Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Multilevel Monte Carlo Methods for simulation of sensitivities

M.Sc. High Performance Computing

Harry Hamilton
Supervisor: Darach Golden

School of Mathematics
Trinity College Dublin

Contents

1 Introduction

Monte Carlo methods are commonly used in computational finance to compute $\mathbb{E}[P]$. P is the discounted payoff depending on some underlying asset's price $S(t)$, which is governed by an evolution stochastic differential equation (SDE) of the form

$$dS(t) = a(S,t)dt + b(S,t)dW(t) \tag{1}$$

Just as essential is the computation of Greeks, the first and second order derivatives of the prices with respect to input parameters such as the interest rate, current asset price and volatility. In practical purposes this is of vital importance with respect to both price changes of the underlying asset and changes in the model parameters. The former gives a measure of risk exposure in a portfolio, that is, how the value of a portfolio will change given a change in the underlying asset price. The latter reason is more subtle but of particular consequence in computational finance as in this case we are not trying to measure risk exposure but rather sensitivity with respect to misspecifications of the model parameters.

The ubiquity of the Monte Carlo approach in finance can be mainly attributed to its computational efficiency for problems in high-dimensions involving multiple assets and interest rates but also due to the ease of parallelisation across compute-clusters. Our focus in this project is on Multilevel Monte Carlo (MLMC), first introduced by Giles. While the cost of Monte Carlo does not grow with stochastic dimension it carries the disadvantage of being very slow to converge. MLCM seeks to address this issue by using a hierarchy of discretisations of increasing accuracy. We use a large number of coarse samples to capture variability and take a few fine samples to eliminate bias due to discretisation. This approach reduces variance and leads to better convergence.

Another development over the last decade has been the design and implementation of efficient Automatic differentiation, which comes in two flavours. The first of these is forward mode, in which a tangent-linear version of a program is built. The cost pf computing sensitivity of each output of a program with respect to the inputs in this mode is to the number of inputs. This carries the same computational complexity as a finite difference calculation but withe the benefit that derivatives are computed with machine accuracy.

The second approach is Reverse mode Automatic differentiation, more commonly called adjoint automatic differentiation (AAD). This approach involves producing an adjoint program, which allows computation of the gradient at some small cost multiple of running the original program. This means that for a large number of input parameters this approach quickly outperforms forward mode.

The goal of this paper is to study the interplay between MLMC and AAD in the context of computational finance and to address the advantages and disadvantages of each approach.

2 Theory

2.1 Financial Option Theory

We define an option to be a contract between two parties where the value is determined by the future price of some underlying asset. The buyer has the right but not the obligation to participate in a transaction. Types of options are usually specified by the exercise type and the payoff function.

Three common exercise types are European, American, and Bermudan style options. A European style contract may only be exercised at the options time of maturity. An American style contract may be exercised at any time up to the expiration date. For this reason a European style contract will never carry more worth than an American style contract with strike price and expiration date held constant. The final common form of contract is Bermudan style contracts, which may only be exercised on specified dates. This type of contract is usually found in interest rate and swap markets.

The payoff function of an option is based on a set of parameters such as the strike price K along with the price of some underlying asset. The functions which we shall be simulating will be Lipshitz continuous. While simulation of discontinuous payoffs via MLMC is possible, it carries a unique set of challenges which reduce the effectiveness of such a method. These will be discussed in the results section. A portfolio is said to be self-financing if there are no external infusions or withdrawals of capital.

To take an example, consider an asset price process given by the sequence $S = (S_t)$ for $t \geq 0$. A European call option gives the holder of the contract the the right to purchase one unit of the asset at the time of maturity T. Thus the payoff at T is given by $max(0, S_T - K)$. A European put gives the holder the right to sell the stock at the strike price K at the expiration date. So in this case the payoff is given by $max(0, K - S_T)$.

It is worth mentioning that the Black-Scholes model carries several heavy assumptions. We shall not concern ourselves with these for the purposes of simulation but they bear mentioning in the context of comparing the reality of the market as compared to the models constructed in computational finance. The first of these is the Efficient Market Hypothesis, which states that asset prices are a reflection of all available information. Made by Eugene Fama in the 1970 article "Efficient Capital Markets" [1] from observations on security markets, it was meant to convey that when new information arises, it is quickly Incorporated into the price. Other assumptions made are infinite liquidity of the market, continuity of prices, the ability to trade continuously and a lack of transaction costs.

2.2 Greeks

As was discussed in the Introduction, the derivatives of option prices with respect to the input parameters are a essential diagnostic tool for measuring the sensitivity of a portfolio to changes in the market, which may yield not just changes in stock price, but other parameters such as volatility. To examine these sensitivities of the payoff S, we introduce the standard notation

Definition 1.

$$\Delta = \frac{\partial S}{\partial S_0}$$

$$\Gamma = \frac{\partial^2 S}{\partial S_0^2}$$

$$\rho = \frac{\partial S}{\partial r} \tag{2}$$

$$\Theta = \frac{\partial S}{\partial t}$$

$$\nu = \frac{\partial S}{\partial \sigma}$$

If a portfolio is insensitive with respect to small changes of the above parameters, it is said to be neutral.

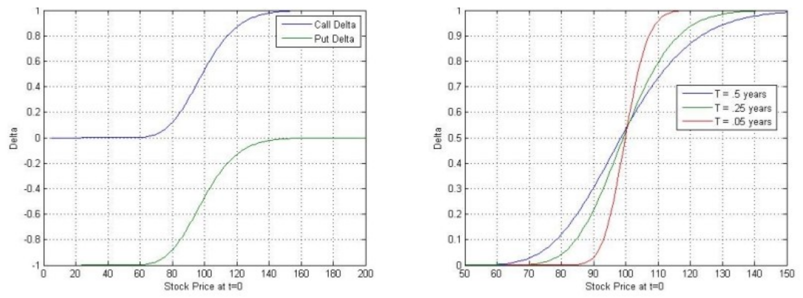


Figure 1: Δ for European put and call and against time to maturity

We can see from Figure 4 that $\Delta_{\text{put}} = \Delta_{\text{call}} - e^{rT}$. Δ measures sensitivity to changes in the price of the underlying asset.

Γ of an option measures the sensitivity of an options Δ against changes in the underlying asset.

ν of an option is the sensitivity against volatility changes.

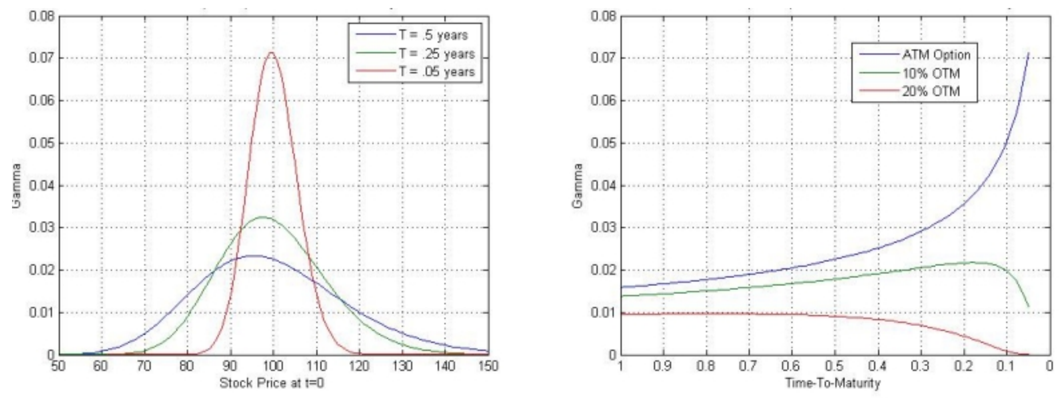


Figure 2: Γ for European option as function of stock price and time to maturity

Γ of an option measures the sensitivity of an options Δ against changes in the underlying asset.

Θ of an option is the sensitivity of the pricing function to a negative change in time to maturity.

2.3 Discretisation of Stochastic differential equations

SDE's come in the general form (1). The function $S(t)$ satisfies the integral equation

$$S(t) = S(0) + \int_0^T a(S(t'),t)\,dt' + \int_0^T b(S(t'),t)\,dt' \tag{3}$$

For a proof of existence and uniqueness see [2]. In simple cases, such as in the case of The SDE underlying the Black-Scholes model, integration via coefficient matching is possible. Consider

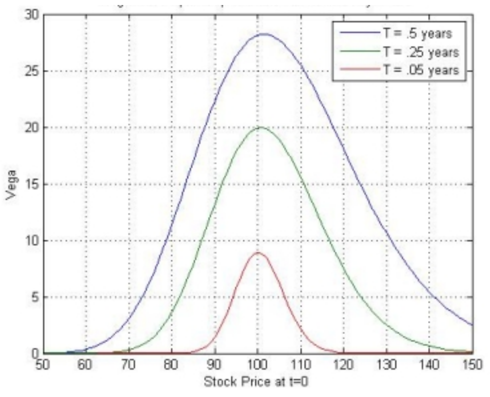
$$dS(t) = rS(t)dt + \sigma S(t)dW(t), S(0) = S_0 \geq 0 \tag{4}$$

where r, the risk-free interest rate and σ, the volatility are constant. We want a solution of the form $S(t) = g(t, W(t))$, which means

$$dS(t) = \big(\partial_t g + \frac{1}{2}\partial_x^2 g\big)dt + \sigma\partial_x g dW(t)$$

and so we must have

$$rg = \partial_t g + \frac{1}{2}\partial_x^2 g,\ and\ \sigma g = \partial_x g$$



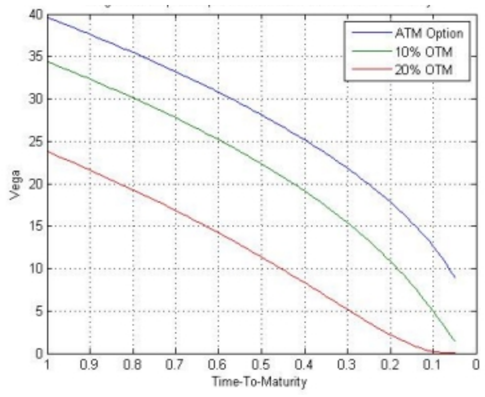


Figure 3: ν for European option as function of stock price and time to maturity

Putting these equations together we find $g(t,x) = e^{\sigma x + (r-\sigma^2/2)t}$ and thus

$$S(t) = S_0 e^{(r-\sigma^2/2)t + \sigma W(t)} \tag{5}$$

In order to perform a simulation we need to first discretise SDE (1). The simplest choice is the Euler discretisation,

$$\hat{S}_{n+1} = \hat{S}_n + a(\hat{S}_n, t_n)h + b(\hat{S}_n, t_n)\Delta W_n \tag{6}$$

with N timesteps and $h = T/N$. The Brownian increments ΔW_n are independant normal variables with mean zero and variance h. The quality of this approximation implroves with incresed N, though this can lead to error propagation under certain conditions. Our measure of error is divide into two types, strong and weak. The strong error measures how close S is approximated by $\hat{S}$, written as $\mathbb{E}[S_T - \hat{S}_T]$. The weak error for some function f is given by $|\mathbb{E}[f(S_T)] - \mathbb{E}[f(\hat{S}T)]|$.

In our case we will use the Milstein scheme, which has stronger convergence properties. [2]

$$\hat{S}_{n+1} = \hat{S}_n \left(1 + rh + \sigma\Delta W_n + \frac{\sigma^2}{2}(\Delta W_n^2 - h)\right) \tag{7}$$

n denotes the timestep with $t_n = nh$ and a_n, b_n referring to a and b evaluated at $\hat{S}_n, t_n$. Just as with the Euler discretisation ΔW_n is the Brownian increment $W_{n+1} - W_n$.

2.4 Monte Carlo

Given (1), we want to approximate the expected value of the solution $Q = \mathbb{E}[P]$. For now we consider P to just be some scalar value of interest. A Monte

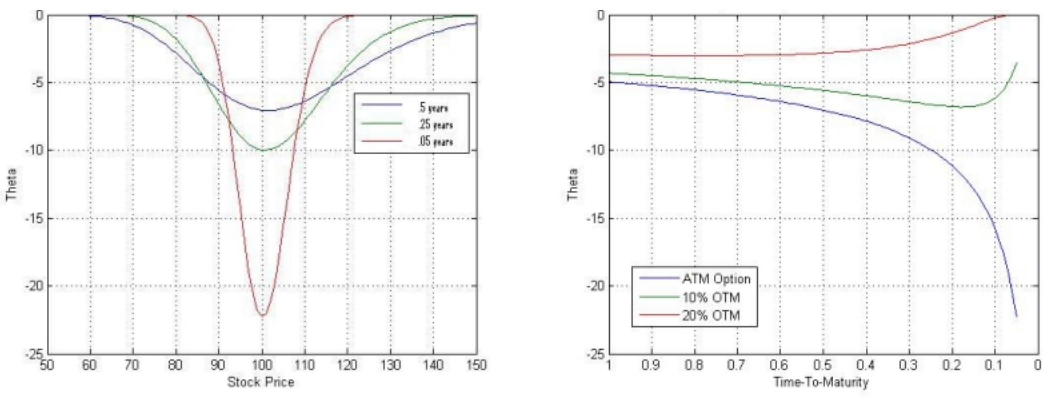


Figure 4: ν for European option as function of stock price and time to maturity

Carlo estimate of this value is a equally weighted average of the values $P(\omega)$ for N independant samples ω from some given probability space $(\Omega, \mathfrak{F}, \mathbb{P})$. This our estimate is given by the sum

$$Q_N = \mathbb{E}[P]_N = N^{-1} \sum_{n=1}^{N} P(\omega^{(n)}) \tag{8}$$

We can break down where error comes from in this approach by looking at the terms in the mean squared error (MSE) of this calculation

$$\begin{aligned} \mathbb{E}\left[(Q_N - \mathbb{E}[Q])^2\right] &= \mathbb{E}\left[(Q_N - \mathbb{E}[Q_N] + \mathbb{E}[Q_N] - \mathbb{E}[Q])^2\right] \\ &= \mathbb{E}\left[(Q_N - \mathbb{E}[Q_N])^2\right] + \mathbb{E}\left[(\mathbb{E}[Q_N] - \mathbb{E}[Q])^2\right] \\ &+ (\mathbb{E}\left[2(Q_N - \mathbb{E}[Q_N])(\mathbb{E}[Q_N] - \mathbb{E}[Q])\right] \\ &= \mathbb{V}[Q_N] + (\mathbb{E}[Q_N] - \mathbb{E}[Q])^2 \\ &= \mathbb{V}[Q_N] + (\mathbb{E}[Q_N] - Q)^2 \end{aligned} \tag{9}$$

We now use two known results about random variables to simplify this expression,

$$\mathbb{V}\left[\sum_{n=1}^{N} \omega^{(n)}\right] = \sum_{n=1}^{N} \mathbb{V}[\omega^{(n)}] + \sum_{n \neq p} \mathbb{C}ov[\omega^{(n)}, \omega^{(p)}]$$
$$\mathbb{V}[a\omega^{(n)}] = a^2 \mathbb{V}[\omega^{(n)}]$$

Note that each $\omega^{(n)}$ is drawn independently so $Cov[\omega^{(n)}, \omega^{(p)}] = 0$, $n \neq p$. Applying these formulas we get

$$\mathbb{V}[Q_N] = \mathbb{V}\left[N^{-1} \sum_{n=1}^{N} P(\omega^{(n)})\right] = N^{-2} \sum_{n=1}^{N} \mathbb{V}\left[P(\omega^{(n)})\right] = N^{-1} \mathbb{V}[Q] \tag{10}$$

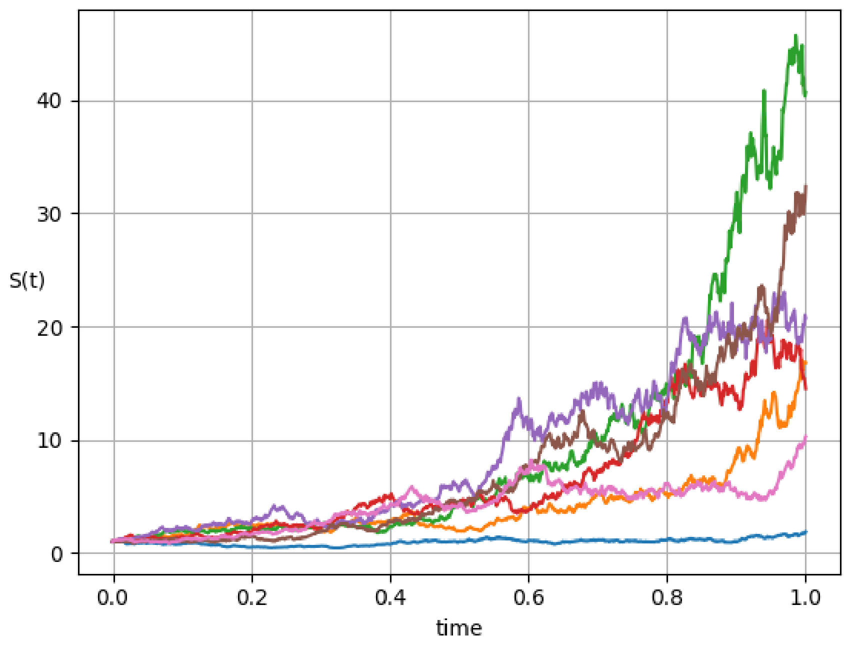


Figure 5: Six path simulations of SDE using Milstein discretisation

Finally, applying this to our MSE calculation gives us

$$\mathbb{E}\left[(Q_N - \mathbb{E}[Q])^2\right] = N^{-1}\mathbb{V}[Q] + (\mathbb{E}[Q_N] - Q)^2 \tag{11}$$

We have broken the MSE of this estimator into two terms, each of which describes a source of error.

The fist term, $N^{-1}\mathbb{V}[Q]$ is the variance of the estimator which arises as a result of using a finite number of samples. From [4] we can see that the root mean square (RMS) error of this procedure is $O(N^{-1/2})$. As a result to get to some desired accuracy ϵ we require $N = O(\epsilon^{-2})$ sample. This is one of the weaknesses of the Monte Carlo method as the computation of each sample $P(\omega^{(n)})$ can be very costly. This is the problem that MLMC seeks to address by reducing the estimator variance.

The second term, $(\mathbb{E}[Q_N] - Q)^2$, is bias due to using an approximation of P. Ideally, the stochastic error from variance of the estimator and deterministic bias from the mesh resolution should be balanced but in practice the choice of resolution is often made arbitrarily.

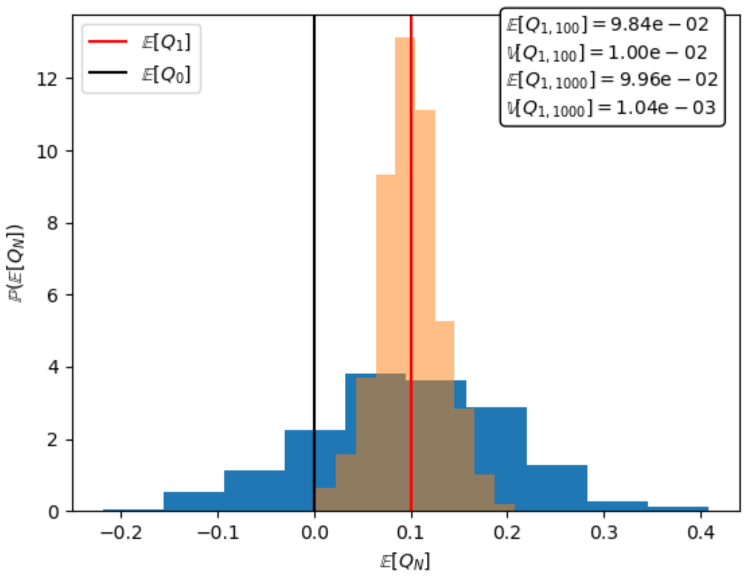


Figure 6: Error due to bias from mesh resolution

2.5 Multilevel Monte Carlo

We begin this section by discussing two-level Monte Carlo method. The classic approach to reducing variance in Monte Carlo simulation was to use control variates. We can use the following estimator

$$N^{-1}\sum_{n=1}^{N}\left\{f^{(n)} - \lambda\left(g^{(n)} - \mathbb{E}[g]\right)\right\} \tag{12}$$

where g is a control variate with correlation ρ to f. If λ is chosen to be optimal, $\rho\sqrt{\mathbb{V}[f]/\mathbb{V}[g]}$ then the variance of this estimator is reduced by factor $1-\rho^2$

A two-level Monte Carlo method works from the same principle. Suppose we want to estimate $\mathbb{E}[P_1]$ but is cheaper to compute $P_0 \approx P_1$. Then using the identity

$$\mathbb{E}[P_1] = \mathbb{E}[P_0] + \mathbb{E}[P_1 - P_0] \tag{13}$$

we can construct a two level estimator

$$N_0^{-1}\sum_{n=1}^{N_0} P_0^{(n)} + N_1^{-1}\sum_{n=1}^{N_1}\left(P_1^{(n)} - P_0^{(n)}\right) \tag{14}$$

Setting C_0 and C_1 to be the cost of computing a sample of P_0 and $P_1 - P_0$, the variane is minimised by choosing $N_1/N_0 = \sqrt{V_1/C_1}/\sqrt{V_0/C_0}$, there V_0 and V_1 are the respective variances.

We now address a full MLMC method. Suppose we have Monte Carlo simulations with time steps $h_l = 2^{-l}T$, for $l = 0, ..., L$. So on the coarsest level the simulation uses just one time step while on the finest level the simulation uses 2^L time steps. Let P be the payoff and let $\hat{P}_l$ denote its approximation using a SDE discretisation with time step h_l. Extending our identity from (13), we see that

$$\mathbb{E}[\hat{P}_L] = \mathbb{E}[\hat{P}_0] + \sum_{l=1}^{L} \mathbb{E}[\hat{P}_l - \hat{P}_{l-1}] \tag{15}$$

This gives us a representation for the finest level of resolution as the sum of the level 0 expected value and a series of correction terms between levels l and $l-1$. Letting $\hat{Y}_0$ be the estimator for $\mathbb{E}[\hat{P}_0]$ using N_0 samples, our estimator for $\mathbb{E}[\hat{P}_l - \hat{P}_{l-1}]$ is given by

$$\hat{Y}_l = N_l^{-1} \sum_{i=1}^{N_l} \left(\hat{P}_l - \hat{P}_{l-1} \right) = N_l^{-1} \sum_{i=1}^{N_l} \hat{Y}_l^i \tag{16}$$

The variance is minimised by choosing N_l to be proportional to $\sqrt{V_l h_l}$. We can generalize our error analysis from (11) with the following theorem.

Theorem 1. *Let P be a function of a solution to* (1) *for a given Brownian path $W(t)$ and let $\hat{P}_l$ be the corresponding approximation using the discretisation a level l with 2^l time steps of width $h_l = M^{-l}T$.*

If there exist independent estimators $\hat{Y}_l$ with computational complexity C_l based on N_l samples and there are positive constants $\alpha \geq \frac{1}{2}$, β, γ c_1, c_2, c_3 such that

1. $\mathbb{E}[\hat{Y}_l] = \begin{cases} \mathbb{E}[\hat{P}_0] & l = 0 \\ \mathbb{E}[\hat{P}_l - \hat{P}_{l-1}] & l > 0 \end{cases}$
2. $|\mathbb{E}[\hat{P}_l - P]| \leq c_1 h_l^{\alpha}$
3. $\mathbb{V}(\hat{Y}_l) \leq c_2 h_l^{\beta} N_l^{-1}$
4. $C_l \leq c_3 N_l h_l^{-1}$

Then there is a constant c_4 such that for any $\epsilon < e^{-1}$ there are values for L and N_l resulting in a estimator $\hat{Y} = \sum_{l=0}^{L} \hat{Y}_l$ with $MSE = \mathbb{E}[(\hat{Y} - \mathbb{E}[P])^2] < \epsilon^2$ with complexity C bounded by

$$C \leq \begin{cases} c_4\epsilon^{-2} & \beta > \gamma \\ c_4\epsilon^{-2}(log\epsilon)^2 & \gamma = 1 \\ c_4\epsilon^{-2-(1-\beta)/\alpha} & 0 < \gamma < 1 \end{cases} \tag{17}$$

Proof. See [4] $\square$

This theorem describes where the computation cost of the calculation lies. Conditions 1 and 3 describes an exponential decay in the weak error and the variance respectively, while condition 4 gives an exponential increase in expected cost.

In the case $\beta > \gamma$, the computational cost is dominated by the coarse level where $C_l = O(1)$. In this case we rquire $O(\epsilon)$ samples to achieve accuracy. In this case we have reduced the problem down to a standard Monte Carlo path simulation. In the case $\beta < \gamma$, the computational cost is shifted to the finest level of refinements and thus $C_L = O(\epsilon^{\gamma/\alpha})$. From [2] the largest possible value that α can achieve is $\beta = 2\alpha$. In this case the total cost is $O(C_L)$, the best that we can get. Finally is the case $\beta = \gamma$, the computational cost and the variance is spread evenly across the levels of refinement.

For a Milstein discretisation $\alpha = 1$. While α is often known, β, which determines strong convergence, depends on the payoff shape.

Notice that in the above theorem that the timestep is given as $h_l = M^{-l}T$, where M specifies the degree to which timesteps are refined at each level. It was found in [**gmeiner**] that for discretisations of elliptic PSEs $M = 2$ is optimal. Giles found in [**giles1**] that the computational cost of all levels is propotional to $2\epsilon^{-2}(log\epsilon)^2 f(M)$ where

$$f(M) = \frac{M - M^{-1}}{(logM)^2} \tag{18}$$

We can see from 9 that the minimum of this function is near $M = 7$. There is an interplay here between choosing a beneficial value for M and have an optimal number of levels.

We can construct a convergence test by noting that $\mathbb{E}[P - \hat{P}_L] \approx (M - 1)^{-1}E[\hat{P}_L - \hat{P}_{L-1}]$. Plugging this into the estimator gives us

$$\left(\sum_{l=0}^{L} \hat{Y}_l\right) + (M-1)^{-1}\hat{Y}_L = \frac{M}{M-1}\left(\hat{Y}_0 + \sum_{l=0}^{L}\left(\hat{Y}_l - M^{-1}\hat{Y}_{l-1}\right)\right) \tag{19}$$

Viewing the estimator like this allows us to look at $\hat{Y}_l - M^{-1}\hat{Y}_{l-1}$ to see if the bias is sufficiently small. In [4] it was determined that the correct choice of convergence test is

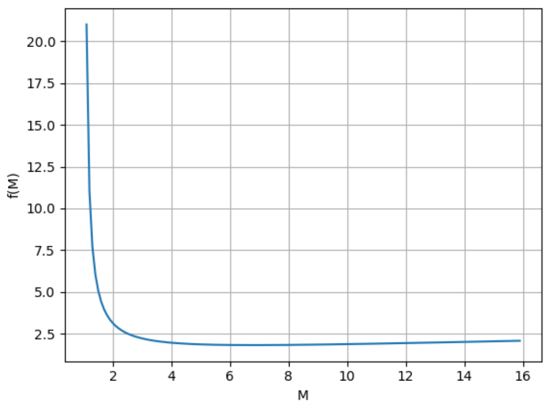


Figure 7: Plot of the function $f(M) = \frac{M - M^{-1}}{(log M)^2}$

$$|\hat{Y}_l - M^{-1}\hat{Y}_{l-1}| < \frac{1}{\sqrt{2}}(M^2 - 1)\epsilon \tag{20}$$

2.6 Multilevel Monte Carlo for Greeks

We now turn our attention to applying these methods to the computation of Greeks. We first consider a classic method for computing Greeks and use the results as a motivation for a multilevel estimator.

For this discussion let $\hat{S} = \hat{S_n}$ for $0 \geq n \geq N$ be the discrete solution to equation (1) at time steps $t_0, ...t_N$ with a corresponding set of Brownian increments. Let $\hat{P}$ and $\hat{V}$ be the payoff estimator and estimated option value respectively.

We approach these methods via example and refer to [9] for further details and justifications of results. Consider a European call with discounted payoff

$$P = e^{-rT}(S_T - K)^+ = e^{-rT}max(0, S_T - K) \tag{21}$$

Our first technique is the pathwise sensitivities approach, with which we will compute Δ and ν. The payoff is Lipshitz, which allows us to apply this method [9]. From equation (7), we see that we may write our Milstein discretisation of the SDE as

$$\hat{S}_{n+1} = \hat{S}_n \left(1 + rh + \sigma \Delta W_n + \frac{\sigma^2}{2}(\Delta W_n^2 - h)\right) = \hat{S}_n D_n \tag{22}$$

with

$$D_n = 1 + rh + \sigma \Delta W_n + \frac{\sigma^2}{2}(\Delta W_n^2 - h) \tag{23}$$

Differentiating with respect to some parameter θ gives us

$$\partial_\theta \hat{S}_{n+1} = \partial_\theta \hat{S}_n . D_n + \hat{S}_n . \partial_\theta D_n \tag{24}$$

Differentiating for Δ specifically we have

$$\begin{aligned} \partial_{S_0} \hat{S}_0 &= 1 \\ \partial_{S_0} \hat{S}_{n+1} &= \partial_{S_0} \hat{S}_n . D_n \end{aligned} \tag{25}$$

Differentiating for Δ specifically we have

$$\begin{aligned} \partial_\sigma \hat{S}_0 &= 0 \\ \partial_\sigma \hat{S}_{n+1} &= \partial_\sigma \hat{S}_n . D_n + \hat{S}_n (\Delta W_n + \sigma(\Delta W_n^2 - h)) \end{aligned} \tag{26}$$

This process gives us the pieces we need to construct a multilevel estimator. Using a fine discretisation with $N_f = 2^l$ and coarse discretisation with $N_c = 2^{l-1}$. Then the estimator on level l is given by

$$\hat{Y}_l = N_l^{-1} \sum_{i=1}^{N_l} \left(\frac{\partial P}{\partial S_{N_f}} \frac{\partial \hat{S}_{N_f}}{\partial \theta} \right)^{(l)} - \left(\frac{\partial P}{\partial S_{N_c}} \frac{\partial \hat{S}_{N_c}}{\partial \theta} \right)^{(l-1)} \tag{27}$$

Just as with our previous multilevel estimator, we use the same Brownian motion on the fine and coarse level. We will see in our results section that this process reduces the rate of convergence of this new estimator. This is due to the discontinuity of $\partial_S P$. We will discuss some possible workarounds in the future work section but for now we will move onto an alternate approach to multilevel methods.

2.7 Forward mode AD

Mathematically describing AD is a straightforward task, with the real difficulties arising from practical implementation. Before moving on to the more powerful adjoint model for automatic differentiation we will discuss the simpler tangent-linear model, which is much simpler to implement.

Definition 2. The Jacobian induces a linear mapping $\nabla F : \mathbb{R}^n \to \mathbb{R}^m$ given by $\mathbf{x}^{(1)} \mapsto \nabla F . \mathbf{x}^{(1)}$ The function $F^{(1)} : \mathbb{R}^{2n} \to \mathbb{R}^m$ defined by

$$\mathbf{y}^{(1)} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}) = \nabla F(\mathbf{x}) . \mathbf{x}^{(1)}$$

, is the tangent-linear model of F.

Writing the directional derivatives $\mathbf{y}^{(1)}$ and $\mathbf{x}^{(1)}$ as partial derivatives and applying the chain rule we have

$$\mathbf{y}^{(1)} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \mathbf{s}} = \nabla F(\mathbf{x}) . \mathbf{x}^{(1)} \tag{28}$$

A tangent-linear derivative code transforms an implementation of the function $\mathbf{y} = F(\mathbf{x})$ into $(\mathbf{y}^{(1)}, \mathbf{y}) = F(\mathbf{x}, \mathbf{x}^{(1)})$ which computes the function value and the directional derivative. The natural approach to achieving this is overloading, which will be discussed in the implementation section.

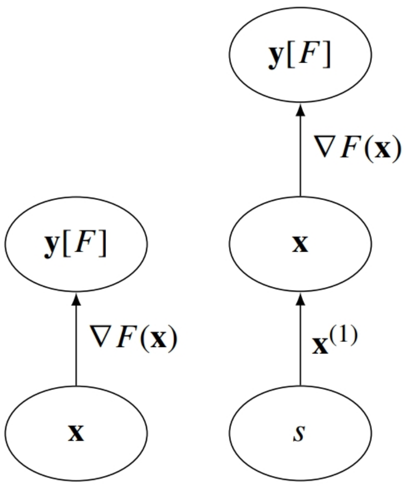


Figure 8: Tangent-linear extension of $\mathbf{y} = F(\mathbf{x})$. Sourced from [7]

2.7.1 Reverse mode AD

The tangent-linear approach to computing the gradient of a multivariate function $F : \mathbb{R}^n \to \mathbb{R}^m$ has a computational cost which grows with the number of variables n. For real world applications, in our case computing Greeks in large portfolios, this number can often grow extremely large very quickly [7]. As a simple example consider a function which takes one minute to compute with $n = 10^6$. We begin with a definition from functional analysis.

Definition 3. The adjoint of a linear operator $\nabla F : \mathbb{R}^n \to \mathbb{R}^m$ is $(\nabla F)^* : \mathbb{R}^m \to \mathbb{R}^n$, where

$$\langle (\nabla F)^* . \mathbf{y}_{(1)}, \mathbf{x}^{(1)} \rangle_{\mathbb{R}^n} = \langle \mathbf{y}_(1), \nabla F . \mathbf{x}^{(1)} \rangle_{\mathbb{R}^n}$$

Theorem 2. $(\nabla F)^* = (\nabla F)^T$

From this definition we can see that if we choose the adjoint of the output $\mathbf{y}_{(1)}$ to be orthogonal to $\mathbf{y}^{(1)} = \nabla F(\mathbf{x}).\mathbf{x}^{(1)}$, then the adjoint of the input $\mathbf{x}_{(1)} = \nabla F(\mathbf{x})^T.\mathbf{y}_{(1)}$ will be orthogonal to $\mathbf{x}^{(1)}$.

We can now define an adjoint model for computing the gradient.

Definition 4. The Jacobian induces a linear mapping $\nabla F^T : \mathbb{R}^m \to \mathbb{R}^n$ given by $\mathbf{x} \mapsto \nabla F.\mathbf{x}^{(1)}$ given by $\mathbf{y}_{(1)} \mapsto \nabla F^T.\mathbf{y}_{(1)}$. Define the function $F_{(1)} : \mathbb{R}^{m+n} \to \mathbb{R}^n$ where

$$\mathbf{x}_{(1)} = F_{(1)}(\mathbf{x}, \mathbf{y}_{(1)}) = \nabla F(\mathbf{x})^T.\mathbf{y}_{(1)}$$

to be the adjoint model of F.

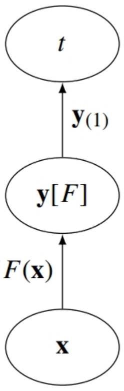


Figure 9: Adjoint extension of $\mathbf{y} = F(\mathbf{x})$. Sourced from [7]

Just as with the tangent-linear model we can write the directional derivatives as partial derivatives and apply the chain rule to get

$$\mathbf{x}_{(1)} = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)\left(\frac{\partial \mathbf{s}}{\partial \mathbf{y}}\right) = \nabla F(\mathbf{x}).\mathbf{x}^{(1)} \tag{29}$$

A adjoint model derivative code transforms the implementation of a function $\mathbf{y} = F(\mathbf{x})$ into $(\mathbf{y}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)}) = F_{(1)}(\mathbf{y}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)})$. We will discuss several toy models in the implementation section to illustrate the difficulty of efficiently generating AAD code.

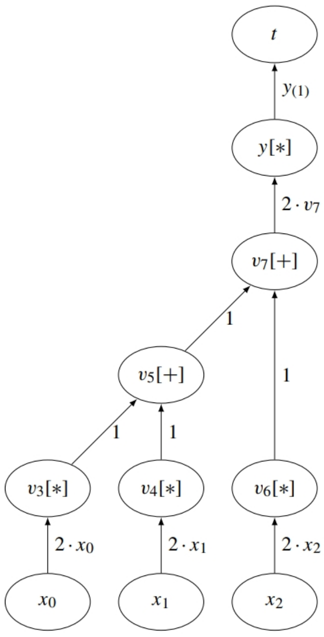


Figure 10: Adjoint extension of $\mathbf{y} = F(\sum_{i=1}^{3} x_i^2)^2$. Sourced from [7]

3 Algorithms

3.1 Mersenne Twister

We will be begin our discussion of the algorithms used over the course of this project with a brief discussion of the Mersenne Twister, the pseudorandom number generator used for generating random numbers. One of the main apeals of this PRNG is its very long period $2^{19937} - 1$ while managing to pass most tests for statistical randomness [6].

The primary challenge arising from randomness generation was the choice over how to generate multiple streams of pseudo random numbers in parallel. There are two obvious techniques that we can use to achieve this. The first of these is leap-frogging, which primes the PRNG to return the nth value starting from the kth position in the stream. This method avoids having each process throw away a possibly large sequence of pseudo random numbers but in some cases can lead to poor statistical properties in the generated sequence.

The alternate approach which was used in this case is the skip-ahead technique. I this method the sequence pseudo random numbers is partitioned into k streams of up to n numbers. While this approach involves the *ith* processor making *kn* calls to the PRNG before having access to its stream of variates, this was still found to be faster than the slowdown in variate computation when using the leap-frog approach. We refer to [8] for more details.

3.2 Mulilevel Monte Carlo simulation

Our aim with Multilevel Monte Carlo is to reduce the variance of our calculation. Rather than computing $\mathbb{E}[P]$ directly the approach is to exploit the identity $\mathbb{E}[P_L] = \mathbb{E}[P_0] + \sum_{l=1}^{L} \mathbb{E}[Y_l]$ to estimate the mean on the coarsest level $l = 0$ and updating this value with estimators for each level.

1. Let $L = 0$
2. Estimate V_L with $N_L = 10^4$ samples
3. Find optimal $N_l, l = 0, ...L$
4. Evaluate extra samples at each level for new N_l
5. If $L \geq 2$ test for convergence with (20)
6. If $L < 2$ or not converged, $L := L + 1$, go to 2

Optimal N_l is given by

$$N_l = \left\lceil 2\epsilon^{-2}\sqrt{V_l h_l}\left(\sum_{l=0}^{L}\sqrt{V_l/h_l}\right)\right\rceil \tag{30}$$

In the MLMC method there are three layers on which parallelism is natural to discuss. For a small cluster computing individual samples in parallel is an effective strategy but as the number of processes grows the small number of samples on the finer grid levels becomes too small to exploit the parallelism. The first and most obvious place to look at parallelisation strategies is the estimators of the individual layer, $l = 0, ..., L$, each of which may be computed in parallel.

Within each layer l the samples $\hat{Y}_l^i$ for $i = 1, ..., N_l$ may be computed in parallel. To do both of these concurrently we need to split our process communicator. Since the number of levels is typically small doing both of these strategies together is an effective strategy for maximising parallelism in the program. We shall see haw this was achieved in the Implementation section. There is one more level of parallelism that was not implemented, that of parallelisation of the solver itself.

3.3 AAD

The natural way to implement forward mode automatic differentiation is via overloading. recall from definition (2) that from the Jacobian we can construct a natural transformation $\mathbf{x}^{(1)} \mapsto \nabla F.\mathbf{x}^{(1)}$. In order to do this in practice we can define an augmented data type which contains this $\mathbf{x}^{(1)}$ along with the input vector $\mathbf{x}$. The arithmetic operators along with basic trancendental functions are then overloaded to propagete the directional derivative along with the primary calculation. Often the only thing that needs to be changed in the driver routine of the code is a change of type.

From the theoretical model that was constructed in section 2 we can also begin to construct an algorithm to calculate all of the adjoints of a function.

1. Evaluate the sequence of operations that make up the calculation. These are of the form $b = f(a_1, ..., a_n)$. Keep track of each intermediate a and b so that we can compute the partial derivatives $\partial_{a_i} f$.

2. Set the adjoint of the final result to 1 and set all the other adjoints to zero.

3. For each operation conduct the n adjoint operations

$$\bar{a}_i += \partial_{a_i} f.\bar{b}$$

   in the reverse order to the evaluation.

Note that we know we can perform step 2 in this process directly from our definition (3) of the adjoint of a linear operator, which allows us to choose a suitable set of orthogonality relations at this step.

4 Implementation

4.1 Tangent-Linear Model

We begin our implementation of forward mode by defining a class TLtype with double precision members, which we call x and t for the value and tangent respectively. A constructor is defined which initializes the value and tangent to zero when the class is first called. The assignment operator $=$ is overloaded so that it returns a copy of the value rather than the tangent unless specifically aliased to do so.

```
//TLtype.hpp
#ifndef TLTYPE_
#define TLTYPE_

class TLtype{
  public:
    double v;
    double t;

    TLtype(const double&);
    TLtype();
    TLtype& operator=(const TLtype&);
};

TLtype operator*(const TLtype&,const TLtype&);
TLtype operator+(const TLtype&,const TLtype&);
TLtype operator-(const TLtype&,const TLtype&);

TLtype sin(const TLtype&);
TLtype cos(const TLtype&);
TLtype exp(const TLtype&);

#endif
```

We require an implementation of each of the relevant operators along with intrinsic functions. Addition and subtraction are simple cases as the derivative is linear and so just distributes over the sums.

```
//Overloaded addition
TLtype operator+(const TLtype& x1,const TLtype& x2){
  TLtype temp;
  temp.v=x1.v+x2.v;
  temp.t=x1.t+x2.t;
  return temp;
}
//Overloaded subtraction
TLtype operator-(const TLtype& x1,const TLtype& x2){
  TLtype temp;
  temp.v=x1.v-x2.v;
  temp.t=x1.t-x2.t;
  return temp;
}
```

Implementing a multiplication operator $*$ is somewhat more complicated. For the value terms just multiply as normal but for the tangent we have $d(f.g) = df.g + f.dg$ so we need to apply the product rule to get the correct term. For ease of use we could also overload the / operator for functions of the form f/g, which would involve applying the quotient rule to the tangent but this is not a strictly necessary addition as we can always rewrite this case so that we are applying $*$.

```
//Overloaded multiplication
TLtype operator*(const TLtype& x1,const TLtype& x2){
  TLtype temp;
  temp.v=x1.v*x2.v;
  temp.t=x1.t*x2.v+x1.v*x2.t;//Product rule
  return temp;
}
```

Each tangent of our intrinsic function follows the same pattern involving an application of the chain rule. We list the implementation of cosine here.

```
//Cosine
TLtype cos(const TLtype& x){
  TLtype temp;
  temp.v=cos(x.v);
  temp.t=-sin(x.v)*x.t;//Chain rule
  return temp;
}
```

We now have the necessary pieces for forward mode automatic differentiation. We now need to look at how our modifications have altered how we implement fuctions. For example, lets look at the function $y = \left(\sum_{i=1}^{n} x_i^2\right)^2$. One possible implementation of this is

```
void f(double *x, double &y){
  y=0;
  for(int i=0;i<n;i++){
    y=y+x[i]*x[i];
  }
  y=y*y;
}
```

To apply our AD implementation we only need to modify the double to our custom type TLtype.

```
void f(TLtype *x, TLtype &y){
  y=0;
  for(int i=0;i<n;i++){
    y=y+x[i]*x[i];
  }
  y=y*y
}
```

We can see the primary disadvantage of the tangent-linear approach in the driver routine for this example. We have a loop over the dimension of the function we want to compute the gradient of. While this process can be easily parallelised for many practical problems the dimension still grows too quickly for this to be a viable method. Our implementation of reverse mode differentiation will seek to address this deficiency.

```
int main(){
  TLtype x[n]; TLtype y;

  for(int i=0;i<n;i++){
    x[i]=1;
  }
  //Loop over dimension computing each tangent
  for(int i=0;i<n;i++){
    x[i].t=1; f(x,y);
    x[i].t=0;
    cout << y.t <<endl;
  }
  return 0;
}
```

4.2 Adjoint Model

To construct our implementation of the adjoint model we first need to build a tape. This is a data structure representing the directed acyclic graph making up our function. In the code snippets provided, the tape is implemented as a statically allocated array of entries indexed by their position in the array.

```
class A_TapeEntry{
  public:
    int oc;
    int arg1;
    int arg2;
    double v;
    double a;
    A_TapeEntry() : oc(A_UNDEF), arg1(A_UNDEF),
                    arg2(A_UNDEF), v(0), a(0) {};
};

class Atype{
  public:
    int va;
    double v;
    Atype() : va(A_UNDEF), v(0) {};
    Atype(const double&);
    Atype& operator=(const Atype&);
};
```

```
Atype operator*(const Atype&, const Atype&);
Atype operator+(const Atype&, const Atype&);
Atype operator-(const Atype&, const Atype&);

Atype sin(const Atype&);
Atype cos(const Atype&);
Atype exp(const Atype&);

void A_PrintTape();
void A_InterpretTape();
void A_ResetTape();

```

The operation associated with the tape entry is given by oc. Operation codes are defined in preprocessing as macros. $arg1$ and $arg2$ carry addresses for the first and second arguments of the operation respectively. The current value and adjoint are stored in v and a respectively.

Just as with our implementation of the tangent-linear model, we also need to build a custom data type, in this case to replace doubles. This new class contains the position in tape va and the current value of a variable v.

Each operator and intrinsic function records information on the tape. For example, take multiplication.

```
Atype operator*(const Atype& x1, const Atype& x2){
  Atype temp;
  Atape[Avac].oc = A_MUL;
  Atape[Avac].arg1 = x1.va;
  Atape[Avac].arg2 = x2.va;
  Atape[Avac].v = temp.v = x1.v*x2.v;
  temp.va= Avac++;
  return temp;
}
```

The current index on tape is given by $Avac$. We begin by recording the appropriate operation code, in this case A_MUL for multiplication. the virtual addresses of the arguments $x1$ and $x2$ are the recorded and finally the new value after performing the operation. The tape index $Avac$ is the incremented.

Key to our implementation is interpreting the tape, which is equivalent to performing back propagation. Starting from the final index on the tape and moving back to the beginning at each entry on the tape we check the operation code *oc* to the operation performed at this step of the calculation. In our implementation we used a switch statement to determine how to proceed. We will first take addition as an example, which has operation code A_ADD.

```
switch(Atape[i].oc){
      case A_ASG : {
        Atape[Atape[i].arg1].a += Atape[i].a;
        break;
      }
      case A_ADD : {
        Atape[Atape[i].arg1].a += Atape[i].a;
        Atape[Atape[i].arg2].a += Atape[i].a;
        cout << Atape[Atape[i].arg1].a << endl;
        break;
      }
      ...
```

In this case the operation code at position i in the tape is found to be A_ADD. Just as with the tangent-linear model, for addition and subtraction we can do a simple accumulation of each of the collected arguments due to the distributive property of each of these operations. We will now look at multiplication, which lacks this feature at the adjoint term and thus requires a careful application of the product rule.

```
case A_MUL : {
  Atape[Atape[i].arg1].a += Atape[Atape[i].arg2].v * Atape[i].a;
  Atape[Atape[i].arg2].a += Atape[Atape[i].arg1].v * Atape[i].a;
        break;
}
```

We can see in this example how we have translated a manual implementation of differentiation for a specific problem into a system which does so automatically and can be applied to a wide range of problems. In a manual implementation the general strategy is to begin by duplicating the active data segment of our problem, in this case the value v stored as a double. In the forward section of the code we store values that are lost due to overwriting as the calculation continues as well as leaving scopes. At the reverse section we then need assignment-level adjoint code like what we can see in our multiplication example.

Now that we have shown the methodology for adjoint differentiation, we will demonstrate how to apply this approach in the specific context of Black-Scholes. A primary concern is that this code carries the assumption that the multivariate function that we apply this method to is differentiable as implemented. This requirement at the preprocessing step limits application of this method without the use of some techniques for smoothing the function is question, some of which we will discuss in the further work section. The amount of preprocessing that needs to be done in the context of both forward and reverse AD can vary highly with the multivariate function in question. In a simple case a change of types may be sufficient but in a more complicated setting user-defined intrinsic functions are often required. We can see a simple example of this below in the context of a cosine operation and exponentiation, both of which carry a single argument and use the chain rule to accumulate adjoints.

```
case A_COS : {
  Atape[Atape[i].arg1].a -=
    sin(Atape[Atape[i].arg1].v) * Atape[i].a;
break;
}
case A_EXP : {
  Atape[Atape[i].arg1].a += Atape[i].v * Atape[i].a;
  break;
}
```

4.3 Multilevel Monte-Carlo

In order to minimize the number of factors on which the results depend on and keep timings comparable, for each of our test cases we set the refinement factor to $M = 2$, which we found from finding the minima of equation (20) is not optimal.

There are several other parameters which we wanted to compute along the main calculation, the most important of which are α, β and γ, which arise from our discussion of the complexity theorem for Multilevel simulation. These are essential indicators of the performance of our simulation as the weak error is given by $O(2^{-\alpha l})$, the variance is $O(2^{-\beta l})$ and the sample cost is $O(2^{-\gamma l})$. As mentioned it cannot be assumed for every calculation that these parameters are known a priori so we need to perform a regression calculation to estimate them. We do this in the driver function of our code *mlmc.cpp* after checking for input values and if there is any errors in the function parameters, for example choosing the minimum level of refinement to be larger than the maximum.

```
void regression(int N, float *x, float *y, float &a, float &b){

  float sum0=0.0f, sum1=0.0f, sum2=0.0f, sumy0=0.0f, sumy1=0.0f;

  for (int i=0; i<N; i++) {
    sum0  += 1.0f;
    sum1  += x[i];
    sum2  += x[i]*x[i];

    sumy0 += y[i];
    sumy1 += y[i]*x[i];
  }

  a = (sum0*sumy1 - sum1*sumy0) / (sum0*sum2 - sum1*sum1);
  b = (sum2*sumy0 - sum1*sumy1) / (sum0*sum2 - sum1*sum1);
}
```

In the mlmc function in the same cpp file this is called via

```
regression(L,x,y,alpha,sum);
alpha = fmax(alpha,0.5f);
```

where x and y are arrays and sum is calculated in a convergence test. In our main function in *options.cpp*, we have a for loop which runs through each of the option types we want to do a multilevel simulation with. For each option we call two testing files which will identify the performance of our code. The first of these tests is in *mlmc_test.cpp*. In order to get the performance parameters, we start by looping through the levels $l = 0, ..., L$ of our simulation. This is where we can implement the first level of parallelism. This loop can be easily divided among processors and does not require any communication between them until the post processing step where the results at each level are pulled together to form the full estimator. To find the estimate at level l along with the running cost of the simulation the function

```
void level_l(int l, int N, double *sums)
```

is called. This function performs the path simulation for level l with N paths. This can again be done in parallel. One strategy to achieve parallelism on the level of both levels and samples is to use sub-communicators. By splitting the MPI_COMM_WORLD communicator using MPI_Comm_split, we can now have multiple processes assigned to each level. The primary challenge of using this approach is load balancing of processors between levels. As the optimal number of levels L may not be known a priori, practical implementation of level parallelism my not always be possible. In this case sample parallelism may be applied alone, which has the benefit of avoiding the problem of load balancing altogether. A further level of parallelism in this loop is possible at the level of the partial differential equation solver, which carries the most importance in a large simulation environment. Suppose we have P processors to distribute. From [**gmeiner**] we can see that on a typical supercomputer $P > \sum_{l=0}^{L} N_l$ and in particular $P >> N_L$. As a result of this if we wanted to perform a larger simulation solver parallelism would be necessary to fully exploit the system. Since we are not doing computations of this size we can stay with a two level parallelism strategy. At the end of each loop each communicator does a gather on the performance parameters to an array containing each number of interest.

The values of interest are stored as follows.

```c
double sums[7];
  float *cost = (float *)malloc((L+1)*sizeof(float));
  float *del1 = (float *)malloc((L+1)*sizeof(float));
  float *del2 = (float *)malloc((L+1)*sizeof(float));
  float *var1 = (float *)malloc((L+1)*sizeof(float));
  float *var2 = (float *)malloc((L+1)*sizeof(float));
  float *chk1 = (float *)malloc((L+1)*sizeof(float));
  float *kur1 = (float *)malloc((L+1)*sizeof(float));
```

For $l = 0, ..., L$ these arrays hold C_l, $\mathbb{E}[P_l - P_{l-1}]$, $\mathbb{E}[P_l]$, $\mathbb{V}[P_l - P_{l-1}]$, $\mathbb{V}[P_l]$, consistency and kurtosis respectively.

In our loop they are calculated as follows

```c
for (int l=0; l<=L; l++) {
  for (int m=0; m<7; m++) sums[m] = 0.0;

  mlmc_l(l,N,sums);

  for (int m=0; m<7; m++) sums[m] = sums[m]/N;

  cost[l] = sums[0];
  del1[l] = sums[1];
  del2[l] = sums[5];
  var1[l] = fmax(sums[2]-sums[1]*sums[1], 1e-10);
  var2[l] = fmax(sums[6]-sums[5]*sums[5], 1e-10);

  kur1[l]  = (      sums[4]
                - 4.0*sums[3]*sums[1]
                 + 6.0*sums[2]*sums[1]*sums[1]
                 -
 3.0*sums[1]*sums[1]*sums[1]*sums[1] )
        / (var1[l]*var1[l]);

  if (l==0)
     chk1[l] = 0.0f;
  else
    chk1[l] = sqrtf((float) N) *
              fabsf(  del1[l]  + del2[l-1]  -del2[l] )
     (3.0f*(sqrtf(var1[l]) + sqrtf(var2[l-1])+ sqrtf(var2[l])));
```

where the sum array contains the parameters computed from the simulation. Of particular importance is checking for kurtosis on the finest level L. We refer to [10] for a formal definition but roughly speaking kurtosis provides a measure of outliers in a distribution. In our context, this means that when we apply the refined estimator to correct the current prediction the correction is being dominated by a few rare paths. Getting this result suggest that either the number of levels should be lowered or the refinement factor M should be changed.

We now perform another set of tests to find the cost of our multilevel Monte Carlo simulation and compare this result to a standard Monte Carlo method. We first do a multilevel Monte Carlo simulation to find N_l and C_l for each l. The cost of the simulation can be found by multiplying these terms at each level and summing.

5 Results

Applying the same notation used in the theory section we have that

$$\frac{\partial S}{\partial \theta} \approx \hat{Y}_0 + \sum_{l=1}^{L} \hat{Y}_l \tag{31}$$

These estimators are given by equation (27) to be $\hat{Y}_l \approx \mathbb{E}\left[\frac{\partial \hat{P}_f}{\partial \theta} - \frac{\partial \hat{P}_c}{\partial \theta}\right]$. Setting $M = 2$ the level samples are given by $N_f^{(l)} = 2^l$ and $N_c^{(l)} = 2^{l-1}$ with corresponding time steps $h = h_f^{(l)} = T/N_f^{(l)} = T/2^l$ and $h_c^{(l)} = T/N_c^{(l)} = T/2^{l-1}$.

From our discussion of the multilevel Monte Carlo complexity theorem in the theory section we find that

$$\mathbb{E}[\hat{Y}_l] = O(h^{\alpha}) \tag{32}$$

For the variance we have a similar result

$$\mathbb{V}_l = O(h^{\beta}) \tag{33}$$

By taking the logarithm of these results we get two useful identities $log\mathbb{E}[\hat{Y}_l] \sim -l\alpha$ and $log\mathbb{V}_l \sim -l\beta$. This result mirrors our application of regression to estimate the complexity parameters in the previous section.

In the following section we will discuss application of multilevel Monte Carlo to both options pricing and sensitivity calculation

5.1 European option

The first option that was considered is a European call, which has the discounted payoff

$$P = e^{-rT}(S_T - K)^{+} = e^{-rT}max(0, S_T - K) \tag{34}$$

At the expiry time we have

$$\mathbb{E}\left[P(\hat{S}_N) - P(S_T)\right] = O(h) \tag{35}$$

The mean squared error is $O(h^2)$ which gives us a variance of

$$\mathbb{V}\left[P(\hat{S}_N) - P(S_T)\right] = O(h^2) \tag{36}$$

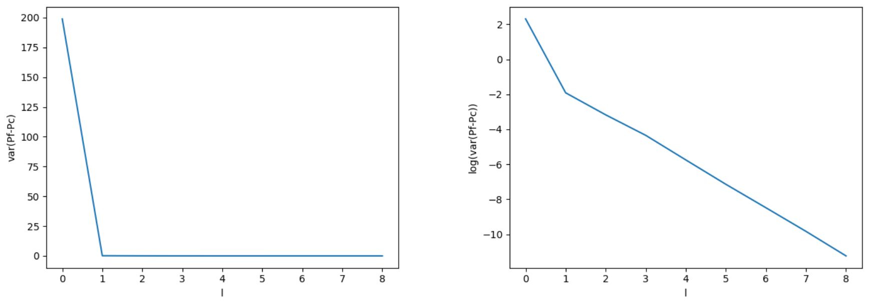


Figure 11: Variance of $P_f - P_c$ with l for European call

We can see from figure 11 that the variance of the correction estimators quickly drops for coarse levels but begins to stabilize as the refinement increases. We can see from the steep drop from $l = 0$ to $l = 1$ that setting

$M = 2$ in this case was the correct decision, as the corrections become so small so quickly that increasing the number of samples at each level would result in a possible large slowdown for little gain in terms of convergence.

By plotting the variance of P_f alongside the variance of $P_f - P_c$ in 13 we can observe a substantial improvement in variance reduction. As stated in the introduction, the large variance of the Monte Carlo method is the primary barrier in regards to convergence rates, particularly in higher dimensions where Monte Carlo is one of the only practical options for quadrature. For this reason the observed variance reduction translates to a significant speedup.

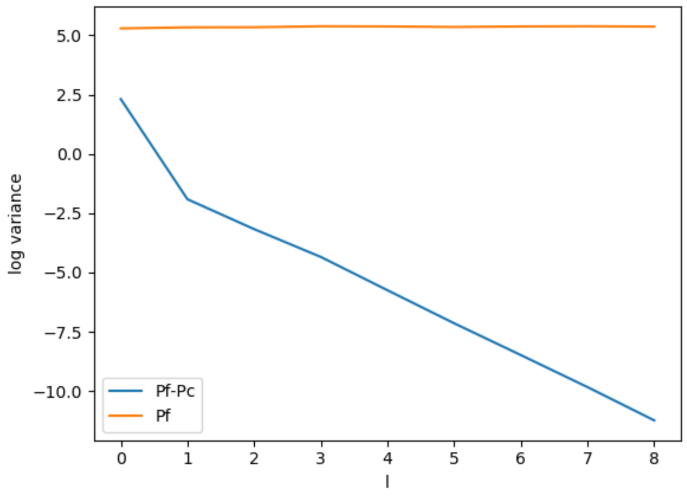


Figure 12: Variance of $P_f - P_c$ and P_f for European call

For Greeks, the convergence of multilevel methods is increased due to the discontinuity of the function. We can observe this by comparing estimates of β, which governs variance $O(2^{-\beta})$ and α, which governs weak convergence $O(2^{-\alpha})$.

Estimator	α	β	Complexity
P	1.0	2.0	$O(\epsilon^{-2})$
Δ	1.0	0.8	$O(\epsilon^{-2.2})$
ν	1.0	1.0	$O(\epsilon^{-2} log(\epsilon^2))$

From this we can observe that there is a trade off between the variance reduction that we get from splitting up a Monte Carlo estimator and increased variance due to discontinuities. In the further work section we will discuss some of the proposed approaches to this problem and how they fit into the framework we have built.

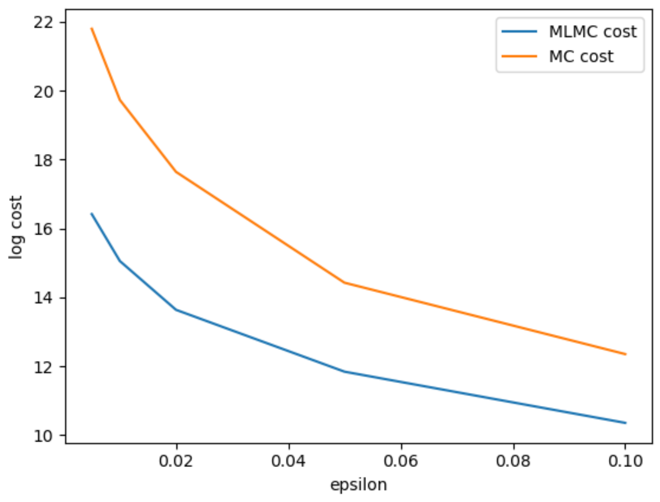


Figure 13: Cost of standard Monte Carlo and Multilevel Monte Carlo against accuracy ϵ

5.2 Asian Option

We use an Asian option as another example. The payoff of an Asian option, as opposed to the European call we just looked at, is dependant on the average price of an underlying asset over some given period, rather than on some specified date. This averaging usually comes in the form of the mean of the underlying assets price over the time to maturity subtracting the strike price K. More formally, the discounted payoff of an Asian option is given by

$$P = e^{-rT} max(0, \bar{S} - K) \tag{37}$$

where the average $\bar{S}$ is given by

$$\bar{S} = \frac{1}{T} \int_0^T S(t) dt \tag{38}$$

As with the European call payoff, the payoff function for an Asian option is also Lipschitz so we can apply the methods we have already discussed without many changes. For our purposes we will stick to the simplest choice of approximation

$$\bar{\hat{S}}\frac{1}{T}\sum_{n=0}^{T/h-1}\frac{h}{2}(\hat{S}_n+\hat{S}_{n+1}) \tag{39}$$

A better approximation is constructed in [9] using a Brownian bridge. As the payoff is Lipschitz we can differentiate the average to get

$$\frac{\partial\bar{\hat{S}}}{\partial\theta}=\frac{1}{2N}\left(\frac{\partial\hat{S}_0}{\partial\theta}+\frac{\partial\hat{S}_N}{\partial\theta}\right)+\frac{1}{N}\sum_{n=1}^{N-1}\frac{\partial\hat{S}_n}{\partial\theta} \tag{40}$$

We now have a form that we can apply our multilevel method to. On the fine level we simulate $\hat{S}_n^f$ along with $\partial_\theta\hat{S}_n^f$. On the coarse level we simulate $\hat{S}_n^c$ along with $\partial_\theta\hat{S}_n^c$. With equation (40) we can then compute the average on each level.

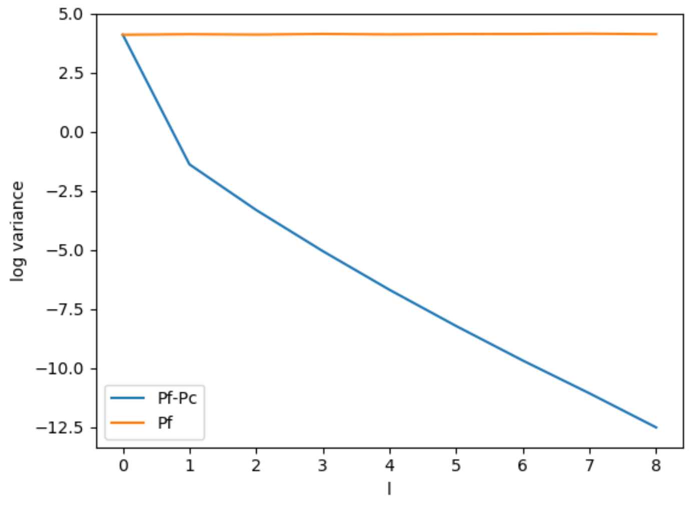


Figure 14: Variance of $P_f - P_c$ with l for European option

We can see from figure 18 that the reduction in variance is similar to what we found in the case of the European call, with the variance falling rapidly for low values of l and steadying as the refinement increased. This is a much more significant result than previously however, as unlike the European call there is no known general analytical solution for the price of an Asian option.

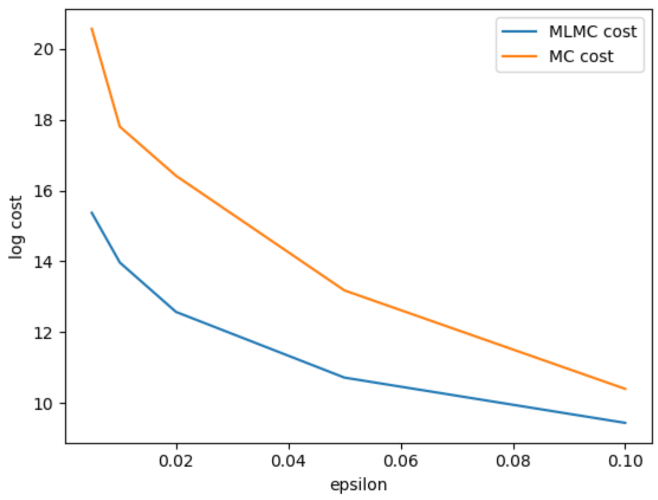


Figure 15: Cost of standard Monte Carlo and Multilevel Monte Carlo against accuracy ϵ

In comparison to a European call, we found in 16 that the cost of multilevel Monte Carlo at low accuracy was closer to standard Monte Carlo for a Asian option but as the desired accuracy ϵ increased, the multilevel method was able to out compete the standard method with a saving similar to the European case. We estimated our complexity parameters to be the following

Estimator	α	β	Complexity
P	0.8	2.2	$O(\epsilon^{-2})$
Δ	0.9	1.0	$O(\epsilon^{-2.2})$
ν	1.0	1.2	$O(\epsilon^{-2} log(\epsilon^2))$

It should be noted for completeness that the type of Asian option we chose to simulate uses arithmetic averaging. This means that each price is given equal weight in the sum. Another common style is to use an index weighted average. In this case, newer prices are given a higher weight that older prices by way of an exponential decline.

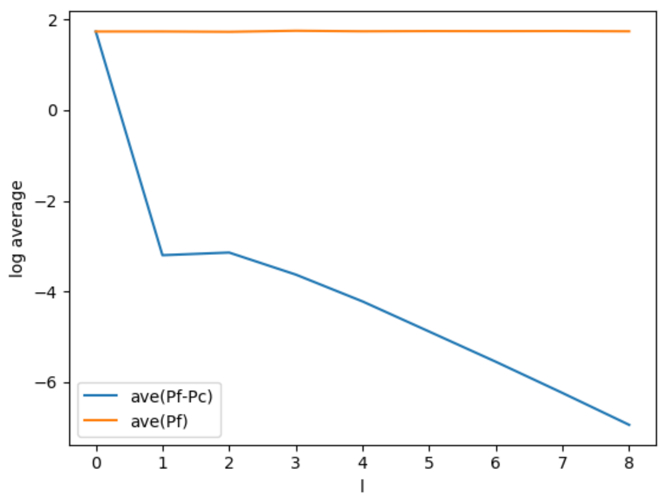


Figure 16: Estimators at each level $\mathbb{E}[P_f - P_c]$ and $E[P_f]$

5.3 Lookback Option

The final option that we consider is the lookback option, which is an option whose payoff is dependant on the minimum or maximum value the underlying asset obtains over the life of the option. In our case this will be the minimum, with the payoff function given by

$$P = S_T - minS_t \tag{41}$$

In [3] it was found that we can get good results using a Milstein scheme using the following result for the minimum value

$$\hat{S}_{min} = min(\hat{S}_n - qb_n\sqrt{h}) \tag{42}$$

where b_n is the volatility at the nth time step and $q \approx 0.5826$ is a correction constant to account for discrete sampling bias. A better choice for the Milstein discretisation is to use the following to simulate the minima at each time step.

$$\hat{S}_{min,n} = \frac{1}{2}\left(\hat{S}_n + \hat{S}_{n+1} - \sqrt{(\hat{S}_{n+1} - \hat{S}_n)^2 - 2b_n^2 hlogU_n}\right) \tag{43}$$

U_n is a uniform random variable on $[0, 1]$. In the case of lookback options, we have a nice formula for Δ

$$\Delta = \frac{\partial P}{\partial S_0} = \frac{\partial(S_T - S_{min})}{\partial S_0} = \frac{P}{S_0} \tag{44}$$

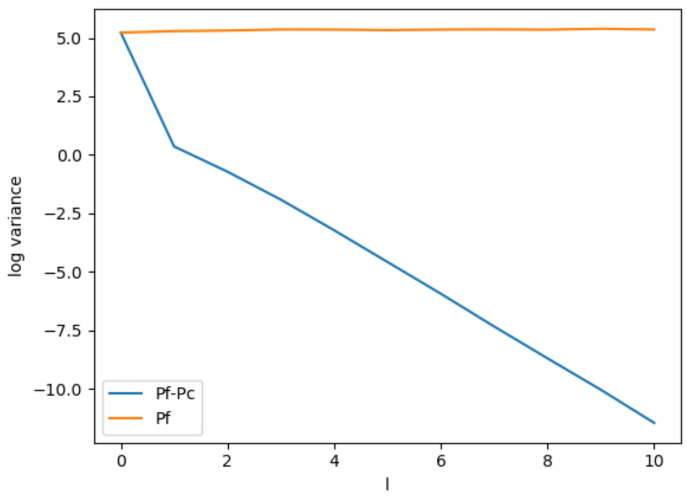


Figure 17: Variance of $P_f - P_c$ with l for European option

In the case of lookback option we instead focused our attention on pricing the option rather than computing some other sensitivity such as ν. Since the payoff is again Lipschitz we can in theory use pathwise sensitivities to construct an estimator but in practice applying this method is extremely ungainly and result in correction estimators which are difficult to predict [9]. Our simulation gave us $\alpha \approx 0.8$ and $\beta \approx$ which gives a complexity estimate of $O(\epsilon^{-2})$.

5.4 AAD

We will now apply what we learned in the implementation section to a simple European model to get an idea of how a manual implementation of AAD looks in practice.

```
//Adjoint function
double C(
  const double S0,
  const double r,
  const double y,
  const double sig,
  const double K,
  const double T,
```

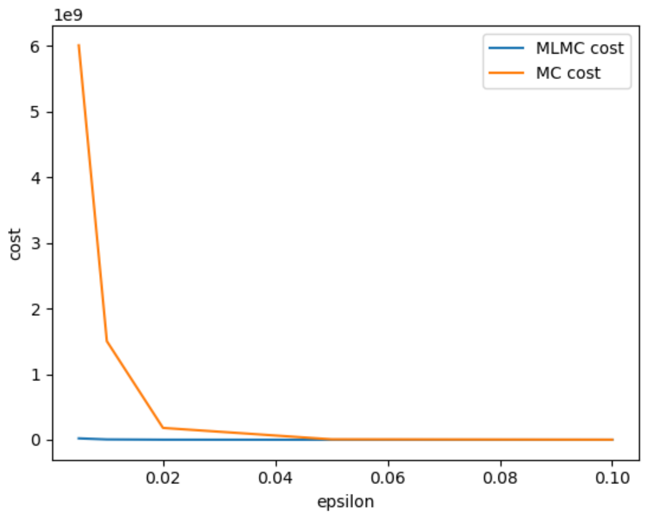


Figure 18: Cost of Multilevel Monte Carlo and standard Monte Carlo comparison. Here we don't use a logarithm scale to get a sense of the savings of this method

```
  //Adjoints
  //
  const bool calcAdjoints = false,
  double *S0_ = nullptr,
  double *r_ = nullptr,
  double *y_ = nullptr,
  double *sig_ = nullptr,
  double *K_ = nullptr,
  double *T_ = nullptr,

  const double C_ = 1.0
  )
{...
```

To reiterate our approach, we begin by evaluating, that is, performing the sequence of operations that result in the final calculation. We keep track of each intermediate result and operation by way of duplication of the active data segment of the code, which in this case looks as follows

```
  ...
  const double sqrtT = sqrt(T);
  const double df = exp(-r*T);
  const double F = S0*exp((r-y)*T);
  const double std = sig*sqrtT;
  const double d = log(F/K)/std;
  const double d1 = d+0.5*std;
  const double d2 = d-0.5*std;
  const double nd1 = normalCdf(d1);
  const double nd2 = normalCdf(d2);
  const double v = df*(F*nd1-K*nd2);

  if(!calcAdjoints) return v;
  ...
```

The next step is to compute the adjoint operation for each operation in the constructed sequence. We begin by initializing the adjoint of our value v and then running through each step in reverse.

```
...
  //Accumulation step
  if(sig_) *sig += std*sqrtT;
  if(T_) *T_ += 0.5*std_*sig/sqrtT;
  if(S0_) *S0_ += F_*F/S0;
  if(r_) *r_ += F_*T*F;
  if(y_) *y_ -= F_*T*F;
  if(T_) *T += F_*(r-y)*F;
  if(r_) *r += -df_*df*T;
  if(T_) *T += -df_*df*r;

  return v;
}
```

Each of the if statements contain a variable onto which we are performing adjoint accumulation, a process which can be summarized by the following diagram

$$b = f(a_1, ..., a_n) \rightarrow \hat{a}_i + = \frac{\partial f}{\partial a_i}\hat{b} \tag{45}$$

6 Further work

Over the course of this report we have explored some common variance reduction techniques in the context of Monte Carlo simulation of options and their sensitivities, as well as how they compare to two different models of automatic differentiation. We have purposely restricted our attention to options with Lipschitz payoffs, that is, the payoff functions have the property

$$|f(x_1) - f(x_2)| \leq L|x_1 - x_2| \tag{46}$$

While many common options do have this property it is far from being universal, which carries with it the question of how to extend the results we have derived using pathwise sensitivities to the wider context of discontinuous payoffs.

One area of interest, particularly in the context of parallel simulation is Vibrato Monte Carlo. For simplicity we will use a Euler discretisation (7) to explain this method. Consider the final timestep of this is given by

$$\hat{S}_N = \hat{S}_{N-1} + a(\hat{S}_{N-1}, t_{N-1})h + b(\hat{S}_{N-1}, t_{N-1})\Delta W_{N-1} \tag{47}$$

Rather than generating a value for ΔW_{N-1} using a random number generator, we instead consider the distribution of possible values. Setting $W := (\Delta W_0, ..., \Delta W_{N-2})$ in this approach we can consider $\hat{S}_N$ to be a Normal distribution with mean

$$\mu_W = \hat{S}_{N-1} + a(\hat{S}_{N-1}, t_{N-1})h \tag{48}$$

and standard deviation

$$\sigma_W = b(\hat{S}_{N-1}, t_{N-1})\sqrt{h} \tag{49}$$

Now we have that for a particular path described by some vector W the expected payoff is now given by

$$\mathbb{E}[f(\mu_W + \sigma_W Z)] \tag{50}$$

where Z is a unit normal random variable. The main advantage of this procedure is that for each of these path vectors W, the payoff is smooth, with $O(h^{-1/2})$ near the strike price and near zero elsewhere, which results in a $O(h^{-1/2})$ variance.

This method caries another major advantage in that it can be parallelised just as easily as standard Monte Carlo calculation. The analogy of vibrato in regards to rapidly changing pitch in music and this method is a follows. Rather than having a precise value as our output as in standard Monte Carlo this method instead results in outputs values which follow a narrow probability distribution.

Of note in regards to our automatic differentiation models is that we only discussed the computation of first order derivatives and did not discuss the computation of sensitivities such as Γ. While theoretically AAD over AAD is a natural extension and can even be produced via modification of the overloaded operators and tape used to implement first order AAD, in practice it has not yet found relevence in practical applications. Instead, the standard method is to use AAD to compute first order derivatives and then switch to a finite difference approximation, or bumping as it is known in the financial sector, to compute higher derivatives. An area ripe for further work would be the use of AAD and a multilevel method, which scales much better to high dimensional problem, as an alternate approach.

References

[1] Eugene F. Fama. "Efficient Capital Markets: A Review of Theory and Empirical Work". In: *The Journal of Finance* 25.2 (1970). DOI: http://efinance.org.cn/cn/fm/Efficient%20Capital%20Markets%20A%20Review%20of%20Theory%20and%20Empirical%20Work.pdf.

[2] M.B. Giles. "Improved multilevel Monte Carlo convergence using the Milstein scheme". In: (). DOI: https://people.maths.ox.ac.uk/gilesm/files/mcqmc06.pdf.

[3] M.B. Giles. "Vibrato Monte Carlo sensitivities". In: (). DOI: https://people.maths.ox.ac.uk/gilesm/files/mcqmc08.pdf.

[4] Michael B. Giles. "Multilevel Monte Carlo Path Simulation". In: *Operations Research* 56.3 (2007). DOI: https://people.maths.ox.ac.uk/gilesm/files/OPRE_2008.pdf.

[5] Desmond J. Higham. "An Introduction to Multilevel Monte Carlo for Option Valuation". In: *International Journal of Computer Mathematics* (2015). DOI: https://arxiv.org/pdf/1505.00965.pdf#page22.

[6] Archana Jagannatam. "Mersenne Twister – A Pseudo Random Number Generator and its Variants". In: ().

[7] Uwe Naumann. *The Art of Differentiating Computer Programs - An Introduction to Algorithmic Differentiation.* SIAM, 2011. ISBN: 978-1-61197-206-1.

[8] Antoine Savine. *Modern Computational Finance: AAD and Parallel Simulations*. Wiley, 2018. ISBN: 978-1119539452.

[9] M.B. Giles Sylvestre Burgos. "Technical report The computation of Greeks with Multilevel Monte Carlo ". In: (2010). DOI: `https://arxiv.org/pdf/1102.1348.pdf#page59`.

[10] Eric W Weisstein. *Kurtosis*. URL: `https://mathworld.wolfram.com/Kurtosis.html#:~:text=The%20kurtosis%20of%20a%20theoretical%20distribution%20is%20defined,implemented%20in%20the%20Wolfram%20Language%20as%20Kurtosis%20`.