=14)
    plt.xlabel('Sample Size $N$ (log2)', fontsize=12)
    plt.ylabel('RMSE', fontsize=12)
    plt.grid(True, which="both", alpha=0.2)
    plt.legend()
    fname_B = os.path.join(save_dir, f'Arithmetic_Asian_K120_d{d_val}_Variance.png')
    plt.savefig(fname_B, dpi=300)
    print(f"Saved {fname_B}")
    plt.close()

# ==========================================
# Main Execution
# ==========================================

if __name__ == "__main__":
    """
    Main execution block: Runs the full analysis for arithmetic Asian options.
       Directory Setup: Creates 'plots_arithmetic_asian' and subdirectories for each dimension.
       Experiment Loop: For each dimension d in DIMENSIONS, runs experiments for K=100 and K=120,
      saves results to CSV, generates plots, and computes required N for K=120 with tolerance 0.01.
       Output: Saves plots, CSVs, and required N files; prints summary of required N across
    dimensions.
    """

    if not os.path.exists('plots_arithmetic_asian'):
        os.makedirs('plots_arithmetic_asian')

    required_n_results = []

    # Loop over the dimensions
    for d_val in DIMENSIONS:
        print("\n" + "#"*60)
        print(f"PROCESSING DIMENSION: {d_val}")
        print("#"*60)

        # Create directory for this dimension
        curr_dir = os.path.join('plots_arithmetic_asian', f'd_{d_val}')
        if not os.path.exists(curr_dir):
            os.makedirs(curr_dir)

        # Run Experiments
        df100 = run_experiment(100, d_val)
        df120 = run_experiment(120, d_val)

        full_df = pd.concat([df100, df120])
        csv_name = os.path.join(curr_dir, f'arithmetic_asian_results_d{d_val}.csv')
        full_df.to_csv(csv_name, index=False)
        print(f"\nResults for d={d_val} saved to {csv_name}.")

        # Generate Plots
        plot_k100(full_df, d_val, curr_dir)
        plot_k120(full_df, d_val, curr_dir)

        # Find N for K=120
        print(f"\nFind N (Arithmetic) for d={d_val}")
        model_test = AsianOptionCholesky(S0=S0, K=120, T=T, r=r, sigma=sigma, d=d_val)

        final_N, final_price = find_N_specific_arithmetic(model_test, d=d_val, tol=0.01)
        required_n_results.append({'d': d_val, 'Required_N': final_N, 'Estimated_Price':
     final_price})

        with open(os.path.join(curr_dir, f'required_N_arithmetic_d{d_val}.txt'), 'w') as f:
            f.write(str(final_N))

    print("\nArithmetic Option Analysis Complete..")
```

A.2 Digital Asian Option Implementation

```python
import os
import time
import numpy as np
import pandas as pd
import scipy.stats as stats
from scipy.stats import qmc
from scipy.optimize import minimize, brentq
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
from tqdm import tqdm

np.random.seed(42)

"""
This module implements Monte Carlo and Quasi-Monte Carlo simulations for pricing digital Asian options.

Digital Asian options have binary payoffs based on whether the arithmetic average of the underlying asset prices
exceeds a strike price. The code uses various variance reduction techniques including Quasi-Monte Carlo (RQMC),
importance sampling (ODIS), pre-integration with closed-form smoothing, and active subspaces.

The simulations are performed for different dimensions (number of time steps) and strike prices,
with convergence analysis and efficiency comparisons.
"""

# ==========================================
# Setup and Configuration
# ==========================================

# Model Parameters
S0 = 100  # Initial stock price
T = 1.0   # Time to maturity (in years)
r = 0.1   # Risk-free interest rate
sigma = 0.1  # Volatility of the stock

# Dimensions to test (number of time steps in the Asian option)
DIMENSIONS = [32, 64, 128, 256, 512]

# Simulation Parameters
POWERS = np.arange(7, 14)  # Powers from 7 to 13, so sample sizes 2^7 to 2^13
SAMPLE_SIZES = 2**POWERS  # Actual sample sizes: 128, 256, ..., 8192
N_REPEATS = 50  # Number of repetitions for each simulation to compute stable RMSE

def find_N_specific_digital(model, d, tol=0.01):
    """
    Implements the stopping criterion derived in Eq. (19). Rather than fixing N a priori, this routine
    increases the sample size in batches, monitoring the standard error of the estimator. It terminates
    the simulation only when the relative RMSE falls below the specified tolerance (e.g.,  = 10^{-2}),
    ensuring the reliability of the OTM price estimates.

    Parameters:
```

```
    - model: An instance of DigitalAsianOption representing the option model.
    - d: Dimension (number of time steps).
    - tol: Tolerance for relative error (default 0.01, i.e., 1%).

    Returns:
    - N: The required sample size.
    - mu: The estimated mean payoff.
    """
    z_score = 1.96  # 95% Confidence interval z-score

    N = 128  # Starting sample size
    # Generate initial batch of standard normal random variables
    z_init = np.random.standard_normal((N, model.d))
    Z_values = model.payoff_digital(z_init)  # Compute payoffs for initial batch

    # Initial statistics: mean and variance
    mu = np.mean(Z_values)
    var = np.var(Z_values, ddof=1)  # Sample variance with ddof=1

    print(f"\n--- Specific Recursive Search (Digital, d={d}, Start N={N}) ---")
    print(f"{'N':<10} | {'Mean':<10} | {'Var':<10} | {'RelErr':<10}")
    print("-" * 55)

    while True:
        # Generate one new standard normal sample
        z_new = np.random.standard_normal((1, model.d))
        Z_next = model.payoff_digital(z_new)[0]  # Payoff for the new sample

        mu_old = mu  # Save previous mean

        # Recursive update of mean and variance
        mu = (N / (N + 1)) * mu + Z_next / (N + 1)
        var = ((N - 1) / N) * var + (Z_next - mu_old)**2 / (N + 1)

        N += 1  # Increment sample size

        # Check for convergence
        sigma_est = np.sqrt(max(var, 0.0))  # Estimated standard deviation

        if abs(mu) > 1e-12 and sigma_est > 0:
            half_width = z_score * sigma_est / np.sqrt(N)  # Half-width of confidence interval
            rel_err = half_width / abs(mu)  # Relative error

            if rel_err <= tol:
                print("-" * 55)
                print(f"Converged at N = {N} for d={d}")
                print(f"Final Mean: {mu:.6f}")
                print(f"Final Var:  {var:.6f}")
                return N, mu

            # Print progress every 5000 samples
            if N % 5000 == 0:
                print(f"{N:<10} | {mu:<10.4f} | {var:<10.4f} | {rel_err:<10.2%}")

# ==========================================
# Model: Digital Asian Option
# ==========================================

```

```python
class DigitalAsianOption:
    """
    Represents a digital Asian option using Cholesky decomposition for path generation.

    This class models the underlying asset paths using correlated Brownian motions,
    discretized into 'd' time steps. The payoff is binary based on the arithmetic average
    of the stock prices exceeding the strike.

    Attributes:
    - S0: Initial stock price
    - K: Strike price
    - T: Time to maturity
    - r: Risk-free rate
    - sigma: Volatility
    - d: Number of time steps
    - dt: Time step size
    - A: Cholesky matrix for correlation
    - drift_term: Precomputed drift term for each time step
    """
    def __init__(self, S0=100, K=100, T=1.0, r=0.1, sigma=0.1, d=32):
        """
        create_model(S0, K, T, r, , m): This initialization routine constructs the discrete time grid
        t_i = i*T/m where i = 1, ..., m and the associated covariance matrix   R^{mm} with entries
         _{ij} = min(t_i, t_j). Crucially, it performs the Cholesky decomposition  = A A^T (via numpy.linalg.cholesky).
        The resulting lower-triangular matrix A is stored to facilitate the linear mapping W = A z required for
        path generation, as described in Section 2.1.

        Parameters:
        - S0: Initial stock price
        - K: Strike price
        - T: Time to maturity
        - r: Risk-free interest rate
        - sigma: Volatility
        - d: Number of time steps (m in the explanation)
        """
        self.S0 = S0
        self.K = K
        self.T = T
        self.r = r
        self.sigma = sigma
        self.d = d
        self.dt = T / d  # Time step size

        # 1. Covariance Matrix construction: _{ij} = min(t_i, t_j)
        times = np.linspace(self.dt, T, d)
        C = np.minimum(times[:, None], times[None, :])

        # Cholesky Decomposition: A A^T =
        self.A = np.linalg.cholesky(C)

        # Precompute drift for simulation: (r - 0.5 ^2) * t_i
        self.drift_term = (self.r - 0.5 * self.sigma**2) * times

    def get_S_mean(self, z):
        """
        get_s_mean(z, model): Computes the realization of the random variable (1/m) _{i=1}^m S_{t_i}.
        Given a standard normal vector z  R^m, it first recovers the Brownian motion path W = A z.
        It then applies the geometric Brownian motion mapping S_{t_i} = S0 exp((r  ^2/2) t_i +  W_{t_i})
        in a vectorized manner. This function serves as the numerical evaluation of the underlying asset process.

        Parameters:
        - z: Array of standard normal random variables, shape (N, d)

        Returns:
        - Arithmetic mean of stock prices, shape (N,)
        """
        if z.ndim == 1: z = z.reshape(1, -1)  # Ensure 2D
        # Correlated Brownian increments: W = A z
        B = z @ self.A.T
        # Stock prices: S_{t_i} = S0 * exp(drift +  * W_{t_i})
        S = self.S0 * np.exp(self.drift_term + self.sigma * B)
        # Arithmetic mean: (1/m)  S_{t_i}
        return np.mean(S, axis=1)

    def payoff_digital(self, z):
        """
        payoff_digital(z, model): Evaluates the discounted payoffs e^{-r T} _i(z).
        Implements Eq. (5), returning e^{-r T} if the arithmetic mean exceeds K, and 0 otherwise.

        Parameters:
        - z: Array of standard normal random variables, shape (N, d)

        Returns:
        - Discounted binary payoffs, shape (N,)
        """
        S_mean = self.get_S_mean(z)  # Compute arithmetic mean
        p = np.where(S_mean > self.K, 1.0, 0.0)  # Binary payoff
        return np.exp(-self.r * self.T) * p  # Discounted

    def payoff_arithmetic(self, z):
        """
        payoff_arithmetic(z, model): Evaluates the discounted payoffs e^{-r T} _i(z) for the arithmetic option.
        Returns e^{-r T} max(S_mean - K, 0). Used as a proxy for gradient estimation in active subspaces.

        Parameters:
        - z: Array of standard normal random variables, shape (N, d)

        Returns:
        - Discounted arithmetic payoffs, shape (N,)
        """
        S_mean = self.get_S_mean(z)  # Compute arithmetic mean
        return np.exp(-self.r * self.T) * np.maximum(S_mean - self.K, 0)  # Discounted payoff

# ========================================
# Directions & Shifts
# ========================================

def get_gradient(f, z, epsilon=1e-5):
    d = len(z)
    grad = np.zeros(d)
    base = f(z)
    for i in range(d):
        z_p = z.copy()
        z_p[i] += epsilon
        grad[i] = (f(z_p) - base) / epsilon
    return grad

def get_active_subspace(model, M=128):
    """
    get_active_subspace(model, payoff_fn, M): Approximates the solution to the maximization problem (14).
    It estimates the gradient covariance matrix C = E[     ^T] using a pilot sample of size M = 128.
    Gradients are computed via finite differences (get_gradient). The function returns the leading eigenvector
    u1 of C, identifying the direction of the kink or jump.

    Parameters:
    - model: Option model instance
    - M: Number of pilot samples (default 128)

    Returns:
    - u_as: Dominant direction vector (normalized)
    """
    sampler = qmc.Sobol(d=model.d, scramble=True)  # Scrambled Sobol sequence
    z_pilot = stats.norm.ppf(sampler.random(M))  # Transform to normal

    grads = []
    # Use Arithmetic payoff for gradient estimation (as proxy for digital)
    for i in range(M):
        g = get_gradient(model.payoff_arithmetic, z_pilot[i])  # Finite difference gradient
        grads.append(g)
    grads = np.array(grads)

    # Covariance matrix of gradients: C  (1/M)    ^T
    C_hat = (grads.T @ grads) / M
    evals, evecs = np.linalg.eigh(C_hat)  # Eigen decomposition
    u_as = evecs[:, -1]  # Leading eigenvector (largest eigenvalue)

    # Standardize sign: ensure mean moves up with positive u
    test_z = u_as * 0.1
    if model.get_S_mean(test_z)[0] < model.get_S_mean(-test_z)[0]:
        u_as = -u_as
    return u_as

def get_z1_direction(d):
    """Direction for z1: [1, 0, ... 0]"""
    u = np.zeros(d)
    u[0] = 1.0
    return u

def householder_matrix(u):
    """Orthogonal matrix Q where first column is u"""
    d = len(u)
    e1 = np.zeros(d); e1[0] = 1.0
    sign = np.sign(u[0]) if u[0] != 0 else 1
    v = u + sign * np.linalg.norm(u) * e1
    v = v / np.linalg.norm(v)
    H = np.eye(d) - 2 * np.outer(v, v)
    return H * (-sign)

def get_odis_shift(model):
    """
    get_odis_shift_digital(model): Determines the optimal drift * for the binary option.
    Instead of the generic likelihood maximization, it solves a constrained geometric problem:
    it minimizes the distance ||z||^2 / 2 subject to (z) > 0. This identifies the point on the limit
    surface  (z) = 0 with the highest probability density, solved via the SLSQP algorithm.

    Parameters:
    - model: Option model

    Returns:
    - Optimal shift vector *
    """
    def constr(z):
        # Constraint: S_mean(z) - K = 0 (on the exercise boundary)
        return model.get_S_mean(z)[0] - model.K

    def obj(z):
        # Objective: minimize ||z||^2 / 2
        return 0.5 * np.sum(z**2)

    x0 = np.ones(model.d) * 0.1  # Initial guess
    cons = ({'type': 'eq', 'fun': constr})  # Equality constraint
    res = minimize(obj, x0, method='SLSQP', constraints=cons, tol=1e-4)  # Constrained optimization

    if not res.success:
        return np.zeros(model.d)  # Fallback to zero shift

    return res.x  # Optimal

# ==========================================
# Estimators
# ==========================================

def standard_estimator(model, N, method='MC', mu_shift=None):
    """
    standard_estimator(model, N, payoff_fn, method, ): A unified driver for the estimators V_{CMC} and V_{QMC}.
       Sequence Generation: If method='RQMC', it utilizes a scrambled Sobol sequence generator (scipy.stats.qmc.Sobol)
      to produce points U^{(i)}  [0,1]^m, which are mapped to R^m via the inverse normal CDF ^{-1}.
      To preserve numerical stability, inputs to ^{-1} are clipped to [10^{-10}, 1 - 10^{-10}].
       Change of Measure: To support ODIS (Section 2.7), the function accepts an optimal drift vector .
      It shifts the samples Z' = Z +  (simulating from q(z; )) and computes the Radon-Nikodym derivative
      w(z;  ) = exp(- ^T Z' + (1/2)|| ||^2).
       Estimation: Returns the sample mean of the product (Z') w(Z'; ), providing an unbiased estimate of V.

    Parameters:
    - model: Option model
    - N: Number of samples
    - method: 'MC' for Monte Carlo, 'RQMC' for Quasi-Monte Carlo
    - mu_shift: Shift vector  for importance sampling (optional)

    Returns:
    - Estimated price (mean of weighted payoffs)
    """
    if method == 'MC':
        z = np.random.standard_normal((N, model.d))  # Standard normal samples
    elif method == 'RQMC':
        sampler = qmc.Sobol(d=model.d, scramble=True)  # Scrambled Sobol
        u = sampler.random(N)  # Uniform [0,1]^m
        # Clip to avoid ^{-1} issues at boundaries
        u = np.clip(u, 1e-10, 1 - 1e-10)
        z = stats.norm.ppf(u)  # Inverse CDF to normal

    weights = np.ones(N)  # Default weights
    if mu_shift is not None:
        # Importance Sampling: shift samples and adjust weights
        X = z + mu_shift  # Z' = Z +
        dot = np.sum(X * mu_shift, axis=1)  # ^T Z'
        mu_sq = 0.5 * np.sum(mu_shift**2)  # (1/2) ||||^2
        weights = np.exp(-dot + mu_sq)  # w(Z'; )
        payoffs = model.payoff_digital(X)  # (Z')
    else:
        payoffs = model.payoff_digital(z)  # (Z)

    return np.mean(payoffs * weights)  # Sample mean of  w

def pre_int_estimator_closed_form(model, N, u, Q, mu_perp=None, method='RQMC'):
    """
    vector_pre_int_digital(...): These functions implement the smoothed estimators by evaluating the inner
    integral p(x_{-1}) defined in Eq. (6).
       Root Finding: For every sample z_{-1} in the subspace R^{m-1}, the routine defines the implicit function
      g(v) = Mean(S(v u1 + Q_{-1} z_{-1})) - K. It employs Brent's method to find the critical value v* = (z_{-1})
      where the payoff activates.
       Analytic Smoothing: For the Digital option, the discontinuous indicator is replaced by the smooth tail
      probability (-v*). For the Arithmetic option, the function computes the conditional expectation
      E[(S_mean - K)+ | z_{-1}] analytically using Gaussian integrals, removing the derivative discontinuity.

    Parameters:
    - model: Option model
    - N: Number of samples in perpendicular subspace
    - u: Direction vector (e.g., active subspace or z1)
    - Q: Orthogonal matrix from Householder
    - mu_perp: Shift in perpendicular direction (optional)
    - method: 'MC' or 'RQMC'

    Returns:
    - Estimated price using pre-integration
    """
    d_perp = model.d - 1  # Dimension of perpendicular subspace R^{m-1}
    if method == 'MC':
        z_perp = np.random.standard_normal((N, d_perp))  # Samples in R^{m-1}
    else:
        sampler = qmc.Sobol(d=d_perp, scramble=True)
        z_perp = stats.norm.ppf(sampler.random(N))  # Quasi-random

    weights = np.ones(N)  # IS weights
    if mu_perp is not None:
        # Apply IS in perpendicular direction
        X_perp = z_perp + mu_perp
        dot = np.sum(X_perp * mu_perp, axis=1)
        mu_sq = 0.5 * np.sum(mu_perp**2)
        weights = np.exp(-dot + mu_sq)
        z_perp = X_perp  # Use shifted samples

    # Precompute matrices
    U_perp = Q[:, 1:]  # Perpendicular directions
    Au = model.A @ u  # A u
    AU_perp_z_perp = (model.A @ U_perp) @ z_perp.T  # A U_{-1} z_{-1}

    beta = model.sigma * Au  # Coefficient for exponential
    const_log_S = np.log(model.S0) + model.drift_term  # Log S0 + drift terms

    estimates = np.zeros(N)

    # Search range for v*
    BOUND = 30.0

    for i in range(N):
        # Contribution from perpendicular part
        perp_contribution = model.sigma * AU_perp_z_perp[:, i]
        alpha = np.exp(const_log_S + perp_contribution)  # Alpha coefficients

        def g(v):
            # g(v) = E[S_mean | z_{-1}] - K, where S_mean depends on v
            return np.mean(alpha * np.exp(beta * v)) - model.K

        # Find v* such that g(v*) = 0
        try:
            if g(-BOUND) * g(BOUND) < 0:
                v_star = brentq(g, -BOUND, BOUND)  # Root in [-BOUND, BOUND]
            else:
                v_star = -BOUND if g(0) > 0 else BOUND  # Boundary case
        except ValueError:
             v_star = -BOUND if g(0) > 0 else BOUND

        # Closed-form smoothing: replace indicator with (-v*)
        val = np.exp(-model.r * model.T) * stats.norm.cdf(-v_star)
        estimates[i] = val * weights[i]  # Weighted estimate

    return np.mean(estimates)  # Average over samples

# ==========================================
# Simulation Logic
# ==========================================

def run_experiment(K_target, d_val):
    """
    Runs simulations for a given strike price K_target and dimension d_val.

    Computes RMSE for various estimators and returns results as a DataFrame.
    """
    print(f"\n--- Running Experiment for K = {K_target}, d = {d_val} (Digital) ---")
```

```python
    model = DigitalAsianOption(S0=S0, K=K_target, T=T, r=r, sigma=sigma, d=d_val)  # Initialize model

    # Compute directions for pre-integration
    u_z1 = get_z1_direction(d_val)  # z1 direction
    Q_z1 = np.eye(d_val)  # Identity for z1
    u_as = get_active_subspace(model)  # Active subspace direction
    Q_as = householder_matrix(u_as)  # Rotation matrix

    # Compute ODIS shift only for OTM case (K=120)
    mu_opt = None
    mu_perp_as = None
    if K_target == 120:
        print("  -> Computing ODIS shift...")
        mu_opt = get_odis_shift(model)  # Optimal shift
        mu_local = Q_as.T @ mu_opt  # Transform to local coords
        mu_perp_as = mu_local[1:]  # Perpendicular component

    # Compute ground truth using high-accuracy pre-integration
    print("  -> Computing Ground Truth...")
    if K_target == 120:
        true_val = pre_int_estimator_closed_form(model, 2**17, u_as, Q_as, mu_perp_as, 'RQMC')
    else:
        true_val = pre_int_estimator_closed_form(model, 2**17, u_as, Q_as, None, 'RQMC')
    print(f"  -> Truth: {true_val:.6f}")

    # Define methods based on strike
    methods = {}
    if K_target == 100:  # ATM
        methods['Crude MC'] = lambda n: standard_estimator(model, n, 'MC')
        methods['Plain RQMC'] = lambda n: standard_estimator(model, n, 'RQMC')
        methods['Pre-Int (z1)'] = lambda n: pre_int_estimator_closed_form(model, n, u_z1, Q_z1,
 None, 'RQMC')
        methods['Pre-Int (AS)'] = lambda n: pre_int_estimator_closed_form(model, n, u_as, Q_as,
 None, 'RQMC')

    elif K_target == 120:  # OTM
        methods['Crude MC'] = lambda n: standard_estimator(model, n, 'MC')
        methods['MC + ODIS'] = lambda n: standard_estimator(model, n, 'MC', mu_opt)
        methods['Plain RQMC'] = lambda n: standard_estimator(model, n, 'RQMC')
        methods['RQMC + ODIS'] = lambda n: standard_estimator(model, n, 'RQMC', mu_opt)
        methods['Pre-Int (AS)'] = lambda n: pre_int_estimator_closed_form(model, n, u_as, Q_as,
 None, 'RQMC')
        methods['Pre-Int (AS) + ODIS'] = lambda n: pre_int_estimator_closed_form(model, n, u_as,
 Q_as, mu_perp_as, 'RQMC')

    results = []
    total_ops = len(SAMPLE_SIZES) * len(methods) * N_REPEATS  # Progress tracking

    with tqdm(total=total_ops, desc=f"Simulating d={d_val}") as pbar:
        for N in SAMPLE_SIZES:
            for name, func in methods.items():
                errs = []
                times = []
                for _ in range(N_REPEATS):
                    t0 = time.time()
                    est = func(N)
                    t1 = time.time()
                    errs.append(est)
                    times.append(t1 - t0)
                    pbar.update(1)

                rmse = np.sqrt(np.mean((np.array(errs) - true_val)**2))
                avg_time = np.mean(times)
                results.append({'K': K_target, 'N': N, 'Method': name, 'RMSE': rmse, 'Time':
    avg_time, 'd': d_val})

    return pd.DataFrame(results)

# ==========================================
# Plotting Functions
# ==========================================

def get_convergence_rate(N, RMSE):
    slope, intercept = np.polyfit(np.log(N), np.log(RMSE), 1)
    return slope

def plot_k100(df, d_val, save_dir):
    """
    K=100 Plot: Convergence and efficiency analysis for digital option.
    Left: Convergence (Log2 N vs RMSE)
    Right: Cost (Time vs RMSE)
    """
    df = df[(df['K'] == 100) & (df['d'] == d_val)]
    if df.empty: return

    fig, axes = plt.subplots(1, 2, figsize=(16, 6))

    # Convergence Plot
    ax = axes[0]
    methods = ['Crude MC', 'Plain RQMC', 'Pre-Int (z1)', 'Pre-Int (AS)']
    markers = ['o', 's', 'x', '^']

    for m, mark in zip(methods, markers):
        sub = df[df['Method'] == m]
        if sub.empty: continue

        slope = get_convergence_rate(sub['N'], sub['RMSE'])
        label_str = f"{m} (Rate $\\approx N^{{{slope:.2f}}}$)"
        ax.loglog(sub['N'], sub['RMSE'], marker=mark, linestyle='-', label=label_str, base=2)

    Ns = df['N'].unique()
    # Reference Lines
    if not df[df['Method']=='Crude MC'].empty:
        ref_mc = Ns**(-0.5) * (df[df['Method']=='Crude MC']['RMSE'].iloc[0] * Ns[0]**0.5)
        ax.loglog(Ns, ref_mc, 'k--', alpha=0.3, label='$O(N^{-0.5})$', base=2)

    if not df[df['Method']=='Plain RQMC'].empty:
        ref_qmc = Ns**(-1.0) * (df[df['Method']=='Plain RQMC']['RMSE'].iloc[0] * Ns[0]**1.0)
        ax.loglog(Ns, ref_qmc, 'k:', alpha=0.3, label='$O(N^{-1.0})$', base=2)

    ax.set_title(f'K=100 (Digital No-Quad, d={d_val}): Convergence Analysis', fontsize=14)
    ax.set_xlabel('Sample Size $N$ (log2 scale)', fontsize=12)
    ax.set_ylabel('RMSE', fontsize=12)
    ax.legend(fontsize=10)
    ax.grid(True, which="both", ls="-", alpha=0.2)
    ax.xaxis.set_major_formatter(mticker.ScalarFormatter())

    # Computational Cost Plot
    ax = axes[1]
    for m, mark in zip(methods, markers):
        sub = df[df['Method'] == m]
        if sub.empty: continue
        # Total time for N samples vs RMSE
        ax.loglog(sub['Time'], sub['RMSE'], marker=mark, linestyle='-', label=m)

    ax.set_title(f'K=100 (Digital No-Quad, d={d_val}): Efficiency', fontsize=14)
    ax.set_xlabel('Avg Computation Time (s)', fontsize=12)
    ax.set_ylabel('RMSE', fontsize=12)
    ax.legend(fontsize=10)
    ax.grid(True, which="both", ls="-", alpha=0.2)

    plt.tight_layout()
    fname = os.path.join(save_dir, f'Digital_NoQuad_K100_d{d_val}_Analysis.png')
    plt.savefig(fname, dpi=300)
    print(f"Saved {fname}")
    plt.close()

def plot_k120(df, d_val, save_dir):
    """
    K=120 Plots: Comprehensive comparison and variance reduction focus for digital option.
    Plot A: Comprehensive (All methods).
    Plot B: Variance Reduction Focus (Method vs Method+ODIS).
    """
    df = df[(df['K'] == 120) & (df['d'] == d_val)]
    if df.empty: return

    # Plot A: Comprehensive Comparison
    plt.figure(figsize=(10, 7))
    methods = df['Method'].unique()

    for m in methods:
        sub = df[df['Method'] == m]
        if sub.empty: continue
        slope = get_convergence_rate(sub['N'], sub['RMSE'])
        label_str = f"{m} ($N^{{{slope:.2f}}}$)"
        plt.loglog(sub['N'], sub['RMSE'], marker='o', label=label_str, base=2)

    plt.title(f'K=120 (Digital No-Quad, d={d_val}): Comprehensive Comparison', fontsize=14)
    plt.xlabel('Sample Size $N$ (log2)', fontsize=12)
    plt.ylabel('RMSE', fontsize=12)
    plt.grid(True, which="both", alpha=0.2)
    plt.legend()
    fname_A = os.path.join(save_dir, f'Digital_NoQuad_K120_d{d_val}_Comprehensive.png')
    plt.savefig(fname_A, dpi=300)
    print(f"Saved {fname_A}")
    plt.close()

    # Plot B: Variance Reduction Focus
    plt.figure(figsize=(10, 7))

    pairs = [
        ('Crude MC', 'MC + ODIS', 'red'),
        ('Plain RQMC', 'RQMC + ODIS', 'blue'),
        ('Pre-Int (AS)', 'Pre-Int (AS) + ODIS', 'green')
```

```python
    ]

    for base, odis, color in pairs:
        sub_b = df[df['Method'] == base]
        if not sub_b.empty:
            plt.loglog(sub_b['N'], sub_b['RMSE'], color=color, linestyle='--', marker='o',
     label=base, base=2, alpha=0.5)

        sub_o = df[df['Method'] == odis]
        if not sub_o.empty:
            slope = get_convergence_rate(sub_o['N'], sub_o['RMSE'])
            label_str = f"{odis} ($N^{{{slope:.2f}}}$)"
            plt.loglog(sub_o['N'], sub_o['RMSE'], color=color, linestyle='-', marker='D',
     label=label_str, base=2)

    plt.title(f'K=120 (Digital No-Quad, d={d_val}): Variance Reduction Impact', fontsize=14)
    plt.xlabel('Sample Size $N$ (log2)', fontsize=12)
    plt.ylabel('RMSE', fontsize=12)
    plt.grid(True, which="both", alpha=0.2)
    plt.legend()
    fname_B = os.path.join(save_dir, f'Digital_NoQuad_K120_d{d_val}_Variance.png')
    plt.savefig(fname_B, dpi=300)
    print(f"Saved {fname_B}")
    plt.close()

# ==========================================
# Main Execution
# ==========================================

if __name__ == "__main__":
    # Main execution for digital Asian option analysis
    if not os.path.exists('plots_digital_asian_2'):
        os.makedirs('plots_digital_asian_2')

    required_n_results = []

    # Loop over the dimensions
    for d_val in DIMENSIONS:
        print("\n" + "#"*60)
        print(f"PROCESSING DIMENSION: {d_val}")
        print("#"*60)

        # Create directory for this dimension
        curr_dir = os.path.join('plots_digital_asian_2', f'd_{d_val}')
        if not os.path.exists(curr_dir):
            os.makedirs(curr_dir)

        # Run Experiments
        df100 = run_experiment(100, d_val)
        df120 = run_experiment(120, d_val)

        full_df = pd.concat([df100, df120])
        csv_name = os.path.join(curr_dir, f'digital_option_results_2_d{d_val}.csv')
        full_df.to_csv(csv_name, index=False)
        print(f"\nResults for d={d_val} saved to {csv_name}.")

        # Generate Plots
        plot_k100(full_df, d_val, curr_dir)
        plot_k120(full_df, d_val, curr_dir)

        # Find N for K=120
        print(f"\nFind N (Digital) for d={d_val}")
        model_test = DigitalAsianOption(S0=S0, K=120, T=T, r=r, sigma=sigma, d=d_val)

        final_N, final_price = find_N_specific_digital(model_test, d=d_val, tol=0.01)
        required_n_results.append({'d': d_val, 'Required_N': final_N, 'Estimated_Price':
 final_price})

        with open(os.path.join(curr_dir, f'required_N_digital_d{d_val}.txt'), 'w') as f:
            f.write(str(final_N))

    print("\nDigital Option Analysis Complete.")
```

AI Use Disclosure In compliance with point 3 of the project rules, we disclose the usage of AI assistance in the development of this project. AI tools were utilized for the following specific tasks:

- **Code Optimization and Plotting:** The AI assisted in organizing the Python code structure for better modularity and readability, as well as generating the scripts used to produce the convergence and efficiency plots presented in the report.
- **Active Subspace Implementation for Digital Options:** The AI provided guidance on handling the lack of useful gradients in the Digital Asian option payoff (which is piecewise constant). Specifically, it helped implement the strategy of using the Arithmetic Asian option as a smooth proxy to estimate the gradient covariance matrix and identify the active subspace directions effectively.