A Nested Multilevel Monte Carlo Framework using Reduced-Precision Arithmetic

Jason K.Q. Tay

Lady Margaret Hall

University of Oxford

A thesis submitted for the degree of

MSc in Mathematical Modelling and Scientific Computing

Trinity 2025

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Mike Giles, for his invaluable supervision, guidance, and generous availability, even during his commitments at academic conferences in the United States over the summer. I also warmly congratulate him on his election as a fellow of the Royal Society earlier this year.

I am also grateful to our Course Director, Dr Kathryn Gillow, for her unwavering commitment and dedication to the MMSC course and its cohort. Her passion for teaching and guidance throughout the year has been truly commendable.

I would like to thank my peers in the MMSC cohort. From late nights in the Mathematical Institute cramming for exams and the thesis, to formal halls and evening basketball sessions, the memories we have made will remain with me for a lifetime. I would like to wish everyone the greatest success and happiness in their future endeavours.

Last, but by no means least, I would like to thank my parents, siblings, and close friends for their never-ending encouragement and support. In particular, I am indebted to my mother for her time and care in proofreading this thesis.

Abstract

Multilevel Monte Carlo (MLMC) is a variance-reduction technique widely utilised to reduce the computational cost of stochastic simulations, particularly in option pricing. The idea is to approximate expectations using multiple resolutions in the discretisation of the underlying stochastic differential equation. Although effective, MLMC remains computationally demanding, motivating the use of lower-precision arithmetic to accelerate simulations. In this thesis, we investigate the use of reduced precision floating-point arithmetic within a nested MLMC framework on CPUs. Several lookup table methods for Gaussian random number generation were developed and vectorised using AVX-512 intrinsics, achieving genuine FP16 speedups over FP32. A cost model for payoff path calculations was also introduced to quantify the savings from reduced precision. Numerical experiments showed that mixed-precision MLMC schemes using reduced-precision arithmetic can achieve substantial computational savings, but accumulated half-precision rounding errors at fine levels compromise convergence, resulting in smaller-than-theoretical weak convergence α and variance decay β. To overcome this, we propose an adaptive-precision scheme that switches from a mixed-precision FP16–FP32 to an FP32-only scheme beyond a cut-off level. This preserves the efficiency gain from reduced precision at coarse levels while restoring stability at fine levels. Our results demonstrate that the adaptive mixed-precision MLMC introduced in this work successfully provides efficient and reliable simulations for option pricing. In contrast to FPGA-based fixed-point approaches in prior work, this thesis introduces CPU-based floating-point nesting as a novel framework for reduced-precision MLMC.

Gen AI Statement

Portions of this thesis were assisted by generative AI tools (OpenAI ChatGPT), which were used solely for language refinement, code debugging (MATLAB, C++), and LaTeX template formatting. All mathematical derivations, implementations, results, and analysis are entirely the author's own, with supervision and guidance from Prof. Giles.

Contents

1 Introduction **1**

1.1 Intel Hardware and SIMD Vectorisation 2

1.1.1 Scalar versus Vector Processing 3

1.1.2 Intel AVX-512 Overview 4

1.2 Floating-Point and Mixed-Precision Arithmetic 4

1.3 Geometric Brownian Motion and European Call Option 6

1.4 Monte Carlo Methods for SDEs 7

1.4.1 Multilevel Monte Carlo 8

1.4.2 Accuracy and Convergence of Multilevel Monte Carlo 10

1.4.3 Multilevel Monte Carlo Algorithm 11

1.4.4 Nested Multilevel Monte Carlo 11

1.5 Aim and Contributions of this Work 13

2 Approximating the Gaussian Distribution **15**

2.1 The Inverse Gaussian Cumulative Distribution Function 16

2.2 Piecewise Constant Approximation on Uniform Intervals (Method 1) 18

2.3 Piecewise Linear Approximation on Dyadic Intervals (Method 2) . . . 19

2.4 Piecewise Linear Approximation on Superdyadic Intervals (Method 3) 21

2.5 Comparison of the Three Inversion Methods 22

2.5.1 Storage and Computational Cost 22

2.5.2 Approximation Accuracy and Scaling 23

3 Gaussian Random Number Generation Algorithm **26**

3.1 LUT Generation and Lookup Procedure 26

3.2 Implementing SIMD Vectorisation 28

3.2.1 Piecewise Constant Approximation 29

3.2.2 Piecewise Linear Approximation (Dyadic and Superdyadic) . . 29

3.3 Benchmarking Methodology 30

3.3.1 Benchmark Setup 31
3.3.2 Benchmark Procedure 31
3.3.3 Baseline Comparison 32
3.4 Comparative Performance Analysis 33
3.4.1 Timing Results 33
3.4.2 FP16 – FP32 Coupling Analysis 35

4 Cost Model for Payoff Path Calculation 36
4.1 Path Cost Measurement Framework 37
4.2 Numerical Results and Discussion 38

5 Nested Multilevel Monte Carlo Framework 41
5.1 Classic Nested Multilevel Monte Carlo 41
5.1.1 Numerical Results and Discussion 42
5.2 Adaptive–Precision Nested Multilevel Monte Carlo 48
5.2.1 Numerical Results and Discussion 48
5.3 Discussion and Conclusion 52

6 Conclusion 54

A Subnormal Numbers 55

B Signed Error Plots for LUT Approximations 56

C Alternate Piecewise Linear Approximation Algorithm 58

D AVX-512 Intrinsics for SIMD Vectorisation 59

E Payoff Path Calculations 64

F Complete Multilevel Monte Carlo Simulations 66

Bibliography 72

Chapter 1

Introduction

Since Boyle's first explicit application of Monte Carlo (MC) simulation to option pricing in 1977, path simulation has become the primary tool for estimating the expected value of financial payoffs in computational finance [5, 22]. The expected value is based on the solution of stochastic differential equations (SDEs) which consider the evolution of asset prices, interest rates, volatilities and other quantities. At its core, standard MC is straightforward and embarrassingly parallel, but achieving high accuracy requires computing a large number of costly random SDE path approximations, with a computational cost of $\mathcal{O}(\varepsilon^{-3})$ for an accuracy of $\mathcal{O}(\varepsilon)$. This has motivated research into various variance and cost reduction techniques as described in [16].

Among these methods is the multilevel Monte Carlo (MLMC), which was proposed in 2008 by Giles [15] and has since been adapted to numerous applications and even combined with other variance reduction methods such as the quasi-Monte Carlo [21]. The main idea behind MLMC is to approximate the expectation by using levels of increasing time-stepping resolutions when numerically solving the SDEs and allocating an optimal number of samples at each level, minimising the overall computational cost subject to the desired bound on the variance. The computational savings stem from taking many coarse samples from lower levels and combining them with few fine samples from the finer levels. In the simplest case of a Lipschitz payoff and Euler discretisation, the computational cost to achieve an accuracy of $\mathcal{O}(\varepsilon)$ is reduced to $\mathcal{O}\big(\varepsilon^{-2}(\log\varepsilon)^2\big)$, a significant reduction compared to the standard MC.

While SDEs like the Geometric Brownian Motion (GBM) do not require a nested expectation structure to be solved numerically, adopting a nested MLMC implementation in which each level-specific estimator is itself computed using a dedicated estimator can lead to further computational speedups [17]. A particularly effective approach is to exploit mixed-precision arithmetic through a nesting framework. From accelerating neural networks [34] to mathematical optimisation [8], the utilisation of mixed-precision methodology has proven success in various scientific fields. Similar work that have studied the potential of mixed-precision computation in the MLMC framework include Sheridan-Methvan and Haas [24, 38], both of which were past projects supervised by Prof. Giles at Oxford. This thesis builds upon the fine-grained, per-variable precision optimisation methodology in [24] to standard floating-point formats (FP16 and FP32), while simultaneously adapting the approximate Gaussian variable generation techniques in [38] for efficient low-precision implementation within the nested MLMC for pricing financial options.

An important motivation for adopting reduced precision in a nested MLMC framework is the native support for half-precision (FP16) vector instructions on modern hardware, such as the Intel Xeon CPUs. These enable many floating-point operators with roughly half the energy and memory footprint of single precision (FP32), which remains available when higher accuracy is required. This eliminates the reliance on FPGAs for low-precision arithmetic [24] and leverages modern SIMD instructions (AVX-512) for Gaussian generation and payoff computation.

As such, the main goal of this thesis is to build a nested MLMC framework that exploits modern Intel hardware's native support for half-precision arithmetic. The objective is to improve computational efficiency and reduce energy consumption in large-scale derivative pricing and risk assessment, aligning with current priorities in algorithmic design and computational finance [23, 43].

The following sections of this chapter first provide an overview of the Intel hardware and vectorisation capabilities leveraged in this work, before introducing the core preliminaries on reduced-precision arithmetic and MLMC methods that underpin the methodology developed in this thesis.

1.1 Intel Hardware and SIMD Vectorisation

Modern high-performance computing platforms, such as the Intel Xeon Gold 6538Y+ used in this thesis [12], are equipped with vectorised arithmetic capabilities. They exploit data parallelism, allowing multiple data elements to be processed simultaneously through Single Instruction, Multiple Data (SIMD) execution [2, 25, 26]. MC simulations, and particularly the MLMC framework considered in this thesis, are embarrassingly parallel and thus extremely applicable to SIMD acceleration. By leveraging the Intel AVX-512 instruction set and the hardware's native support for half-precision arithmetic [9], this thesis aims to maximise computational throughput while using nested MLMC to avoid any loss of accuracy.

1.1.1 Scalar versus Vector Processing

Historically, processors executed instructions as scalars, in which each arithmetic operation was applied to a single pair of operands before proceeding to the next [41]. In contrast, vector processing combines multiple operands into wide registers and applies the same instruction to all of them in parallel.

To conceptually visualise the distinct advantages of vector processing, consider the operation representing $a + b = c$ with arrays a, b, and c. The operations $(+, =)$ act on each element one at a time in scalar processing, while in vector processing, it replicates the same functionality using vectorised SIMD operations for a specified vector width. On modern hardware, the cost difference between vector and scalar counterparts are near negligible [38]. As such, suppose the register has a vector width of 4 integers, the vectorised program only requires a quarter of the original number of scalar operations as illustrated in Figure 1.1. Thus, the vectorised SIMD task provided in the example expects a 4x speedup compared to scalar processing.

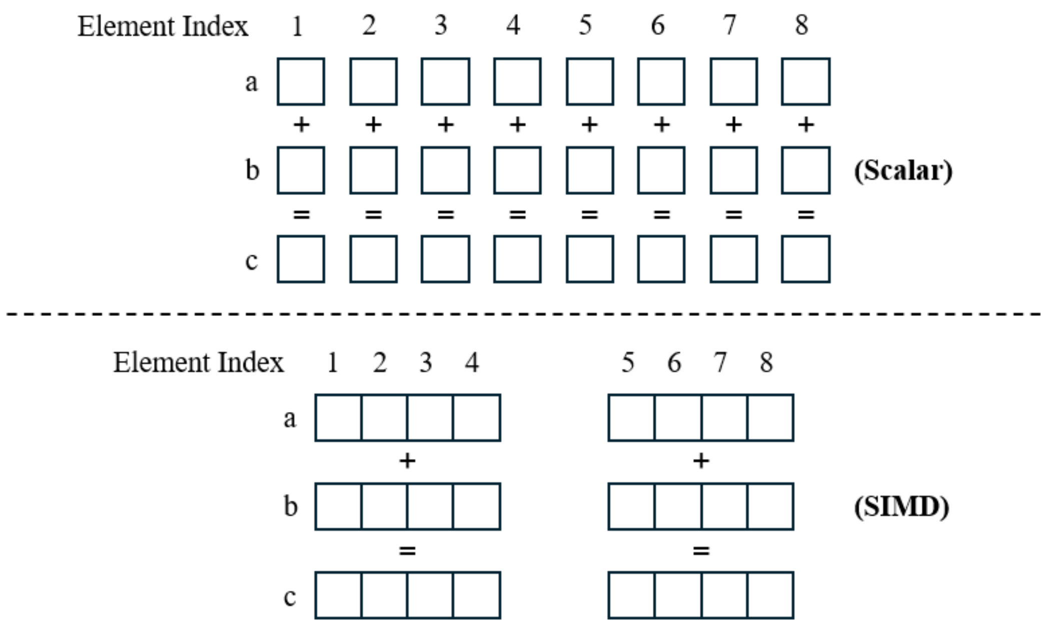


Figure 1.1: Processing of operation $a + b = c$ with a vector width of 4 integers. **(Above)** Scalar. **(Below)** SIMD. Adapted from [38].

1.1.2 Intel AVX-512 Overview

AVX-512 is part of Intel's SIMD advanced vector extension (AVX) instruction set. As the name suggests, it consists of 512-bit wide vectors, having 32 vector registers, and 16 masking registers which are 64-bit wide. These registers enable the same operation to be applied to multiple data elements in parallel as shown in Figure 1.1. Internally, the Intel Gold 6538Y+ processor is capable of storing 32 FP16 values or 16 FP32 values at once.

This doubling of register usage when using FP16 allows twice as many elements to be processed per instruction compared to FP32, which serves as a key motivation for utilising mixed-precision arithmetic as it enables a true hierarchical efficiency that is essential for MLMC. A schematic of a 512-bit register is shown in Figure 1.2, where each element stores part of an array to be processed in parallel.

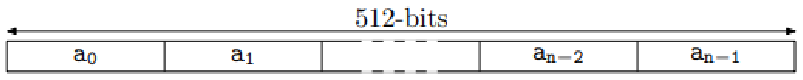


Figure 1.2: Schematic of an AVX-512 vector register holding n elements of an array a. Taken from [38].

.

From a programming perspective, AVX-512 operations are accessible through specific compiler intrinsics for FP16 and FP32 arithmetic, memory access, and data conversion. These intrinsics provide a low-level interface that maps closely to machine instructions, enabling fine-grained control over throughput and memory behaviour. While other approaches for vectorisation exist, such as compiler auto-vectorisation and OpenMP SIMD directives, the focus of this thesis is on manual vectorisation using AVX-512 intrinsics. The specific implementation and emulation strategies are presented in Section 3.2.

1.2 Floating-Point and Mixed-Precision Arithmetic

Floating-point numbers are used to represent real numbers in computers according to the IEEE-754 standard. A normalised floating-point value in binary format is computed as

$$x = (-1)^s \times 2^{E-b} \times (1 + F), \tag{1.1}$$

where s is the sign bit, E is the unsigned value stored in the e-bit exponent field, F is the fractional mantissa ($0 \leq F < 1$), and the bias $b = 2^{e-1}-1$ represents both positive and negative exponents using unsigned binary integers. This representation enables the encoding of a wide range of real numbers using fixed-length binary formats [3].

The IEEE-754 standard defines several common formats, including half precision (FP16) and single precision (FP32), which differ in the numbers of bits allocated to each component as described in Table 1.1 [32].

Table 1.1: IEEE-754 bit layouts for FP16 and FP32. Bias values are derived from the exponent bits and are not additional stored bits.

Components	**FP16**	**FP32**
Sign bits, s	1	1
Mantissa bits, F	10	23
Exponent bits, e	5	8
Bias, $b = 2^{e-1} - 1$	15	127

The precision of a floating-point value is determined by the number of mantissa bits, otherwise referred to as machine epsilon or unit roundoff:

$$\begin{aligned} \delta_{\text{FP16}} &= 2^{-10} \approx 9.8 \times 10^{-4}, \\ \delta_{\text{FP32}} &= 2^{-23} \approx 1.2 \times 10^{-7}. \end{aligned} \tag{1.2}$$

In contrast, the dynamic range of a floating-point value is determined by the number of exponent bits. The smallest and largest magnitudes for a format with exponent bits e and bias b are 2^{1-b} and 2^b, respectively. This yields the following ranges:

$$\begin{aligned} \text{FP16} &\in [2^{-14}, 2^{15}] \approx [6.1 \times 10^{-5}, 3.3 \times 10^{4}], \\ \text{FP32} &\in [2^{-126}, 2^{127}] \approx [1.2 \times 10^{-38}, 1.7 \times 10^{38}]. \end{aligned} \tag{1.3}$$

This illustrates the main appeal of half-precision as FP16 is twice as compact and can potentially deliver up to a theoretical 2x speedup on modern hardware with native support compared to FP32. However, FP16's reduced precision and narrower dynamic range make it vulnerable to underflow, overflow, and rounding errors during numerical simulations. These issues are especially relevant in SDEs systems, such as pricing financial options, where quantities can span several orders of magnitude and each timestep requires operations (like square roots and multiplication) that are sensitive to rounding errors in low-precision arithmetic.

For a complete specification of the IEEE-754 standard, including the handling of values below the defined dynamic range in (1.3) through subnormal numbers instead of immediate underflowing, see Appendix A and [29].

1.3 Geometric Brownian Motion and European Call Option

Monte Carlo methods are applicable across a broad range of systems, such as SDEs and high-dimensional problems arising in physics [33], finance [42], and engineering [1]. In this thesis, we consider a classical example in computational finance: the simulation of a Geometric Brownian Motion (GBM) for pricing a European vanilla call option [4]. Options are financial derivatives that offer the investor the right but not the obligation to exercise an asset at a predetermined strike price and future maturity date [14]. The payoff function for the European vanilla call option is modelled as

$$P = \max(S_T - K, 0), \tag{1.4}$$

where S_T is the simulated asset price at maturity T, and K is the strike price.

Mathematically, the asset price S_t evolves according to the GBM [39]:

$$dS_t = rS_t\,dt + \sigma S_t\,dW_t, \tag{1.5}$$

where r is the interest rate, σ is the volatility of the asset, and W_t is a standard Brownian motion that introduces randomness into the system. The SDE (1.5) is solved numerically by applying the Euler-Maruyama scheme

$$S_{i+1} = S_i + S_i\Big(r\,h + \sigma\,\sqrt{h}\,Z_i\Big), \tag{1.6}$$

which discretises time into steps $h = T/N$, and generates Gaussian increments $\Delta W_i = \sqrt{h}\,Z_i$, where $Z_i \sim \mathcal{N}(0, 1)$.

Solving this numerical scheme produces the simulated asset price at maturity S_T, and is consequently used to solve the payoff P of the call option as expressed in (1.4). However, this payoff represents the option's value at maturity T, instead of present value P_0 at initial time $t = 0$, where it must be discounted using the risk-free interest rate r

$$P_0 = e^{-rT}\max(S_T - K, 0). \tag{1.7}$$

This procedure, as summarised in Algorithm 1.1, can be implemented in either FP16 or FP32 arithmetic, independent of specific vectorisation details.

Algorithm 1.1 Euler–Maruyama GBM path calculation

Input: Initial asset price S_0, strike price K, interest rate r, volatility σ, maturity T, time steps M

Output: Asset price at maturity S_T, payoff P, discounted payoff P_0

```
$h \leftarrow T/M$
drift $\leftarrow r \times h$
diffusion $\leftarrow \sigma \times \sqrt{h}$
disc $\leftarrow \exp(-r \times T)$
Initialise $S_0$
Generate random Gaussian samples $Z_j \sim \mathcal{N}(0,1)$ for $i = 1, \ldots, M$
for $i = 1$ to $M$ do
    mult1$_i$ $\leftarrow$ diffusion $\times Z_i$
    sum$_i$ $\leftarrow$ drift $+$ mult$_i$
    mult2$_i$ $\leftarrow S_i \times$ sum$_i$
    $S_{i+1} \leftarrow S_i +$ mult2$_i$
end for
$S_T \leftarrow S_{M+1}$
$P \leftarrow \max(S_T - K, 0)$
$P_0 \leftarrow P \times$ disc
```

1.4 Monte Carlo Methods for SDEs

Monte Carlo (MC) simulation is a numerical technique that estimates the expected value of a desired quantity, like the payoff of the call option (1.4), by computing the sample mean across many random paths. For each path i, the payoff of $P^{(i)} = \max(S_T^{(i)} - K, 0)$ is computed, and the option price estimate is

$$\frac{1}{N}\sum_{i=1}^{N} P^{(i)}, \tag{1.8}$$

where N is the number of samples used in the standard MC simulation. The variance of this estimate is $N^{-1}\mathbb{V}[P]$ and the root mean square error (RMSE) is $\mathcal{O}(N^{-1/2})$. Then achieving an accuracy of ε requires $N = \mathcal{O}(\varepsilon^{-2})$ samples and a time step of $h = \mathcal{O}(\varepsilon)$, leading to the total computational cost of $\mathcal{O}(\varepsilon^{-3})$ [15].

One key challenge in simulating SDEs such as the GBM is the discretisation bias that arises when approximating continuous-time dynamics using discrete time steps $h = T/N$. This bias limits the accuracy of the standard MC, especially when using coarse time steps.

1.4.1 Multilevel Monte Carlo

Introduced by Giles [15], the multilevel Monte Carlo (MLMC) addresses the computational inefficiency and discretisation bias in the standard MC by leveraging a hierarchy of time discretisation. Instead of simulating all random paths at the finest resolution, MLMC splits the computational work across multiple levels of increasing resolution, as shown intuitively by the following two-level Monte Carlo estimate

$$\mathbb{E}[P_1] \approx \frac{1}{N_0}\sum_{i=1}^{N_0} P_0^{(i)} + \frac{1}{N_1}\sum_{i=1}^{N_1}\big(P_1 - P_0\big)^{(i)}, \tag{1.9}$$

where the coarse sample is denoted by index 0 and the finest by index 1. $P_1 - P_0$ represents the payoff difference between P_1 and P_0 for the same stochastic sample solved using the discretised Euler-Maruyama scheme (1.6), N_0 is the independent paths on the coarse grid, and N_1 is the paired paths on both the fine and coarse grid computing the payoff difference, where $N_0 \gg N_1$.

Defining C_0 and C_1 to be the cost of computing a single sample of P_0 and $P_1 - P_0$, respectively, the total computational cost of the two-level estimator is $N_0C_0 + N_1C_1$, and if V_0 and V_1 are the variance of P_0 and $P_1 - P_0$, the overall variance is $N_0^{-1}V_0 + N_1^{-1}V_1$. Treating N_0 and N_1 as real variables, the optimal allocation of samples is solved as a constrained minimisation problem using the Lagrange multiplier approach.

This two-level MC framework can be generalised to a multilevel framework of sequence $P_0, \ldots, P_L$, where each P_l is an approximation with time steps of size, $h_l = h_0\, 2^{-l}$ denoted by index $l \in \{0, 1, \ldots, L\}$. The generalised MLMC has the telescoping identity

$$\mathbb{E}[P_L] = \mathbb{E}[P_0] + \sum_{l=1}^{L} \left(\mathbb{E}[P_l - P_{l-1}]\right), \tag{1.10}$$

with the convention $P_{-1} = 0$. The unbiased estimator for $\mathbb{E}[P_L]$ is then

$$\frac{1}{N_0}\sum_{i=1}^{N_0} P_0^{(i)} + \sum_{l=1}^{L}\left\{\frac{1}{N_l}\sum_{i=1}^{N_l}\big(P_l - P_{l-1}\big)^{(i)}\right\}. \tag{1.11}$$

Defining N_l to be the number of samples on level l, C_0 and V_0 to be the cost and variance of a single sample of P_0, and C_l, V_l to be the cost and variance of a single sample of $P_l - P_{l-1}$, the total computational cost of the MLMC estimator is

$$\sum_{l=0}^{L} N_l C_l, \tag{1.12}$$

and the overall variance is

$$\sum_{l=0}^{L} \frac{V_l}{N_l}. \tag{1.13}$$

Similar to the two-level MC, the optimal allocation of samples can be formulated as a constrained minimisation problem for some value of the Lagrange multiplier $\lambda \in \mathbb{R}^+$

$$\min_{N_l} \sum_{l=0}^{L} N_l C_l \quad \text{such that} \quad \sum_{l=0}^{L} \frac{V_l}{N_l} \leq \varepsilon^2, \tag{1.14}$$

where ε is the target root mean square error (RMSE) and user-defined tolerance level. The optimal sample is

$$N_l = \lambda \sqrt{\frac{V_l}{C_l}}, \tag{1.15}$$

and the variance constraint requires that

$$\lambda = \varepsilon^{-2} \sum_{l=0}^{L} \sqrt{V_l\, C_l}, \tag{1.16}$$

leading to the total computational cost of the MLMC estimator

$$\text{Cost}_{\text{MLMC}} = \varepsilon^{-2} \left(\sum_{l=0}^{L} \sqrt{V_l C_l} \right)^2. \tag{1.17}$$

As [16] details, the computational cost of MLMC depends critically on the trend of the product $V_l C_l$ across levels l. Table 1.2 summarises the impact of these trends.

Table 1.2: Computational cost of MLMC under different variance-cost trends

Trend of $V_l C_l$	**Computational Cost**
Increasing with level l	$\text{Cost}_{\text{MLMC}} \approx \varepsilon^{-2} V_L C_L$
Decreasing with level l	$\text{Cost}_{\text{MLMC}} \approx \varepsilon^{-2} V_0 C_0$

In the first case, the finest level dominates as the cost per sample C_L grows faster than variance decays. Conversely, in the second case, the coarsest level dominates due to $C_0 \ll C_l$ and $V_0 \gg V_L$. Compared to the standard MC cost of approximately $\varepsilon^{-2} V_0 C_L$[1], the MLMC cost is reduced by a factor of V_L/V_0 and a potentially larger C_0/C_L when the finest and coarsest level dominates, respectively.

[1] Assuming that the cost of computing P_L is similar to the cost of computing $P_L - P_{L-1}$, and that $\text{Var}(P_L) \approx \text{Var}(P_0)$.

1.4.2 Accuracy and Convergence of Multilevel Monte Carlo

The performance of the Monte Carlo estimators are quantified in terms of the mean square error (MSE) or root mean square error (RMSE) deviation.

Definition 1.1. If Y approximates $\mathbb{E}[P]$, the RMSE deviation of the estimator P for the expected payoff $\mathbb{E}[P]$ is

$$\text{RMSE} \equiv \sqrt{\text{MSE}} \equiv \sqrt{\mathbb{E}[(Y - \mathbb{E}[P])^2]} = \sqrt{\mathbb{V}[Y] + (\mathbb{E}[Y] - \mathbb{E}[P])^2}, \tag{1.18}$$

where $\mathbb{V}[Y]$ is the variance of estimator Y, and $\text{bias}(Y)^2 \equiv (\mathbb{E}[Y] - \mathbb{E}[P])^2$ is the squared difference between the expected estimator and the true expected value.

The following theorem establishes the convergence and computational complexity under which the MLMC estimator achieves mean squared error ε^2 with bounded computational cost [15, 16].

Theorem 1.1. Let P denote a random variable, and let P_l denote the corresponding level-l numerical approximation. Suppose there exist independent estimators Y_l based on N_l Monte Carlo samples, each with expected cost C_l and variance V_l, and positive constants $\alpha, \beta, \gamma, c_1, c_2, c_3$ such that $\alpha \geq \frac{1}{2}\min(\beta, \gamma)$ and:

(i) $|\mathbb{E}[P_l - P]| \leq c_1 2^{-\alpha l}$ (Weak convergence)

(ii) $\mathbb{E}[Y_l] = \begin{cases} \mathbb{E}[P_0], & l = 0, \\ \mathbb{E}[P_l - P_{l-1}], & l > 0, \end{cases}$ (Estimator expectation)

(iii) $V_l \leq c_2 2^{-\beta l}$ (Variance decay)

(iv) $C_l \leq c_3 2^{\gamma l}$ (Cost growth),

then there exists a positive constant c_4 such that for any $\varepsilon < e^{-1}$ there are values of L and N_l for which the MLMC estimator

$$Y = \sum_{l=0}^{L} Y_l \tag{1.19}$$

has a MSE bound, $\text{MSE} \equiv \mathbb{E}[(Y - \mathbb{E}[P])^2] < \varepsilon^2$ with a computational cost C with bound

$$\mathbb{E}[C] \leq \begin{cases} c_4 \varepsilon^{-2}, & \beta > \gamma, \\ c_4 \varepsilon^{-2} (\log \varepsilon)^2, & \beta = \gamma, \\ c_4 \varepsilon^{-2-(\gamma-\beta)/\alpha}, & \beta < \gamma. \end{cases} \tag{1.20}$$

For a large class of payoff functions, such as the European vanilla option considered in this thesis, the theoretical convergence rates are $\alpha = 1$, $\beta = 1$, and $\gamma = 1$. These serve as a benchmark for interpreting the empirical convergence rates measured in Section 5.

1.4.3 Multilevel Monte Carlo Algorithm

The standard MLMC algorithm as described in Algorithm 1.2 allocates samples across levels to achieve a target RMSE ε. The algorithm initialises with a user-specified number of paths with $L_{\min}$ levels and estimates the variances V_l and costs C_l to compute optimal samples N_l at each level, based on equation (1.15) and (1.16).

To ensure that total error satisfies RMSE $\leq \varepsilon$, the algorithm bounds both variance $\mathbb{V}[Y]$ and squared bias $(\mathbb{E}[P_L - P])^2$ to be less than $\varepsilon^2/2$. Under the assumption $\mathbb{E}[P_l - P] \propto 2^{-\alpha l}$ from Theorem 1.1 and the weak convergence $\mathbb{E}[P - P_L] \leq \varepsilon/\sqrt{2}$, the bias at level L is approximated by

$$\mathbb{E}[P - P_L] = \sum_{l=L+1}^{\infty} \mathbb{E}[P_l - P_{l-1}] \approx \frac{\mathbb{E}[P_L - P_{L-1}]}{2^\alpha - 1}, \tag{1.21}$$

leading to the convergence test

$$\frac{\mathbb{E}[P_L - P_{L-1}]}{2^\alpha - 1} \leq \frac{\varepsilon}{\sqrt{2}}. \tag{1.22}$$

If this convergence test fails, a new level $L + 1$ will be added to reduce bias.

Algorithm 1.2 Multilevel Monte Carlo Algorithm, taken from [17]

```
start with $L = 2$, and initial target of $N_0$ samples on levels $l = 0, 1, 2$
while extra samples need to be evaluated do
    evaluate extra samples on each level
    compute/update estimates for $V_\ell$, $\ell = 0, \ldots, L$
    define optimal $N_\ell$, $\ell = 0, \ldots, L$ as described in (1.15) and (1.16)
    test for weak convergence as defined in (1.22)
    if not converged then
        $L \leftarrow L + 1$, and initialise target $N_L$
    end if
end while
```

1.4.4 Nested Multilevel Monte Carlo

While standard MLMC achieves significant gains, further cost reduction is achievable through a nested MLMC framework, in which the telescoping sum incorporates both half- and single-precision estimators. Specifically, each MLMC level combines FP16 and FP32 path simulations, introducing an inner correction term to address the rounding error bias from using reduced-precision arithmetic. This leads to the nested identity

$$\mathbb{E}[P] = \sum_{l=0}^{L} \Big(\mathbb{E}[\widehat{\Delta P_l}] + \mathbb{E}[\Delta P_l - \widehat{\Delta P_l}] \Big), \tag{1.23}$$

where the presence of hats denotes the payoff difference in half-precision, and the absence of hats denotes the same payoff difference in single-precision, such that

$$\widehat{\Delta P_l} = P_{\text{fp16}}(h_l) - P_{\text{fp16}}(h_{l-1}), \quad \Delta P_l = P_{\text{fp32}}(h_l) - P_{\text{fp32}}(h_{l-1}), \tag{1.24}$$

in which h_l corresponds to fine time steps, and h_{l-1} corresponds to coarse time steps, where $h_l = h_{l-1}\, 2^{-1}$.

This nested framework leads to the following unbiased estimator for $\mathbb{E}[P]$

$$\begin{aligned} &\sum_{l=0}^{L} \Big\{ \frac{1}{\widehat{N}_l} \sum_{i=1}^{\widehat{N}_l} \big[P_{\text{fp16}}(h_l) - P_{\text{fp16}}(h_{l-1})\big]^{(i)} \\ &+ \frac{1}{N_l} \sum_{j=1}^{N_l} \Big\{ [P_{\text{fp32}}(h_l) - P_{\text{fp32}}(h_{l-1})]^{(j)} - [P_{\text{fp16}}(h_l) - P_{\text{fp16}}(h_{l-1})]^{(j)} \Big\} \Big\}. \end{aligned} \tag{1.25}$$

where $\widehat{N}_l$ and $\widetilde{N}_l$ are the number of samples used for the half-precision estimator $\widehat{\Delta P_l}$ and the correction term $\Delta P_l - \widehat{\Delta P_l}$, respectively.

Defining $\widehat{C}_l$, $\widehat{V}_l$ to denote the cost and variance of one sample of $\widehat{\Delta P_l}$, and $\widetilde{C}_l$, $\widetilde{V}_l$ to denote the cost and variance of one sample of $\Delta P_l - \widehat{\Delta P_l}$, then the cost of the nested estimator is

$$\sum_{l=0}^{L} (\widehat{N}_l \widehat{C}_l + \widetilde{N}_l \widetilde{C}_l), \tag{1.26}$$

with an overall variance of

$$\sum_{l=0}^{L} \Big(\frac{\widehat{V}_l}{\widehat{N}_l} + \frac{\widetilde{V}_l}{\widetilde{N}_l} \Big). \tag{1.27}$$

Similarly to the standard MLMC case, the optimal sample allocations $\widehat{N}_l$ and N_l for the nested estimator can be determined by solving a constrained minimisation problem, where the objective is to minimise the total computational cost subject to a fixed upper bound ε^2 on the total variance. Using a Lagrange multiplier $\lambda_{\text{Nested}} \in \mathbb{R}^+$ gives $\widetilde{N}_l = \lambda_{\text{Nested}} \sqrt{\widetilde{V}_l / \widetilde{C}_l}$ and $\widehat{N}_l = \lambda_{\text{Nested}} \sqrt{\widehat{V}_l / \widehat{C}_l}$ for all levels l. Substituting the expressions back into the variance constraint requires that

$$\lambda_{\text{Nested}} = \epsilon^{-2} \sum_{l=0}^{L} \big(\sqrt{\widehat{V}_l \widehat{C}_l} + \sqrt{V_l C_l} \big), \tag{1.28}$$

and accordingly, the total computational cost is

$$\text{Cost}_{\text{Nested}} = \varepsilon^{-2}\Big(\sum_{l=0}^{L}\sqrt{\widehat{V}_l\widehat{C}_l} + \sqrt{\widetilde{V}_l\widetilde{C}_l}\Big)^2. \tag{1.29}$$

This cost expression closely resembles the total cost derived for the standard MLMC in Equation 1.17. However, it is important to note that the cost terms $\widetilde{C}_l$ and $\widehat{C}_l$ are redefined in the nested setting to reflect the use of single- and half-precision estimators, respectively. If $\widetilde{V}_l/\widehat{V}_l \ll \widehat{C}_l/\widetilde{C}_l \ll 1$, the nested MLMC framework yields a cost saving of a factor approximately $\max_l \widehat{C}_l/\widetilde{C}_l$ over the standard MLMC [19, 24]. This highlights the additional layer of computational efficiency from the use of mixed-precision arithmetic, particularly on hardware with native support for reduced-precision arithmetic.

1.5 Aim and Contributions of this Work

The aim of this project is to demonstrate the computational gains achieved through reduced-precision arithmetic in nested MLMC simulations, with a focus on financial derivative pricing. Specifically, we develop a framework combining FP16 coarse-level computations with FP32 – FP16 correction terms to maintain estimator consistency. In particular, this project exploits AVX-512's native FP16 support to avoid reliance on specialised hardware, such as FPGAs, while preserving the MLMC telescoping sum's convergence properties.

The primary contributions of this work can be summarised as follows. Firstly, optimised Gaussian sampling techniques were developed for AVX-512 compatibility by adapting and refining three piecewise polynomial approximations of the inverse cumulative distribution function (CDF). The proposed implementations consistently outperformed Intel's classic Box–Muller method, came close to the performance of Intel's optimised ICDF generator (while still falling slightly behind), and, most importantly, achieved what neither method offered: true hierarchical efficiency in FP16 over FP32 through native vectorised Gaussian sampling, thereby avoiding the performance penalty associated with Intel MKL's FP32-to-FP16 downcasting.

Secondly, a cost model was derived and validated to quantify the computational cost of payoff path calculations, isolating the expenses of random number generation (RNG) from the Euler-Maruyama update and payoff logic. This model provide the essential parameters for optimising sample allocations across levels in the MLMC estimator and precisely quantifying the speedups achieved through reduced precision.

Thirdly, these methods were integrated into a nested MLMC estimator to incorporate reduced-precision arithmetic, with correction terms $\Delta P_l - \widehat{\Delta P_l}$ effectively compensating for FP16 rounding errors and discretisation bias. Finally, the work was extended to an adaptive-precision MLMC scheme, where the precision level of the estimator was dynamically selected across MLMC levels to address the accumulative half-precision rounding error at fine levels that compromise convergence. Together, these extensions demonstrate how reduced and adaptive-precision can be combined with MLMC to yield substantial efficiency gains. The top-level MLMC code was provided by Prof. Giles as a baseline implementation; the contributions of this work lies in extending this implementation with reduced-precision Gaussian sampling, SIMD vectorisation, and adaptive-precision nested MLMC schemes[2].

This work builds upon prior mixed-precision MLMC research [24,38] by integrating precision-aware numerics with hardware-specific vectorisation, achieving measurable speedups while maintaining the accuracy required for financial derivative pricing and similar SDE applications.

[2] All code implementations are available at:
`https://github.com/jasontaykq/Nested_MLMC_Reduced_Precision`.

Chapter 2

Approximating the Gaussian Distribution

A fundamental principle underlying most simulation methods, such as Monte Carlo, is the generation of random variables from specified distributions by transforming uniformly distributed random numbers [35]. In this thesis, the efficient and accurate generation of Gaussian random numbers, particularly the inverse Gaussian CDF (cumulative distribution function) $\Phi^{-1}(\cdot)$, is crucial as they drive the Brownian motion increments required by the Euler-Maruyama scheme for simulating the SDE of the payoff function (1.5).

One such approach is the inverse transform method [22]. Let $\phi(\cdot)$ denote the probability density function (PDF) of the standard Gaussian distribution $X \sim \mathcal{N}(0, 1^2)$, and $\Phi(\cdot)$ its CDF. Given a uniform random variable $u \sim \mathcal{U}(0, 1)$, the standard approach is to transform u by applying the inverse CDF $\Phi^{-1}(\cdot)$, such that $z = \Phi^{-1}(u)$ follows the desired Gaussian distribution, as illustrated in Figure 2.1.

Summarised in [38], benefits of the inverse transform approach include: the efficient generation of Gaussian random variables from a single uniform sample without rejection (crucial for maintaining the low-discrepancy properties of quasi-Monte Carlo), and the applicability to a wide range of distributions beyond the Gaussian while being cheap to compute [40]. Moreover, unlike classical computational methods such as Box-Muller [37], the inverse transform method avoids transcendental functions, making it suitable for FP16 hardware, which lacks native support and is prone to underflow, overflow, and rounding errors when such functions are typically used.

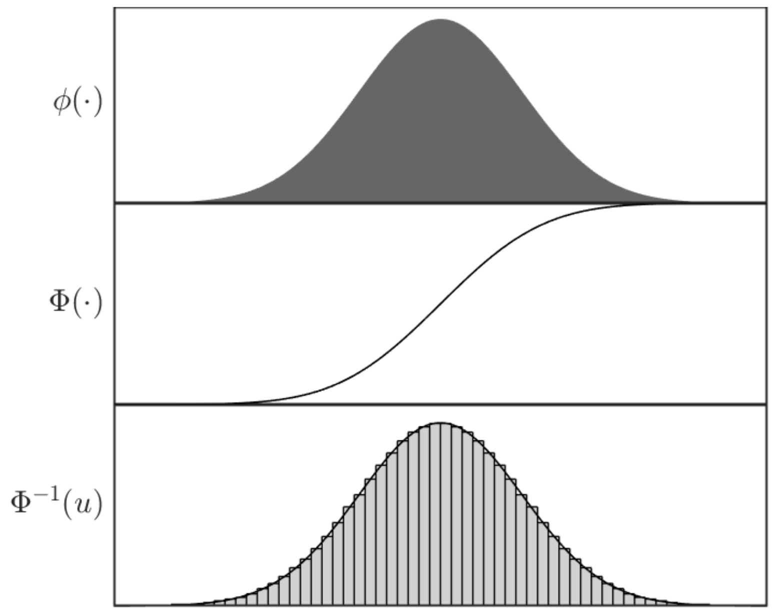


Figure 2.1: A numerical demonstration of the convergence in distribution of the inverse transform method in the standard Gaussian distribution. Adapted from [38].

.

2.1 The Inverse Gaussian Cumulative Distribution Function

The key idea behind inverse transform methods is to approximate the inverse CDF of the Gaussian distribution to a uniform random variable as an input, to output a Gaussian random variable. For the standard Gaussian distribution, the CDF is

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2}\, dt \quad \forall x \in \mathbb{R}. \tag{2.1}$$

The corresponding inverse $\Phi^{-1}(x)$ has the analytic expression

$$\Phi^{-1}(u) = \sqrt{2}\,\mathrm{erf}^{-1}\,(2u-1) \quad \forall u \in (0,1), \tag{2.2}$$

where $\mathrm{erf}\,(z)$ is the error function defined as

$$\mathrm{erf}\,(z) = \frac{2}{\sqrt{\pi}} \int_{0}^{z} e^{-t^2}\, dt. \tag{2.3}$$

In practice, direct simulation using the analytic expression for $\Phi^{-1}(u)$ is computationally expensive and often unsuitable for large-scale or reduced-precision applications, further motivating the need for efficient numerical approximations. Figure 2.2

illustrates the behaviour of the standard Gaussian inverse CDF, $\Phi^{-1}(u)$. The function is symmetric about the point $(0, 1/2)$ and is highly nonlinear near the boundaries $x \to 0$ and $x \to 1$, where it tends toward $-\infty$ and $+\infty$, respectively. As such, this thesis approximates the inverse CDF in $[0, 1/2]$ using a piecewise polynomial function with coefficients stored in a lookup table (LUT), and uses a random bit to reflect the result for $u \in [0.5, 1]$, exploiting symmetry and effectively halving the LUT size to reduce computational cost.

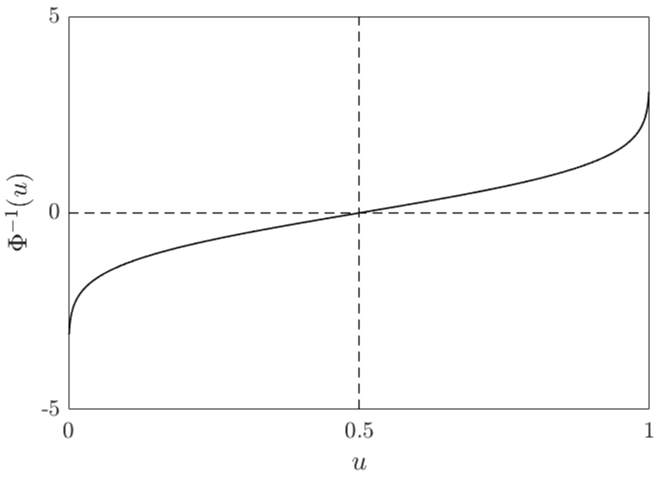
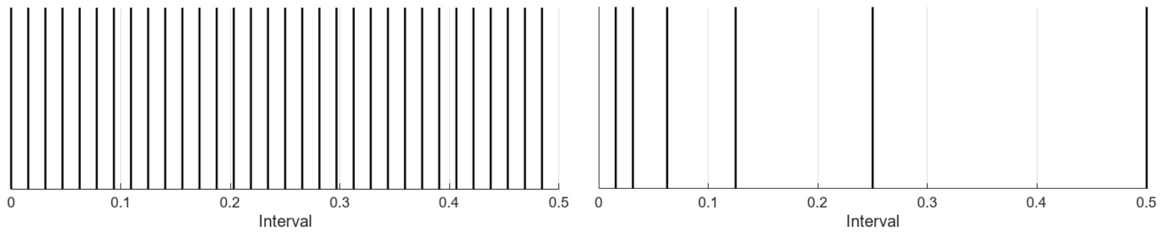


Figure 2.2: Plot of the standard Gaussian inverse CDF $\Phi^{-1}(u)$.

As described in [24], there are several possibilities to segmenting the interval $[0, 1/2]$. The simplest way is to use uniform intervals as shown in Figure 2.3a, but the singularity and highly nonlinear nature at and near the boundary $x = 0$ lead to the need for a large LUT to achieve high accuracy. To address this, [7, 31] proposed hierarchical and non-uniform segmentations that map more closely to the CDF's shape. Such segmentation includes the dyadic interval, where the interval is recursively halved when tending towards $u = 0$ as illustrated in Figure 2.3b.

The following subsections summarise three methods for approximating $\Phi^{-1}(u)$: (1) a piecewise constant approximation on uniform intervals, (2) a piecewise linear approximation on dyadic intervals, and (3) a piecewise linear approximation on superdyadic intervals. Across all methods, the relevant coefficients and representative values for each approximation are precomputed and stored in a LUT to enable fast and efficient evaluation during simulation. The mathematical foundations and derivations of the LUT methods in Sections 2.2 and 2.3 follow the work of [24] closely.



(a) Uniform interval (b) Dyadic interval

Figure 2.3: Illustration of interval segmentations on [0,0.5].

2.2 Piecewise Constant Approximation on Uniform Intervals (Method 1)

The first approximation method constructs a LUT of constant values Z_j that approximate the inverse Gaussian CDF $\Phi^{-1}(u)$ over a given uniformly partitioned interval $\mathcal{I}_j = [u_j, u_{j+1}] \subset [0, 1/2]$. Specifically, the domain $[0, 1/2]$ is divided into 2^{d-1} equal-width intervals, where each grid point is given by $u_j = 2^{-d}j$ for $j = 0, 1, \ldots, 2^{d-1} - 1$, where j indexes the interval. Here, d specifies the number of address bits used in the LUT, determining the total number of intervals and, subsequently, the resolution of the approximation. For a given uniform variable $u \in [0, 1/2]$, the piecewise constant method selects the corresponding interval $\mathcal{I}_j = [u_j, u_{j+1}]$ and assigns Z_j as the approximate value of $\Phi^{-1}(u)$ throughout the interval $\mathcal{I}_j$.

To determine the optimal constant Z_j at each interval, the mean squared error (MSE) between the true inverse CDF $\Phi^{-1}(u)$ and its constant approximation Z_j on $\mathcal{I}_j$ must be minimised:

$$\mathrm{MSE}(Z_j) = \int_{u_j}^{u_{j+1}} \left(Z_j - \Phi^{-1}(u)\right)^2 \, du. \tag{2.4}$$

Differentiating with respect to Z_j and setting the result to zero yields

$$\frac{d}{dZ_j}\mathrm{MSE}(Z_j) = 2\int_{u_j}^{u_{j+1}} (Z_j - \Phi^{-1}(u)) \, du = 0, \tag{2.5}$$

which leads to

$$Z_j = \frac{1}{u_{j+1} - u_j} \int_{u_j}^{u_{j+1}} \Phi^{-1}(u) \, du. \tag{2.6}$$

For a uniform partition $u_{j+1} - u_j = 2^{-d}$, the optimal constant simplifies to

$$Z_j = 2^d \int_{u_j}^{u_{j+1}} \Phi^{-1}(u) \, du. \tag{2.7}$$

In this thesis, the constant values Z_j stored in the LUT are computed as the average of $\Phi^{-1}(u)$ over each interval $\mathcal{I}_j = [u_j, u_{j+1}]$, effectively minimising the (MSE) within each interval. The resulting approximation and associated error plots are shown in Figure 2.4 and Figure B.1[3].

As noted previously, the highly nonlinear nature of $\Phi^{-1}(u)$ near the boundaries where the function tends toward $\pm\infty$ leads to the largest approximation error in this region. Conversely, the approximation improves to acceptable error ranges near $u = 0.5$, where the function becomes nearly linear. While simple to implement, this uniform method is computationally expensive due to its inefficient use of intervals, especially in nonlinear regions. This motivates the adaptive segmentation strategies, such as dyadic intervals, introduced next.

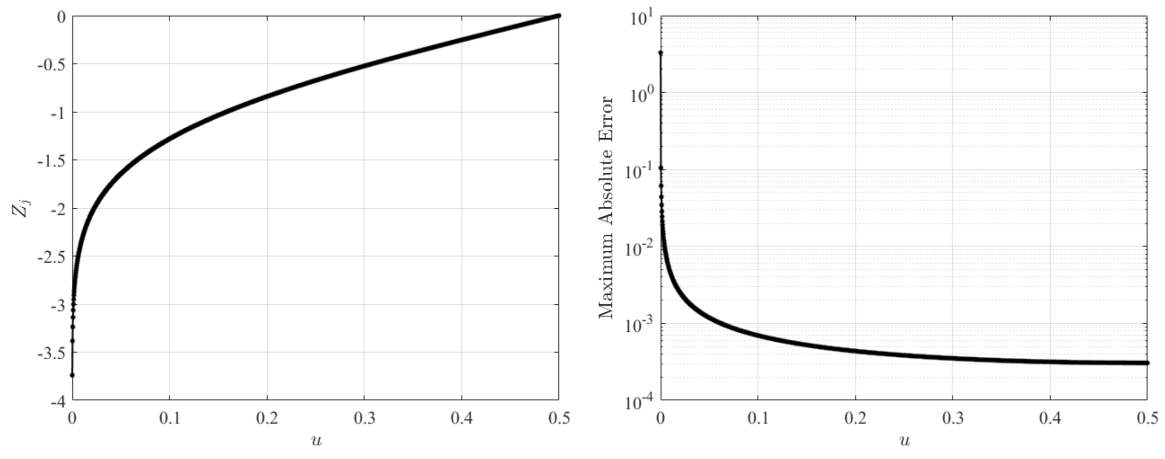


Figure 2.4: **(Left)** Piecewise constant approximation of $\Phi^{-1}(u)$ on uniform intervals. **(Right)** Maximum absolute error for the piecewise constant approximation of $\Phi^{-1}(u)$ on uniform intervals.

2.3 Piecewise Linear Approximation on Dyadic Intervals (Method 2)

In this method, the domain $[0, 1/2]$ is recursively divided into dyadic intervals, in which each interval is recursively halved as the boundary $u = 0$ is approached to best capture the regions where $\Phi^{-1}(u)$ varies most rapidly. The interval containing the point indexed by j is identified by the leading bit i of j and contains the points indexed by the integers $[2^{i-1}, 2^i - 1]$. In binary representation, the dyadic block B_i,

[3] Simulations in Figure 2.4 and B.1 uses $d = 12$, resulting in $2^{11} = 4096$ uniform intervals.

referring to a single dyadic interval, is the set of those j whose binary representation has leading bit i, so that each such interval corresponds to the dyadic grid points $u \in [2^{i-1} \cdot 2^{-d}, 2^{i} \cdot 2^{-d}]$ for $i = 1, 2, \ldots, d-1$.

Each dyadic interval is approximated by a linear polynomial $\tilde{Z}_j = a_j + b_j(u - u_j)$, where $u \in [u_j, u_{j+1}]$ is the uniform variable, and (a_j, b_j) are the optimal interval-specific coefficients. This is the same approach used in the piecewise constant approximation where the resulting coefficients are stored in a LUT of size $2(d-1)$, reflecting the need to store both the intercept and slope for each interval. Decomposing the error in terms of the optimal constant approximation Z_j (as defined in the previous section), gives

$$\text{MSE}(\tilde{Z}_j) = \int_{u_j}^{u_{j+1}} \left(\tilde{Z}_j - Z_j\right)^2 du + \int_{u_j}^{u_{j+1}} \left(Z_j - \Phi^{-1}(u)\right)^2 du, \tag{2.8}$$

where the first term captures the additional error due to the linear fit beyond the optimal constant fit, while the second term is the irreducible error from using the optimal constant alone. This effectively shows that, for a fixed value of d, the approximation error of the dyadic scheme will always exceed that of the piecewise constant approximation Z_j on uniform intervals. However, the dyadic implementation requires exponentially fewer intervals, resulting in a much smaller LUT. A comparative analysis between memory efficiency and approximation accuracy for the different methods will be detailed in Table 2.2.

Noting that the second term on the right-hand side (RHS) of Equation (2.8) is independent of the linear fit parameters and therefore does not affect the minimisation with respect to (a, b). Thus, the minimisation problem reduces to finding the least-squares fit of the linear function to the optimal constant across all intervals, in which the optimal optimal coefficients are determined

$$\arg\min_{a,b} \int_{u_j}^{u_{j+1}} \left(\tilde{Z}_j - Z_j\right)^2 du. \tag{2.9}$$

Equivalently, over all dyadic intervals, the total error introduced by the linear approximation can be minimised as

$$\sum_{j=1}^{2^{d-1}} \left(\tilde{Z}_j - Z_j\right)^2. \tag{2.10}$$

The resulting piecewise linear function and approximation error are shown in Figures 2.5 and B.2[4], respectively. The signed error plot exhibits an oscillatory behaviour, with the amplitude of the error dampening as $u \to 0.5$. This is consistent

[4] Simulations in Figure 2.5 and B.2 uses $d = 12$, resulting in 11 dyadic intervals.

with the increasing linearity of $\Phi^{-1}(u)$ in this region, allowing the linear approximation to closely track the inverse CDF. The observed peaks in the error correspond to the dyadic grid points, which reflects maximum deviation between the linear fit and $\Phi^{-1}(u)$ at these point. In contrast, the local minimums typically occur at the midpoints of each interval, where the linear approximation achieves the best fit.

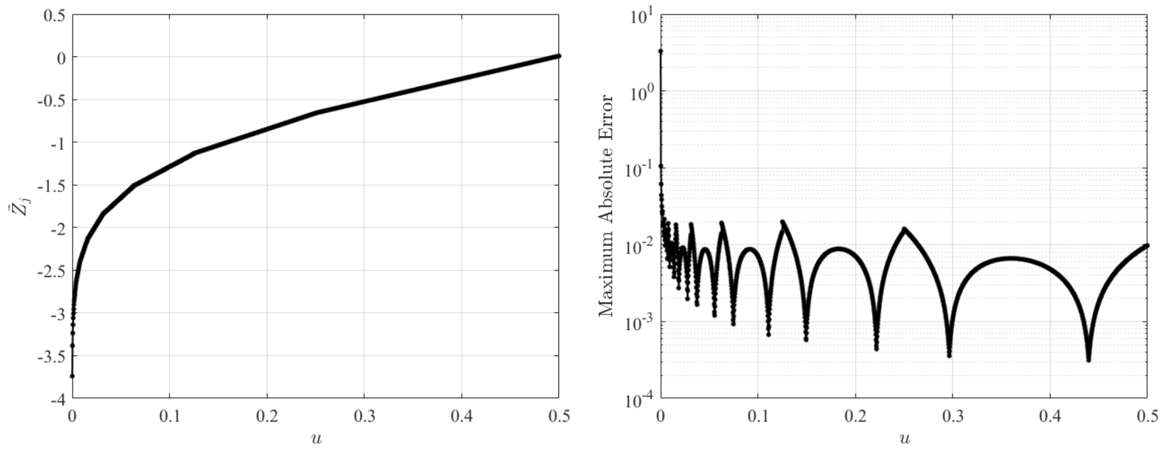


Figure 2.5: **(Left)** Piecewise linear approximation of $\Phi^{-1}(u)$ on dyadic intervals. **(Right)** Maximum absolute error for the piecewise linear approximation of $\Phi^{-1}(u)$ on dyadic intervals.

2.4 Piecewise Linear Approximation on Superdyadic Intervals (Method 3)

This method applies the same local linear approximation scheme as in the dyadic case, but partitions the domain using superdyadic intervals to provide finer granularity. Specifically, the interval length is reduced by a geometric factor of $\sqrt{2}$ at each refinement, such that

$$\Delta u_i = (\sqrt{2})^{-i} = 2^{-i/2}, \quad u \in \left[2^{(-d+i-1)/2},\ 2^{(-d+i)/2}\right], \quad i = 1, 2, \ldots, d-1. \tag{2.11}$$

This segmentation yields twice as many intervals as the standard dyadic scheme for a given d, allocating greater resolution to the highly nonlinear regions near $u = 0$.

As before, each interval is approximated by a linear polynomial

$$\tilde{Z}_j = a_j + b_j(u - u_j), \quad u \in [u_j, u_{j+1}], \tag{2.12}$$

where the coefficients (a_j, b_j) are precomputed to minimise the mean squared error within the interval. The LUT therefore requires $4(d-1)$ entries in total, reflecting the need to store two coefficients per interval across $2(d-1)$ superdyadic intervals.

The resulting quality of the superdyadic approximation is illustrated in Figures 2.6 and B.3, which show the piecewise linear fit and its associated signed error for $d = 12$, corresponding to 22 superdyadic intervals.

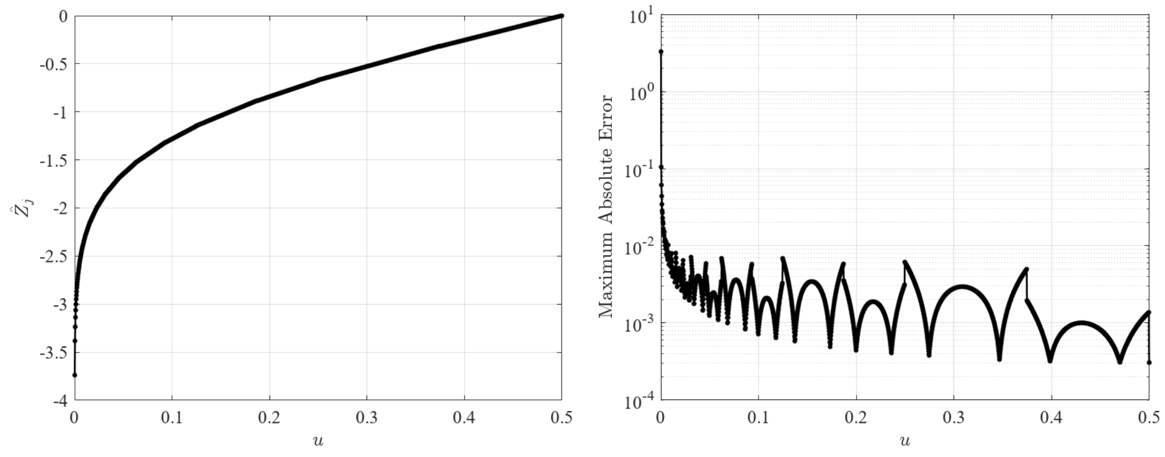


Figure 2.6: **(Left)** Piecewise linear approximation of $\Phi^{-1}(u)$ on superdyadic intervals. **(Right)** Maximum absolute error for the piecewise linear approximation of $\Phi^{-1}(u)$ on superdyadic intervals.

2.5 Comparison of the Three Inversion Methods

2.5.1 Storage and Computational Cost

Table 2.1 summarises the storage requirements and evaluation of the three inversion methods. The uniform piecewise constant method has the simplest evaluation rule, requiring only a direct table lookup with complexity $\mathcal{O}(1)$, independent of bit-width d. However, a trade-off for this simple runtime cost is the need for larger LUT sizes, which grows exponentially with bit-width d, specifically 2^{d-1}. This makes the uniform piecewise constant method memory intensive when high accuracy is required, as the exponential growth of the LUT size rapidly increases memory usage and cache demands.

In contrast, the dyadic and superdyadic methods use significantly smaller LUTs that grow only linearly with d. For dyadic segmentation, the LUT requires $2(d-1)$ entries, while the superdyadic requires $4(d-1)$. This represents a significant reduction in storage cost compared to the uniform method, particularly at large values of

d. However, the trade-off behind these adaptive schemes is the slightly more complex evaluation cost. With the dyadic and superdyadic segmentations, the interval index can be obtained in $\mathcal{O}(1)$ by decoding the exponent of the floating-point representation of u (and equivalently, using $j = \log_2(u)$ when $u \in (0, 1/2)$). The per-sample evaluation then reduces to a small number of operations: load the optimal gather and slope coefficients (a_j, b_j) for the interval and perform one fused multiply-add (FMA) for the linear interpolation. Thus, the evaluation cost is theoretically similar to the piecewise constant at $\mathcal{O}(1)$, independent of d, with a small fixed arithmetic overhead for interpolation. This exponent-bit indexing is inspired by [38].

Table 2.1: Storage and evaluation cost on CPUs for the approximate methods.

Approximation method	**LUT size**	**Evaluation cost**
Piecewise Constant-Uniform	2^{d-1}	Look-up: $\mathcal{O}(1)$
Piecewise Linear-Dyadic	$2(d-1)$	Exponent-bit index + 1 FMA: $\mathcal{O}(1)$
Piecewise Linear-Superdyadic	$4(d-1)$	Exponent-bit index + 1 FMA: $\mathcal{O}(1)$

In summary, the uniform method is suitable in contexts where arithmetic cost dominates and memory is abundant. In contrast, the dyadic and superdyadic schemes offer a better balance, achieving a constant-cost $\mathcal{O}(1)$ evaluation through bit-level indexing. The superdyadic scheme generally yields higher accuracy, since its segmentation is based on a $\sqrt{2}$-scaled geometric series as opposed to the standard factor-2 series used in dyadic schemes. Thus, this results in a slightly more complex exponent-mantissa test when compared to the dyadic implementation.

2.5.2 Approximation Accuracy and Scaling

While storage and evaluation cost determine computational feasibility, the accuracy of the approximation is the criterion that ultimately determines the suitability of each method. Table 2.2 provides a quantitative comparison of the three methods: reporting MSE, maximal approximated values at $u \to 0$, and LUT sizes across varying bit-width d.

Recalling equation (2.8), the MSE of the dyadic and superdyadic piecewise linear approximations is always bounded below by that of the piecewise Constant-Uniform. Despite this, the advantage of the Constant-Uniform method is not always obvious. At $d = 10$, the differences in MSE across all three methods are slight, having only clear distinctions at $d \geq 12$, where the Constant-Uniform method requires exponentially larger LUTs. For example, at $d = 12$ the Constant-Uniform scheme achieves an MSE of 3.13×10^{-5} but at the cost of 2048 entries, whereas the dyadic scheme attains a comparable MSE of 6.23×10^{-5}, nearly twice as large, but only needs 22 entries. The superdyadic scheme, with 44 entries, reaches 3.48×10^{-5}, nearly matching the piecewise constant's accuracy at a significantly smaller LUT size. At $d = 14$, the gap between the methods widens significantly, with the uniform method requiring 8192 entries, compared to just 26 for dyadic and 52 for superdyadic, representing two to three orders of magnitude greater memory efficiency for the adaptive schemes.

The "maximal value" column of Table 2.2 highlights the ability of each scheme to approximate the tail behaviour of $\Phi^{-1}(u)$ as $u \to 0$ or 1, corresponding to the two tails of the Gaussian distribution. Similar to the trends observed thus far, the Constant-Uniform scheme requires a large LUT to achieve extreme tail values, whereas both dyadic and superdyadic methods reproduce the exact same tail values using LUTs that are two or three orders of magnitude smaller.

Table 2.2: Comparison of MSE, maximal values at $u \to 0$, and LUT sizes for the approximation methods across varying bit-width d.

Method	$\boldsymbol{d}$	**MSE**	**Maximal value**	**LUT size**
Piecewise Constant-Uniform	10	1.50×10^{-4}	± 3.18	512
Piecewise Constant-Uniform	12	3.13×10^{-5}	± 3.37	2048
Piecewise Constant-Uniform	14	6.75×10^{-6}	± 4.08	8192
Piecewise Linear-Dyadic	10	1.91×10^{-4}	± 3.37	18
Piecewise Linear-Dyadic	12	7.26×10^{-5}	± 3.74	22
Piecewise Linear-Dyadic	14	4.81×10^{-5}	± 4.08	26
Piecewise Linear-Superdyadic	10	1.53×10^{-4}	± 3.37	36
Piecewise Linear-Superdyadic	12	3.48×10^{-5}	± 3.74	44
Piecewise Linear-Superdyadic	14	1.01×10^{-5}	± 4.08	52

Figure 2.7 illustrates how MSE decays as d increases for the three approximation methods. Paired with the convergence analysis from [20], it is observed that as $d \to \infty$, the Constant-Uniform method gives MSE $\to 0$, since the interval length tends to zero and each segment reproduces the true value exactly. In contrast, the dyadic segmentation converges to a nonzero error, MSE $\to C$, because only the interval nearest to the singularity at $u = 0$ benefits from refinement as d increases, while the error contribution from the other intervals remains essentially fixed. The superdyadic method improves this by allocating more intervals near $u = 0$, thereby

reducing the constant C relative to the dyadic case, while also not converging to zero.

This explains the plateauing of the dyadic and superdyadic curves in Figure 2.7, while the uniform scheme continues to decline. In practice, however, the exponential LUT growth of the uniform method makes it infeasible for large d, and the adaptive methods remain more efficient for moderate values of d. These scaling properties provide the motivation for the comparative benchmarking in Chapter 3, where the practical efficiency of the different Gaussian generation methods is assessed on modern Intel hardware.

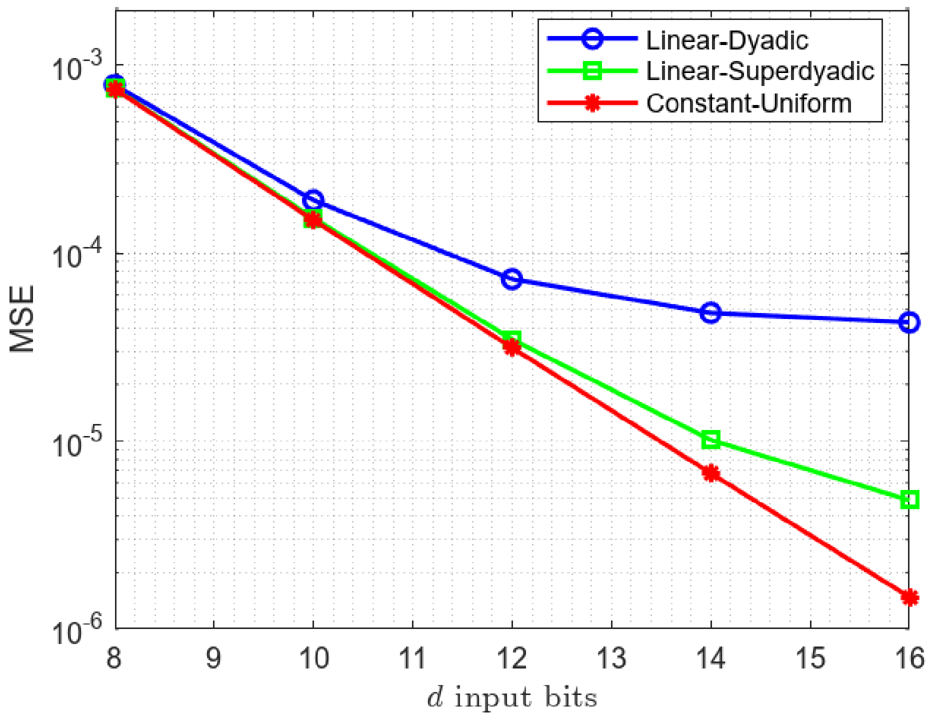


Figure 2.7: MSE for methods 1, 2 and 3 over the bit-width d of the random integer j. Adapted from [24].

Chapter 3

Gaussian Random Number Generation Algorithm

3.1 LUT Generation and Lookup Procedure

In this thesis, the LUT coefficients used for the Gaussian sample generation are precomputed offline in MATLAB, but can be generated in any suitable computational platform. The MATLAB code calculates the optimal piecewise constant, Z_j, or piecewise linear, (a_j, b_j), coefficients for the inverse Gaussian CDF over the relevant interval and grid point. These coefficients were then exported as single-precision (FP32) and half-precision (FP16) values in C++ header files. This approach ensures that the computationally expensive approximation calculations were performed only once offline, while the runtime code required only fast array lookups.

At runtime, uniform random variables $u \in [0, 1]$ in FP32 were generated using Intel MKL's `vRngUniform` [27]. FP16 uniforms were avoided to ensure consistent coupling between precision levels and to prevent discretisation errors that could arise from the coarser resolution of FP16 numbers mapping to different LUT bins, which would break the correlation essential for effective variance reduction.

These uniform variables are then mapped to Gaussian-distributed samples following the lookup procedure for the piecewise constant approximation outlined in Algorithm 3.1. To avoid numerical instability and index out-of-bound errors near the tails of the distribution (as the Gaussian CDF tends to $-\infty$ and $+\infty$ near 0 and 1), the uniform variable u is clamped within the interval $[10^{-7}, 1 - 10^{-7}]$. To exploit the symmetry of the Gaussian distribution, samples with $u > 0.5$ are reflected about the point $(0.5, 0)$, and the sign of the output is inverted accordingly. This effectively reduces the LUT size by half with a small computational overhead.

Algorithm 3.1 Gaussian sample generation for piecewise constant approximation

```
Input: Uniform random number $u \in [0, 1]$ (in FP32), LUT coefficients array LUT
Output: Gaussian sample $z_{\texttt{precision}}$ ($\texttt{precision}$ = FP16 or FP32)
  $N \leftarrow \text{length}(\texttt{LUT})$
  Clamp $u$ within $[10^{-7}, 1 - 10^{-7}]$ to avoid tails
  if $u \leq 0.5$ then
      idx $\leftarrow [u \times 2 \times N]$
      if idx $\geq N$ then idx $\leftarrow N - 1$
      $z \leftarrow \texttt{LUT}[\text{idx}]$
  else
      idx $\leftarrow [(1 - u) \times 2 \times N]$
      if idx $\geq N$ then idx $\leftarrow N - 1$
      $z_{\texttt{precision}} \leftarrow -\texttt{LUT}[\text{idx}]$
  end if
```

For the piecewise linear approximation, uniform random variables are generated as 16-bit unsigned integers $u \in [0, 65535]$. This integer representation is important for the piecewise linear approximation method, as it allows for the direct extraction of the sign bit and magnitude bits through efficient bit-level operations, avoiding the rounding issues that would occur with FP32 inputs[5].

The algorithm, described in Algorithm 3.2, begins by decomposing the input integer u into a sign bit and a 15-bit magnitude. The magnitude bits are then reinterpreted directly as a floating-point value u_0 in the desired target precision (FP16 or FP32), which efficiently maps the integer input into the floating-point domain while preserving its proportional value. Similarly to the piecewise constant, u_0 is also clamped to avoid tails, but now within the interval $[2^{-14}, 0.5]$, where 2^{-14} is the smallest positive FP16 normal.

To summarise, Algorithm 3.1 outlines the lookup procedure for piecewise constant approximations, offering simplicity and fast indexing, as it directly maps uniform samples to discrete LUT values without interpolation. In contrast, the piecewise linear approximation interpolates within the dyadic or superdyadic segments and selecting the segment in $\mathcal{O}(1)$ through exponent-bit indexing, which replaces the

[5] It is not advisable to use FP32 uniform inputs for the piecewise linear approximations due to the rounding errors introduced during the initial conversion and index calculation steps, which can break the coupling between precision levels even when coefficients are extracted from the correct intervals.

$\mathcal{O}(d)$ linear search (described in Algorithm C.1) and reduces the per-sample cost to a single FMA interpolation $A[\texttt{idx}] + B[\texttt{idx}] \times u_0$. The algorithm and evaluation cost for this exponent-bit indexing method, inspired by [38], is summarised in Algorithm 3.2 and Table 2.1, respectively.

Algorithm 3.2 Gaussian sample generation for piecewise linear approximation

```
Input: 16-bit unsigned integer $u_{\text{int}} \in [0, 65535]$, LUT coefficients arrays A, B
Output: Gaussian sample $z_{\texttt{precision}}$ (precision $=$ FP16 or FP32)
Constant: FP16 Exponent Bias=15, FP32 Exponent Bias=127
  $N \leftarrow \text{length}(\texttt{A}) - 1$
  Extract sign bit: sgn_bit $\leftarrow (u_{\text{int}} \gg 15)\&1$
  Extract magnitude bit: mag_bits $\leftarrow u_{\text{int}}\&$0x7FFF
  if precision $=$ FP16 then
      $u_0 \leftarrow$ reinterpret mag_bits as _Float16
  else
      $u_0 \leftarrow$ convert mag_bits from FP16 to float
  end if
  Clamp:
  $u_0 \leftarrow \max(u_0, 6.1035 \times 10^{-5})$
  $u_0 \leftarrow \min(u_0, 0.5)$
  if precision $=$ FP32 then
      bits $\leftarrow$ reinterpret $u_0$ as uint32
      $e \leftarrow (\texttt{bits} \gg 23)\&$0xFF                        ▷ Exponent-bit indexing
      idx $\leftarrow (N + 2) - (\texttt{FP32 Exponent Bias} - e)$
  else
      bits $\leftarrow$ reinterpret $u_0$ as uint16
      $e \leftarrow (\texttt{bits} \gg 10)\&$0x1F                        ▷ Exponent-bit indexing
      idx $\leftarrow (N + 2) - (\texttt{FP16 Exponent Bias} - e)$
  end if
  idx $\leftarrow$ clamp(idx, 0, $N$)
  $z \leftarrow \texttt{A}[\texttt{idx}] + \texttt{B}[\texttt{idx}] \times u_0$
  if sgn_bit $= 1$ then
      $z_{\texttt{precision}} \leftarrow -z$
  end if
```

3.2 Implementing SIMD Vectorisation

This section describes SIMD implementations of the LUT-based Gaussian generators. The logic follows Section 1.1: AVX-512 processes 16 lanes per iteration in FP32 and

32 lanes in FP16, enabling the expected hierarchical speedup of the FP16 Gaussian generator over FP32.

3.2.1 Piecewise Constant Approximation

The SIMD implementation of the piecewise-constant method follows Algorithm 3.1 using AVX-512 intrinsics. For each vector block of uniforms, values are first clamped to $[10^{-7}, 1 - 10^{-7}]$ to avoid tails. A lane-wise mask identifies $u < 0.5$ and a masked blend forms the mirrored inputs $\tilde{u} = \min(u, 1 - u)$. Integer indices are produced by scaling with $2N_{\text{LUT}}$, truncating (taking the floor) to integers and clamping to $[0, N_{\text{LUT}} - 1]$. The corresponding constants are then gathered from the LUT, and the sign is restored using the original mask.

In FP32, 16 lanes are processed per vector loop, with LUT entries fetched by Intel's 32-bit gather. In FP16, the loop produces 32 results per iteration and preserves coupling with the FP32 generator by using FP32 uniforms for binning and sign decisions[6], avoiding the extra quantisation errors that direct FP16 uniforms would introduce. Because there is no native 16-bit gather, the FP16 LUT is stored in a duplicated 32-bit layout (each 16-bit value replicated into a 32-bit word), and values are retrieved using Prof. Giles's emulated FP16 gather routine [18]. This routine permutes packed 16-bit indices into 32-bit gather lanes, gathers against the duplicated table, and reinterprets the results back into FP16 form.

For completeness, a scalar cleanup loop is implemented to handle any remaining samples that are not divisible by the vector width, following the same steps. Table D.1 maps the steps of Algorithm 3.1 to the AVX-512 intrinsics used in this implementation.

3.2.2 Piecewise Linear Approximation (Dyadic and Superdyadic)

The AVX-512 implementations for the piecewise linear approximations use the exponent-bit indexing trick in Algorithm 3.2, adapted from [38]. Both the dyadic and superdyadic implementations take 16-bit unsigned integers as input, where the most significant bit encodes the sign and the remaining 15 bits represent a half-precision magnitude in $[0, 0.5]$.

For each vector block, the sign bits are extracted and stored for later restoration. The magnitude bits are reinterpreted as half-precision values, converted to the target

[6] Other input types, e.g. 32-bit integers, are also suitable. However, usage of FP16 is not recommended as it only adds quantisation error and breaks coupling without offering any speed gain.

precision, and clamped to the safe interval $[2^{-14}, 0.5]$ to avoid subnormals and ensure correct exponent extraction. This ensures that the coupling between FP16 and FP32 samples is preserved. The exponent bits are then obtained by reinterpreting the floating-point value as an integer.

The key distinction between the dyadic and superdyadic implementations is the indexing rule. The dyadic method uses an exponent-only selection

$$\mathrm{idx}_{\mathrm{dyadic}} \;=\; \mathrm{clamp}\big((N+2)-(b-e),\ 0,\ N\big),$$

where N is the maximum valid index, b is the exponent bias, and e is the extracted exponent. The superdyadic method refines each dyadic bin by splitting it at the mantissa threshold corresponding to the $\sqrt{2}$ geometric series:

$$\mathrm{idx}_{\mathrm{superdyadic}} \;=\; \mathrm{clamp}\big(2j+\mathbb{I}(m \geq m_{\sqrt{2}}),\ 0,\ N\big),$$

where $j \in [0, J-1]$ is the dyadic exponent index (with J the number of dyadic bins underlying the superdyadic scheme, typically $J = \lceil N/2 \rceil$), m is the mantissa of u's normalized significand, $m_{\sqrt{2}}$ is the corresponding mantissa for $\sqrt{2}$, and $\mathbb{I}$ is the indicator function [6]. The indicator function for a set of real number A is defined as

$$\mathbb{I}_A(x) = \mathbb{I}\{x \in A\} = \begin{cases} 1 & \text{for } x \in A, \\ 0 & \text{for } x \notin A. \end{cases} \tag{3.1}$$

The precomputed coefficients (A, B) are stored in aligned arrays. The methods use efficient permutes instead of expensive gathers for coefficient selection, evaluate $z = A + B\,u$ with a single FMA, and restore the sign by flipping the sign bit where the input's sign bit was set.

Overall, the dyadic scheme will be faster due to a smaller instruction count (no mantissa threshold test) than the superdyadic scheme. Tables D.2 and D.3 map the steps of Algorithm 3.2 to the AVX-512 intrinsics used in this implementation.

3.3 Benchmarking Methodology

This section outlines the experimental framework established to assess the computational performance of the three Gaussian sample generation methods presented in Chapter 2. The framework considers implementations in both single-precision (FP32) and half-precision (FP16), benchmarking them against Intel MKL's (Math Kernel Library) `vRngGaussian` [10], a highly optimised Gaussian generator function.

3.3.1 Benchmark Setup

All benchmarks were conducted on an Intel Xeon Gold 6538Y+ CPU which supports native AVX-512 vector instructions including half-precision floating point operations [9,12], albeit lacking operator overloading for syntactic sugar. The benchmark codebase consists entirely of C++ files compiled using the Intel compiler `icpx` [13,28], alongside the libraries and flags shown in Listing 3.1.

Listing 3.1: Libraries and compiler flags used in the MakeFile

```
LIB := -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread
   -lm -ldl

FLAGS := -std=c++17 -O3 -xHost -mavx512fp16
```

Notably, OpenMP parallelisation flags were excluded as the benchmark focuses exclusively on vectorisation performance [30]. This design choice, in line with the guidance of Prof. Giles, ensures that the reported speedups isolate the contribution of AVX-512 vectorisation without including effects from multithreading. As such, the improvements observed in Section 3.4.1 and 4.2 ($\sim$ 1.1–1.3$\times$ for FP16 over FP32) should be interpreted as vectorisation-only gains. Substantial further acceleration is possible when OpenMP is enabled: where Prof. Giles observed speedups of up to 64$\times$ from multithreaded execution.

3.3.2 Benchmark Procedure

The benchmarking process involves measuring the time taken to generate 10 million Gaussian samples and taking the average to obtain per-sample timings for each LUT method and precision configurations. Each benchmark begins with three preliminary warm-up runs that are discarded to avoid skew from initialisation overheads such as dynamic memory allocation and cache loading.

Subsequent timed runs consist of generating 10 million Gaussian samples per iteration, repeated $R = 20$ times. The recorded results are summarised as the mean $\pm$ standard deviation (SD) across runs, thereby accounting for system noise and variability. High-resolution clocks, specifically `clock_gettime` with the `CLOCK_MONOTONIC` flag, are employed to ensure precise timing measurements [36].

3.3.3 Baseline Comparison

Intel's MKL `vRngGaussian` serves as the target baseline for the benchmarks. The function offers three separate Gaussian generation methods: the Box-Muller transform, a variant Box-Muller 2, and the inverse CDF (ICDF) method [11]. The Box-Muller transform generates a Gaussian distributed random number x through a pair of independent uniformly distributed numbers (u_1, u_2) through the reflection $x = \sqrt{-2 \ln u_1} \sin 2\pi u_2$. However, this method generates only one output for each pair of uniform numbers, limiting computational efficiency.

The Box-Muller 2 variant addresses this inefficiency by simultaneously generating two independent Gaussian variables for each pair of uniform random numbers (u_1, u_2):

$$x_1 = \sqrt{-2 \ln u_1} \sin 2\pi u_2 \quad ; \quad x_2 = \sqrt{-2 \ln u_1} \cos 2\pi u_2. \tag{3.2}$$

Alternatively, Intel offers the ICDF method which generates Gaussian samples by numerically approximating the inverse Gaussian CDF. This enables the `ICDF` method to gain significant speedups by avoiding costly transcendental functions and runtime approximations compared to the Box-Muller techniques.

It is critical to note that Intel MKL's `vRngGaussian` function does not natively support FP16 output. The FP16 results in Table 3.1 were obtained by generating FP32 samples and then explicitly casting them to FP16. This downcasting operation introduces an additional overhead that negates the expected computational savings for half-precision. This inefficiency fundamentally limits the performance gains achievable in MLMC frameworks, where coarser (lower precision) levels are assumed to be computationally cheaper. Consequently, these results strongly motivate the use of our LUT methods that natively support FP16 and FP32, enabling the true hierarchical efficiency essential for MLMC.

Table 3.1: Per-sample time (in ns), reported as mean $\pm$ SD over $R = 20$ runs with 10 million samples/run, using Intel's `vRngGaussian`.[7]

Generation Method	**FP16**	**FP32**
Box Muller	2.60 ± 0.24 ns	2.051 ± 0.060 ns
Box Muller 2	1.83 ± 0.21 ns	1.45 ± 0.21 ns
ICDF	1.257 ± 0.067 ns	0.923 ± 0.068 ns

[7] All standard deviations (SD) are rounded to two significant figures, and mean values are reported to match the precision of their corresponding SD.

In the subsequent analysis, the LUT-based Gaussian generators will be benchmarked against the Box Muller and ICDF generation method. The Box Muller transform is included as it represents the most widely used and generic approach, serving as a natural baseline. The ICDF method, on the other hand, is the fastest implementation in the Intel library and thus provides a strict performance benchmark. Combined, these comparisons enables the evaluation the relative improvements and competitiveness of the discussed LUT methods over conventional techniques such as the Box-Muller method, and against Intel's optimised ICDF Gaussian generator.

3.4 Comparative Performance Analysis

3.4.1 Timing Results

Table 3.2 summarises the per-sample timings for each LUT method, with results given separately for the "transform-only" stage (conversion of uniforms into Gaussian samples) and the "end-to-end" stage (including uniform RNG overhead). All methods successfully produced Gaussian samples with mean ≈ 0 and variance ≈ 1 within numerical tolerance[8].

As expected and noted in Table 2.1, the linear–dyadic implementation was the fastest among the three, benefitting from the smallest LUT size and an $\mathcal{O}(1)$ evaluation cost. For the "transform-only" case, the FP16 Linear–Dyadic achieved 0.266 ns/sample, corresponding to a 2.06× speedup over its FP32 counterpart (as shown in Table 3.3). The Linear-Superdyadic followed closely at 0.281 ns/sample, and although it was slightly slower in absolute terms, it delivered the largest relative FP16 acceleration at 2.75×. The Constant–Uniform FP16 also benefitted from FP16 acceleration, albeit at a more modest 1.11× speedup. When uniform RNG overheads are included, end-to-end runtimes add a fixed ≈ 1.105 ns/sample to each method. As a result, the relative spread between methods narrows (while the absolute gaps remain roughly unchanged). FP16 still provides consistent improvements; among the linear schemes, the superdyadic method yields the largest end-to-end speedup at 1.34×.

[8] The Constant-Uniform has an LUT size of 512 (bit-width $d = 10$), while the Linear-Dyadic and Linear-Superdyadic have an LUT size of 26 and 52 (bit-width $d = 14$), respectively.

Table 3.2: Per-sample time (in ns) reported as mean $\pm$ SD over $R = 20$ runs with 10 million samples/run for the three LUT methods. *Transform-only* times only the LUT transform; *End-to-end* includes the uniform RNG runtime of ≈ 1.105 ns/sample.

Method	**Transform-only**	**End-to-end**
Constant–Uniform FP32	0.807 $\pm$ 0.077 ns	1.91 $\pm$ 0.11 ns
Constant–Uniform FP16	0.729 $\pm$ 0.053 ns	1.83 $\pm$ 0.10 ns
Linear–Dyadic FP32	0.535 $\pm$ 0.071 ns	1.64 $\pm$ 0.13 ns
Linear–Dyadic FP16	0.266 $\pm$ 0.056 ns	1.37 $\pm$ 0.10 ns
Linear–Superdyadic FP32	0.742 $\pm$ 0.061 ns	1.85 $\pm$ 0.11 ns
Linear–Superdyadic FP16	0.281 $\pm$ 0.064 ns	1.39 $\pm$ 0.12 ns

Table 3.3: FP16 vs. FP32 speedups (FP32 time/FP16 time) of the three LUT methods, calculated from the mean values recorded in Table 3.2.

Method	**Transform-only Speedup**	**End-to-end Speedup**
Constant–Uniform	1.11×	1.04×
Linear–Dyadic	2.06×	1.20×
Linear–Superdyadic	2.75×	1.34×

Table 3.4 benchmarks the LUT implementations against Intel MKL's Box Muller and ICDF. All three LUT schemes clearly outperform Box–Muller in both FP16 and FP32, in line with expectations (dyadic is fastest, superdyadic second, and constant third). Against MKL's optimised ICDF, however, the LUT schemes are slower in FP32, roughly at a 2× factor. Nonetheless, the FP16 versions of the linear schemes remain competitive, within about 8% to 9% of the ICDF performance. This shows that despite ICDF being the fastest generator, the LUT methods uniquely deliver genuine FP16 speedups, something `vRngGaussian` lack.

Table 3.4: Relative end-to-end speedup of the three LUT methods against Intel MKL `vRngGaussian` Box Muller and ICDF.

Method	**Box Muller**		**ICDF**	
	FP16	**FP32**	**FP16**	**FP32**
Constant-Uniform	1.42×	1.07×	0.69×	0.48×
Linear-Dyadic	1.90×	1.25×	0.92×	0.56×
Linear-Superdyadic	1.87×	1.11×	0.91×	0.50×

3.4.2 FP16 – FP32 Coupling Analysis

To assess suitability for variance reduction in MLMC, coupling accuracy was evaluated by generating paired FP16 and FP32 Gaussian streams from identical uniform inputs. Table 3.5 summarises the mean and maximum absolute differences, the variance of the differences, and the proportion of "large" deviations exceeding 10^{-3} ($|\Delta z| > 10^{-3}$).

The Constant-Uniform method achieved near-ideal coupling, with mean absolute differences on the order of 10^{-8}, maximum deviations bounded around 10^{-3}, and no samples exceeded the $|\Delta z| > 10^{-3}$ threshold. This hints at Constant-Uniform being a suitable Gaussian generator for MLMC purposes, ensuring that corrections between precision levels do not introduce uncontrolled variance or heavy tails.

The piecewise linear methods also coupled well but displayed some notable deviations. Mean differences and variances remained small (10^{-8} to 10^{-7}), comparable to the piecewise constant case. However, a small percentage of samples ($\sim$ 2% to 3%) exhibited large differences where $|\Delta z| > 10^{-3}$. These larger deviations and outliers likely arise from half-precision rounding when evaluating $z = A + B\,u$, despite bins being correctly identified. While such deviations will not affect standard European option pricing or MLMC variance estimates in practice, they could degrade weak convergence compared with the Constant-Uniform approach.

Table 3.5: Coupling Analysis of FP16–FP32 over paired streams; reporting absolute mean/max difference in Δz, variance of the difference, and % of samples with a large difference ($|\Delta z| > 10^{-3}$), across 10 million samples.

LUT Method	**Mean Diff.**	**Max. Diff.**	**Variance**	**% of Large Diff.**
Constant-Uniform	6.28×10^{-8}	9.69×10^{-4}	4.18×10^{-8}	0%
Linear-Dyadic	1.04×10^{-7}	3.71×10^{-3}	1.61×10^{-7}	1.93%
Linear-Superdyadic	3.79×10^{-8}	2.15×10^{-3}	1.46×10^{-7}	3.00%

In summary, the LUT methods demonstrate strong performance. They consistently outperform Box-Muller, approach MKL's ICDF in FP16, and most importantly, provide genuine low-precision acceleration for nested MLMC testing. The Constant-Uniform method offers the most stable FP16–FP32 coupling, while the linear methods achieve greater speedups at the cost of occasional outliers. This trade-off suggests that Constant-Uniform is preferable when coupling between precision is priority, while dyadic and superdyadic offer faster alternatives when absolute throughput is the primary concern.

Chapter 4

Cost Model for Payoff Path Calculation

This chapter extends the RNG benchmarking introduced in Chapter 3 to the full Euler–Maruyama payoff path calculation for a European call option. As described in Section 1.3, this corresponds to evaluating

$$P(h_l) = e^{-rT} \max\bigl(S_T(h_l) - K, 0\bigr), \tag{4.1}$$

where the level-l step size is $h_l = T/M_l$ with $M_l = 2^l M_0$ (typically $M_0 = 1$), and hence $h_l = h_{l-1}/2$.

Definition 4.1. Let $m \in \{\text{Constant-Uniform}, \text{Linear-Dyadic}, \text{Linear-Superdyadic}\}$ denote the Gaussian LUT method and $p \in \{\text{FP16}, \text{FP32}\}$ the precision. A single Gaussian sample refers to one standard normal variate, generated from one uniform sample plus its associated transform, with the per-Gaussian RNG time expressed as

$$g_p^{(m)} = u + r_p^{(m)} \quad \text{[ns/Gaussian Sample]} \tag{4.2}$$

where u is the uniform-generation time and $r_p^{(m)}$ is the transform-only time for method m at precision p (see Table 3.2). For a level l path with $M_l = T/h_l$ steps, the corresponding per-path-RNG time is

$$G_p^{(m)}(h_l) \;=\; M_l\, g_p^{(m)} \quad \text{[ns/path]}. \tag{4.3}$$

Notation. For precision $p \in \{\text{FP16}, \text{FP32}\}$ and level l with $M_l = T/h_l$ steps, this work defines

$$t_p(h_l) \;\text{[ns/path]} \qquad \text{and} \qquad \tau_p(h_l) \equiv \frac{t_p(h_l)}{M_l} \;\text{[ns/path–step]}, \tag{4.4}$$

where $t_p(h_l)$ is the per-path-calculation time (Euler–Maruyama + payoff + discount), and $\tau_p(h_l)$ is the corresponding per-step-calculation time, where Gaussian RNG is excluded from both timings. Quantifying t_p, τ_p, and $G_p^{(m)}$ (using the measurements in Section 3 and Section 4.2) is essential for evaluating FP16/FP32 speedups and for initialising the level-wise MLMC costs in Chapter 5.

4.1 Path Cost Measurement Framework

In the standard MLMC estimator, the level-l contribution is the fine-coarse difference computed entirely in FP32, in which the cost of one FP32 fine-coarse sample $\Delta P_l = P_{\text{fp32}}(h_l) - P_{\text{fp32}}(h_{l-1})$ is

$$C_l = \underbrace{t_{\text{fp32}}(h_l) + t_{\text{fp32}}(h_{l-1})}_{\text{FP32 per-path-calculation cost}} + \underbrace{G_{\text{fp32}}^{(m)}(h_l)}_{\text{End-to-end per-path FP32 RNG cost}}, \tag{4.5}$$

where $t_{\text{fp32}}(h)$ is the per-path-calculation time, and $G_{\text{fp32}}^{(m)}(h)$ is the per-path RNG time as defined in Definition 4.1.

For the nested mixed-precision scheme (see Section 1.4.4), the cheap FP16 pair is evaluated and corrected with a FP32−FP16 difference computed on the same random stream, in which the cost of one FP16 fine-coarse sample $\widehat{\Delta P}_l = P_{\text{fp16}}(h_l) - P_{\text{fp16}}(h_{l-1})$ is

$$\widehat{C}_l = \underbrace{t_{\text{fp16}}(h_l) + t_{\text{fp16}}(h_{l-1})}_{\text{FP16 per-path-calculation cost}} + \underbrace{G_{\text{fp16}}^{(m)}(h_l)}_{\text{End-to-end per-path FP16 RNG cost}}. \tag{4.6}$$

Although it may seem counterintuitive to exclude the coarse-path RNG $G_p^{(m)}(h_{l-1})$ from the end-to-end RNG cost in both FP32 and FP16 fine-coarse pair, this is actually a deliberate design choice. In the MLMC implementation, the coarse Brownian increment over h_{l-1} is constructed by summing two fine-level increments over $h_l = h_{l-1}/2$, hence the pair uses only M_l Gaussian draws; in which the additional vector additions are negligible compared with the cost of RNG and Euler-Maruyama updates. This is a key implementation for computational savings as the RNG dominates the cost calculation path.

For the correction $\Delta P_l - \widehat{\Delta P}_l$, the algorithm generates a single shared uniform stream and transforms it into both precisions at each fine step. Hence, each correction sample only pays one uniform-generation time u, two transform times $r_{\text{fp32}}^{(m)}$ and $r_{\text{fp16}}^{(m)}$, together with M_l and M_{l-1} per-step-calculation costs τ_p. This leads to the per-sample correction cost $\Delta P_l - \widehat{\Delta P}_l$ of

$$\widetilde{C}_l = \left(\tau_{\text{fp32}} + \tau_{\text{fp16}}\right)\left(M_l + M_{l-1}\right) + M_l\left(u + r_{\text{fp32}}^{(m)} + r_{\text{fp16}}^{(m)}\right) \tag{4.7}$$

Let $\widehat{N}_l$ and $\widetilde{N}_l$ denote the sample counts for the FP16 pair $\widehat{\Delta P_l}$ and the correction term $\Delta P_l - \widehat{\Delta P_l}$, respectively, the expected level cost in the nested MLMC scheme (1.25) is then

$$\text{Cost}_l^{\text{Nested}} = \widehat{N}_l \widehat{C}_l + \widetilde{N}_l \widetilde{C}_l \tag{4.8}$$

which reflects the cost model derived in (1.26) and feeds directly into the allocation formulas in (1.28) and (1.29). All results obtained in the subsequent section are obtained with AVX-512 vectorised implementations, as detailed in Algorithm 1.1 and Table D.4.

4.2 Numerical Results and Discussion

Euler-Maruyama per-path-calculation Timings

Figure 4.1 shows the per-path-calculation and per-path-step-calculation timings for both FP16 and FP32 across levels $l \in \{0, 1, \dots, 9\}$, corresponding to step size $M \in \{1, 2, \dots, 512\}$. For completeness, the full numerical results (mean ± SD over 20 runs) are provided in Table E.1.

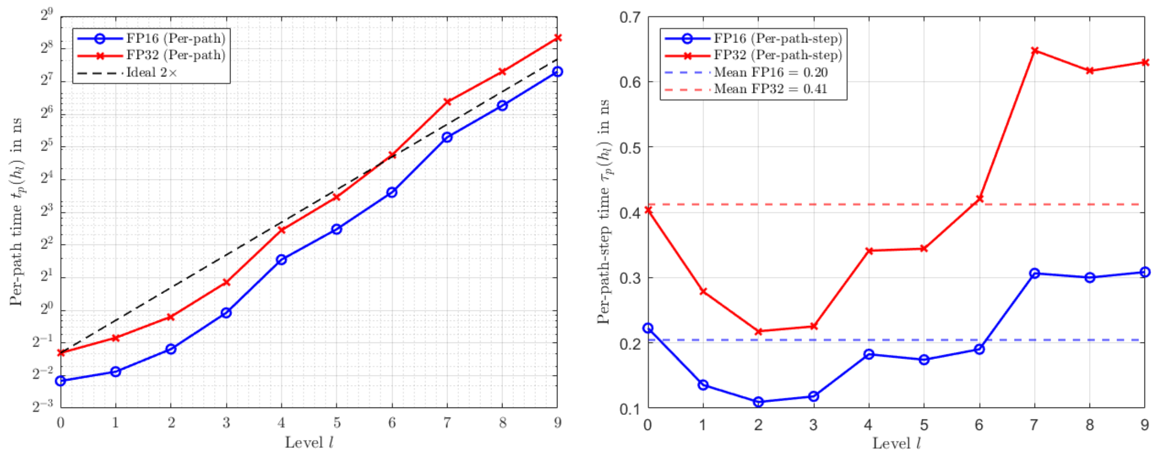


Figure 4.1: **(Left)** Per-path time (ns) in FP16 and FP32 against level l. The dashed line is an Ideal 2× reference anchored at the FP16 level-0 per-path cost. **(Right)** Per-path-step time (ns) in FP16 and FP32; near-flat curves indicate near-linear scaling with l. RNG timings are excluded from both figures.

The FP16 and FP32 per-path curves (left) grow almost linearly with the level l and follow the *Ideal* 2× guide closely, confirming that the payoff-path cost essentially scales linearly with the number of steps M_l. Across all levels, FP16 remains strictly below FP32, with a gap close to a factor of two. However, slight discrepancies from the *Ideal* 2× guide appear at both small and large l, likely due to the negligible setup overheads at small l, and to the working set of Gaussian increments exceeding cache storage at large l, leading to additional memory traffic that increases the slope.

The per-path-step curves (right) are nearly level independent, with an FP16 mean of around $\tilde{\tau}_{\text{fp16}} \approx 0.20$ ns/step and FP32 around $\tilde{\tau}_{\text{fp32}} \approx 0.41$ ns/step. The ratio $\tau_{\text{fp32}}/\tau_{\text{fp16}} \approx 2.0$ quantifies the expected 2× SIMD advantage of native FP16 paths over FP32. As mentioned, the "U-shape" observed at lower levels is likely attributable to negligible overhead; the increase for $l \geq 7$ likely stems from the cache-capacity effects, which is more pronounced for the FP32 as data spill into the L3 or DDR cache.

RNG Timings

Figure 4.2 (left) reports the end-to-end RNG time per sample $g_p^{(m)}(h_l) = u + r_p^{(m)}(h_l)$ for the three Gaussian-based LUT method $m \in \{$Constant-Uniform, Linear-Dyadic, Linear-Superdyadic$\}$ and both precision $p \in \{$FP16, FP32$\}$. The curves are nearly constant across l, indicating that $g_p^{(m)}$ is effectively level-independent. The progressive increase for $l \geq 8$ resembles the per-path-step time behaviour in Figure 4.1, likely stemming from cache-capacity effects.

Conversely, Figure 4.2 (right) shows the end-to-end FP16 speedup

$$\frac{g_{\text{fp32}}^{(m)}}{g_{\text{fp16}}^{(m)}} = \frac{u + r_{\text{fp32}}^{(m)}}{u + r_{\text{fp16}}^{(m)}}, \tag{4.9}$$

As the uniform component u is common to both precisions, it compresses the speedup relative to the transform-only ratios as described in Table 3.2. The Constant-Uniform and dyadic scheme are nearly level-invariant, with a speedup factor of $\approx 1.13\times$ and $\approx 1.20\times$, respectively. By contrast, the superdyadic scheme begins with a speedup factor of above 2× at small l and gradually decreases and eventually converges to ~ 1.4 by level $l = 9$. This tail-off effect occurs due to data spilling into L3 and DDR memory, where transfer time dominates and the cost of generation and transformation becomes negligible, leading to the asymptotic speedup reported in Table 3.3.

These general behaviours are entirely consistent with the single-level ($M = 1$) 10 million Gaussian samples benchmark in Chapter 3 (see Table 3.2 and 3.3). The ranking of the computational cost and LUT runtime follows the same ranking (dyadic is the fastest, superdyadic is second, and constant is third), and the relative FP16 speedup factors resemble the values presented in Table 3.3 (with the exception of the large superdyadic speedup at early levels).

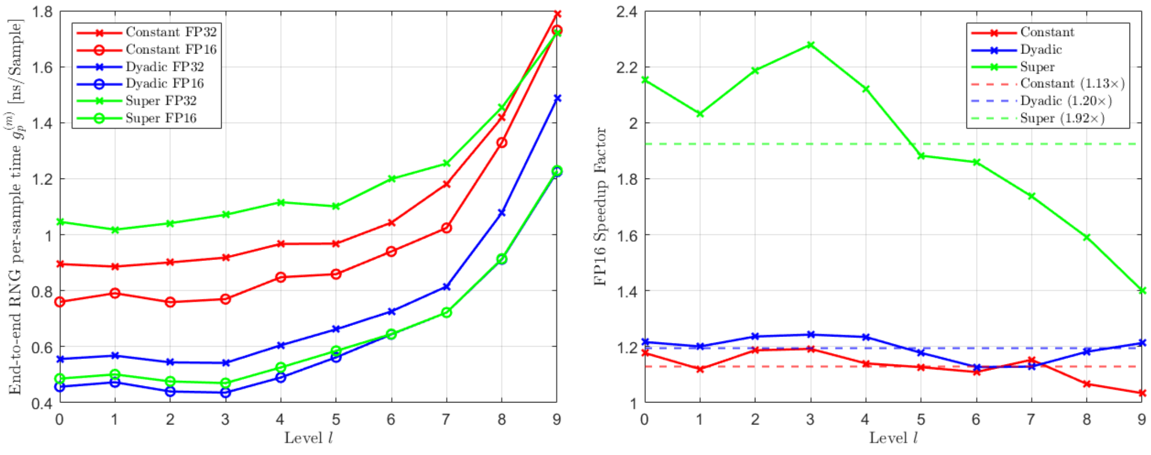


Figure 4.2: **(Left)** End-to-end RNG per-sample time $g_p^{(m)}(h_l) = u + r_p^{(m)}(h_l)$ for the Gaussian-based LUT methods in FP16 and FP32 across level l. **(Right)** Corresponding FP16 speedup $(u + r_{\text{fp32}}^{(m)})/(u + r_{\text{fp16}}^{(m)})$ against level l. Runtime results are recorded in Table E.2.

Since $\tau_{\text{fp32}}/\tau_{\text{fp16}} \approx 2$ and $g_{\text{fp32}}^{(m)}/g_{\text{fp16}}^{(m)}$ is nearly flat in l across all LUT methods, the pair level speedup is inherently level-invariant. These calibrated cost constants will be used directly in the level-wise MLMC model in Chapter 5.

Chapter 5

Nested Multilevel Monte Carlo Framework

5.1 Classic Nested Multilevel Monte Carlo

The preceding chapters establish the foundations of this work: (1) the efficient, coupled Gaussian RNG in both FP16 and FP32 (Chapter 3), and (2) the sample cost model for payoff path simulation (Chapter 4). This chapter integrates these components into the mixed-precision nested MLMC framework, initially outlined in Section 1.4.4, using Prof. Giles's FP64/FP32 mixed-precision MLMC driver as the baseline. The contribution in this work is the adaptation of this framework to an FP32/FP16 setting to evaluate the practicality of reduced-precision arithmetic in option pricing.

The core principle of the nested estimator is to decompose the level-wise difference into two parts: a computationally cheap half-precision term and a more costly high-precision correction. This allows the majority of computations to be performed in half-precision, with a small number of FP32 − FP16 corrections to eliminate the discretisation and rounding error bias. The expectation of the payoff is given by

$$\mathbb{E}[P] = \sum_{l=0}^{L} \Big(\underbrace{\mathbb{E}[\widehat{\Delta P_l}]}_{\text{Half-precision estimate}} + \underbrace{\mathbb{E}[\Delta P_l - \widehat{\Delta P_l}]}_{\text{FP32}-\text{FP16 correction term}} \Big), \tag{5.1}$$

where $\widehat{\Delta P_l} = P_{\text{fp16}}(h_l) - P_{\text{fp16}}(h_{l-1})$ denotes the half-precision difference, and $\Delta P_l = P_{\text{fp32}}(h_l) - P_{\text{fp32}}(h_{l-1})$ denotes the corresponding single-precision difference.

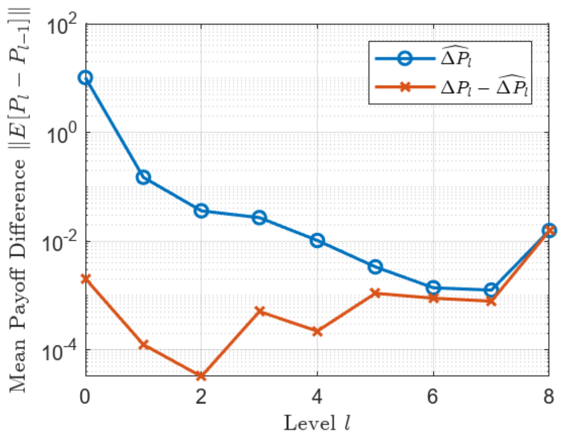
Having defined the estimator, the work in this section presents the numerical results where the mixed-precision approach is applied uniformly across all levels $l = 0, \dots, L$ until convergence or until the maximum refinement level $L_{\max} = 10$ is reached. The simulations price a European call option, as defined in Section 1.3, using the Euler-Maruyama discretisation of the GBM. The simulation setup, including asset parameters, MLMC control settings, and convergence test configurations, is summarised in Table 5.1.

Table 5.1: Asset parameters, MLMC control settings, and convergence test configurations parameters used in simulations.

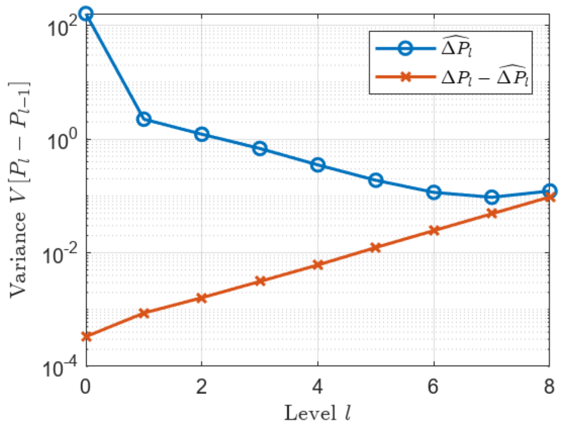
Parameter	**Value**	**Description**
Asset parameters		
S_0	100	Initial asset price.
K	100	Strike price.
r	0.05	Interest rate.
σ	0.2	Asset volatility.
T	1	Time to maturity.
MLMC control settings		
N_0	200	Initial number of pilot samples per level.
L_{min}	2	Minimum refinement level required before adaptive level growth through the bias check.
L_{max}	10	Maximum refinement level, beyond which no further levels are added even if weak convergence test fails.
Convergence test configurations		
N_{test}	$20,000$	Number of MC samples in convergence tests.
L_{test}	8	Number of levels in convergence tests.
ε	$0.005 \times 2^k,\ k = 0, \ldots, 4$	Target RMSE values across different test runs.

5.1.1 Numerical Results and Discussion

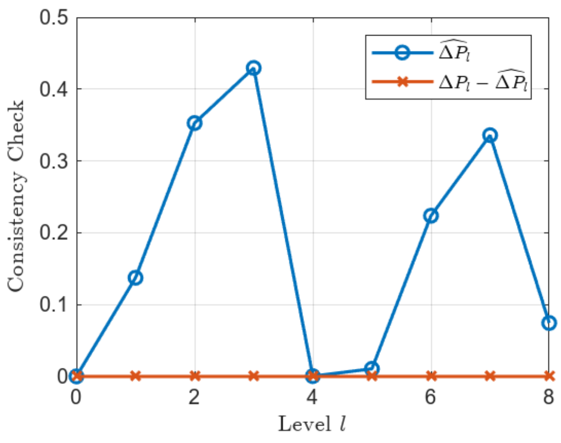
The numerical behaviour of the classic nested MLMC (using the Linear-Dyadic Gaussian generator) can be assessed through four diagnostics: mean payoff difference, variance difference, consistency check, and kurtosis, as shown in Figure 5.1. Each plot contains two curves: the blue line corresponds to the half-precision estimate $\widehat{\Delta P_l}$, while the red line corresponds to the FP32−FP16 correction term $\Delta P_l - \widehat{\Delta P_l}$.



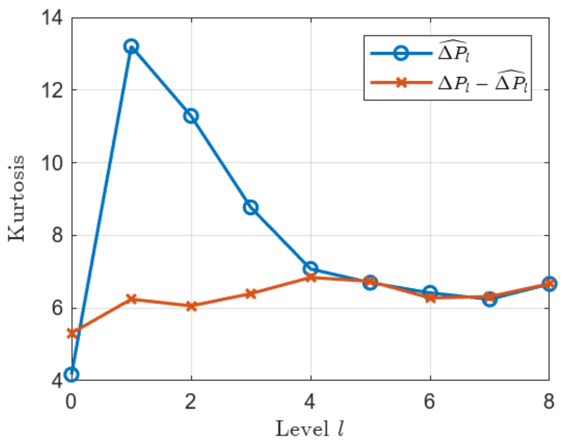
(a) Mean payoff difference $\|\mathbb{E}[P_l - P_{l-1}]\|$ across levels l



(b) Variance $\mathbb{V}[P_l - P_{l-1}]$ across levels l.



(c) Consistency check across levels l.



(d) Kurtosis of payoff difference distribution across levels l.

Figure 5.1: Numerical results for a European call option using the Euler-Maruyama discretisation of the GBM SDE in a classic nested MLMC (**Linear-Dyadic**).

The top two plots (Figures 5.1a and 5.1b) show the absolute mean payoff difference and variance difference across refinement levels, based on an initial number of 2×10^4 MC samples. The payoff difference decays at a rate of $\alpha \approx 0.73$, while the variance decays at $\beta \approx 0.63$, as reported in Table 5.2. At coarse levels, both the $\widehat{\Delta P_l}$ curves exhibit the expected decay, but at finer levels the decay flattens, with the variance difference even increasing slightly at $l = 8$. This behaviour is attributed to accumulated rounding error in half-precision arithmetic, which dominates beyond a critical refinement level L_{critical}. The same effect is also visible in the $\Delta P_l - \widehat{\Delta P_l}$ correction term, which remains small on coarse levels but grows at finer levels, reflecting the mismatch between FP32 and FP16 path calculations. As a result, both the regression estimates α and β are reduced.

The bottom two plots (Figures 5.1c and 5.1d) show the consistency check and kurtosis. The consistency measure remains < 1 across all levels, and kurtosis decreases slightly with level. Both behaviours are consistent with literature [16], confirming that the level differences are correctly computed and that the variance estimates remain statistically reliable. These diagnostics also prevent the risk of programming errors or instability caused by rare outliers.

Table 5.2 summarises the regression estimates for α, β, and γ across the different LUT methods. The Constant-Uniform generator achieves $\alpha \approx 1.04$, $\beta \approx 0.68$, and $\gamma \approx 1.0$, while the Linear-Dyadic and Linear-Superdyadic generator yield similar results with smaller weak convergence rates. This reduction in α is a direct result of the poorer FP16–FP32 coupling observed in Table 3.5 for the piecewise linear methods, where interpolation rounding errors introduce a bias into the telescoping sum. The most significant observation, however, is the smaller-than-theoretical[9] variance decay rate of $\beta \approx 0.67$, indicating that variance does not diminish rapidly with increasing levels. In the context of MLMC, this implies larger sample allocation for correction terms at fine levels, thereby reducing efficiency.

Table 5.2: Linear regression estimates of the classic nested MLMC parameters across different LUT-based methods. The parameters are defined as in Theorem 1.1, where α is the weak/bias convergence, β is the variance decay, and γ is the cost growth.

LUT Method	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	$\boldsymbol{\gamma}$
Constant-Uniform	1.04	0.68	1.00
Linear-Dyadic	0.73	0.67	1.00
Linear-Superdyadic	0.86	0.67	1.00

This smaller-than theoretical variance decay does not reflect a methodological or computational flaw, but rather the practical limitations of reduced-precision arithmetic, where rounding error dominate at finer levels. Nevertheless, these regression estimates are critical as they directly inform the complexity analysis presented next.

[9] As noted in Section 1.4.2, the theoretical convergence rates for the European vanilla option are $\alpha = 1$, $\beta = 1$, and $\gamma = 1$. This is supported by the numerical simulation of a standard MLMC in FP32, shown in Table F.1 and Figure F.1.

Computational Complexity and Cost Analysis

Figure 5.2 shows the optimal sample allocation N_l across levels for five separate MLMC runs with target RMSE $\varepsilon = 0.005 \times 2^k$ for $k = 0, \dots, 4$. Circles ($\circ$) indicate the number of samples assigned to the half-precision estimator $\widehat{\Delta P_l}$, while crosses ($\times$) denote the allocation to the FP32 − FP16 correction term $\Delta P_l - \widehat{\Delta P_l}$. As expected, the number of levels increases as the error tolerance ε tightens to meet the weak error requirement.

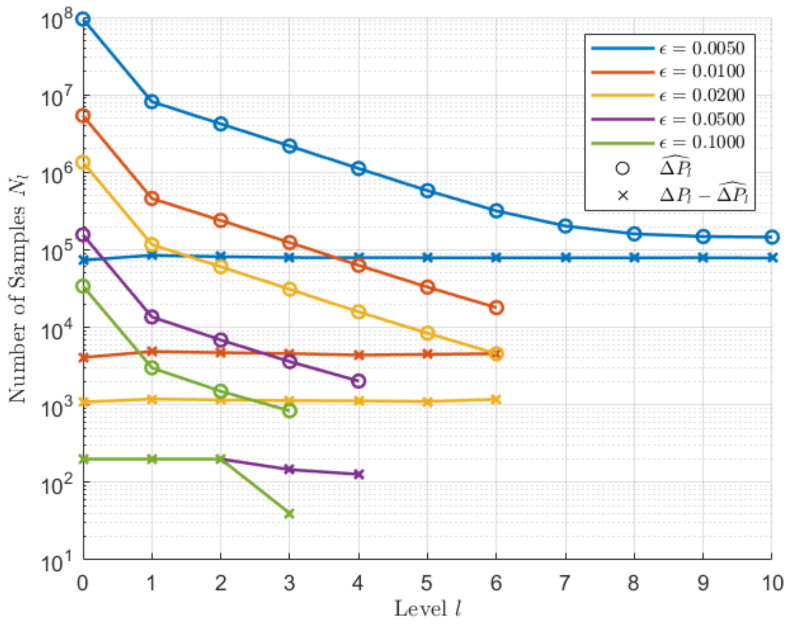


Figure 5.2: Optimal sample allocation N_l across levels l for varying target RMSE values ε using the **Linear-Dyadic LUT**. Circles ($\circ$) denote the number of samples assigned to $\widehat{\Delta P_l}$, while crosses ($\times$) denote the allocation to the correction term $\Delta P_l - \widehat{\Delta P_l}$.

Most $\widehat{\Delta P_l}$ samples are allocated on the coarsest levels, where computations are relatively cheap, while finer levels receive fewer samples due to their higher cost. A further observation is that the allocation to correction terms increases slightly under stricter error tolerances. This reflects the weaker variance decay caused by half-precision rounding errors at fine levels, which require additional correction samples $\Delta P_l - \widehat{\Delta P_l}$ to correct the offset of the mismatch between precisions. This highlights the hidden computational cost associated with reduced-precision arithmetic at higher levels of refinement.

A unique behaviour arises with the Linear-Dyadic LUT. At the strictest error tolerance ($\varepsilon = 0.005$), the MLMC fails to achieve weak convergence and terminates at the maximum refinement level $L_{\max} = 10$. This contrasts with the Constant-Uniform and Linear-Superdyadic LUTs, both of which achieve convergence within the same maximum refinement level, supported by their larger weak convergence α values shown in Table 5.2.

While the numerical results presented thus far validated the behaviour of the nested MLMC framework, the central question remains: whether the nested MLMC using reduced precision actually delivers tangible computational savings. To address this, the relationship between the target RMSE ε and the total computational cost was examined. This allows a direct comparison of the $\mathcal{O}(\varepsilon^{-3})$ complexity of standard MC with the optimal $\mathcal{O}(\varepsilon^{-2})$ scaling of MLMC. This enables a clear comparison of practical performance against theoretical cost scaling from literature, while quantifying the effect of mixed-precision arithmetic.

Figure 5.3 compares computational costs for MLMC and standard MC frameworks, implemented in both single precision (FP32) and mixed-precision (FP16–FP32), using the Linear-Dyadic LUT. The horizontal axis shows accuracy ε, while the vertical axis shows cost scaled by ε^2. This scaling follows [15], ensuring that an algorithm like the MLMC with asymptotically optimal complexity will yield an approximately constant curve as ε decreases.

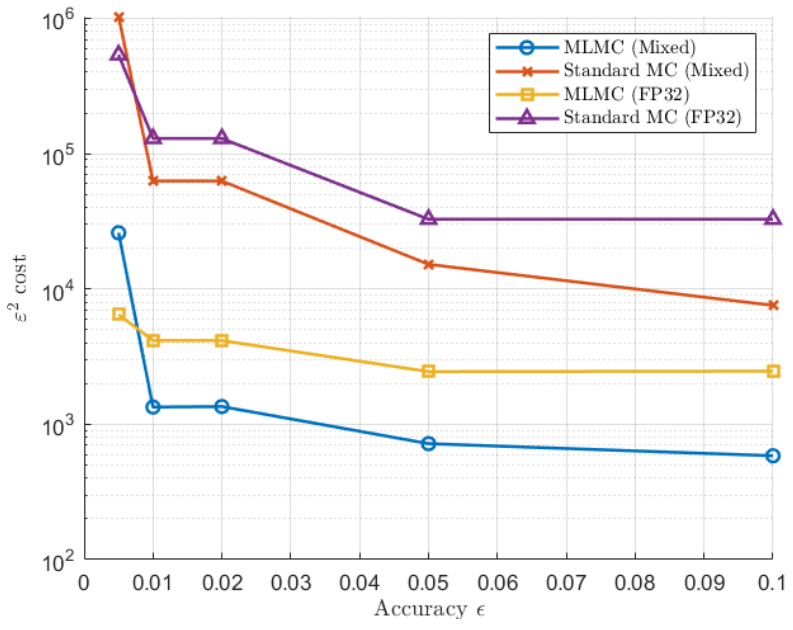


Figure 5.3: Cost comparison of standard MC and classic MLMC in both FP32 and mixed-precision (FP16, FP32) using the **Linear-Dyadic LUT**. The plot shows the variation in computational cost $C \times \varepsilon^2$ against target accuracy ε.

The results confirm theoretical predictions. Across all accuracies, MLMC outperforms standard MC, at substantially lower cost. The gap widens significantly as ε decreases below 0.05, reflecting MLMC's superior variance reduction across levels, confirming the expected complexity reduction from $\mathcal{O}(\varepsilon^{-3})$ to $\mathcal{O}(\varepsilon^{-2})$. In contrast, standard MC exhibits the anticipated cost increase as accuracy requirements tighten.

As described in Section 1.4.4, introducing mixed-precision arithmetic within the nested MLMC framework further enhances efficiency. Mixed-precision MLMC consistently achieves the lowest computational cost across all target tolerances, while satisfying the RMSE targets. The only exception occurs at $\varepsilon = 0.005$ with the Linear-Dyadic Gaussian generator, where weak convergence fails. Overall, these results demonstrate that nested MLMC with reduced-precision arithmetic remains numerically stable and cost-efficient, even under strict error tolerances, provided weak convergence is satisfied.

Key Takeaways

The numerical results presented in this section are consistent across all Gaussian LUT methods, and can be similarly observed with the Constant-Uniform and Linear-Superdyadic generators (Figures F.2 and F.3). Their larger weak convergence rates enable MLMC to converge successfully within the maximum refinement level $L_{\max} = 10$, even at the strictest error tolerance $\varepsilon = 0.005$. For comparison, the full FP32 MLMC diagnostic plots are provided in Figures F.1, confirming the expected variance decay and stable sample allocations, consistent with theoretical convergence ($\alpha = 1, \beta = 1, \gamma = 1$).

In summary, two key conclusions emerge from these experiments. First, MLMC consistently outperforms standard MC both theoretically and in practice, validating its optimal complexity and practical efficiency for SDE simulations. Second, the results show that a nested MLMC using reduced-precision arithmetic can deliver additional cost savings without compromising accuracy. This finding highlights the potential of reduced-precision MLMC frameworks as a practical framework for high-performance option pricing and related applications in computational finance[10].

At the same time, the experiments also reveal a limitation: reduced-precision MLMC exhibits smaller-than-theoretical weak convergence α and variance decay β, due to half-precision rounding errors accumulating and dominating at finer levels.

[10] Reduced-precision MLMC refers specifically to combining FP16 and FP32 arithmetic, which is distinct from conventional FP64/FP32 mixed-precision approaches.

This motivates the adaptive-precision approach introduced next in Section 5.2, where the estimator switches from mixed-precision to a pure single-precision FP32 MLMC beyond a specified cut-off level $l = \widetilde{L}$. Such a hybrid strategy retains the effiency gains of half-precision arithmetic on coarse levels, while simultaneously restoring convergence and improved variance decay at fine levels.

5.2 Adaptive–Precision Nested Multilevel Monte Carlo

This section introduces the adaptive-precision nested MLMC, in which the estimator switches from mixed-precision MLMC (FP16–FP32) to a pure FP32-only MLMC beyond a cut-off level $l = \widetilde{L}$. This modification addresses the deterioration in weak convergence and variance decay in the fully mixed-precision estimator. In particular, once refinement passes a critical level L_{critical}, rounding errors in half-precision dominates, flattening both bias convergence and variance reduction.

By introducing the cut-off before this regime, the adaptive estimator retains the efficiency of reduced precision on coarse levels, while relying on full FP32 at finer levels to correct rounding errors and restore stable convergence. The resulting expectation can be expressed as

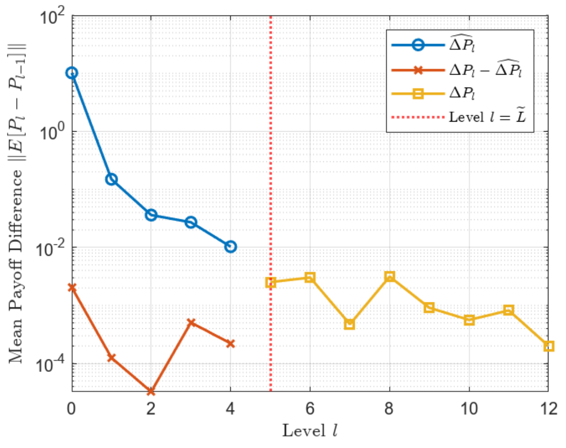
$$\mathbb{E}[P] = \sum_{l=0}^{\widetilde{L}} \Big(\mathbb{E}[\widehat{\Delta P_l}] + \mathbb{E}[\Delta P_l - \widehat{\Delta P_l}] \Big) + \sum_{l=\widetilde{L}}^{L} \mathbb{E}[\Delta P_l], \tag{5.2}$$

where $\widehat{\Delta P_l} = P_{\text{fp16}}(h_l) - P_{\text{fp16}}(h_{l-1})$ denotes the half-precision difference, and $\Delta P_l = P_{\text{fp32}}(h_l) - P_{\text{fp32}}(h_{l-1})$ the single-precision difference.

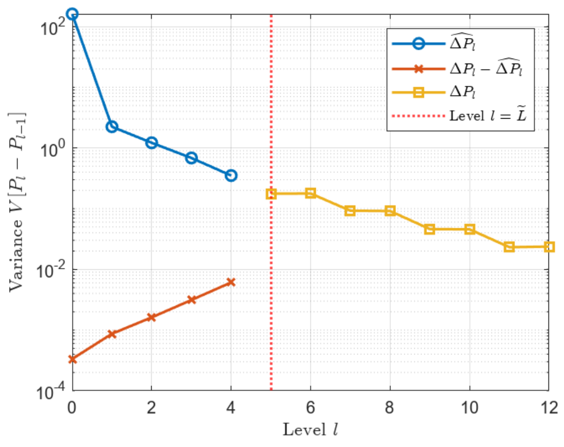
To ensure a fair comparison, the numerical simulations in the following section adopt the same simulation parameters as in Table 5.1. But for the convergence experiments, however, the number of test levels was increased to $L_{\text{test}} = 14$ to capture the asymptotic behaviour more reliably. This is particularly important for the adaptive scheme, where additional post cut-off levels $l \geq \widetilde{L}$ are required to obtain accurate regression estimates.

5.2.1 Numerical Results and Discussion

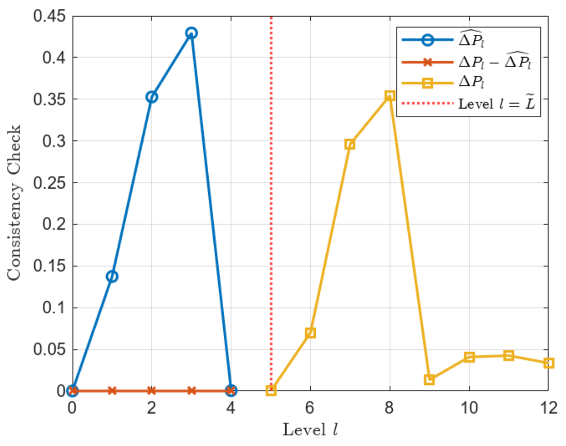
The numerical behaviour of the adaptive-precision MLMC (using the Linear-Dyadic generator) is shown in Figure 5.4. Each diagnostic plot contains three curves: the blue line corresponds to the half-precision estimate $\widehat{\Delta P_l}$, the red line to the FP32−FP16 correction term $\Delta P_l - \widehat{\Delta P_l}$, and the yellow line to the pure FP32-only estimate ΔP_l. The vertical dotted red line at $l = \widetilde{L} = 5$ indicates the cut-off level.



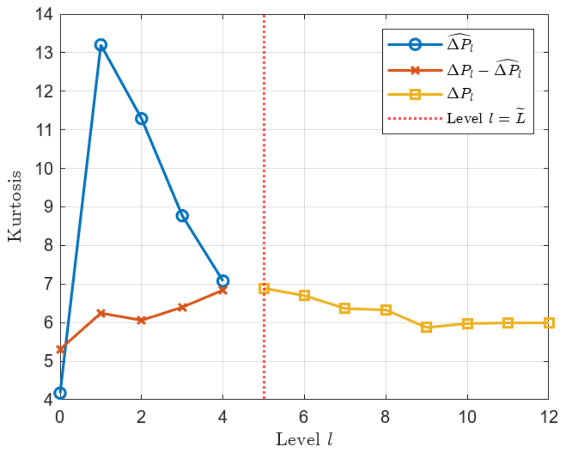
(a) Mean payoff difference $\|\mathbb{E}[P_l - P_{l-1}]\|$ across levels l



(b) Variance $\mathbb{V}[P_l - P_{l-1}]$ across levels l.



(c) Consistency check of FP16–FP32 correction term across levels l.



(d) Kurtosis of payoff difference distribution across levels l.

Figure 5.4: Numerical results for a European call option using the Euler-Maruyama discretisation of the GBM SDE in an adaptive-precision MLMC with fixed cut-off level at $l = \widetilde{L} = 5$ (**Linear-Dyadic**).

The top two plots (Figures 5.4a and 5.4b) show the mean payoff difference and the variance across refinement levels. For $l = 0, \ldots, 4$, the results are identical to the classic mixed-precision MLMC. Beyond the cut-off $l = \widetilde{L}$, the adaptive scheme switches to pure FP32 and the payoff difference continues to decay at near-theoretical weak convergence and decay rates. Unlike the classical mixed-precision scheme, no flattening or increase is observed on fine levels, demonstrating that the cut-off successfully prevents deterioration from half-precision rounding errors. This improvement is reflected in the regression estimates of $\alpha \approx 0.86$ and $\beta \approx 0.96$, compared with the degraded values of $\alpha \approx 0.73$ and $\beta \approx 0.67$ observed in the fully mixed-precision scheme using the Linear-Dyadic Gaussian generator.

The bottom two plots (Figures 5.4c and 5.4d) show the consistency check and kurtosis. As in the classic scheme, consistency remains below 1 and kurtosis remains close to zero both before and after the cut-off. This confirms the numerical stability and robustness of the adaptive scheme.

Overall, these results demonstrate the advantages of introducing a cut-off level that switches from mixed-precision to pure FP32. This prevents the accumulation of half-precision rounding errors on fine levels and restores stable weak convergence and variance decay. The improvement is evident in Table 5.3, with both α and β increasing across all LUT methods, converging towards the theoretical value of (1,1,1).

Table 5.3: Linear regression estimates of the adaptive-precision nested MLMC parameters across different LUT-based methods. The parameters are defined as in Theorem 1.1, where α is the weak/bias convergence, β is the variance decay, and γ is the cost growth.

LUT Method	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	$\boldsymbol{\gamma}$
Constant-Uniform	1.00	0.97	1.00
Linear-Dyadic	0.86	0.96	1.00
Linear-Superdyadic	0.86	0.96	1.00

Computational Complexity and Cost Analysis

Figure 5.5 shows the optimal sample allocation N_l for adaptive-precision MLMC runs with target RMSE values $\varepsilon = 0.005 \times 2^k$ for $k = 0, \ldots, 4$. As before, circles ($\circ$) denote samples for the half-precision estimator $\widehat{\Delta P_l}$, crosses ($\times$) the FP32 − FP16 correction term $\Delta P_l - \widehat{\Delta P_l}$, and squares ($\square$) the FP32-only estimate ΔP_l. The vertical red line at $l = \widetilde{L} = 5$ marks the cut-off level.

A key difference from the classic nested MLMC (Figure 5.2) is that the adaptive scheme converges at $l = 8$, rather than reaching the maximum refinement level $l = L_{\max}$, under the strictest error tolerance $\varepsilon = 0.005$. As the error tolerance ε is relaxed, the number of levels required decreases as expected. This confirms the effectiveness of the adaptive scheme: the cut-off level prevents half-precision rounding error from dominating the fine levels and avoids the weak convergence failures observed in the mixed-precision case.

As in the classical nested MLMC, the largest number of samples is allocated to coarse levels, with allocations decreasing at finer resolutions. Pre cut-off $l \leq \widetilde{L} = 5$, the structure closely resembles the mixed-precision allocation of Figure 5.2 closely, while post cut-off $l \geq \widetilde{L} = 5$ mirrors the pure-FP32 scheme shown in Figure F.1.

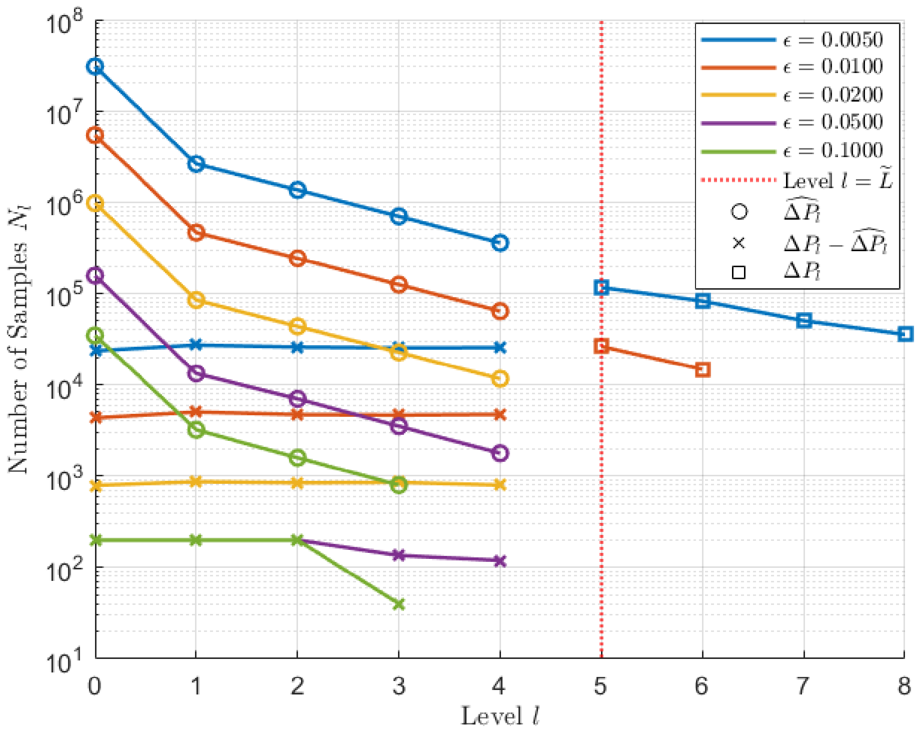


Figure 5.5: Optimal sample allocation N_l across levels l for varying target RMSE values ε for the adaptive-precision MLMC (using the **Linear-Dyadic LUT**). Circles ($\circ$) denote the number of samples assigned to $\widehat{\Delta P_l}$, crosses ($\times$) denote the allocation to $\Delta P_l - \widehat{\Delta P_l}$, and squares ($\square$) denote the allocation to ΔP_l. The vertical dotted red line at $l = \widetilde{L} = 5$ indicates the cut-off level.

Figure 5.6 compares the computational costs for the standard FP32-only MLMC (yellow), classic nested mixed-precision MLMC (blue), and the adaptive-precision nested MLMC (red). The results demonstrate two key points.

First, the adaptive-precision scheme delivers the computational cost across all error tolerances, significantly outperforming the FP32-only MLMC. Second, unlike the mixed-precision scheme, the adaptive approach maintains stability even at the strictest tolerance $\varepsilon = 0.005$, confirming that the cut-off successfully mitigates the accumulated half-precision rounding errors on fine levels. Overall, these results highlight the adaptive scheme as the most effective configuration, combining the savings of reduced precision with stability of full FP32 on fine levels.

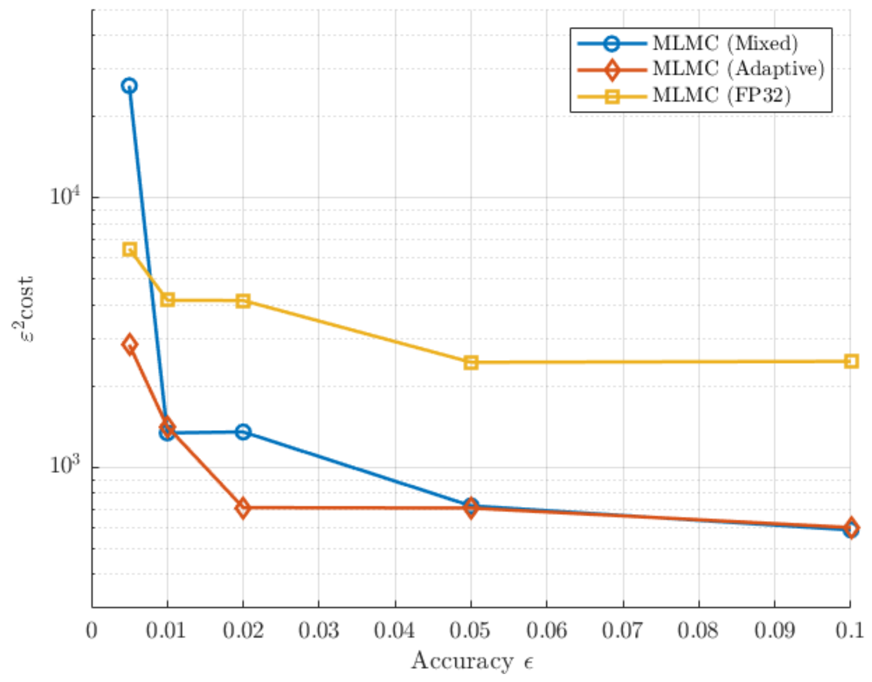


Figure 5.6: Cost comparison of standard MLMC, classic nested MLMC, and adaptive-precision nested MLMC using the **Linear-Dyadic LUT**. The plot shows the variation in computational cost $C \times \varepsilon^2$ against target accuracy ε.

Figures F.4 and F.5 show analogous results for the Constant-Uniform and Linear-Superdyadic LUTs, exhibiting the same behaviours discussed above for the adaptive-precision scheme.

5.3 Discussion and Conclusion

This section examined the numerical performance of both the classic nested MLMC estimator and the adaptive-precision extension. The results highlight the practical benefits and limitations of reduced-precision arithmetic in a nested MLMC framework.

For the classic mixed-precision scheme, the use of half-precision arithmetic on coarse levels initially delivers substantial efficiency gains, since the coarsest levels dominate the total cost. This explains why the largest savings are achieved at these levels, and why reduced precision is most impactful in this regime. However, at fine levels, the accumulation of rounding errors degrades performance. Regression estimates confirm this, with weak convergence and variance decay rates reduced to $\alpha \approx 0.73$

and $\beta \approx 0.67$ when using the Linear-Dyadic LUT, compared with theoretical values of $(1, 1, 1)$. This behaviour is consistent with observations in [38], which showed that rounding effects worsen as grids are refined. In particular, the Linear-Dyadic LUT fails to converge under strict tolerances, demonstrating the vulnerability of the scheme beyond a critical refinement level L_{critical}.

The adaptive-precision scheme addresses this limitation by switching to full FP32 beyond a cut-off level of $l = \widetilde{L} = 5$. This hybrid approach retains the efficiency of half-precision on coarse levels (keeping majority of cost savings), while eliminating the rounding-error bias on fine levels. As a result, the regression estimates improve substantially across all LUTs: $\alpha \approx 0.83$ and $\beta \approx 0.96$ for the Linear-Dyadic. Cost comparisons confirm that the adaptive scheme achieves the lowest computational cost across all error tolerances, outperforming both FP32-only and fully mixed-precision MLMC.

It is worth noting that not all implementations experience the same degradation on fine levels with reduced-precision. For example, [24] did not observe this effect in FPGA-based simulations, since the optimisation framework automatically increased precision on refined grids. Similarly, adopting higher-order discretisations such as Milstein could place even greater emphasis on the coarse-level savings, where reduced precision has the strongest impact.

Taken together, these findings demonstrate that reduced-precision arithmetic can be successfully applied within MLMC for option pricing, provided it is carefully combined with adaptive strategies. The adaptive-precision approach tested in this work offers a practical balance between computational savings and numerical stability, and points towards broader applications of mixed-precision MLMC in high-performance computational finance and related fields.

Chapter 6

Conclusion

This thesis explored the potential computational gains achievable through reduced-precision arithmetic in a nested MLMC framework. The first stage compared LUT-based methods for generating low-precision Gaussian random variables, achieving genuine FP16 speedup over FP32 when vectorised using AVX-512 intrinsics - an advantage unavailable in Intel's MKL generator yet essential for hierarchical MLMC gains. We then assessed a sample cost model for payoff path calculations to quantify component costs and visualise the computational savings of the framework.

Numerical experiments confirmed that the mixed-precision MLMC scheme using reduced-precision arithmetic delivers cost savings. However, half-precision rounding errors degrade convergence at fine levels, resulting in smaller-than-theoretical weak convergence α and variance decay β. To address this limitation, an adaptive-precision extension was developed, introducing a cut-off level that switches the estimator to an FP32-only framework. This hybrid strategy retained the computational benefits of FP16 at coarse levels while restoring stability and convergence at fine levels.

Future work should include long-term comparisons against FPGA and GPU (CUDA) implementations, with particular focus on energy and cost efficiency. On the CPU side, the AVX-512 vectorisation techniques in this work could be extended with OpenMP multithreading, which has demonstrated up to 64$\times$ speedups in separate tests by Prof. Giles. In parallel, the AVX-512 vectorisation techniques developed in this work may be pursued by future postgraduate research in Prof. Giles's group, particularly for applications to Lévy processes and related stochastic models.

Overall, this work demonstrates that adaptive mixed-precision MLMC using reduced-precision arithmetic enables efficient and reliable simulations for option pricing and related applications, extending the FPGA-based fixed-point approaches in [24] with a CPU-based floating-point nesting framework.

Appendix A

Subnormal Numbers

A critical feature of the IEEE-754 standard is the handling of numbers smaller than the smallest normal value 2^{-14} and 2^{-126} for FP16 and FP32 (although extremely rare), respectively. Rather than assigning these values as zero immediately (underflow), the standard supports subnormal (or denormal) numbers. When the exponent field E is zero, the number is interpreted as:

$$x_{\text{subnormal}} = (-1)^s \times 2^{1-b} \times (F), \tag{A.1}$$

where s is the sign bit, F is the fractional mantissa ($0 \leq F < 1$), and the bias is defined as $b = 2^{e-1} - 1$. This allows for a representation down to $2^{-24} \approx 5.96 \times 10^{-8}$ for FP16 and $2^{-149} \approx 1.4 \times 10^{-45}$ for FP32. While essential for maintaining relative accuracy near zero and avoiding catastrophic cancellation, arithmetic on subnormal numbers can be significantly slower on many hardware architectures. This is a particularly relevant consideration for the MLMC method, as the fundamental operation of calculating correction terms $\mathbb{E}[P_\ell - P_{\ell-1}]$ inherently involves taking the difference between two correlated, and therefore nearly equal, quantities.

Appendix B

Signed Error Plots for LUT Approximations

For completeness, we include the signed error plots of the LUT approximations in Appendix A. These plots illustrate the oscillatory bias patterns of the approximations, complementing the maximum absolute error results presented in the main text.

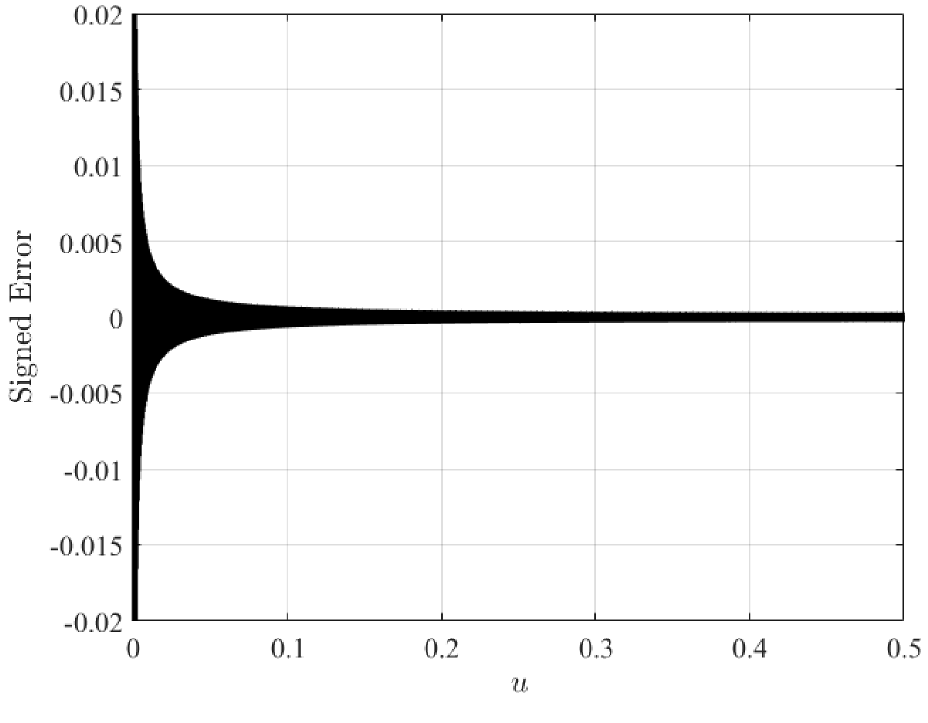


Figure B.1: Signed error for the piecewise constant approximation of $\Phi^{-1}(u)$ on uniform intervals.

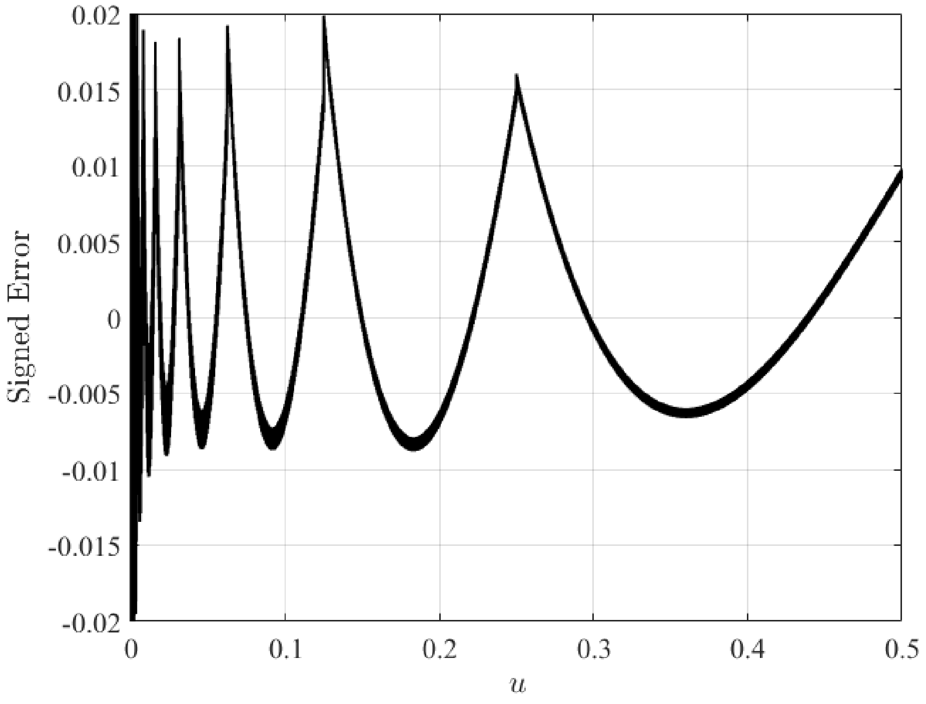


Figure B.2: Signed error for the piecewise linear approximation of $\Phi^{-1}(u)$ on dyadic intervals.

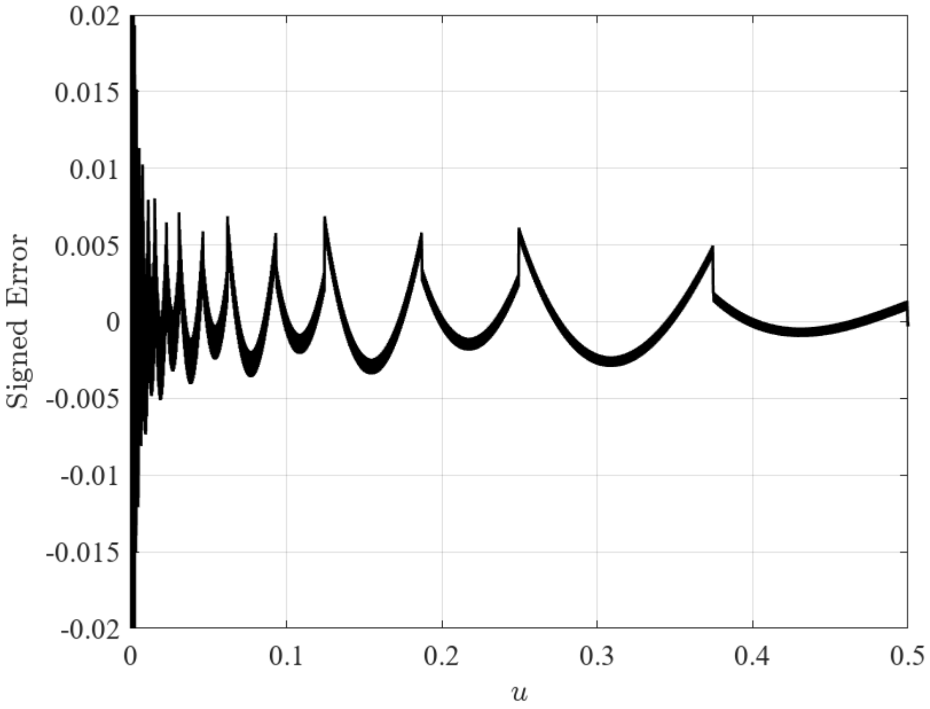


Figure B.3: Signed error for the piecewise linear approximation of $\Phi^{-1}(u)$ on superdyadic intervals.

Appendix C

Alternate Piecewise Linear Approximation Algorithm

Instead of the exponent-bit indexing technique described in Section 3, one may locate the appropriate dyadic or superdyadic segment using a linear search over the edge array. This leads to an evaluation cost that grows linearly with d, since the algorithm requires a linear search through the `EDGES` array, followed by a constant number of multiply-add operations, as described in Algorithm C.1.

Algorithm C.1 Gaussian sample generation for piecewise linear approximation (using linear search)

Input: Uniform random number $u \in [0, 1]$ (in `FP32`), dyadic edge array `EDGES`, LUT coefficients arrays `A`, `B`

Output: Gaussian sample z (`FP16` or `FP32`, depending on precision desired)

```
$N \leftarrow \text{length}(\texttt{EDGES}) - 1$
Clamp $u$ within $[10^{-7}, 1 - 10^{-7}]$ to avoid tails
if $u < 0.5$ then
    $\texttt{idx} \leftarrow$ index where $\texttt{EDGES}[\texttt{idx}] \leq u < \texttt{EDGES}[\texttt{idx} + 1]$
    if $\texttt{idx} < 0$ then $\texttt{idx} \leftarrow 0$
    if $\texttt{idx} \geq N$ then $\texttt{idx} \leftarrow N - 1$
    $z \leftarrow \texttt{A}[\texttt{idx}] + \texttt{B}[\texttt{idx}] \times (u - \texttt{EDGES}[\texttt{idx}])$
else
    $\texttt{idx} \leftarrow$ index where $\texttt{EDGES}[\texttt{idx}] \leq (1 - u) < \texttt{EDGES}[\texttt{idx} + 1]$
    if $\texttt{idx} < 0$ then $\texttt{idx} \leftarrow 0$
    if $\texttt{idx} \geq N$ then $\texttt{idx} \leftarrow N - 1$
    $z \leftarrow -\left(\texttt{A}[\texttt{idx}] + \texttt{B}[\texttt{idx}] \times ((1 - u) - \texttt{EDGES}[\texttt{idx}])\right)$
end if
```

Appendix D

AVX-512 Intrinsics for SIMD Vectorisation

This appendix records the exact AVX-512 intrinsics used for SIMD implementations in this work. Table D.1, D.2 and D.3, follows the sequential step by step procedure in Algorithm 3.1 and 3.2 for the Gaussian-based LUT methods, respectively.

Similarly, Table D.4 presents the AVX-512 intrinsics used for the SIMD implementation for the independent payoff paths in half- and full-precision, respectively. It follows the step-by-step procedure outlined in Algorithm 1.1.

Table D.1: Mapping of Algorithm 3.1 steps for the piecewise constant approximation to AVX-512 intrinsics.

Algorithm 3 Step	**AVX-512 Intrinsics**	**Description**
Clamp u within $[10^{-7}, 1 - 10^{-7}]$	`_mm512_max_ps`, `_mm512_min_ps`	Enforces lower and upper bounds to avoid out-of-range indices in the LUT.
Half-interval test $u < 0.5$	`_mm512_cmp_ps_mask`	Produces a lane mask indicating which uniforms belong to the lower half.
Mirror $u \mapsto 1 - u$ if $u \geq 0.5$	`_mm512_mask_blend_ps`	Blends original and mirrored uniforms using the mask to map all inputs to $[0, 0.5]$.
Index computation	`_mm512_mul_ps`, `_mm512_cvttps_epi32`, `_mm512_min_epi32`	Multiplies by LUT scale, converts to integer indices, and clamps to valid range.
LUT lookup	`_mm512_i32gather_ps` (FP32), or custom `gather_fp16` (FP16)	Gathers Gaussian constants from the FP32 or FP16 duplicated LUT.
Apply sign (odd symmetry about $u = 0.5$)	`_mm512_sub_ps` (negate), `_mm512_mask_blend_ps`	Restores correct sign for upper-half samples.
Store output	`_mm512_storeu_ps` (FP32), `_mm512_cvtps_ph`+ `_mm512_storeu_ph` (FP16)	Packs results back into memory in the chosen precision.

Table D.2: Mapping of Algorithm 3.2 for the piecewise linear (dyadic vs. superdyadic), FP32: mapping of steps to AVX-512 intrinsics.

Algorithmic step	**Dyadic (FP32) Intrinsics**	**Superdyadic (FP32) Intrinsics**
Clamp u to $[2^{-14}, 0.5]$	`_mm512_max_ps`, `_mm512_min_ps`	`_mm512_max_ps`, `_mm512_min_ps`
Exponent / mantissa decode	`_mm512_castps_si512`, `_mm512_srli_epi32`, `_mm512_and_epi32`	`_mm512_castps_si512`, `_mm512_srli_epi32`, `_mm512_and_epi32`
Mantissa threshold at $\sqrt{2}$	Not applicable	`_mm512_cmp_epi32_mask`
Index remap & clamp [11]	`_mm512_sub_epi32`, `_mm512_min_epi32`, `_mm512_max_epi32`	`_mm512_sub_epi32`, `_mm512_slli_epi32`, `_mm512_mask_add_epi32`, `_mm512_min_epi32`
Select (A, B) coefficients	`_mm512_permutexvar_ps` (A, B)	`_mm512_i32gather_ps` (A), `_mm512_i32gather_ps` (B)
Evaluate $z = A + B\,u$	`_mm512_fmadd_ps`	`_mm512_fmadd_ps`
Restore sign	`_mm512_mask_xor_ps`	`_mm512_mask_xor_ps`
Store	`_mm512_storeu_ps`	`_mm512_storeu_ps`

[11] Implementation note: In practice, the mantissa threshold test for the superdyadic split ($m \geq m_{\sqrt{2}}$) is replaced by an equivalent exponent-of-squares check: squaring the magnitude and comparing the exponent of u^2, following a suggestion by Prof. Mike Giles. The same approach is applied in the FP16 implementation, as detailed in Table D.3.

Table D.3: Mapping of Algorithm 3.2 for the piecewise linear (dyadic vs. superdyadic), FP16: mapping of steps to AVX-512 intrinsics.

Algorithmic step	**Dyadic (FP16) Intrinsics**	**Superdyadic (FP16) Intrinsics**
Clamp u to $[2^{-14}, 0.5]$	`_mm512_max_ph`, `_mm512_min_ph`	`_mm512_max_ph`, `_mm512_min_ph`
Exponent / mantissa decode	`_mm512_castph_si512`, `_mm512_and_si512`, `_mm512_srli_epi16`	`_mm512_castph_si512`, `_mm512_and_si512`, `_mm512_srli_epi16`
Mantissa threshold at $\sqrt{2}$	Not applicable	`_mm512_cmp_epi16_mask`
Index remap & clamp	`_mm512_sub_epi16`, `_mm512_min_epi16`, `_mm512_max_epi16`	`_mm512_slli_epi16`, `_mm512_mask_add_epi16`, `_mm512_min_epi16`
Select (A, B) coefficients	`_mm512_maskz_permute..` ..`xvar_epi16` (A, B)	`_mm512_maskz_permute..` ..`xvar_epi16` (A, B)
Evaluate $z = A + B\,u$	`_mm512_fmadd_ph`	`_mm512_fmadd_ph`
Restore sign	`_mm512_mask_mul_ph`	`_mm512_mask_mul_ph`
Store	`_mm512_storeu_ph`	`_mm512_storeu_ph`

Table D.4: Mapping of Algorithm 1.1 (Euler–Maruyama GBM payoff path) to AVX-512 intrinsics used in this thesis.

Algorithmic step	**FP32 intrinsics**	**FP16 intrinsics**
Initialise constants rh, $\sigma\sqrt{h}$, K, 1, disc	`_mm512_set1_ps`	`_mm512_set1_ph`
Initialise state $S \leftarrow S_0$	`_mm512_set1_ps`	`_mm512_set1_ph`
Per-step loop $(i = 1, \ldots, M)$*:*		
·Load Z_i	`_mm512_loadu_ps`	`_mm512_loadu_ph`
·FMA: $tmp \leftarrow \sigma\sqrt{h}\, Z_i + rh$	`_mm512_fmadd_ps`	`_mm512_fmadd_ph`
·Add one: $tmp \leftarrow tmp + 1$	`_mm512_add_ps`	`_mm512_add_ph`
·Update: $S \leftarrow S \cdot tmp$	`_mm512_mul_ps`	`_mm512_mul_ph`
Terminal difference $diff \leftarrow S - K$	`_mm512_sub_ps`	`_mm512_sub_ph`
Mask for $(S{-}K)^+ : m \leftarrow (diff > 0)$	`_mm512_cmp_ps_mask`	`_mm512_cmp_ph_mask`
Discounted payoff: $(S{-}K)^+ \cdot \text{disc}$	`_mm512_mask_mul_ps`	`_mm512_mask_mul_ph`
Store payoff vector	`_mm512_storeu_ps`	`_mm512_storeu_ph`

Appendix E

Payoff Path Calculations

Table E.1 reports full per-path and per-path-step timings (in ns) for FP16 and FP32 arithmetic across levels l, corresponding to step counts $M = 2^l$. Values are recorded as the mean $\pm$ standard deviation over $R = 20$ runs, with RNG costs excluded. Speedup and scaling factors are derived from these measurements. Table E.2 compares the end-to-end per-sample RNG timings $g_p^{(m)}$ for the Gaussian-based LUT methods in FP16 and FP32. Reported values highlight that FP16 provides a consistent performance advantage with the Linear-Superdyadic method achieving the highest relative speedups (mean $\sim 1.9\times$).

Table E.1: Full per-path and per-path-step timings (ns) for FP16 and FP32 across levels $l \in \{0, 1, 2, \dots, 9\}$, corresponding to step count $M \in \{1, 2, 4, 8, \dots, 512\}$. Timings are recorded in mean $\pm$ SD over $R = 20$ runs; RNG times excluded. Figure 4.1 is derived from these data.

M	**Per-path (ns)**		**Per-path-step (ns)**		**Speedup**	**Scaling**
	FP16	**FP32**	**FP16**	**FP32**		
1	0.223 ± 0.022	0.403 ± 0.060	0.223 ± 0.022	0.403 ± 0.060	$1.81\times$	$1.00\times$
2	0.271 ± 0.016	0.557 ± 0.034	0.1355 ± 0.0080	0.279 ± 0.017	$2.06\times$	$1.38\times$
4	0.438 ± 0.071	0.87 ± 0.15	0.109 ± 0.018	0.218 ± 0.038	$1.99\times$	$2.16\times$
8	0.944 ± 0.059	1.80 ± 0.16	0.118 ± 0.0074	0.225 ± 0.019	$1.91\times$	$4.47\times$
16	2.92 ± 0.50	5.5 ± 1.0	0.183 ± 0.032	0.341 ± 0.065	$1.87\times$	$13.54\times$
32	5.57 ± 0.61	11.0 ± 1.5	0.174 ± 0.019	0.344 ± 0.046	$1.98\times$	$27.31\times$
64	12.2 ± 1.6	26.9 ± 6.1	0.190 ± 0.025	0.420 ± 0.096	$2.21\times$	$66.66\times$
128	39.2 ± 4.4	83 ± 11	0.308 ± 0.035	0.648 ± 0.089	$2.11\times$	$205.57\times$
256	76.8 ± 7.0	158 ± 12	0.300 ± 0.027	0.616 ± 0.049	$2.05\times$	$391.24\times$
512	158 ± 23	322 ± 38	0.308 ± 0.045	0.630 ± 0.074	$2.04\times$	$799.61\times$

Table E.2: Comparison of end-to-end per-sample timings $g_p^{(m)}$ for the different Gaussian-based LUT method in FP16 and FP32 across levels l [ns/Sample].

l	**Constant-Uniform**			**Linear-Dyadic**			**Linear-Superdyadic**		
	FP16	**FP32**	**Speedup**	**FP16**	**FP32**	**Speedup**	**FP16**	**FP32**	**Speedup**
0	0.760	0.895	1.18×	0.457	0.556	1.22×	0.486	1.046	2.15×
1	0.791	0.886	1.12×	0.473	0.568	1.20×	0.501	1.018	2.03×
2	0.759	0.901	1.19×	0.440	0.544	1.24×	0.476	1.041	2.19×
3	0.770	0.918	1.19×	0.436	0.542	1.24×	0.470	1.071	2.28×
4	0.848	0.967	1.14×	0.490	0.605	1.23×	0.526	1.116	2.12×
5	0.859	0.968	1.13×	0.562	0.662	1.18×	0.585	1.101	1.88×
6	0.940	1.043	1.11×	0.644	0.726	1.13×	0.645	1.199	1.86×
7	1.024	1.181	1.15×	0.722	0.815	1.13×	0.722	1.254	1.74×
8	1.329	1.418	1.07×	0.912	1.078	1.18×	0.914	1.454	1.59×
9	1.730	1.789	1.03×	1.225	1.486	1.21×	1.229	1.721	1.40×
Mean	0.981	1.097	1.13×	0.636	0.758	1.20×	0.655	1.202	1.92×

Appendix F

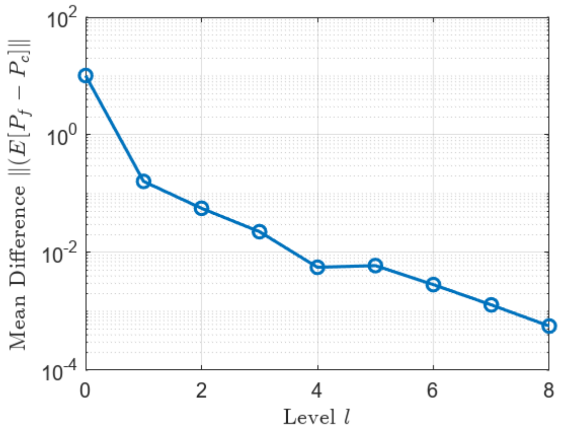
Complete Multilevel Monte Carlo Simulations

For completeness, this appendix presents the full set of diagnostic plots for the pure FP32 MLMC framework, analogous to those shown for the mixed FP16/FP32 scheme in Figure 5.1 of the main text. These plots illustrate the expected behaviour of MLMC under standard single-precision arithmetic and serve as a control baseline against which the mixed-precision results may be compared. Table F.1 presents the linear regression estimates for the pure FP32 MLMC across the different LUT methods explored in this thesis.

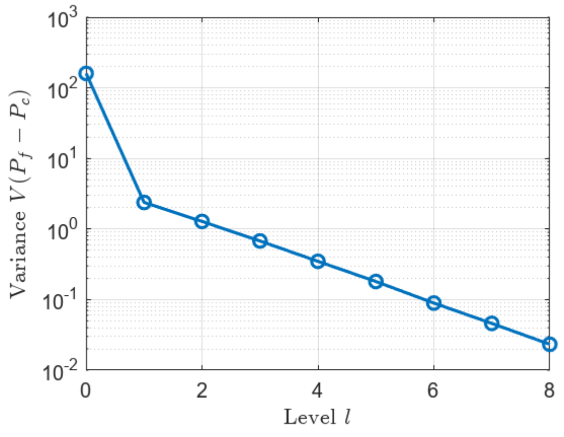
Table F.1: Linear regression estimates of MLMC parameters across different LUT-based methods. The parameters are defined as in Theorem 1.1, where α is the weak/bias convergence, β is the variance decay, and γ is the cost growth.

LUT Method	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	γ
Constant-Uniform	1.00	0.97	1.00
Linear-Dyadic	0.93	0.96	1.00
Linear-Superdyadic	0.94	0.96	1.00

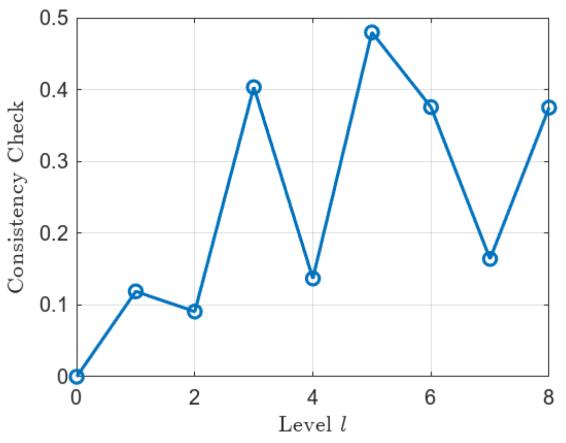
Figures F.2, F.3, F.4, and F.5 present the numerical results of the classic and adaptive-precision mixed-precision nested MLMC for the Constant-Uniform and Linear-Superdyadic LUT.



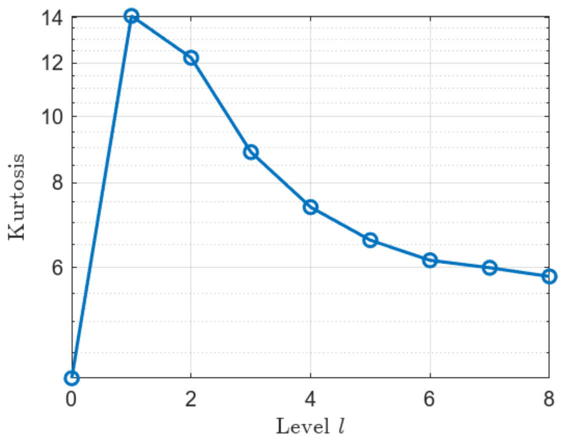
(a) Mean payoff difference $\|E[P_l - P_{l-1}]\|$ across levels l



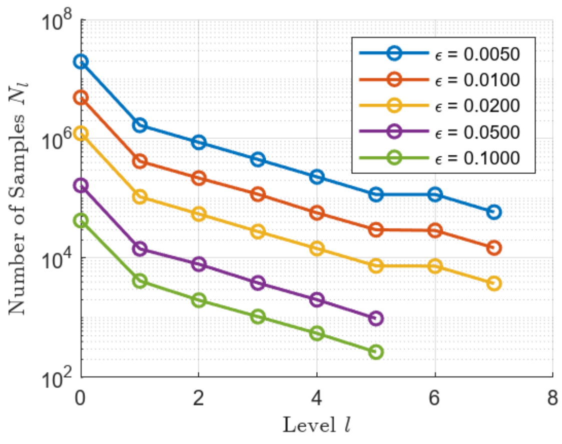
(b) Variance $V[P_l - P_{l-1}]$ across levels l.



(c) Consistency check of FP16–FP32 correction term across levels l.

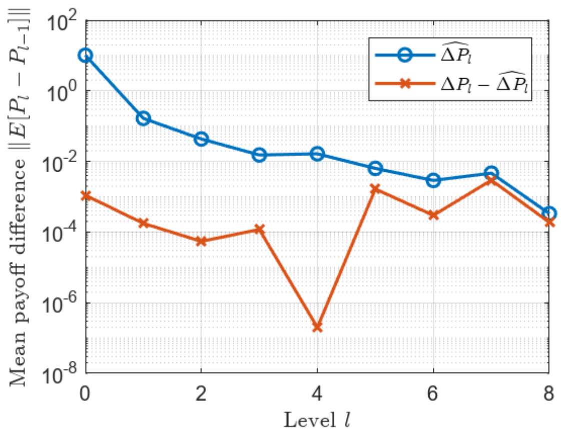


(d) Kurtosis of payoff difference distribution across levels l.

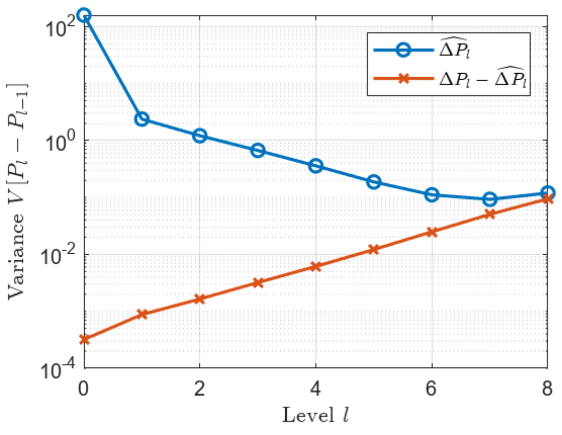


(e) Optimal sample allocation N_l across levels l, for varying error tolerances ε.

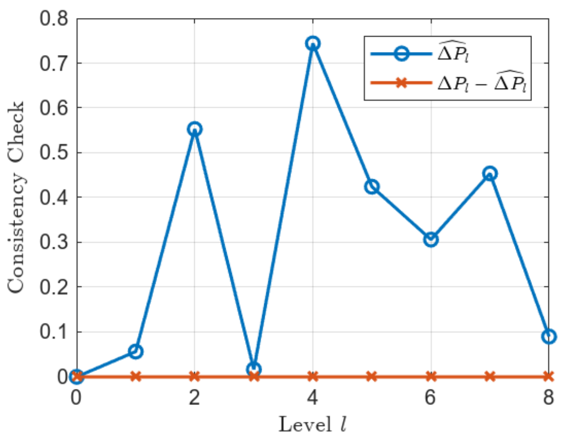
Figure F.1: Numerical results for a European call option using the Euler-Maruyama discretisation of the GBM SDE in a standard FP32-only MLMC.



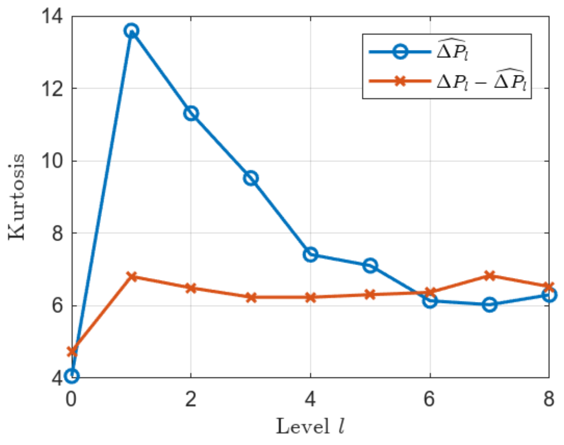
(a) Mean payoff difference $\|\mathbb{E}[P_l - P_{l-1}]\|$ across levels l



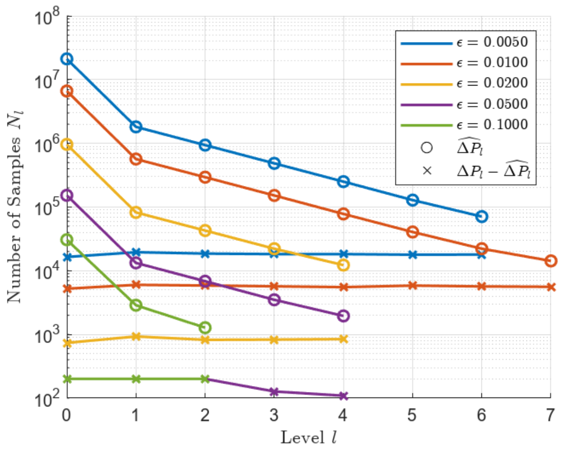
(b) Variance $\mathbb{V}[P_l - P_{l-1}]$ across levels l.



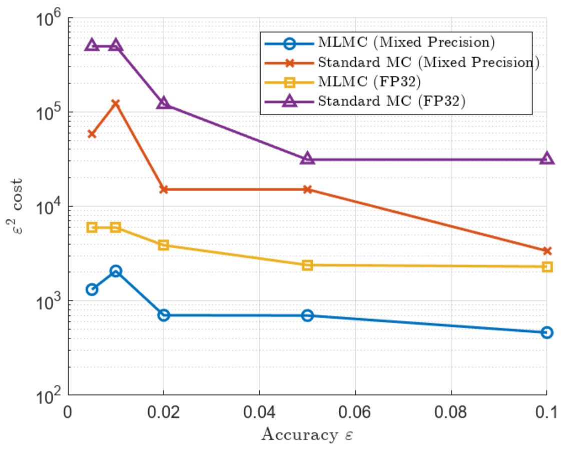
(c) Consistency check of FP16–FP32 correction term across levels l.



(d) Kurtosis of payoff difference distribution across levels l.

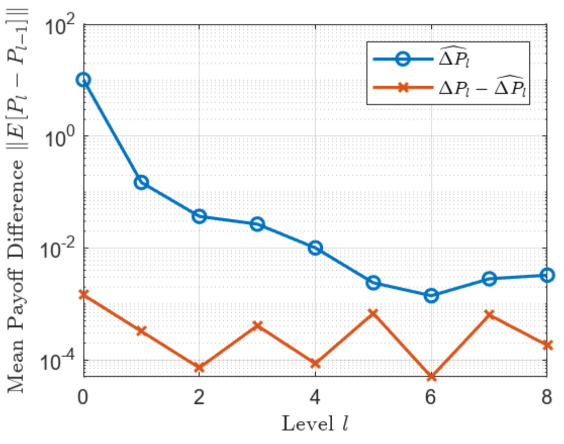


(e) Optimal sample allocation N_l across levels l, for varying error tolerance ε.

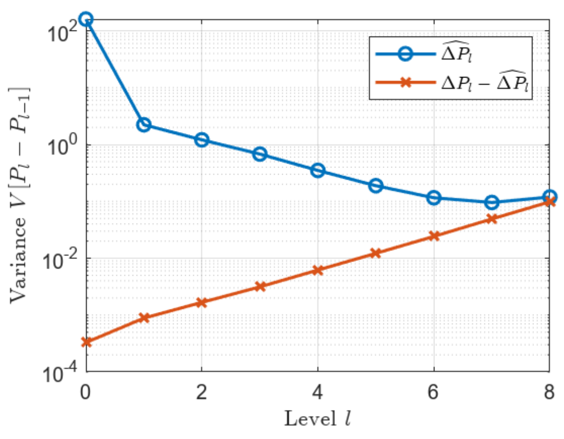


(f) Cost comparison of standard MC and classic MLMC.

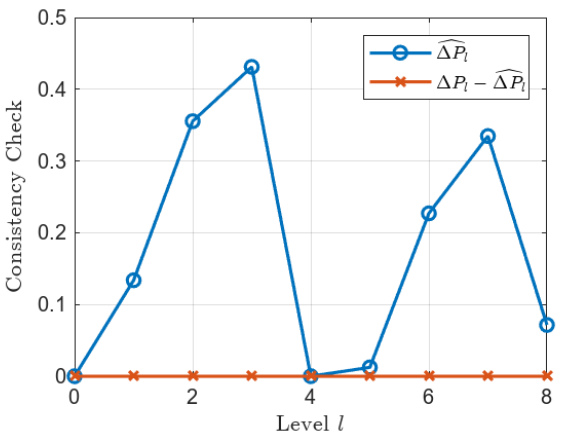
Figure F.2: Numerical results for a European call option using the Euler-Maruyama discretisation of the GBM SDE in a classic nested MLMC (**Constant-Uniform**).



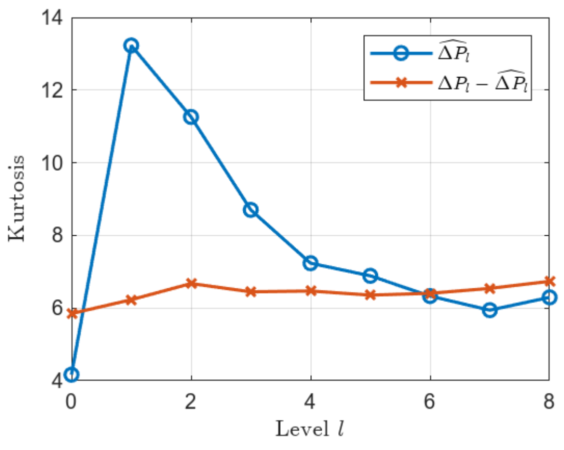
(a) Mean payoff difference $\|\mathbb{E}[P_l - P_{l-1}]\|$ across levels l



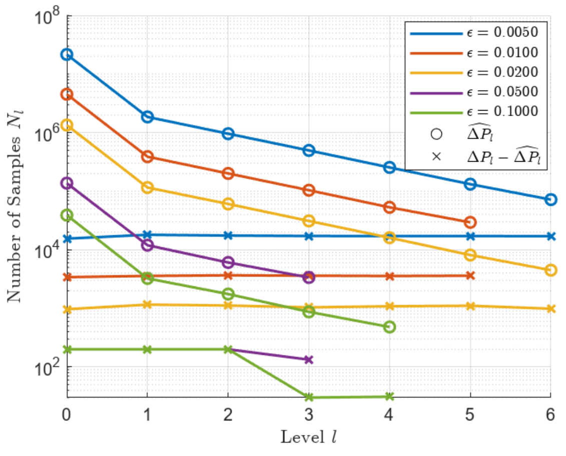
(b) Variance $\mathbb{V}[P_l - P_{l-1}]$ across levels l.



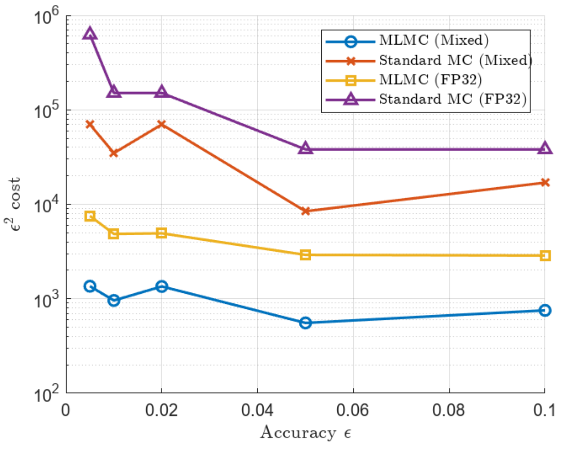
(c) Consistency check of FP16–FP32 correction term across levels l.



(d) Kurtosis of payoff difference distribution across levels l.

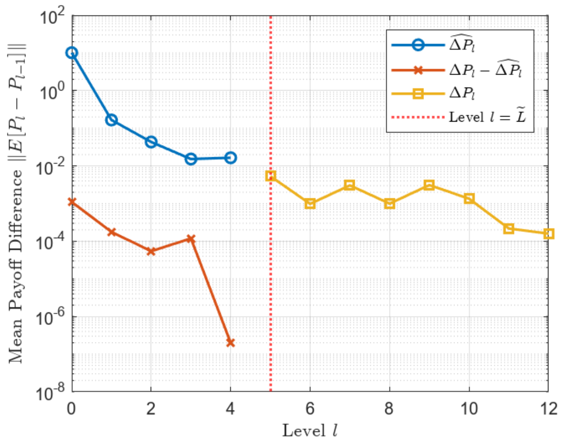


(e) Optimal sample allocation N_l across levels l, for varying error tolerance ε.

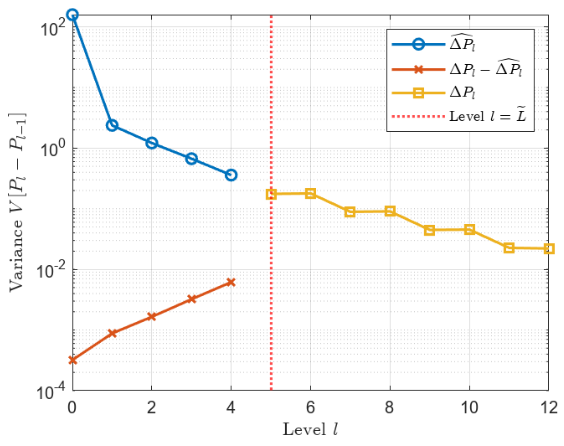


(f) Cost comparison of standard MC and classic MLMC.

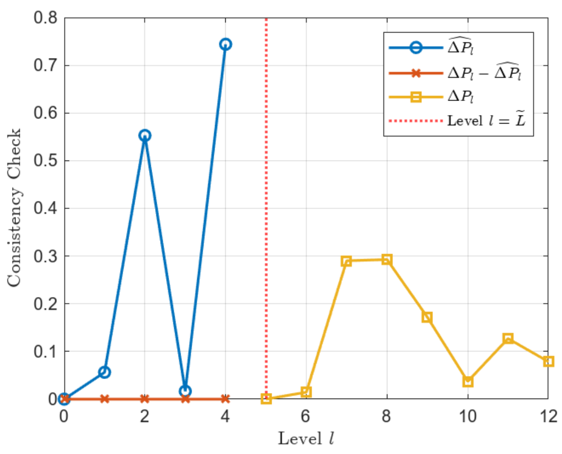
Figure F.3: Numerical results for a European call option using the Euler-Maruyama discretisation of the GBM SDE in a classic nested MLMC (**Linear-Superdyadic**).



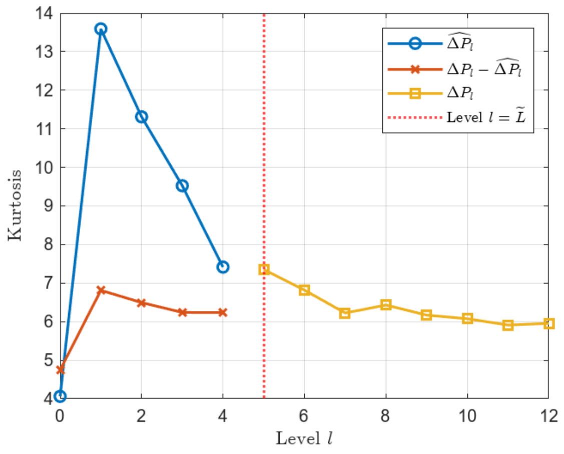
(a) Mean payoff difference $\|\mathbb{E}[P_l - P_{l-1}]\|$ across levels l



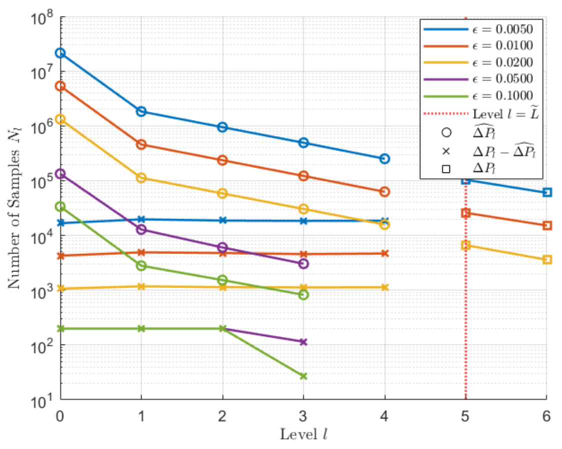
(b) Variance $\mathbb{V}[P_l - P_{l-1}]$ across levels l.



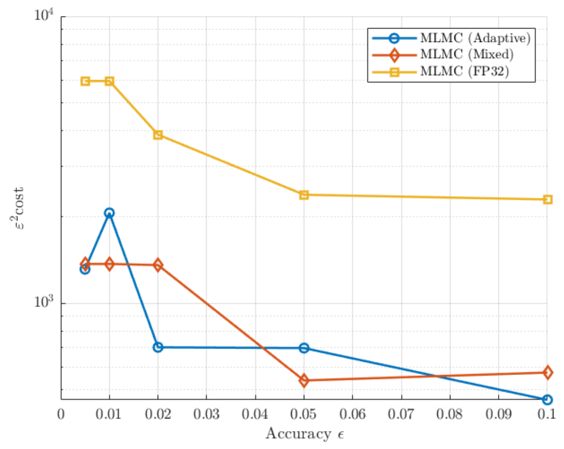
(c) Consistency check of FP16–FP32 correction term across levels l.



(d) Kurtosis of payoff difference distribution across levels l.

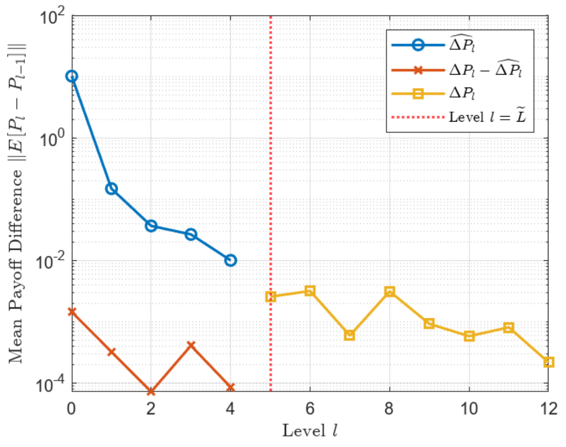


(e) Optimal sample allocation N_l across levels l, for varying error tolerance ε.

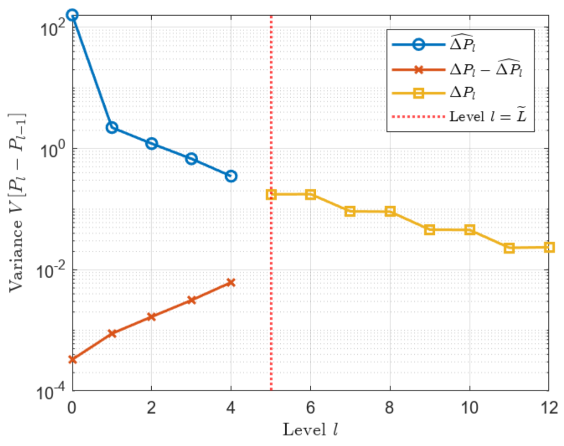


(f) Cost comparison of standard MLMC, classic nested MLMC, and adaptive MLMC.

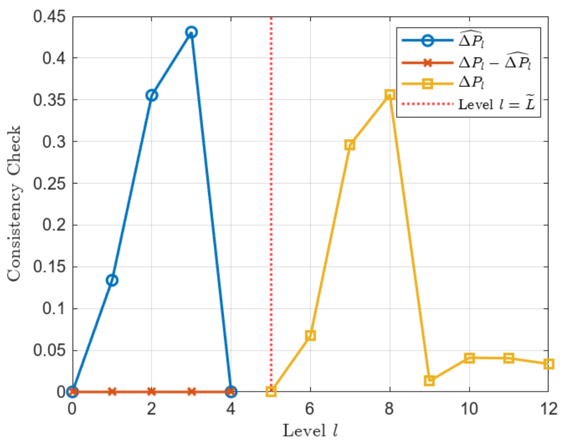
Figure F.4: Numerical results for a European call option using the Euler-Maruyama discretisation of the GBM in an adaptive-precision MLMC (**Constant-Uniform**).



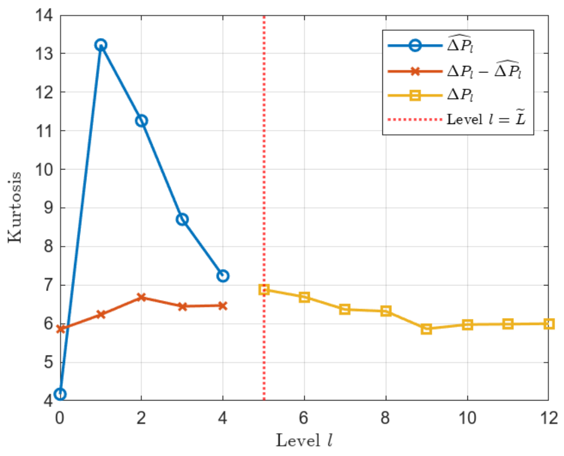
(a) Mean payoff difference $\|\mathbb{E}[P_l - P_{l-1}]\|$ across levels l



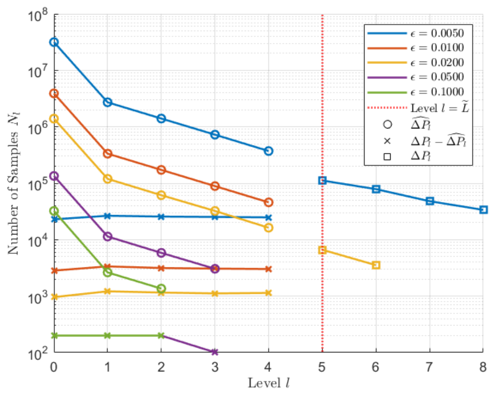
(b) Variance $\mathbb{V}[P_l - P_{l-1}]$ across levels l.



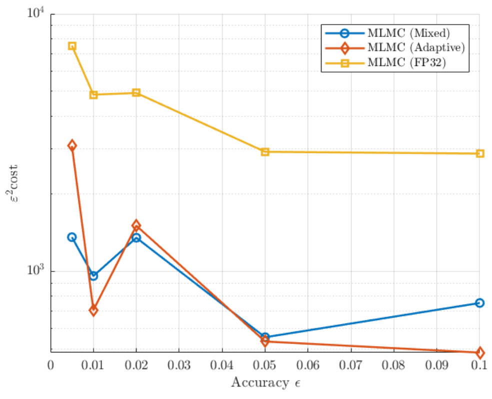
(c) Consistency check of FP16–FP32 correction term across levels l.



(d) Kurtosis of payoff difference distribution across levels l.



(e) Optimal sample allocation N_l across levels l, for varying error tolerance ε.



(f) Cost comparison of standard MLMC, classic nested MLMC, and adaptive MLMC.

Figure F.5: Numerical results for a European call option using the Euler-Maruyama discretisation of the GBM in an adaptive-precision MLMC (**Linear-Superdyadic**).

Bibliography

[1] J.G. Amar. The Monte Carlo method in science and engineering. *Computing in science & engineering*, 8(2):9–19, 2006.

[2] ARM Limited. *Types of Parallelism*, 2021. Document ID: 101574-0601.

[3] S. Boldo, C.P. Jeannerod, G. Melquiond, and J.M. Muller. Floating-point arithmetic. *Acta Numerica*, 32:203–290, 2023.

[4] M. Bouzoubaa. Pricing Vanilla Options. In *Equity Derivatives Explained*, pages 38–53. Springer, 2014.

[5] P.P. Boyle. Options: A Monte Carlo approach. *Journal of Financial Economics*, 4(3):323–338, 1977.

[6] G.L. Chen. *Indicator Variables.* San José State University, Math 261A: Regression Theory & Methods, 2020.

[7] R.C.C. Cheung, D.U. Lee, W. Luk, and J.D. Villasenor. Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method. *IEEE transactions on very large scale integration (VLSI) systems*, 15(8):952–962, 2007.

[8] G. Chow, W. Luk, and P. Leong. A mixed precision methodology for mathematical optimisation. pages 33–36, 2012.

[9] Intel Corporation. *AVX-512 FP16 Instruction Set for Intel Xeon Processor-based Products Technology Guide*, 2023. Document ID: 669773.

[10] Intel Corporation. *Intel oneAPI Math Kernel Library (oneMKL): Documentation for vRngGaussian*, 2023.

[11] Intel Corporation. *Intel oneAPI Math Kernel Library (oneMKL): Random Number Generators Naming Conventions* , 2023.

[12] Intel Corporation. *Intel Xeon Gold 6538Y Processor (60M Cache, 2.20 GHz) Specifications*, 2023.

[13] Intel Corporation. *Porting Guide for ICC Users to DPC++ or ICX*, 2023.

[14] M. Davis, V. Panas, and T. Zariphopoulou. European option pricing with transaction costs. *SIAM Journal on Control and Optimization*, 31(2):470–493, 1993.

[15] M. Giles. Multilevel Monte Carlo path simulation. *Operations research*, 56(3):607–617, 2008.

[16] M. Giles. Multilevel Monte Carlo methods. *Acta Numerica*, 24:259–328, 2015.

[17] M. Giles. MLMC for nested expectations. *Contemporary computational mathematics-A celebration of the 80th birthday of ian sloan*, pages 425–442, 2018.

[18] M. Giles. *AVX-512 vector intrinsics*. Mathematical Institute, University of Oxford, 2025. Personal Webpage, Header file for operator-overloaded fp16x32 class (alias to `__m512h`).

[19] M. Giles and O. Sheridan-Methven. Analysis of nested multilevel monte carlo using approximate normal random variables. *SIAM/ASA Journal on Uncertainty Quantification*, 10(1):200–226, 2022.

[20] M. Giles and O. Sheridan-Methven. Approximating inverse cumulative distribution functions to produce approximate random variables. *ACM Transactions on Mathematical Software*, 49(3):1–29, 2023.

[21] M. Giles and B.J. Waterhouse. Multilevel quasi-monte carlo path simulation. *Advanced Financial Modelling, Radon Series on Computational and Applied Mathematics*, 8:165–181, 2009.

[22] P. Glasserman. Monte Carlo methods in financial engineering. *Springer*, 53, 2004.

[23] J. Gonzalez-Conde, A. Rodriguez-Rozas, E. Solano, and M. Sanz. Simulating option price dynamics with exponential quantum speedup. *arXiv preprint arXiv:2101.04023*, 2022.

[24] I.B. Haas and M. Giles. A nested MLMC framework for efficient simulations on FPGAs. *Monte Carlo Methods and Applications*, (0), 2025.

[25] C. J Hughes. *Single-instruction multiple-data execution.* Springer Nature, 2022.

[26] Intel Corporation. *Introduction to Accelerator and Parallel Programming*, 2023. Technical Article.

[27] Intel Corporation. *VSL Vector Random Number Generators: vRngUniform (Fortran)*, 2024.

[28] Intel Corporation. *DPC++/C++ Compiler Developer Guide and Reference*, 2025.

[29] W. Kahan. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[30] A Kiessling. An introduction to parallel programming with OpenMP. In *The University of Edinburgh, A Pedagogical Seminar*, volume 76, 2009.

[31] D.U. Lee, R.C.C. Cheung, W. Luk, and J.D. Villasenor. Hierarchical segmentation for hardware function evaluation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(1):103–116, 2008.

[32] P. Markstein. The new IEEE-754 standard for floating point arithmetic. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2008.

[33] E.S. Oran, C.K. Oh, and B.Z. Cybyk. Direct simulation Monte Carlo: recent advances and applications. *Annual Review of Fluid Mechanics*, 30(1):403–441, 1998.

[34] E.H. Park, D.Y. Kim, and S.J. Yoo. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. pages 688–698, 2018.

[35] C.P. Robert, G. Casella, and G. Casella. *Introducing Monte Carlo methods with R*, volume 18. Springer, 2010.

[36] Rutgers University. C Tutorial: Use Linux's high resolution clock.

[37] D.W. Scott. Box-Muller transformation. *Wiley Interdisciplinary Reviews: Computational Statistics*, 3(2):177–179, 2011.

[38] O. Sheridan-Methven. *Nested multilevel Monte Carlo methods and a modified Euler-Maruyama scheme utilising approximate Gaussian random variables suitable for vectorised hardware and low-precisions.* PhD thesis, University of Oxford, 2021.

[39] K. Suganthi and G. Jayalalitha. Geometric brownian motion in stock prices. In *Journal of Physics: Conference Series*, volume 1377, page 012016. IOP Publishing, 2019.

[40] D.B. Thomas, L. Howes, and W. Luk. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72, 2009.

[41] J. E. Thornton. Design of a computer: The Control Data 6600. *Scott, Foresman & Co.*, 1970.

[42] H. Wang. *Monte Carlo simulation with applications to finance.* CRC Press, 2012.

[43] A. Zalani. Low-Latency Machine Learning for Options Pricing: High-Speed Models and Trading Performance. *Journal of Computer Science and Technology Studies*, 7(5):65–72, 2025.