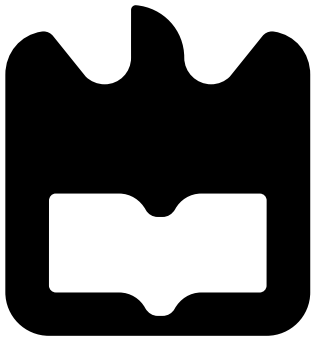
Universidade de Aveiro Departamento de Matemática,
2022

José Nuno
Catarino Brito

Diferenciação Automática Adjunta para o Erro Quadrático Médio envolvendo Expetações

Automatic Adjoint Differentiation for Mean Squared Error involving Expectations



José Nuno Catarino Brito

Diferenciação Automática Adjunta para o Erro Quadrático Médio envolvendo Expetações

Automatic Adjoint Differentiation for Mean Squared Error involving Expectations

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Matemática e Aplicações, realizada sob a orientação científica do Doutor Evgeny Lakshtanov, Investigador do Centro de Investigação e Desenvolvimento em Matemática e Aplicações

o júri / the jury

presidente / president	**Professora Doutora Isabel Maria Simões Pereira,** Professora Associada, Universidade de Aveiro
vogais / examiners committee	**Doutor Dmitrii Legatiuk,** Chair Of Mathematics, University Of Erfurt
	Doutor Evgeny Lakshtanov, Equiparado a Investigador Principal, Universidade de Aveiro

agradecimentos / acknowledgements

À minha mãe, que por muitas vezes ter perdido a paciência comigo, sempre esteve ao meu lado em todos momentos bons e maus, bem como o meu irmão, mesmo não estando tão presente ultimamente, foi sempre um suporte na minha vida.

Ao meu orientador Evgeny, mesmo tendo, às vezes, ficado sem paciência, sempre me apoiou em todo o meu percurso académico e dar os primeiros passos no mundo profissional. Ao professor Uwe pelos conselhos e pela ajuda que me forneceu nos últimos tempos.

Aos meus grandes parceiros de treino e amigos claro, Fábio e Pedro, pelas muitas gargalhadas entre nós e pelos vários conselhos de vida e profissional que me ajudou em diversas etapas da minha vida.

Às minhas amigas, pelas conversas bastante animadas e saídas, mesmo sendo poucas, que me fizeram sair da minha área de conforto, espairecer e aproveitar um pouco do que a vida fornece. Eu tentarei interagir mais para haver mais ocasiões memoráveis deste tipo.

Palavras Chave Diferenciação Automática Adjunta; Computação em paralelo; Simulação de Monte-Carlo; Opções Europeias; Erro Quadrático Médio; C++.

Resumo

Esta dissertação terá como objetivo, explicar como calcular gradientes do Erro Quadrático Médio, $\frac{1}{n}\sum_{i=1}^{n}(Ey_i - C_i)^2$, uma vez que é aplicado em métodos de calibração de modelos estocásticos, utilizando a Diferenciação Automática Adjunta e computação em paralelo.

Com o envolvimento de valores médios, a aplicação da Simulação de Monte-Carlo será necessária para aproximação de possíveis valores esperados tal como, métodos de estimação distintos e apropriados para a computação das suas derivadas parciais correspondentes.

Por conseguinte, neste documento, será inicializado com a introdução de certos conceitos necessários, como a Diferenciação Automática Adjunta, Simulação de Monte-Carlo, assim como, Call Options de Opções Europeias uma vez que, será utilizado, para um caso prático, um modelo estocástico simples de Call Options europeias aplicando a diferenciação para um certo conjunto de parâmetros.

Por fim, tais conceitos introduzidos previamente, serão aplicados para o objetivo principal, analisando os resultados teóricos e práticos de que se pode retirar com os diferentes métodos de estimação aplicados.

Keywords Automatic Adjoint Differentiation; Parallelizationo Monte-Carlo Simulation; European Options; Mean Squared Error; C++.

Abstract

This dissertation will have as an objective, explain how to calculate gradients of Mean Squared Error, $\frac{1}{n}\sum_{i=1}^{n}(Ey_i - C_i)^2$, since it's applied in calibration methods of stochastic models, using the Automatic Adjoint Differentation and parallelization.

With the involviment of mean values, the application of the Monte-Carlo Simulation will be needed to approximate possible expected values as well as, distinct and appropriate estimation methods for the computation of its correspondents partial derivaties.

Therefore, in this document, will be initialized with the introduction of certain concepts needed, like the Automatic Adjoint Differentiation, the Monte-Carlo Simulation, as well as, Call Options of European Options since, will be used, for a pratical case, a simple stochastic model of European Call Options applying the differentiation for a certain group of parameters.

Finally, such concepts introduzed previously, will be applied for the primary objective, analizing the theoretical and pratical results from which you can withdraw with the different estimation methods applied.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Through this introductory chapter, it'll be presented the motivation for this work, the objective we wish to accomplish as well as how the document is going to be structured.

1.1 Motivation

Automatic Adjoint Differentiation is a powerful and one of the most important tools in the financial area. Is a differentiation method capable of calculating derivatives of highly complex functions in an incredible speed compared to others differentiation methods, for example Finite Differentiation. Therefore, the Automatic Adjoint Differentiation is used in other areas, like Machine Learning, for the quick computation of gradients of loss functions. Another important reason is the accuracy of the results, which are computed with an exact precision since, when it comes to the derivatives obtained with Finite Differentiation, it might result in truncation errors. Knowing the main advantages of this differentiation method, shows that it's a faster and precise algorithm to compute gradients. There is a more detailed discussion about the differentiation method in [3, 4, 5, 6, 7, 8, 9].

In the financial market, it's used expected values when it comes to decide if an investment is worthy or not. To determine those values, it's used the Monte Carlo Simulation, that considers multiple outcomes that may occur, giving the possibility to estimate the expected values, having an insight of how to proceed in a certain investiment.

When it comes to discover how accurate an estimator, it's usually applied the Mean Squared Error equation, since it corresponds to calculate the mean value of the squares of the errors, which are between the estimator and the observed value. Hence, it's one of the most widely used as a minimizing loss function in optimization of stochastic problems and so, it requires to compute the gradient. Consequently, this procedure might take some time, if dealing with multiple parameters.

Therefore, it is obtained the resources needed, when it comes to compute the gradient of Mean Squared Error involving expectations and, most important, when dealing a stochastic function containing a high number of parameters. There is a work done, where it considers some of the topics mentioned before, in the article [2], in which this dissertation work is based on. There, it can be found some estimation methods that can be applied for the computation of the partial derivatives of the loss function in study.

1.2 Objectives and Structure of the Dissertation

The objective of this work is to obtain an estimation method capable of calculating the partial derivatives of the Mean Squared Error involving expectations (for a set of European Call Options, it is discussed afterwards in the dissertation) with Automatic Adjoint Differentiation, faster without jeopardizing the precision of the results compared to one know to be able to estimate, the estimation method can be found in the article [1]. Thus, we created a program with a non-real set of European Call Options, with estimation methods and an Automatic Adjoint Differentiation program, for the sole purpose of estimating partial derivatives, analyze the time of execution and the precision of the estimates. The intention is to compare the estimation methods and corroborate with conclusions in [2].

In addition to this chapter, where it corresponds to the introduction of the work presented, in Chapter 2, discusses the methodology and the state of art. Chapter 3, corresponds to the discussion of the estimations methods and the case study, showing all the results as well as a discussion for each of them. Chapter 4, it summarizes all the work and the most important conclusions. There is also the Appendix, where it will contain theoretical proofs, examples, and the implemented program codes.

Chapter 2

Context and State of Art

This chapter is responsible to present and discuss all the important subjects of this dissertation. In the section 2.1, enunciates the Adjoint Differentiation as well as Automatic Adjoint Differentiation and how it computes the gradients. In the section 2.2, explains and analyzes what a European Call Option and how to calculate (this is a common option contract in the financial market and used for the case study). Finally, in the section 2.3, discusses what is a Monte Carlo Simulation and its properties.

2.1 Adjoint Differentiation (AD)

The Adjoint Differentiation corresponds to an algorithm with purpose of calculating partial derivatives in an incredible speed with the application of the chain rule method.

The idea behind this algorithm is the ability to decompose a function, defined as it is stated in [11], into sub-expressions of arithmetic operations and/or elementary functions, recursively, from the inputs of the function towards the outputs, this step is called forward mode. Afterwards, from the outputs, it calculates all derivatives with respect to the sub-expressions until it obtains the partial derivatives of the function, this last step is denoted by reverse mode. The procedure of this algorithm can be found in the article [5] for a detailed analysis.

Because, of the way this algorithm computes the partial derivatives, from the outputs to the inputs, the computational cost is proportional to the number of outputs. On the other hand, compared to other differentiation methods for example, Finite Differentiation, the computational cost is proportional to the number of inputs. Concluding that, the Adjoint Differentiation is advantageous when dealing with functions with a higher number of inputs compared to the number of outputs.

In cases, the Adjoint Differentiation is dealing with high dimensions functions, in pen and paper, this takes a long time and might lead to calculation errors. Therefore, there is the Automatic Adjoint Differentiation, which is a computer program that automatically applies the Adjoint Differentiation.

2.1.1 Automatic Adjoint Differentiation (AAD)

As mentioned before, and according to the information found in the article [8], the Automatic Adjoint Differentiation is a high-level computer program with the capability of applying

the Adjoint Differentiation automatically, saving all the calculations and operations in memory.

The results obtained through this program are faster and exact since, it uses the benefits of the Adjoint Differentiation as well as the computational speed of a computer.

According to the articles [1, 2], the Automatic Adjoint Differentiation proceeds with an algorithm constituted by a function.

$$F : \mathbb{R}^n_{(x_1,\ldots,x_n)} \longrightarrow \mathbb{R}^m_{(y_1,\ldots,y_m)}$$

with the objective of finding the Jacobian of F, which means, $J = F' = \left[\frac{\partial y_j}{\partial x_i}\right]$ for $1 \leq i \leq n$ e $1 \leq j \leq m$, the AAD maps a vector $\lambda \in \mathbb{R}^m$ onto a set of linear combinations of $\frac{\partial y_j}{\partial x_i}$,

$$\left\{ \sum_{j=1}^{m} \lambda_j \frac{\partial y_j}{\partial x_i},\ i = 1,\ \ldots,\ n \right\}, \tag{2.1}$$

containing all the partial derivatives of the function, F.

In Appendix A is possible to observe, with more detail, how the Automatic Adjoint Differentiation proceeds for the calculation of all partial derivatives of a function with more inputs compared to the number of outputs with a practical example.

Sometimes, certain problems involve the calculation of expectations, Ey, and, therefore, the differentiation of such elements, $\frac{\partial}{\partial x}Ey$, is inevitable.

By the analysis of the information retrieved in the article [9] the Automatic Adjoint Differentiation proceeds to the calculation of Ey executing the algorithm that computes y multiple times, calculating in the end its mean value through parallel computations, the same procedure is applied in $\frac{\partial}{\partial x}Ey$ but, these results have a huge memory consumption, since it evaluates y multiple times. The article [9] presents a better detailed explanation of the reason of such phenomenon.

Appendix B shows a solution presented in the article [9], on how to avoid this problem and proceed to the computation in a generalized way.

2.2 European Call Options

A European Option, according to the information found in [12], is a type of option contract where the investor is authorized to execute it at the time established in the beginning of the contract. A Call Option represents the right of the investor to acquire an underlying security or stock.

With the information found in [10, pp. 3-7] and following the same line of reasoning, a European Call Option is given by the expected value of the payoff, where the payoff is

$$(S(T) - K)^+ = \max\{0, S(T) - K\},$$

with $S(T)$ representing the stock price at a time T in the future, maturity time (when it can be executed), and K the strike price. The payoff, the Call Option, works as follows: if at time T, $S(T) > K$, the investor executes it and the profit is $S(T) - K$; if, the opposite occurs, there is no gain. As said before, a European Call Option is represented as $E[(S(T) - K)^+]$.

According to [10, pp. 3-7], $S(T)$ are random variables, since they represent values in the future, and their behaviour are described with the Black-Scholes model, [10, pp. 24-25], through the stochastic differential equation

$$\frac{\partial S(T)}{S(T)} = rdt + \sigma dW(T) \tag{2.2}$$

where caracterizes a Brownian Motion which is represented by a normal distribution with mean 0 and variance T, r is the risk-free interest rate and σ the volatility of the stock price. The values $S(T)$ can be represented by the solution of (2.2), which is

$$S(T) = S(0)e^{\left(\left[r-\frac{1}{2}\sigma^2\right]T+\sigma W(T)\right)}$$

where it can be reformulated as follows,

$$S(T) = S(0)e^{\left(\left[r-\frac{1}{2}\sigma^2\right]T+\sigma\sqrt{T}Z\right)},$$

with Z, a standard normal random variable, resulting in, $S(T)$ following a log-normal distribution[1].

Appendix C, demonstrates how the Automatic Adjoint Differentiation is applied to European Call Options finding the partial derivatives for the parameters of volatility, σ, and the risk-free interest rate, r.

2.3 Monte Carlo Simulation

A Monte Carlo Simulation, according to [10, pp. 1-3], is a numerical technique with the primary objective of estimating the expected value of a certain stochastic process through a series of simulations that characterize the process under study. Therefore, it approximates the estimation to the expected value of the stochastic process with a normal distribution with a certain mean value, Y, and a standard deviation of, $\frac{\sigma}{\sqrt{N_{\text{mc}}}}$, increasing the precision of the results when N_{mc}, the number of simulations, tends to infinite.

Supposing that, it is necessary to compute $Y := Ey$, the expected value of y, where it stands for random values given by a certain distribution, the Monte-Carlo Simulation simulates independent and identical distributed random values, $y_1, \ldots, y_{N_{\text{mc}}}$, the value N_{mc} represents the number of simulations of Monte-Carlo, and calculates the estimator of Y with the average value of the summation of all y_j, with $j = 1, \ldots, N_{\text{mc}}$, which is given by,

$$\bar{Y} = \frac{1}{N_{\text{mc}}} \sum_{j=1}^{N_{\text{mc}}} y_j. \tag{2.3}$$

By the Central Limit Theorem, for numerous simulations, like $N_{\text{mc}} \longrightarrow \infty$, $\bar{Y}$ converges to a normal distribution with mean Y and standard deviation of $\frac{\sigma}{\sqrt{N_{\text{mc}}}}$, and the estimations of Y to the corresponding expected value as well as, their standard deviations, which converge to 0 at a convergence rate of $\frac{1}{N_{\text{mc}}}$.

[1] Once, the variable Z follows a normal distribution, making $S(T)$, a exponential, a log-normal distribution

Chapter 3

Case Study

3.1 Introduction

This case study will concentrate on how to compute gradients of the Mean Squared Error involving expectations, characterized by the equation,

$$MSE = \frac{1}{m}\sum_{i=1}^{m}\left(Ey_i - \hat{C}_i\right)^2 \tag{3.1}$$

with the Automatic Adjoint Differentiation and the application of two distinctive estimation approaches, analyzing on how it works and the performance of the theoretical results. Each variable in (3.1) is defined as m, corresponding to the size of the set, $\hat{C}_i$, representing constant values, y_i, is given by,

$$F : \mathbb{R}^{N+M}_{(x_1,\ldots,x_N|a_1,\ldots,a_M)} \longrightarrow \mathbb{R}^{m}_{(y_1,\ldots,y_m)}$$

with N the number of parameters, M the number of random variables, m, the number of outputs, x_k and a_k represent the parameters and random variables, respectively, and Ey_i are expectations values.

With the intentation of creating a computer program in a C++ environment with a AAD tool incorporated to evaluate the estimates of both approaches, it is used computational parallelization to improve, even more, the performance of the computation.

In addition to this introductory section, the second section, demonstrates the application of two of the estimation methods, [1, 2], analyzing the capability of estimating different partial derivatives of (3.1) as well, the respective computational costs. The third section corresponds to an improvement of the two estimation approaches using computational parallelization when implemented in a C++ environment. The last section corresponds to the practical case study to a set of European Call Options, through the implementation of both estimation approaches into a program in a C++ environment incorporated with a AAD tool. The results retrieved from the developed programs are analyzed and compared to corroborate with the theoretical results.

3.2 Derivation of Mean Squared Error

This section will concentrate in the computation of the gradients of Mean Squared Error involving expectations, using the method of Automatic Adjoint Differentiation and the Monte-

Carlo Simulation, given by

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(Ey_i - \hat{C}_i)^2. \tag{3.2}$$

Therefore, the derivation of (3.2) with respect of a parameter x_k is given by,

$$\begin{aligned}\frac{\partial MSE}{\partial x_k} &= \frac{\partial}{\partial x_k}\left(\frac{1}{m}\sum_{i=1}^{m}\left(Ey_i - \hat{C}_i\right)^2\right) = \frac{2}{m}\sum_{i=1}^{m}\left(Ey_i - \hat{C}_i\right)\frac{\partial Ey_i}{\partial x_k} \\ &= \frac{2}{m}\sum_{i=1}^{m}\left(Ey_i - \hat{C}_i\right)E\left[\frac{\partial y_i}{\partial x_k}\right] = E\left[\frac{2}{m}\sum_{i=1}^{m}\left(Ey_i - \hat{C}_i\right)\frac{\partial y_i}{\partial x_k}\right].\end{aligned} \tag{3.3}$$

As it's possible to observe from the equation (3.3), the Automatic Adjoint Differentiation method is applied to y_i, which is given by the forward algorithm,

$$F : \mathbb{R}^{M+N}_{(a_1,\,...,\,a_M|x_1,\,...,\,x_N)} \longrightarrow \mathbb{R}^{m}_{(y_1,\,...,\,y_m)} \tag{3.4}$$

with N parameters and M independent random variables, afterwards, it's applied the reverse algorithm, R, in (3.4) with the adjoint value , λ_i, represented in (2.1) in Section 2.1.1, $\frac{2}{m}\left(Ey_i - \hat{C}_i\right)$. Since it'll be necessary to calculate the expected value of y_i, Ey_i, the same process learned in Section 2.3, will be applied with the formula (2.3).

Which of these algorithms will have their own computational cost, for the algorithm forward will be designated by $\mathrm{Cost}(F)$ and for the reverse one it'll be $\mathrm{Cost}(R)$.

From this moment, it'll be introduced the two estimation approaches. The first approach, where it can be found in both articles [1, 2], for the calculating of the gradient estimate of (3.2) is given by,

$$\mathrm{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right) = \frac{1}{N_{\mathrm{mc}}}\sum_{j=1}^{N_{\mathrm{mc}}}\frac{2}{m}\sum_{i=1}^{m}\left(Ey_i - \hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k} \tag{3.5}$$

where x_k corresponding to a parameter, for $k = 1,\ \ldots,\ N$. The next algorithm shows the procedure for calculation of the estimatives.

Algorithm 3.2.1 *1. Calculate Ey_i using the forward algorithm, F, resulting in the computational cost of $N_{\mathrm{mc}} \cdot \mathrm{Cost}(F)$;*

2. Afterwards, apply the reverse algorithm, R, for the computation of AAD of (3.2) *with the adjoint value, λ_i which is represented in* (2.1), *$\frac{2}{n}\left(Ey_i - \hat{C}_i\right)$ fixing a random vector, w_j, $j = 1,\ \ldots,\ N_{\mathrm{mc}}$. Obtaining the following sequence:*

$$\left\{\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i - \hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k},\ k = 1, \ldots, N\right\}.$$

Obtaining the computational cost of $\mathrm{Cost}(F) + \mathrm{Cost}(R)$ in which simulation, since it'll be needed to apply the forward algorithm, for the realization of the reverse algorithm;

3. *Finally, sum all paths* $w_j,\ j = 1,\ \ldots,\ N_{\text{mc}}$ *and divide by the total number of simulations used to obtained the intended estimations.*

The total cost of the algorithm of the gradient estimative of (3.2), G_1, *is given by,*

$$\text{Cost}\,(G_1) = N_{\text{mc}} \cdot (2 \times \text{Cost}(F) + \text{Cost}(R))\,.$$

With the stated **Algorithm 3.2.1**, it'll be necessary to demonstrate that the resulting estimations of this approximation method converges to the corresponding parcial derivative. Thus, the following proposition illustrates that such results tend to the exact value.

Proposition 3.2.1 *We have*

$$E\left[\text{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right)\right] = \frac{\partial MSE}{\partial x_k}$$

and

$$N_{\text{mc}} \cdot \text{Var}\left[\text{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right)\right]$$

is bounded above by a number independent of N_{mc}. *The variable* x_k *to a parameterof* (3.2).

The **Proposition 3.2.1** proof can be found in the Appendix D.2. Therefore, the proposition says that the estimates calculated by (3.5) of the different deivations of (3.2) converge to the exact results applying the Central Limit Theorem for a high number of simulations, $N_{\text{mc}} \to \infty$. The same situation happens for their respective standard deviation, $\sqrt{\text{Var}\left[\text{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right)\right]}$, where they converge to 0 at a convergence rate of $\frac{1}{\sqrt{N_{\text{mc}}}}$, as it is displayed in the Monte-Carlo Simulation in [10].

The next estimation approach used, where it can be found in the article [2], is introduced as an improvement torwards the previous approach in terms of its computational cost without impairing the precison of the results coming from the algorithm. The estimation approach of (3.2) with respect to parameter of x_k is given by,

$$\text{Est}_2\left(\frac{\partial MSE}{\partial x_k}\right) = \frac{1}{N_{\text{mc}} - 1} \sum_{j=2}^{N_{\text{mc}}} \frac{2}{m} \sum_{i=1}^{m} \left(S_i(w_{j-1}) - \hat{C}_i\right) \frac{\partial y_i(w_j)}{\partial x_k}$$

where

$$S_i(w_{j-1}) = \frac{1}{j-1} \sum_{l=1}^{j-1} y_i(w_l)$$

with x_k corresponding to a parameter, for $k = 1,\ \ldots,\ N$. The following algorithm demonstrates the procedure for calculation of the estimatives of (3.2).

Algorithm 3.2.2 *1. Fixing a random vector* w_j, $j = 2,\ \ldots,\ N_{\text{mc}}$ *and, afterwards, is applied the reverse algorithm, R, with* $\lambda_i = \frac{2}{n}\left(S_i(w_{j-1}) - \hat{C}_i\right)$. *Thus, one obtains the next sequence,*

$$\left\{\frac{2}{n}\sum_{i=1}^{n}(S_i(w_{j-1}) - C_i)\frac{\partial y_i(w_j)}{\partial x_k},\ k = 1, \ldots, N\right\}.$$

To apply the reverse algorithm, R, it'll be necessary to firstly apply the forward algorithm, F, obtaining a cost of $\text{Cost}(F) + \text{Cost}(R)$*;*

2. *Finally, sum all paths w_j, $j = 2, \ldots, N_{\text{mc}}$ and divide by the total number of simulations used to obtained the intended estimations.*

The total cost of the algorithm of the gradient estimative of (3.2)*, G_2, is given by,*

$$\text{Cost}\left(G_2\right) = N_{\text{mc}} \cdot \left(\text{Cost}(F) + \text{Cost}(R)\right).$$

Like the way it was performed for the **Algorithm 3.2.1**, it'll be necessary to show that the resulting estimations calculated from the **Algorithm 3.2.2** that converge to the respective exact value. This way, the following proposition illustrates such results.

Proposition 3.2.2 *We have*

$$E\left[\text{Est}_2\left(\frac{\partial MSE}{\partial x_k}\right)\right] = \frac{\partial MSE}{\partial x_k}$$

and

$$N_{\text{mc}}^{\frac{2}{3}} \cdot \text{Var}\left[\text{Est}_2\left(\frac{\partial MSE}{\partial x_k}\right)\right]$$

is bounded above by a number independent of N_{mc}. The variable x_k to a parameterof (3.2)*.*

The previous proposition's proof can be found in Appendix D.3. From the conclusions drawn from the **Proposition 3.2.2**, the estimates of the partial derivatives of (3.2) converge to the respective partial derivatives when applying the Central Limit Theorem for a number of simulations tending to infinity, $N_{\text{mc}} \to \infty$, the same behaviour can be seen in the respective standard deviation, $\sqrt{\text{Var}\left[\text{Est}_2\left(\frac{\partial MSE}{\partial x_k}\right)\right]}$, that converges to the value 0 at a convergence rate of $\frac{1}{\sqrt[3]{N_{\text{mc}}}}$. The article [2], offers a justification to why doesn't show a convergence as it's stated in the Monte-Carlo Simulation, a convergence rate of $\frac{1}{\sqrt{N_{\text{mc}}}}$.

3.3 Parallelization of the Computation

The implementation of the algorithms presented in the previous section, will be done in the C++ programming environment, using a specific library for the resolution of the Automatic Adjoint Differentiation (AAD), this library corresponds to the one from MatLogica Ltd, more information can be found in their website [16].

In the same way as in the article [2], the **Algorithms 3.2.1** and **3.2.2** will use the parallelized versions of both the algorithms, forward and reverse, mentioned in the Section 2.1.1, that will have the capacity to process c independent sets of input data with an AAD tool that will be synchronized with an Intel AVX512 architecture, with the value c corresponding to $8 \cdot \{Number\ of\ Cores\}$. Thus, the parallelize versions F_v and R_v, that correspond to the forward, F, and reverse, R, algorithms respectively, will have the following computational costs,

$$\text{Cost}\,(F_v) = \frac{K_F}{c} \cdot \text{Cost}(F)$$
$$\text{Cost}\,(R_v) = \frac{K_R}{c} \cdot \text{Cost}(R)$$

where K_F and K_R corresponds to correction coefficients that reflets the quality of the AAD tool. So, applying this new parallelize versions in the **Algorithms 3.2.1** e **3.2.2**, replacing the algorithms F by F_v and R by R_v, the new computational costs will become

$$\text{Cost}\,(G_1) = N_{\text{mc}} \cdot \left(\frac{2 \times K_F}{c} \cdot \text{Cost}(F) + \frac{K_R}{c} \cdot \text{Cost}(R)\right)$$
$$\text{Cost}\,(G_2) = N_{\text{mc}} \cdot \left(\frac{K_F}{c} \cdot \text{Cost}(F) + \frac{K_R}{c} \cdot \text{Cost}(R)\right)$$

where G_1 and G_2 represent the gradient estimates of $\frac{1}{n}\sum_{i=1}^{n}\left(Ey_i - \hat{C}_i\right)^2$ calculated from the **Algorithms 3.2.1** and **3.2.2**, respectively.

In the next section both algorithms will be applied for a set of non-real European Options, using the new parallelized methods of AAD, where it'll be analyzed and compared the convergence of the estimates of the partial derivatives of $\frac{1}{n}\sum_{i=1}^{n}\left(Ey_i - \hat{C}_i\right)^2$.

3.4 Results and Discussion

With the algorithms presented in Section 3.1 and with the implemented codes for the respective **Algorithms 3.2.1** and **3.2.2**, Appendix E, this section focus in evaluate the precision and computational costs, in this case will correspond to the execution time of the programs, on the estimate gradients of

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Ey_i - \hat{C}_i\right)^2 \tag{3.6}$$

as well as if the convergence of the respective variances corresponds to the theoretical results presented in Section 3.1. Therefore, different partial derivatives of (3.6) will be calculated for a set of non-real European Call Options. Namely, it is considered the following parameters:

- K_i - strike prices;
- $S(0)$ - initial stock price with a value of 100;
- T_i - maturity times;
- $sigma_i$ - knots of a piecewise constant function of volatilities;
- r_i - knots of a piecewise constant function of risk-free interest rate;
- y_i - Call Options $(S(T_i) - K_i)^+$ where $S(T_i)$ is given by,

$$S(T_i) = S(T_{i-1})e^{\left(\left[r_{T_i} - \frac{1}{2}\sigma_{T_i}^2\right] \times (T_i - T_{i-1}) + \sigma_{T_i}\sqrt{T_i}Z\right)},$$

with $i = 1, \ldots, m$, $T_0 = 0$, and $S(T_0) = S(0)$;

- Ey_i - expectations of the Call Options $(S(T_i) - K_i)^+$;
- m - the size of the set;
- $\hat{C}_i$ - observed prices;

Parameters	*European Option* 1	*European Option* 2	*European Option* 3	*European Option* 4	*European Option* 5
T	1	2	3	4	5
σ	0.005	0.004	0.0045	0.003	0.0035
r	0.1	0.11	0.09	0.12	0.13
K	95	105	115	90	100
$\hat{C}$	15.7925	33.6074	67.7963	207.187	474.268

Table 3.1: Parameters used for the set of $m = 5$ call options of European Options, for an intial stock price $S(0) = 100$.

The parameters found in the previous table, corresponds to the ones applied in codes with the **Algorithms 3.2.1** e **3.2.2** implemented for the computation of the observed prices, $\hat{C}_i$ in (3.6). To compute Ey_i, which differs from $\hat{C}_i$ and calculate the gradients, there are two sets of partial derivatives that will be evaluated: the first one, which will be designated as *Vega*, corresponds to the partial derivatives with respect to σ_i, and so, it is initiated with different values compared to the Table 3.1; the second set, denoted by *Rho*, contains the partial derivatives is with respect to r_k, also initialized with different values.

To analyze the convergence of the results of each of the programs created, there will be applied a different number of simulations, 10^5, 10^6, 10^7 and 10^8, analyzing each member of the different gradients computated by the two programs and their respective variances to corroborate the theoretical results retrieved in Section 3.2.

The Tables 3.2 and 3.3 represent all the estimated gradients, calculated with the two algorithms, with respect to σ_k (*Vega*) and to r_k (*Rho*), for 10^7 simulations.

As it can be observed from previous tables, the different estimate gradients of (3.6) using the Automatic Adjoint Differention resulting from the two estimation approaches, applying the same number of simulations, 10^7, present similar results as it was proven in the first statement of both **Propositions 3.2.1** and **3.2.2**, converging to the respective exact value. Another observation is the execution time of **Algorithm 3.2.1** compared to **Algorithm 3.2.2**, which is slower, verifying the theoretical costs presented in Section 3.2 and refuting the conclusions retrived from the article [2].

The next results are used to analyze the precision of the estimates obtained by both algorithms, as well the second statement in each of the **Propositions 3.2.1** and **3.2.2**, analyzing, firtsly, the variances of estimates obtained from the **Algorithm 3.2.1** for 10^5, 10^6, 10^7 and 10^8 simulations.

According to the results of the variances of each estimations, for *Vega* and *Rho*, obtained through the **Algorithm 3.2.1**, are represented in the Tables 3.4 and 3.5. It's possible to notice that each variance of the respective members of the estimate gradient are finite for whatever

Vega	*Algorithm 3.2.1*	*Algorithm 3.2.2*
Time (μs)	3772928	2329670
$\frac{\partial MSE}{\partial \sigma_1}$	348.303	348.265
$\frac{\partial MSE}{\partial \sigma_2}$	696.289	696.215
$\frac{\partial MSE}{\partial \sigma_3}$	1033.88	1033.77
$\frac{\partial MSE}{\partial \sigma_4}$	1335.38	1335.25
$\frac{\partial MSE}{\partial \sigma_5}$	1425.55	1425.44

Table 3.2: Estimated gradient of MSE with respect to σ_i, volatility, for $N_{\text{mc}} = 10^7$, using both Algorithms, 3.2.1 and 3.2.2.

Rho	*Algorithm 3.2.1*	*Algorithm 3.2.2*
Time (μs)	3543793	2268920
$\frac{\partial MSE}{\partial r_1}$	250.474	250.406
$\frac{\partial MSE}{\partial r_2}$	500.56	500.425
$\frac{\partial MSE}{\partial r_3}$	746.527	746.334
$\frac{\partial MSE}{\partial r_4}$	963.163	962.931
$\frac{\partial MSE}{\partial r_5}$	1050.34	1050.13

Table 3.3: Estimated gradient of MSE with respect to r_i, risk-free interest rate, for $N_{\text{mc}} = 10^7$, using both Algorithms, 3.2.1 and 3.2.2.

N_{mc}	10^5	10^6	10^7	10^8
Time (μs)	34022	388629	3772928	36257694
Var MSE_1	0.0261154	0.0026272	0.000261965	$2.62496e-5$
Var MSE_2	0.0526428	0.00529476	0.000529709	$5.30297e-5$
Var MSE_3	0.0785114	0.007896	0.000791706	$7.92139e-5$
Var MSE_4	0.0985318	0.00991266	0.000992271	$9.92393e-5$
Var MSE_5	0.0906596	0.00914299	0.000913487	$9.13612e-5$

Table 3.4: Variances and execution times for different values of N_{mc} using the Algorithm 3.2.1 for Var $MSE_k = \frac{\partial MSE}{\partial \sigma_k}$.

N_{mc}	10^5	10^6	10^7	10^8
Time (μs)	33352	401884	3543793	36081695
Var MSE_1	$2.35309e-6$	$2.36345e-7$	$2.36624e-8$	$2.36753e-9$
Var MSE_2	$9.4104e-6$	$9.45183e-7$	$9.463e-8$	$9.46819e-9$
Var MSE_3	$2.10962e-5$	$2.1189e-6$	$2.1214e-7$	$2.12256e-8$
Var MSE_4	$3.6093e-5$	$3.62512e-6$	$3.62938e-7$	$3.63133e-8$
Var MSE_5	$4.57242e-05$	$4.59261e-6$	$4.59775e-7$	$4.6001e-8$

Table 3.5: Variances and execution times for different values of N_{mc} using the Algorithm 3.2.1 for Var $MSE_k = \frac{\partial MSE}{\partial \sigma_k}$.

number of simulations used, N_{mc}. The convergence of the results, demonstrates a convergence rate of $\frac{1}{N_{\text{mc}}}$, meaning, their respective standard deviation show a convergence at a rate of $\frac{1}{\sqrt{N_{\text{mc}}}}$ as it was theorized in **Proposition 3.2.1** and corroborating the results of the article [2] and with the Monte-Carlo's convergence property, where it can be found in [10, pp. 1-3].

Concluding that, according to the statements given by the **Proposition 3.2.1** and with results presented in the Tables 3.2 and 3.3 and with the Tables 3.4 and 3.5, related to the **Algorithm 3.2.1** proves that the estimates converge to the exact value of the partial derivatives of (3.6), when the number of simulations tending to infinity.

Finally, the Tables 3.6 and 3.7 show the variances of the respective estimate gradients of (3.6) obtained from the **Algorithm 3.2.2** for a certain number of simulations, 10^5, 10^6, 10^7 and 10^8.

N_{mc}	10^5	10^6	10^7	10^8
Time (μs)	21721	251495	2329670	25052998
Var MSE_1	0.0256897	0.00260281	0.00026194	$2.62453e-5$
Var MSE_2	0.052124	0.00525073	0.00052972	$5.30217e-5$
Var MSE_3	0.0780895	0.00783717	0.000791801	$7.92029e-5$
Var MSE_4	0.0984905	0.00984705	0.000992484	$9.9227e-5$
Var MSE_5	0.0911358	0.00909177	0.000913781	$9.13519e-5$

Table 3.6: Variances and execution times for different values of N_{mc} using the Algorithm 3.2.2 for Var $MSE_k = \frac{\partial MSE}{\partial \sigma_k}$.

N_{mc}	10^5	10^6	10^7	10^8
Time (μs)	19803	221739	2268920	17102558
Var MSE_1	0.000425014	$7.01457e-6$	$1.07079e-7$	$3.20655e-9$
Var MSE_2	0.0016849	$2.77801e-5$	$4.24807e-7$	$1.27893e-8$
Var MSE_3	0.00368139	$6.0543e-5$	$9.30506e-7$	$2.84513e-8$
Var MSE_4	0.00598099	$9.79083e-5$	$1.52058e-6$	$4.7956e-8$
Var MSE_5	0.00665603	0.000107899	$1.72146e-6$	$4.16689e-7$

Table 3.7: Variances and execution times for different values of N_{mc} using the Algorithm 3.2.2 for Var $MSE_k = \frac{\partial MSE}{\partial \sigma_k}$.

With the results of the previous two tables, it can be concluded that, the values are finite regardless the number of simulations, N_{mc}. The variances of the estimate partial derivatives converge to 0 at a approximate rate of $\frac{1}{N_{\text{mc}}}$, with the increase of simulations, surpassing the limit established in **Proposition 3.2.2**. The article [2] offers a more detailed explanation about the behaviour of this approximation method.

With the results presented in the Tables 3.2 to 3.3 and Tables 3.6 to 3.7, prove the statements stated by the **Proposition 3.2.2**, resulting that the estimates calculated by the **Algorithm 3.2.2** converge to the exact value of each patial derivative of (3.6), as the number of simulations tends to infinity.

Chapter 4

Conclusion

In this thesis, it was applied, modifed and implemented 2 alternative approximation methods of the calculation of the gradient estimates belonging to the Mean Squared Error for a set of European Call Options, applying the Automatic Adjoint Differentiation with the Monte-Carlo Simulation as well as parallelization of the computation. Chapter 3 shows that both methods produce similar results compared to each other with similar convergence behaviour to the one announced in the Monte-Carlo Simulation, for the same number of simulations. It was also notice that, the second method anounced, possesses a computational cost smaller than the computational cost of the first approach.

Firstly, with the aid of [2, 10], I developed proofs that, with the application of Automatic Adjoint Differentiation and Monte-Carlo Simulation, the gradient estimates for a set of parameters, σ (volatility) and r (risk-free interest rate), the European Call Options converge to the respective exact value, this proofs can be found in Section 2.3.

Lastly, the two approximation methods for the gradient estimates for different parameters of the Mean Squared Error, for a set of European Call Options, following the same line of reasoning of [2] and with the properties of the Monte-Carlo Simulation, where it can be also found in [10], and demonstrate that they converge to the exact value, see Section 3.1 for the theoretical proofs of the **Propositions 3.2.1** and **3.2.2**. Another observation, also important, that can be established is about the computational cost an precision of the results from both algorithms established in Section 3.1, with the implemented codes where they can be consulted in Appendix E, present similar solutions comparing each algorithm and, the computational cost of the **Algorithm 3.2.2** is inferior comparing with the **Algorithm 3.2.1**, as it was theorized in Section 3.1.

Appendix A

Procedure of Automatic Adjoint Differentiation

According to the information provided from the articles [7, 8], the Automatic Adjoint Differentiation is composed with 2 distinctive methods, the forward and the reverse algorithm, where the first one calculates the partial derivatives of a function applying the chain rule starting from each input parameter towards the output parameters, as for the reverse algorithm, it applys the chain rule in the opposite direction, from each output parameter to the input parameters.

For a program implemented with a function,

$$F : \mathbb{R}^n_{(x_1,\ldots,x_n)} \longrightarrow \mathbb{R}^m_{(y_1,\ldots,y_m)}$$

where the objective of finding the Jacobian matrix of F. By the analysis of the article [8], the forward algorithm, initializes each member of a vector v with the input parameters and existing intermediate operations of the function F, and associating to a vector $\dot{v}$ containing all partial derivatives $\frac{\partial v}{\partial x_i}$, which means, $\dot{v} = \nabla_x v$. Next, the derivatives will be calculated and combine with the ones previous calculated through the chain rule.

This algorithm is more efficient when, m, the number of output parameters, is higher than n, the number of input parameters, once it only be necessary to execute this algorithm n times.

Next, a practical example will be presented of how the forward algorithm procedes to compute the partial derivatives of a function under study.

Exemplo A.0.1 *Let F be a function like*

$$F : \mathbb{R}^3_{(x_1,x_2,x_3)} \longrightarrow \mathbb{R}_y$$

onde,

$$F(x_1, x_2, x_3) = y = x_1 \times e^{x_2} + x_3.$$

1. *It's initialized the vector v with the input parameters and existing intermediate operations of the function F, obtaining,*

$$v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ e^{v_2} \\ v_1 \times v_4 \\ v_5 + v_3 \end{bmatrix}.$$

2. *Associate a vector $\dot{v}$, which contain all partial derivatives, $\frac{\partial v}{\partial x}$, to the vector v, resulting in,*

$$\dot{v} = \begin{bmatrix} \dot{v}_1 \\ \dot{v}_2 \\ \dot{v}_3 \\ \dot{v}_4 \\ \dot{v}_5 \\ \dot{v}_6 \end{bmatrix} = \begin{bmatrix} \dot{v}_1 \\ \dot{v}_2 \\ \dot{v}_3 \\ e^{v_2} \cdot \dot{v}_2 \\ v_4 \cdot \dot{v}_1 + v_1 \cdot \dot{v}_4 \\ \dot{v}_5 + \dot{v}_3 \end{bmatrix}.$$

3. *Finally, which partial derivative of F is calculated in the following way,*

$$\dot{v}_i = \frac{\partial x_i}{\partial x_j} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}, \text{ for } 1 \leq i, j \leq n = 3,$$

resulting in the gradient

$$\nabla_x F(x_1, x_2, x_3) = (e^{x_2}, x_1 \cdot e^{x_2}, 1).$$

As it was previously said , the computational cost corresponds to the number of input parameters.

In case of the reverse algorithm applied to F, of the information provided by the article [8], this one is initialized with the forward algorithm, each member of the vector v is initialized with each input parameter and existing intermediate oprations of the function F, where it'll be associated to a vector $\bar{v}$ where it'll be containing all partial derivatives $\frac{\partial y_i}{\partial v}$ but, unlike the forward algorithm, it'll be necessary to initialize the vector v so that it's possible to form $\bar{v}$ since the partial derivatives are calculated from the output parameters to the input ones.

Therefore, this algorithm is more efficient when the number of input parameters is higher than the output parameters, $n > m$, but, even though it has the advantage of being more efficient, faster, it requires more storage memory. In the article [8], shows a type of storage specialized for this algorithm, the Wengert list, and how it works and stores data.

Next, a practical example will be presented, the same in **Example A.0.1**, of how the reverse algortihm procedes to compute the partial derivatives of a function uner study.

Exemplo A.0.2 *Let F be a function like*

$$F : \mathbb{R}^3_{(x_1,x_2,x_3)} \longrightarrow \mathbb{R}_y$$

onde,

$$F(x_1, x_2, x_3) = y = x_1 \times e^{x_2} + x_3.$$

1. *It's initialized the vector v with the input parameters and existing intermediate operations of the function F, applying the forward algorithm obtaining,*

$$v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ e^{v_2} \\ v_1 \times v_4 \\ v_5 + v_3 \end{bmatrix}.$$

2. *Associate a vector $\bar{v}$, which contains all partial derivatives, $\frac{\partial y}{\partial v}$, initializing with the last member through the first one, which means,*

$$\bar{v} = \begin{bmatrix} \bar{v}_1 \\ \bar{v}_2 \\ \bar{v}_3 \\ \bar{v}_4 \\ \bar{v}_5 \\ \bar{v}_6 \end{bmatrix} = \begin{bmatrix} \bar{v}_5 \cdot v_4 \\ \bar{v}_4 \cdot e^{v_2} \\ \bar{v}_6 \\ \bar{v}_5 \cdot v_1 \\ \bar{v}_6 \\ \bar{v}_6 \end{bmatrix}.$$

3. *Finally, each partial derivative of F is initialized as follows,*

$$\bar{v}_{N+M+i} = \frac{\partial y_i}{\partial y_j} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}, \text{ for } 1 \leq i, j \leq m = 1,$$

where N represents the number of input parameters (in this case 3) and M the number of intermediate operations (also 3), resulting in the gradient,

$$\nabla_x F(x_1, x_2, x_3) = (e^{x_2}, x_1 \cdot e^{x_2}, 1).$$

As it was previously said, the computational cost corresponds to the number of output parameters.

For the primary objective of this thesis, the algorithm that is going to be applied will be the reverse one since, the function under study contains multiples input parameters and just a few output parameters.

Appendix B

Automatic Adjoint Differentiation involving Expectations

When Automatic Adjoint Differentiation (AAD) is applied to expectations, Ey, more attention is needed in his differentiation since, as referenced in articles [1, 9], it's mandatory to avoid differentiations of expectations. Next, it'll be presented how AAD computes the results and partial derivatives of expected values.

By the information found in the article [1], the expectation of a variable, Ey, is approximated applying a Monte-Carlo Simulation as follows,

$$Ey \sim \frac{1}{N_{\text{mc}}} \sum_{k=1}^{N_{\text{mc}}} y(w_k),$$

with N_{mc} representing the number of simulations applied, w_k represents a random vector from a standard normal distribution for each simulation k and y, is obtained applying the forward algorithm (Appendix A) in the algorithm implemented with the function under study.

When it comes to the differentiation of expectations, of what is possible to retrive from the article [9], it's mandatory to avoid the differentiation of expectations, thus,

$$\frac{\partial}{\partial x} Ey = E\left[\frac{\partial y}{\partial x}\right],$$

this way, AAD for expectations will be given by an algorithm from the article [9], where, for a better understanding of its procedure, it'll be used the same notations. Therefore, the Automatic Adjoint Differentiation will be applied to the algorithm

$$y = f\left(x_0, \ldots, x_{n-1}\right)$$

where y and x_i represent random values. And, for all $n \leq m \leq N$,

$$x_m = f_m\left(x_{i_1^{(m)}}, \ldots, x_{i_{k^{(m)}}^{(m)}}\right)$$

represent results of intermediate operations, where f_m is an operator of $k^{(m)}$ arguments.

The algorithm suggested by the article [9] is shown below in a generalized and rough way,

Algorithm B.0.1

$$y := x_N$$
$$x_m := f_m\left(x_{i_1^{(m)}}, \ldots, x_{i_{k^{(m)}}^{(m)}}\right) \text{ for all } 1 \leq m < N$$

where k operator represents the expected operator, $x_k = f_k\left(x_{i_1^{(m)}}\right) = E\left[x_{i_1^{(m)}}\right]$.

- *For* $m = N-1,\ \ldots,\ n$:

$$\left.\frac{\partial y}{\partial x_m}\right|_{x_k} = \sum_{l \in I,\, l \neq k} \left.\frac{\partial y}{\partial x_l}\right|_{x_k} \cdot \frac{\partial f_l}{\partial x_m}\left(x_{i_1^{(l)}}, \ldots, x_{i_{k^{(l)}}^{(l)}}\right)$$

  - *For* $j = i_1^{(k)} - 1,\ i_1^{(k)} - 2,\ \ldots,\ n$:

$$\frac{\partial x_{i_1^{(k)}}}{\partial x_m} = \sum_{l \in I} \frac{\partial x_{i_1^{(k)}}}{\partial x_l} \cdot \frac{\partial f_l}{\partial x_m}\left(x_{i_1^{(l)}}, \ldots, x_{i_{k^{(l)}}^{(l)}}\right)$$

    where I represents all indexes l where f_l *depends on* x_m *with the initial condition,*

    * $\left.\frac{\partial y}{\partial x_N}\right|_{x_k} = \left.\frac{\partial y}{\partial y}\right|_{x_k} = 1$;
    * $\frac{\partial x_{i_1^{(k)}}}{\partial x_{i_1^{(k)}}} = 1$;

Then,

$$\frac{\partial y}{\partial x_m} = \left.\frac{\partial y}{\partial x_m}\right|_{x_k} + \left.\frac{\partial y}{\partial x_k}\right|_{x_k} \cdot E\left[\frac{\partial x_{i_1^{(k)}}}{\partial x_m}\right] \text{ with } \frac{\partial x_{i_1^{(k)}}}{\partial x_m} = 0 \text{ for } m > i_1^{(k)}.$$

It's possible to analyse the previously algorithm with a better detail and more information about his procedure in the article [9].

Appendix C

Automatic Adjoint Differentiation for European Call Options

This section concentrates in how to apply the Automatic Adjoint Differentiation (AAD) for european call options, given by

$$y = (S(T) - K)^{+} \tag{C.1}$$

as well as, its expected value, with this, applying the forward algorithm of AAD to the equation (C.1) obtaining the following algorithm,

$$F : \mathbb{R}^{1+5}_{(Z|S(0),\, T,\, r,\, \sigma,\, K)} \longrightarrow \mathbb{R},$$

making it possible to find the value of (C.1) and, with the objective of finding the Jacobian matrix of F, where AAD obtains all linear conbinations of $\frac{\partial y}{\partial x_i}$,

$$\left\{1 \cdot \frac{\partial y}{\partial x_i},\ i = 1,\ \ldots,\ 5\right\}.$$

Note 1 *From this moment forward, certain variables will be named for more simplicity in explaining this matter. I'll designate by* y *and* y_i*, the values of european call options,* $(S(T) - K)^{+}$ *and* $(S_i(T_i) - K_i)^{+}$*, respectivaly, as well as, the expectations of european call options* $E\left[(S(T) - K)^{+}\right]$ *and* $E\left[(S_i(T_i) - K_i)^{+}\right]$*, which will be represented by* Ey *and* Ey_i*, respectivaly.*

The Figure C.1 as well as the Table C.1, demonstrate how the algortihms, forward and reverse, of the Automatic Adjoint Differentiation for the computation of the value and the partial derivatives of y, both algorithms can be represented with an oriented graph and each dot, $\dot{x}_i$, represent the arithmetic operations and/or elementary functions which are decomposed of y and each connection, $\bar{x}_j$, represent the partial derivatives, which are computated as follows,

$$\bar{x}_i = \frac{\partial y}{\partial \dot{x}_i} = \sum_{\dot{x}_j \ sucede \ \dot{x}_i} \bar{x}_j \frac{\partial \dot{x}_j}{\partial \dot{x}_i}.$$

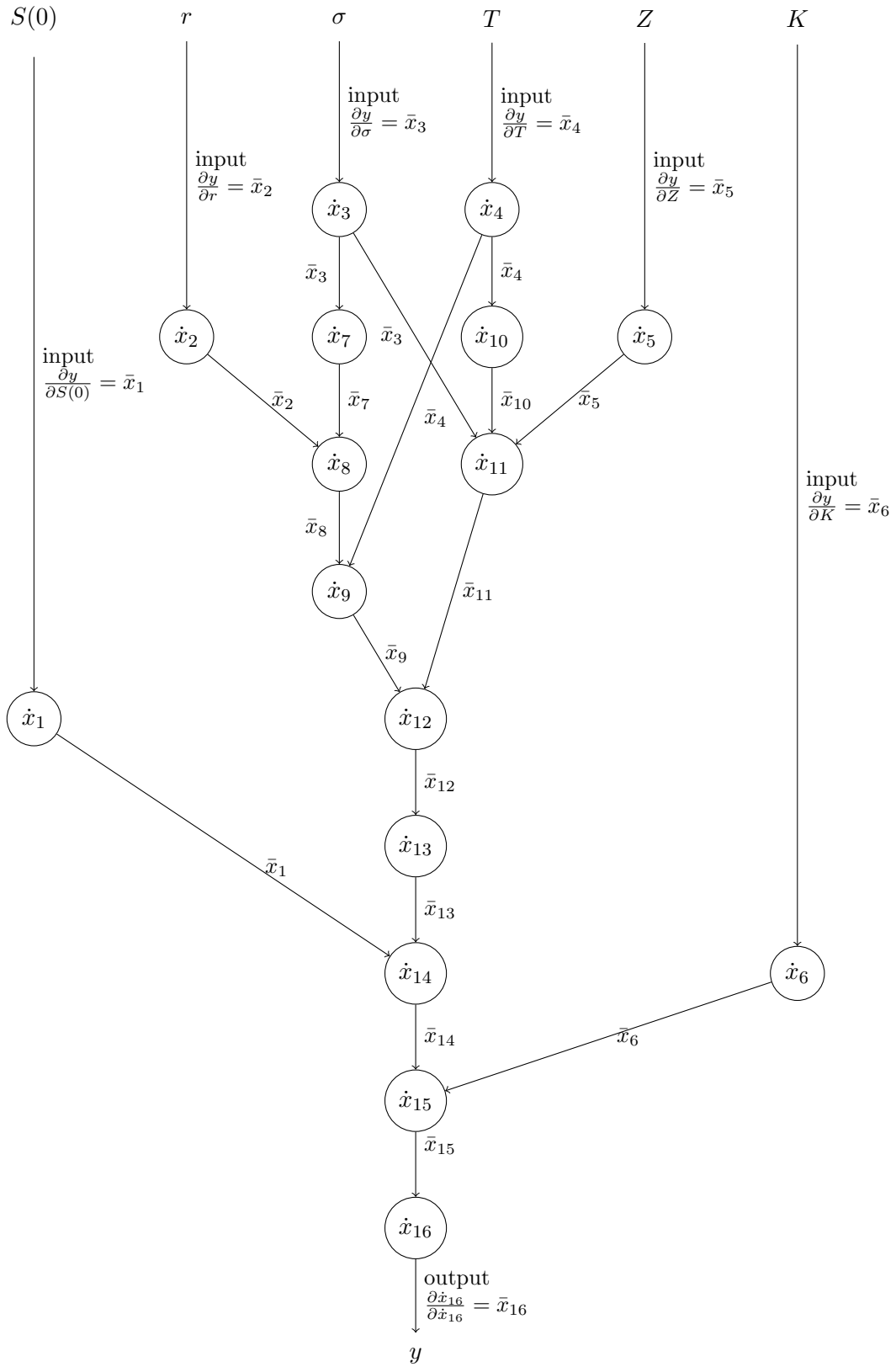


Figure C.1: Computational graph of $y = (S(T) - K)^+$ for *AAD*.

Automatic Adjoint Differentiation for $y = (S(T) - K)^+$		
forward algorithm		reverse algorithm
inputs	$\dot{x}_1 = S(0)$	$\bar{x}_1 = \bar{x}_{14}\frac{\partial \dot{x}_{14}}{\partial \dot{x}_1} = \frac{\partial y}{\partial S(0)}$
	$\dot{x}_2 = r$	$\bar{x}_2 = \bar{x}_8\frac{\partial \dot{x}_8}{\partial \dot{x}_2} = \frac{\partial y}{\partial r}$
	$\dot{x}_3 = \sigma$	$\bar{x}_3 = \bar{x}_7\frac{\partial \dot{x}_7}{\partial \dot{x}_3} + \bar{x}_{11}\frac{\partial \dot{x}_{11}}{\partial \dot{x}_3} = \frac{\partial y}{\partial \sigma}$
	$\dot{x}_4 = T$	$\bar{x}_4 = \bar{x}_{10}\frac{\partial \dot{x}_{10}}{\partial \dot{x}_4} + \bar{x}_9\frac{\partial \dot{x}_9}{\partial \dot{x}_4} = \frac{\partial y}{\partial T}$
	$\dot{x}_5 = Z$	$\bar{x}_5 = \bar{x}_{11}\frac{\partial \dot{x}_{11}}{\partial \dot{x}_5} = \frac{\partial y}{\partial Z}$
	$\dot{x}_6 = K$	$\bar{x}_6 = \bar{x}_{15}\frac{\partial \dot{x}_{15}}{\partial \dot{x}_6} = \frac{\partial y}{\partial K}$
intermediate operations	$\dot{x}_7 = \dot{x}_3^2$	$\bar{x}_7 = \bar{x}_8\frac{\partial \dot{x}_8}{\partial \dot{x}_7}$
	$\dot{x}_8 = \dot{x}_2 - \frac{1}{2}\dot{x}_7$	$\bar{x}_8 = \bar{x}_9\frac{\partial \dot{x}_9}{\partial \dot{x}_8}$
	$\dot{x}_9 = \dot{x}_8 \times \dot{x}_4$	$\bar{x}_9 = \bar{x}_{12}\frac{\partial \dot{x}_{12}}{\partial \dot{x}_9}$
	$\dot{x}_{10} = \sqrt{\dot{x}_4}$	$\bar{x}_{10} = \bar{x}_{11}\frac{\partial \dot{x}_{11}}{\partial \dot{x}_{10}}$
	$\dot{x}_{11} = \dot{x}_3 \times \dot{x}_{10} \times \dot{x}_5$	$\bar{x}_{11} = \bar{x}_{12}\frac{\partial \dot{x}_{12}}{\partial \dot{x}_{11}}$
	$\dot{x}_{12} = \dot{x}_9 + \dot{x}_{11}$	$\bar{x}_{12} = \bar{x}_{13}\frac{\partial \dot{x}_{13}}{\partial \dot{x}_{12}}$
	$\dot{x}_{13} = e^{\dot{x}_{12}}$	$\bar{x}_{13} = \bar{x}_{14}\frac{\partial \dot{x}_{14}}{\partial \dot{x}_{13}}$
	$\dot{x}_{14} = \dot{x}_1 \times \dot{x}_{13}$	$\bar{x}_{14} = \bar{x}_{15}\frac{\partial \dot{x}_{15}}{\partial \dot{x}_{14}}$
	$\dot{x}_{15} = \dot{x}_{14} - \dot{x}_6$	$\bar{x}_{15} = \bar{x}_{16}\frac{\partial \dot{x}_{16}}{\partial \dot{x}_{15}}$
outputs	$\dot{x}_{16} = \max(\dot{x}_{15}, 0) = y$	$\bar{x}_{16} = \frac{\partial y}{\partial \dot{x}_{16}} = \frac{\partial \dot{x}_{16}}{\partial \dot{x}_{16}}$

Table C.1: Operations used by the algorithms forward and reverse for $(S(T) - K)^+$.

Where, each partial derivative of (C.1) is obtained initializing the reverse algorithm as follows,

$$\bar{x}_{N+M+i} = \frac{\partial y_i}{\partial y_j} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}, \text{ for } 1 \leq \text{i, j} \leq \text{m} = 1,$$

where N, the number of inputs, is 6 and M, the number of intermediate operations, is 9.

Applying this method for the calculation of the estimates of the different partial derivatives of Ey, Est $\left(\frac{\partial Ey}{\partial \sigma}\right)$ and Est $\left(\frac{\partial Ey}{\partial r}\right)$. it's known that, using the Monte-Carlo Simulation, the expected value of european call options is given by,

$$Ey \sim \frac{1}{N_{\text{mc}}} \sum_{j=1}^{N_{\text{mc}}} y(w_j),$$

where it's simulated $w_1, \ldots, w_{N_{\text{mc}}}$ (independent and identicaly distributed reandom vectors) provided from a standarized normal distribution, and with N_{mc} representing the number of simulations used to obtaining the expected value of european call options. Thus, the estimates of the different partial derivatives of Ey are obtained as follows,

- Estimative of $\frac{\partial Ey}{\partial \sigma}$: $\text{Est}\left(\frac{\partial Ey}{\partial \sigma}\right) = \frac{1}{N_{\text{mc}}} \sum_{j=1}^{N_{\text{mc}}} \frac{\partial y(w_j)}{\partial \sigma}$;
- Estimative of $\frac{\partial Ey}{\partial r}$: $\text{Est}\left(\frac{\partial Ey}{\partial r}\right) = \frac{1}{N_{\text{mc}}} \sum_{j=1}^{N_{\text{mc}}} \frac{\partial y(w_j)}{\partial r}$.

Therefore, the calculation of such estimative can be found with the following algorithm,

Algorithm C.0.1 *1. Calculate y with the forward algorithm, F, and afterwards, apply the reverse algorithm, R, for the calculation of the AAD of y with the adjoint value, λ, which is represented in* (2.1)*, 1 fixing a random vector w_j, of $j = 1, \ldots, N_{\text{mc}}$. obtaining the following sequence:*

$$\left\{ 1 \frac{\partial y(w_j)}{\partial x_k},\ k = 1, \ldots, N \right\}.$$

Obtaining the computational cosr of $\text{Cost}(F) + \text{Cost}(R)$ *for each simulation.*

2. Finally, sum all paths w_j, $j = 1, \ldots, N_{\text{mc}}$ and divide by the total number of simulations used to obtain the intended estimations.

The total cost of the algorithm is given by,

$$\text{Cost}\left(\frac{\partial Ey}{\partial x_k}\right) = N_{\text{mc}} \cdot (\text{Cost}(F) + \text{Cost}(R)).$$

The following propostion, shows that, the estimated derivatives of Ey with respect to a parameter x_k converge to the respective expected value. The proof of this proposition can be found in Appendix D.1.

Proposition C.0.1 *We have*

$$E\left[\text{Est}\left(\frac{\partial Ey}{\partial x_k}\right)\right] = \frac{\partial Ey}{\partial x_k}$$

and

$$N_{\text{mc}} \cdot \text{Var}\left[\text{Est}\left(\frac{\partial Ey}{\partial x_k}\right)\right]$$

is bounded above by a number independent of N_{mc}. The variable x_k corresponds to a parameter used for the derivation of Ey.

In agreement with the proof of the previous proposition, applying the Central Limit theorem for a highly number of simulations, like $N_{\text{mc}} \to \infty$, the estimate partial derivatives of Ey converge to the respective expected value as well as, the standard deviations, $\sqrt{\text{Var}\left[\text{Est}\left(\frac{\partial Ey}{\partial x_k}\right)\right]}$, converge to the value 0 with a convergence rate of $\frac{1}{\sqrt{N_{\text{mc}}}}$.

Consequently, it´s possible to conclude that the **Algorithm C.0.1**, as the capacity to compute, with success, the gradient estimations of Ey.

Appendix D

Proofs of Propositions

This section contains all the proofs of the **Propositions** scattered through the document, and justifications for their respectives **Algorithms** to compute the gradient estimates that shows convergence at a rate of $\frac{1}{\sqrt{N_{\text{mc}}}}$. Some of these proofs are based of the article [2] since, some of the approximations methods of gradient estimates can be found there.

D.1 Proof of Proposition C.0.1

Proposition 1 *For the estimate*

$$\text{Est}\left(\frac{\partial Ey}{\partial x_k}\right) = \frac{1}{N_{\text{mc}}}\sum_{j=1}^{N_{\text{mc}}}\frac{\partial y(w_j)}{\partial x_k},$$

we have

$$E\left[\text{Est}\left(\frac{\partial Ey}{\partial x_k}\right)\right] = \frac{\partial Ey}{\partial x_k}$$

and

$$N_{\text{mc}}\,\text{Var}\left[\text{Est}\left(\frac{\partial Ey}{\partial x_k}\right)\right] \tag{D.1}$$

is bounded above by a number independent of N_{mc}. The variable x_k corresponds to a parameter used for the derivation of Ey, the expected value of European Call Option, check the Section 2.2.

Proof:

Starting with first statement we get the following computation:

$$\begin{aligned} E\left[\text{Est}\left(\frac{\partial Ey}{\partial x_k}\right)\right] &= E\left[\frac{1}{N_{\text{mc}}}\sum_{j=1}^{N_{\text{mc}}}\frac{\partial y(w_j)}{\partial x_k}\right] \\ &= \frac{1}{N_{\text{mc}}}\sum_{j=1}^{N_{\text{mc}}}E\left[\frac{\partial y(w_j)}{\partial x_k}\right] \\ &= E\left[\frac{\partial y}{\partial x_k}\right] = \frac{\partial Ey}{\partial x_k}. \end{aligned} \tag{D.2}$$

The penultimate equality of the equation (D.2) is due to the fact that w_j are identicaly distributed. For the second statement of the proposition, (D.1), we'll rewrite $\mathrm{Var}\left(\mathrm{Est}\left(\frac{\partial Ey}{\partial x_k}\right)\right)$ as follows:

$$\begin{aligned}
\mathrm{Var}\left(\mathrm{Est}\left(\frac{\partial Ey}{\partial x_k}\right)\right) &= \mathrm{Cov}\left(\mathrm{Est}\left(\frac{\partial Ey}{\partial x_k}\right), \mathrm{Est}\left(\frac{\partial Ey}{\partial x_k}\right)\right) \\
&= \mathrm{Cov}\left(\frac{1}{N_{\mathrm{mc}}}\sum_{j=1}^{N_{\mathrm{mc}}}\frac{\partial y(w_j)}{\partial x_k}, \frac{1}{N_{\mathrm{mc}}}\sum_{j=1}^{N_{\mathrm{mc}}}\frac{\partial y(w_j)}{\partial x_k}\right) \\
&= \sum_{1\le t,l\le N_{\mathrm{mc}}}\frac{1}{N_{\mathrm{mc}}^2}\mathrm{Cov}\left(\frac{\partial y(w_j)}{\partial x_k}, \frac{\partial y(w_j)}{\partial x_k}\right) \\
&= \sum_{j=1}^{N_{\mathrm{mc}}}\frac{1}{N_{\mathrm{mc}}^2}\mathrm{Var}\left(\frac{\partial y(w_j)}{\partial x_k}\right) = \frac{1}{N_{\mathrm{mc}}}\mathrm{Var}\left(\frac{\partial y}{\partial x_k}\right)
\end{aligned}$$

where the covariance is linear and w_j are independent and identical distributed, that's why, it's sufficient to state that

$$\mathrm{Var}\left(\frac{\partial y}{\partial x_k}\right)$$

it's finite[1].

□

D.2 Proof of Proposition 3.2.1

Proposition 2 *For the estimate*

$$\mathrm{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right) = \frac{1}{N_{\mathrm{mc}}}\sum_{j=1}^{N_{\mathrm{mc}}}\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i - \hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k},$$

we have

$$E\left[\mathrm{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right)\right] = \frac{\partial MSE}{\partial x_k} \tag{D.3}$$

and

$$N_{\mathrm{mc}}\,\mathrm{Var}\left[\mathrm{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right)\right] \tag{D.4}$$

is bounded above by a number independent of N_{mc}. The variable x_k corresponds to a parameter used for the derivation of $MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Ey_i - \hat{C}_i\right)^2$.

[1] This condition is satisfied for all natural functions, having finite mean and variance.

Proof:

Following the same line of reasoning as the article [2], for the first statement, (D.3), we get the following computation:

$$\begin{aligned} E\left[\mathrm{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right)\right] &= E\left[\frac{1}{N_{\mathrm{mc}}}\sum_{j=1}^{N_{\mathrm{mc}}}\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i-\hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right] \\ &= \frac{1}{N_{\mathrm{mc}}}\sum_{j=1}^{N_{\mathrm{mc}}}E\left[\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i-\hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right] \\ &= E\left[\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i-\hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right]=\frac{\partial MSE}{\partial x_k}. \end{aligned} \tag{D.5}$$

The penultimate equality of the equation (D.5) is due to the fact that w_j are identicaly distributed. For the second statement of the proposition, (D.4), we'll rewrite $\mathrm{Var}\left(\mathrm{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right)\right)$ as follows:

$$\begin{aligned} \mathrm{Var}\left(\mathrm{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right)\right) &= \mathrm{Cov}\left(\mathrm{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right),\mathrm{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right)\right) \\ &= \mathrm{Cov}\left(\frac{1}{N_{\mathrm{mc}}}\sum_{j=1}^{N_{\mathrm{mc}}}\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i-\hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k},\frac{1}{N_{\mathrm{mc}}}\sum_{j=1}^{N_{\mathrm{mc}}}\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i-\hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right) \\ &= \sum_{1\le t,l\le N_{\mathrm{mc}}}\frac{1}{N_{\mathrm{mc}}^2}\mathrm{Cov}\left(\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i-\hat{C}_i\right)\frac{\partial y_i(w_t)}{\partial x_k},\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i-\hat{C}_i\right)\frac{\partial y_i(w_l)}{\partial x_k}\right) \\ &= \sum_{j=1}^{N_{\mathrm{mc}}}\frac{1}{N_{\mathrm{mc}}^2}\mathrm{Var}\left(\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i-\hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right)=\frac{1}{N_{\mathrm{mc}}}\mathrm{Var}\left(\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i-\hat{C}_i\right)\frac{\partial y_i}{\partial x_k}\right) \end{aligned}$$

where the covariance is linear and w_j are independent and identical distributed, that's why, it's sufficient to state that

$$\mathrm{Var}\left(\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i-\hat{C}_i\right)\frac{\partial y_i}{\partial x_k}\right)$$

it's finite. Since,

$$\begin{aligned} &\mathrm{Var}\left(\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i-\hat{C}_i\right)\frac{\partial y_i}{\partial x_k}\right) \\ &= \sum_{1\le t,l\le n}\frac{4}{n^2}\mathrm{Cov}\left(\left(Ey_t-\hat{C}_t\right)\frac{\partial y_t}{\partial x_k},\left(Ey_l-\hat{C}_l\right)\frac{\partial y_l}{\partial x_k}\right) \\ &= \sum_{1\le t,l\le n}\frac{4}{n^2}\left(Ey_t-\hat{C}_t\right)\left(Ey_l-\hat{C}_l\right)\mathrm{Cov}\left(\frac{\partial y_t}{\partial x_k},\frac{\partial y_l}{\partial x_k}\right). \end{aligned}$$

Where,

$$\left|\mathrm{Cov}\left(\frac{\partial y_t}{\partial x_k}, \frac{\partial y_l}{\partial x_k}\right)\right| = \sqrt{\mathrm{Var}\left(\frac{\partial y_t}{\partial x_k}\right)\mathrm{Var}\left(\frac{\partial y_l}{\partial x_k}\right)}$$

which is finite, since, according to the proof in the **Proposition C.0.1**, $\mathrm{Var}\left(\frac{\partial y}{\partial x_k}\right)$ is finite for $y = (S(T) - K)^+$.

□

D.3 Proof of Proposition 3.2.2

Proposition 3 *For the estimate*

$$\mathrm{Est}_2\left(\frac{\partial MSE}{\partial x_k}\right) = \frac{1}{N_{\mathrm{mc}} - 1}\sum_{j=2}^{N_{\mathrm{mc}}}\frac{2}{n}\sum_{i=1}^{n}\left(\left(S_i(j-1) - \hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right)$$

with

$$S_i(j-1) = \frac{1}{j-1}\sum_{l=1}^{j-1} y_i(w_l),$$

we have

$$E\left[\mathrm{Est}_2\left(\frac{\partial MSE}{\partial x_k}\right)\right] = \frac{\partial MSE}{\partial x_k} \tag{D.6}$$

and

$$N_{\mathrm{mc}}^{\frac{2}{3}}\,\mathrm{Var}\left[\mathrm{Est}_2\left(\frac{\partial MSE}{\partial x_k}\right)\right] \tag{D.7}$$

is bounded above by a number independent of N_{mc}. The parameter x_k corresponds to a parameter used for the derivation of $MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Ey_i - \hat{C}_i\right)^2$.

Proof:

Following the same line of reasoning as the article [2], for the first statement, (D.6), we get the following computation:

$$
\begin{aligned}
E\left[\mathrm{Est}_2\left(\frac{\partial MSE}{\partial x_k}\right)\right] &= E\left[\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=2}^{N_{\mathrm{mc}}}\frac{2}{n}\sum_{i=1}^{n}\left(S_i(j-1)-\hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right] \\
&= E\left[\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=2}^{N_{\mathrm{mc}}}\frac{2}{n}\sum_{i=1}^{n}\left(\frac{1}{j-1}\sum_{l=1}^{j-1}y_i(w_l)-\hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right] \\
&= \frac{1}{N_{\mathrm{mc}}-1}\sum_{j=2}^{N_{\mathrm{mc}}}E\left[\frac{2}{n}\sum_{i=1}^{n}\left(\frac{1}{j-1}\sum_{l=1}^{j-1}y_i(w_l)-\hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right] \\
&= E\left[\frac{2}{n}\sum_{i=1}^{n}\left(\frac{1}{j-1}\sum_{l=1}^{j-1}y_i(w_l)-\hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right], \qquad \text{(D.8)}
\end{aligned}
$$

where,

$$
E\left[S_i(j-1)\right] = E\left[y_i(w_l)\right] = \frac{1}{j-1}\sum_{l=1}^{j-1}E\left[y_i(w_l)\right] = E\left[y_i(w_l)\right] = Ey_i.
$$

Concluding that, the equation (D.8),

$$
E\left[\frac{2}{n}\sum_{i=1}^{n}\left(\frac{1}{j-1}\sum_{l=1}^{j-1}y_i(w_l)-\hat{C}_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right] = E\left[\frac{2}{n}\sum_{i=1}^{n}\left(Ey_i-\hat{C}_i\right)\frac{\partial y_i}{\partial x_k}\right] = \frac{\partial MSE}{\partial x_k}
$$

Now, for the second statement, with

$$
G(j,n) = \frac{2}{n}\sum_{i=1}^{n}\left(S_i(w_j)-C_i\right)
$$

we can rewrite (D.7) as,

$$
\begin{aligned}
&\mathrm{Var}\left(\mathrm{Est}_2\left(\frac{\partial MSE}{\partial x_k}\right)\right) = \mathrm{Var}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=2}^{N_{\mathrm{mc}}}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right) \\
&= \mathrm{Var}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=2}^{\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k} + \frac{1}{N_{\mathrm{mc}}-1}\sum_{j=\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor+1}^{N_{\mathrm{mc}}}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right),
\end{aligned}
$$

where, $\lfloor x \rfloor$, for any value x, returns the largest integer value less than or equal to x, continuing the previous development, we get,

$$\operatorname{Var}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=2}^{\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k}+\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor+1}^{N_{\mathrm{mc}}}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right)$$

$$=\operatorname{Var}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=2}^{\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right)+$$

$$+2\operatorname{Cov}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=2}^{\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k},\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor+1}^{N_{\mathrm{mc}}}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right)+$$

$$+\operatorname{Var}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor+1}^{N_{\mathrm{mc}}}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right).$$

Afterwards, it'll be shown that the 3 terms of the previous equality possess $\mathcal{O}\left(\frac{1}{N_{\mathrm{mc}}^{\frac{2}{3}}}\right)$ when $N_{\mathrm{mc}}\to\infty$. Starting with the first term,

$$\operatorname{Var}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=2}^{\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right)$$

$$\leq\frac{\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor^2}{(N_{\mathrm{mc}}-1)^2}\max_{2\leq t,l\leq\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor}\left(\operatorname{Cov}\left(G(t-1,n)\frac{\partial y_i(w_t)}{\partial x_k},G(l-1,n)\frac{\partial y_i(w_l)}{\partial x_k}\right)\right)$$

$$\leq\frac{\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor^2}{(N_{\mathrm{mc}}-1)^2}\max_{2\leq t,l\leq\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor}\left(\sqrt{\operatorname{Var}\left(G(t-1,n)\frac{\partial y_i(w_t)}{\partial x_k}\right)\operatorname{Var}\left(G(l-1,n)\frac{\partial y_i(w_l)}{\partial x_k}\right)}\right).$$

Given that, $\operatorname{Var}\left(S_i(j-1)\right)$, for whatever $j\in\{1,\,\ldots,\,N_{\mathrm{mc}}\}$, is finite since,

$$\operatorname{Var}\left(S_i(j-1)\right)=\operatorname{Cov}\left(S_i(j-1),S_i(j-1)\right)=\operatorname{Cov}\left(\frac{1}{j-1}\sum_{l=1}^{j-1}y_i(w_l),\frac{1}{j-1}\sum_{l=1}^{j-1}y_i(w_l)\right)$$

$$=\frac{1}{(j-1)^2}\sum_{1\leq t,k\leq j-1}\operatorname{Cov}\left(y_i(w_t),y_i(w_k)\right)=\frac{1}{(j-1)^2}\sum_{l=1}^{j-1}\operatorname{Var}\left(y_i(w_l)\right)=\frac{1}{j-1}\operatorname{Var}\left(y_i(w_1)\right).$$

Where, here, the covariance is linear for each argument and w_j are independent and identical distributed. Being sufficent to show that

$$\operatorname{Var}\left(S_i(w_1)\right)$$

is finite, and so,

$$\operatorname{Var}\left(G(1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right)=\operatorname{Var}\left(\frac{2}{n}\sum_{i=1}^{n}\left(\left(S_i(w_1)-\hat{C}_i\right)\frac{\partial y_i(w_2)}{\partial x_k}\right)\right)$$

$$=\frac{2}{n}\sum_{1\leq t,m\leq n}\operatorname{Cov}\left(\left(S_t(w_1)-\hat{C}_t\right)\frac{\partial y_t(w_2)}{\partial x_k},\left(S_m(w_1)-\hat{C}_m\right)\frac{\partial y_m(w_2)}{\partial x_k}\right),$$

where,

$$\left|\mathrm{Cov}\left(\left(S_t(w_1)-\hat{C}_t\right)\frac{\partial y_t(w_2)}{\partial x_k},\left(S_m(w_1)-\hat{C}_m\right)\frac{\partial y_m(w_2)}{\partial x_k}\right)\right| \\ \leq \sqrt{\mathrm{Var}\left(\left(S_t(w_1)-\hat{C}_t\right)\frac{\partial y_t(w_2)}{\partial x_k}\right)\cdot\mathrm{Var}\left(\left(S_m(w_1)-\hat{C}_m\right)\frac{\partial y_m(w_2)}{\partial x_k}\right)},$$

which is sufficent to claim that

$$\mathrm{Var}\left(\left(S_i(w_1)-\hat{C}_i\right)\frac{\partial y_i(w_2)}{\partial x_k}\right)$$

is finite for whatever $i\in\{1,\ \ldots,\ n\}$. Thus, the first term as $\mathcal{O}\left(\frac{1}{N_{\mathrm{mc}}^{\frac{2}{3}}}\right)$ for $N_{\mathrm{mc}}\to\infty$. The second term, is shown in a similar way,

$$2\,\mathrm{Cov}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=2}^{\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k},\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor+1}^{N_{\mathrm{mc}}}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right) \\ \leq \frac{2N_{\mathrm{mc}}\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor}{(N_{\mathrm{mc}}-1)^2}\max_{2\leq t\leq\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor<l\leq N_{\mathrm{mc}}}\left[\mathrm{Cov}\left(G(t-1,n)\frac{\partial y_i(w_j)}{\partial x_k},G(l-1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right)\right] \\ \leq \frac{2N_{\mathrm{mc}}\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor}{(N_{\mathrm{mc}}-1)^2}\max_{2\leq t\leq\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor<l\leq N_{\mathrm{mc}}}\left[\sqrt{\mathrm{Var}\left(G(t-1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right)}\cdot\sqrt{\mathrm{Var}\left(G(l-1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right)}\right]$$

where $\mathcal{O}\left(\frac{1}{N_{\mathrm{mc}}^{\frac{2}{3}}}\right)$ for $N_{\mathrm{mc}}\to\infty$. For the third and last term, for $N_{\mathrm{mc}}\to\infty$, we get

$$\mathrm{Var}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor+1}^{N_{\mathrm{mc}}}G(j-1,n)\frac{\partial y_i(w_j)}{\partial x_k}\right) \\ =\mathrm{Var}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor+1}^{N_{\mathrm{mc}}}\frac{2}{n}\sum_{i=1}^{n}\left(S_i(w_j)-C_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right).$$

Using,

$$Ey_i\sim\frac{1}{N_{\mathrm{mc}}}\sum_{j=1}^{N_{\mathrm{mc}}}y_i(x_j),$$

the third term of $\mathrm{Var}\left(\mathrm{Est}_2\left(\frac{\partial MSE}{\partial x_k}\right)\right)$ is estimated as follows,

$$
\begin{aligned}
&\operatorname{Var}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor+1}^{N_{\mathrm{mc}}}\sum_{i=1}^{m}\left(S_i(j-1)-C_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right)\\
&\quad\sim \operatorname{Var}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor+1}^{N_{\mathrm{mc}}}\sum_{i=1}^{m}\left(Ey_i+\mathcal{O}\left(\frac{1}{\sqrt{j-1}}\right)-C_i\right)\frac{\partial y_i(w_j)}{\partial x_k}\right)\\
&\quad\sim \operatorname{Var}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor+1}^{N_{\mathrm{mc}}}\left(\sum_{i=1}^{m}\left(Ey_i-C_i\right)\frac{\partial y_i(w_j)}{\partial x_k}+\mathcal{O}\left(\frac{1}{\sqrt{j-1}}\right)\right)\right)\\
&\quad\sim \operatorname{Var}\left(\frac{1}{N_{\mathrm{mc}}-1}\sum_{j=\lfloor\sqrt[3]{N_{\mathrm{mc}}}\rfloor+1}^{N_{\mathrm{mc}}}\sum_{i=1}^{m}\left(Ey_i-C_i\right)\frac{\partial y_i(w_j)}{\partial x_k}+\mathcal{O}\left(\frac{1}{\sqrt[3]{N_{\mathrm{mc}}}}\right)\right)\\
&\quad\sim \operatorname{Var}\left(\operatorname{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right)\right)+\mathcal{O}\left(\frac{1}{N_{\mathrm{mc}}^{\frac{2}{3}}}\right)
\end{aligned}
$$

and, as it was seen in **Propositon D.2**, $\operatorname{Var}\left(\operatorname{Est}_1\left(\frac{\partial MSE}{\partial x_k}\right)\right)$ possess $\mathcal{O}\left(\frac{1}{N_{\mathrm{mc}}}\right)$ for $N_{\mathrm{mc}}\to\infty$, this way, the third term of $\operatorname{Var}\left(\operatorname{Est}_2\left(\frac{\partial MSE}{\partial x_k}\right)\right)$ as $\mathcal{O}\left(\frac{1}{N_{\mathrm{mc}}^{\frac{2}{3}}}\right)$ when $N_{\mathrm{mc}}\to\infty$.

□

Appendix E

Implemented Codes

E.1 Code for Algorithm 3.2.1

```
1
2
3     //Import of all packages and header files for the program, as well
4     //as the header file "LBFGS.h" that contains the LBFGS method and the
5     //"Curves.h" file to create a Piecewise Curve for the volatilities.
6
7     #include <iostream>
8     #include <random>
9     #include <thread>
10    #include "examples.h"
11    #include "Curves.h"
12    #include <LBFGS.h>
13
14    //Libraries from MatLogica AADC library
15    #include <aadc/aadc_matrix.h>
16    #include <aadc/ibool.h>
17    #include <aadc/aadc.h>
18
19    ////////////////////////////////////////////////////////
20
21    using Eigen::VectorXd;
22    using namespace aadc;
23    using namespace LBFGSpp;
24
25    ////////////////////////////////////////////////////////
26    ////////////////////////////////////////////////////////
27
28    class Function_1P {
29
30      typedef double Time;
31      typedef __m256d mmType;
32
33      public:
34
35      //Creation of a stucture to store all variables for each European
36      //Option like, strike price, interest rate, asset price, observed
37      //price and expectation price.
38      template<class vtype>
```

```
39      struct EurOption{
40
41        EurOption(const Time maturity_, const vtype strike_
42          , const double vol_, const double rate_
43          , const int paths_) : maturity(maturity_), strike(strike_)
44          , init_vol(vol_), init_rate(rate_), paths(paths_) {}
45
46        int paths;
47        double grad, init_vol, init_rate, obs_price, total_price, variance;
48        ScalarVector payoffs = ScalarVector(paths, 0.);
49        vtype strike;
50        Time maturity;
51
52      };
53
54      ////////////////////////////////////////////////////////////
55
56      //Initialization of an empty vector of European Options.
57      std::vector<EurOption<idouble>> eos;
58
59      ////////////////////////////////////////////////////////////
60
61      //This function represents the objective funtion
62      //G = (1/m)* sum_{i=0}^{m} (Ey_i - C_i).
63      inline double loss() {
64
65        double sum(0.);
66
67        for (int i = 0; i < number_options; i++) {
68
69          sum += std::pow(eos[i].total_price - eos[i].obs_price, 2);
70
71        }
72
73        return sum/number_options;
74
75      }
76
77      ////////////////////////////////////////////////////////////
78
79      //Stores the interest rate for each European Option.
80      template<class vtype, class vType>
81      class ValuesObs{
82
83        public:
84
85        ValuesObs (const vType& _vals
86          , const ScalarVector& _observations) : vals(_vals)
87          , observations(_observations) {}
88
89        vtype value (const double& obs) {
90
91          int index = interpolatedIndex(obs);
92          return vals[index];
93
94        }
```

```
95
96          int interpolatedIndex (const double& obs) {
97
98            auto i_t = std::lower_bound(
99              observations.begin(), observations.end(), obs
100           );
101           int index_t = std::distance(observations.begin(), i_t);
102           if (index_t == observations.size()) index_t--;
103           return index_t;
104
105         }
106
107         public:
108
109         ScalarVector observations;
110         vType vals;
111
112       };
113
114       ////////////////////////////////////////////////////////
115
116       //Calculates the asset price for a specific period of time for
117       //a random Monte Carlo path.
118       template<class vtype>
119       vtype simulateAssetOneStep(
120         const vtype current_value
121         , Time current_t
122         , Time next_t
123         , ValuesObs<idouble,iVector>& rate_obj
124         , ValuesObs<idouble,iVector>& vol_obj
125         , const vtype& random_sample
126         , int& obs
127       ) {
128
129         //knot of interpolation of the Piecewise linear curve
130         //of volatilities
131         vtype vol = vol_obj.value(obs);
132         vtype rate = rate_obj.value(obs);
133         double dt = (next_t-current_t);
134         //Formula for the calculation of the asset at a certain period of time
135         vtype next_value = current_value * (
136           std::exp(
137             (-vol*vol / 2 + rate)*dt + vol * std::sqrt(dt) * random_sample
138           )
139         );
140
141         return next_value;
142
143       }
144
145       ////////////////////////////////////////////////////////
146
147       //Calculates the payoffs, (S(t_i) - K)^+, for all periods of time t_i.
148       void onePathPricing(
149         idouble asset
150         , ValuesObs<idouble,iVector>& rate_obj
151         , ValuesObs<idouble,iVector>& vol_obj
```

```
152         , const iVector& random_samples
153         , iVector& payoffs
154         , ScalarVector& maturities
155       ) {
156
157         int it_options = 0, t_i = 0;
158         while (it_options < number_options) {
159
160           t_i = 0;
161
162           while (t_i < time_list.size()-1) {
163
164             //Call options formula, (S(t_i) - K)^+ = (S(t_i) - K, 0)
165             asset = simulateAssetOneStep(
166               asset, time_list[t_i], time_list[t_i+1], rate_obj, vol_obj
167               , random_samples[t_i], it_options
168             );
169
170             if (eos[it_options].maturity == time_list[t_i+1]) {
171
172               payoffs[it_options] = std::max(
173                 asset - eos[it_options].strike, 0.
174               );
175               t_i = time_list.size();
176               it_options++;
177
178             } else t_i++;
179
180           }
181
182         }
183
184       }
185
186       ////////////////////////////////////////////////////////////
187
188       Function_1P () {
189
190         iVector aad_vol, aad_random_samples, aad_rate, aad_obs;
191         idouble aad_asset(init_asset);
192
193         aad_vol.resize(number_options), aad_rate.resize(number_options)
194         , aad_random_samples.resize(number_options+1);
195
196         for (int i = 0; i < number_options; i++) {
197
198           eos.push_back(
199             EurOption<idouble>(
200               maturities[i+1], strike_list[i], init_vol_list[i]
201               , init_rate_list[i], paths_per_thread
202             )
203           );
204
205         }
206
207         aad_funcs.startRecording();
208
```

```
209         // Mark vector of random variables as input only.
210         //No adjoints for them.
211         markVectorAsInput(random_arg, aad_random_samples, false);
212 
213         // Mark vector of random variables as input,
214         //but are adjoints for them.
215         markVectorAsInput(vol_arg, aad_vol, true);
216 
217         // Marks the value as imput but the derivative is not required.
218         asset_arg = aad_asset.markAsInput();
219 
220         // Marks the value as imput but the derivative is not required.
221         markVectorAsInput(rate_arg, aad_rate, true);
222 
223         ValuesObs<idouble,iVector> rate_obj(aad_rate, maturities);
224 
225         // Creates a Piecewise linear curve of volatilities.
226         ValuesObs<idouble,iVector> vol_obj(aad_vol, maturities);
227 
228         iVector payoffs(number_options);
229 
230         // Calls the funtion to calculate the payoffs for a
231         //Monte Carlo path only.
232         onePathPricing(
233           aad_asset, rate_obj, vol_obj, aad_random_samples, payoffs
234           , maturities
235         );
236 
237         // All variables are subjected to differentiation.
238         markVectorAsOutput(payoffs_args, payoffs);
239 
240         aad_funcs.stopRecording();
241 
242       }
243 
244       ////////////////////////////////////////////////////////////
245 
246       //Prints the results and the corresponding variables for all European
    Options.
247       void Print_Results(
248       const VectorXd& x_vol
249       , const VectorXd& x_rate
250       , bool result
251       , const int niter
252       ) {
253 
254         for (int i = 0; i < number_options; i++){
255 
256           if (result == true) {
257 
258             std::cout << "\n----------------------------------\nObs "
259               << eos[i].maturity <<"\n";
260 
261           } else {
262 
263             std::cout << "\n----------------------------------\nExp Obs "
264               << eos[i].maturity <<"\n";
```

```
265
266           }
267
268           std::cout << "-----------------------------------\n";
269           std::cout << "Strike = " << eos[i].strike << " / Matur = "
270             << eos[i].maturity << " / ";
271
272           //It prints the obseved prices if the "result" is true or prints
273           //the expectation prices if the "result" is false.
274           if (result == true) {
275
276             std::cout << "Obser Price = " << eos[i].obs_price << "\n";
277
278           } else {
279
280             std::cout << "Expect Price = " << eos[i].total_price << "\n";
281
282           }
283           std::cout << "Vol = " << x_vol[i]
284             << " / Rate = " << x_rate[i] << "\n";
285
286         }
287
288         //Prints the result of "G" in loss().
289         if (result == false) std::cout << "\nLoss  = " << loss() << "\n\n";
290
291       }
292
293       ////////////////////////////////////////////////////////
294
295       //Prints the results and the corresponding variables for
296       //all European Options.
297       void Print_Gradient() {
298
299         //Prints the gradient of "G", which is represented in loss().
300         std::cout << "-----------------------------------\nGradient ("
301           << word << ") = ( ";
302         for (int i = 0; i < number_options-1; i++) {
303
304           std::cout << eos[i].grad << ", ";
305
306         }
307         std::cout << eos[number_options-1].grad
308           << ")\n-----------------------------------\n\n";
309
310         //Prints the standard deviation and variance for each
311         //member of the gradient.
312         for (int i = 0; i < number_options; i++) {
313
314           std::cout << "Standard Deviation of the "
315             << eos[i].maturity <<" member of the gradient = "
316             << sqrt(eos[i].variance/num_mc_paths)
317             << "\nVariance of the "<< eos[i].maturity
318             << " member of the gradient = "
319             << eos[i].variance/num_mc_paths << "\n\n";
320
321         }
```

```
322
323         std::cout << "-----------------------------------\n";
324
325       }
326
327       ////////////////////////////////////////////////////////
328
329       //This function is to calcute the expectations or the gradient,
330       //depending on the variable "step", performing the Monte Carlo
331       //simulation using parallezation for a faster performance.
332
333       void ThreadFuction(
334         const VectorXd& x_vector
335         , const bool step
336         , ScalarMatrix& grad_vector
337         , ScalarVector& price
338         , ScalarVector& grad_total
339       ) {
340
341         //Initialization of certain variables as well as the
342         //Normal Distribution, N(0, 1).
343         ScalarVector prices(number_options, 0.)
344           , vol_vec(number_options, 0.)
345           , rate_vec(number_options, 0.)
346           , grad_vega(number_options, 0.)
347           , grad_rho(number_options, 0.);
348
349         ScalarMatrix grad_vector_vega(number_options, paths_per_thread, 0.)
350           , grad_vector_rho(number_options, paths_per_thread, 0.);
351
352         AVXVector<mmType> mm_price(number_options, mmSetConst<mmType>(0))
353           , mm_grad_vega(number_options, mmSetConst<mmType>(0))
354           , mm_grad_rho(number_options, mmSetConst<mmType>(0))
355           , randoms(number_options+1);
356
357         std::uniform_real_distribution<> normal_distrib(0.0, 1.0);
358         gen.seed(2*17+31);
359
360         if (word == "Vega") {
361
362           for (int i = 0; i < number_options; i++) vol_vec[i] = x_vector[i];
363           rate_vec = init_rate_list;
364
365         } else {
366
367           for (int i = 0; i < number_options; i++) rate_vec[i] = x_vector[i];
368           vol_vec = init_vol_list;
369
370         }
371
372         //This instruction creates the execution context, this is just a
373         //tape of mmType variables.
374         ws = aad_funcs.createWorkSpace();
375
376         for (int mc_i = 0; mc_i < paths_per_thread; ++mc_i) {
377
378           for (int i = 0; i < number_options+1; i++) {
```

```
379
380             for (int c = 0; c < AVXsize; c++) {
381
382               toDblPtr(randoms[i])[c] = normal_distrib(gen);
383
384             }
385
386           }
387
388           //All variables that are marked as Input, have to be inicialized
389           //first before calling the forward algorithm
390           setAVXVector(*ws, random_arg, randoms);
391           setScalarVectorToAVX(*ws, vol_arg, vol_vec);
392           setScalarVectorToAVX(*ws, rate_arg, rate_vec);
393           ws->setVal(asset_arg, mmSetConst<mmType>(init_asset));
394           //This instruction call the Forward algorithm with
395           //preallocated execution context ws.
396           (aad_funcs).forward(*ws);
397           //It resets all adjoints previouslly created.
398           ws->resetDiff();
399           //if "step" == true, then it will only calculate the
400           //observed prices or the expected prices.
401           if (step == true) {
402
403             for (int i = 0; i < number_options; i++) {
404
405               //ws->val(...) will contain the final value of the G.
406               mm_price[i] = aadc::mmAdd(
407                 mm_price[i], ws->val(payoffs_args[i])
408               );
409
410             }
411
412             //if "step" == false, then it will calculate the gradient of G
413           } else {
414
415             for (int i = 0; i < number_options; i++) {
416
417               //Since it is being implemented the adjoint differentiation,
418               //it is required to initialize the adjoint variables,
419               //in this case is, Ey_i - C_i.
420               ws->setDiff(
421                 payoffs_args[i], (
422                   (eos[i].total_price - eos[i].obs_price)*2
423                 )/number_options
424               );
425
426             }
427
428             //This execution calls the backward AD algorithm.
429             (aad_funcs).reverse(*ws);
430
431             for (int i = 0; i < number_options; i++) {
432
433               //The instruction ws->diff(...) will contain
434               //the derivative of the gradient of Vega.
435               mm_grad_vega[i] = aadc::mmAdd(
```

```
436                 ws->diff(vol_arg[i]), mm_grad_vega[i]
437               );
438               grad_vector_vega[i][mc_i] = (aadc::mmSum(
439                 ws->diff(vol_arg[i])
440               ))/(AVXsize);
441               //The instruction ws->diff(...) will contain
442               //the derivative of the gradient of Rho.
443               mm_grad_rho[i] = aadc::mmAdd(
444                 ws->diff(rate_arg[i]), mm_grad_rho[i]
445               );
446               grad_vector_rho[i][mc_i] = (aadc::mmSum(
447                 ws->diff(rate_arg[i])
448               ))/(AVXsize);
449
450             }
451
452           }
453
454         }
455
456         if (step == true) {
457
458           //Sums all the mmType variables.
459           for (int i = 0; i < number_options; i++){
460
461             price[i] = aadc::mmSum(mm_price[i])/num_mc_paths;
462
463           }
464
465         } else {
466
467           for (int i = 0; i < number_options; i++) {
468
469             if (word == "Vega") {
470
471               grad_total[i] = (aadc::mmSum(
472                 mm_grad_vega[i]
473               ))/(num_mc_paths);
474
475             } else {
476
477               grad_total[i] = (aadc::mmSum(
478                 mm_grad_rho[i]
479               ))/(num_mc_paths);
480
481           }
482
483           if (word == "Vega") grad_vector = grad_vector_vega;
484           else grad_vector = grad_vector_rho;
485
486         }
487
488       }
489
490       ////////////////////////////////////////////////////////////
491
492       //This function is used to calculate the observed prices
```

```
493       //since, this European Options are fictional.
494       void operator()(VectorXd& x, const std::string word_) {
495
496         word = word_;
497
498         ScalarVector price_obs(number_options, 0.)
499         , grad_total(number_options, 0.);
500
501         ScalarMatrix grad_vector(number_options, paths_per_thread, 0.);
502
503         //With "true", it will calculate only the observed prices.
504         ThreadFuction(x, true, grad_vector, price_obs, grad_total);
505
506         //It will store the results for the observed prices
507         //to each European Option.
508         for (int i = 0; i < number_options; i++) {
509
510           eos[i].obs_price = price_obs[i];
511
512         }
513
514       }
515
516       ////////////////////////////////////////////////////////
517
518       double operator()(const VectorXd& x, VectorXd& grad) {
519
520         ScalarVector exp_price(number_options, 0.)
521           , grad_total(number_options, 0.);
522         ScalarMatrix grad_vector(number_options, paths_per_thread, 0.);
523         //With "true", it will calculate only the expectations
524         //of the payoffs.
525         ThreadFuction(x, true, grad_vector, exp_price, grad_total);
526         //It will store the results for the expectations to
527         //each European Option.
528         for (int it_options = 0; it_options < number_options; it_options++) {
529
530           eos[it_options].total_price = exp_price[it_options];
531
532         }
533
534         //With "false", it will calculate only the gradient of G.
535         ThreadFuction(x, false, grad_vector, exp_price, grad_total);
536         //It will store the results and calculate the variance
537         //for each member of the gradients.
538         for (int i = 0; i < number_options; i++) {
539
540           eos[i].grad = grad_total[i];
541           grad[i] = grad_total[i];
542           eos[i].variance = 0.;
543
544           //This instruction calculates the variance of each member
545           //of the gradient.
546           for (int mc = 0; mc < grad_vector.cols(); mc++) {
547
548             eos[i].variance += std::pow(
549               grad_vector[i][mc] - eos[i].grad, 2
```

```
550             );
551
552           }
553
554           eos[i].variance /= num_mc_paths;
555
556         }
557
558         double _f = loss();
559
560         return _f;
561
562       }
563
564       private:
565
566       //This instance initializes the AADC library
567       aadc::AADCFunctions<mmType> aad_funcs;
568
569       std::shared_ptr<aadc::AADCWorkSpace<mmType> > ws;
570
571       aadc::VectorArg random_arg, vol_arg, rate_arg;
572
573       aadc::AADCArgument asset_arg;
574
575       aadc::VectorRes payoffs_args;
576
577       int num_mc_paths = 100000000 //Number of Monte-Carlo simulations
578         , AVXsize = aadc::mmSize<mmType>()
579         , iterations //Number of iterations used
580         , paths_per_thread = num_mc_paths / AVXsize
581         , number_options = 5;
582
583       double init_asset = 100.;
584       //Intial volatilities for each European option
585       ScalarVector init_vol_list = {0.005, 0.004, 0.0045, 0.003, 0.0035}
586         , init_rate_list = {0.1, 0.11, 0.09, 0.12, 0.13}
587         , maturities = {0, 1, 2, 3, 4, 5};
588
589       std::vector<Time> time_list = {0, 1, 2, 3, 4, 5};
590
591       std::mt19937_64 gen;
592
593       iVector strike_list = {95., 105., 115., 90., 100.};
594
595       std::string word;
596
597     };
598
599     ////////////////////////////////////////////////////////
600
601     void new_program_1()
602     {
603
604       ScalarVector init_vol_list = {0.005, 0.004, 0.0045, 0.003, 0.0035}
605         , init_rate_list = {0.1, 0.11, 0.09, 0.12, 0.13};
606
```

```
607         int number_options = 5;
608
609         //Gradiente with respect to volatility
610         VectorXd x_vol = VectorXd::Zero(number_options)
611           , x_rate = VectorXd::Zero(number_options)
612           , x = VectorXd::Zero(number_options)
613           , grad = VectorXd::Zero(number_options);
614
615         std::vector<std::string> greeks = {"Vega", "Rho"};
616
617         double f;
618
619         for (int it = 0; it < number_options; it++) {
620
621           x_vol[it] = init_vol_list[it];
622           x_rate[it] = init_rate_list[it];
623
624         }
625
626         Function_1P fun;
627
628         for (int i = 0; i < greeks.size(); i++){
629
630           std::cout << "\n->" << greeks[i] << ":\n";
631
632           if (greeks[i] == "Vega") x = x_vol;
633           else x = x_rate;
634
635           //Calculates the observed prices and prints
636           fun(x, greeks[i]);
637
638           fun.Print_Results(x_vol, x_rate, true, 0);
639
640           // Initial guess
641           if (greeks[i] == "Vega") {
642
643             for (int i = 0; i < number_options; i++) {
644
645               x[i] *= (1+0.2*(double(std::rand()) / RAND_MAX));
646
647             }
648
649           } else {
650
651             for (int i = 0; i < number_options; i++) {
652
653               x[i] *= (1+0.002*(double(std::rand()) / RAND_MAX));
654
655             }
656
657           }
658
659           //Calculates the Expectation prices and the first gradient
660           //of G then it prints.
661           auto base_start = std::chrono::high_resolution_clock::now();
662
663           std::cout << "\n----------------------------------\nMethod 1:";
```

```
664
665         f = fun(x, grad);
666
667         if (greeks[i] == "Vega") fun.Print_Results(x, x_rate, false, 0);
668         else fun.Print_Results(x_vol, x, false, 0);
669
670         fun.Print_Gradient();
671
672         auto base_stop = std::chrono::high_resolution_clock::now();
673
674         std::chrono::microseconds base_time =
675           std::chrono::duration_cast<std::chrono::microseconds>(
676             base_stop - base_start
677           );
678
679         std::cout << "Base time (double): = " << base_time.count()
680           << " microseconds\n\n";
681
682       }
683
684     }
685
```

E.2 Code for Algorithm 3.2.2

```
1
2
3     //Import of all packages and header files for the program, as well
4     //as the header file "LBFGS.h" that contains the LBFGS method and the
5     //"Curves.h" file to create a Piecewise Curve for the volatilities.
6
7     #include <iostream>
8     #include <random>
9     #include <thread>
10    #include "examples.h"
11    #include "Curves.h"
12    #include <LBFGS.h>
13
14    //Libraries from MatLogica AADC library
15    #include <aadc/aadc_matrix.h>
16    #include <aadc/ibool.h>
17    #include <aadc/aadc.h>
18
19    ////////////////////////////////////////////////////////////
20
21    using Eigen::VectorXd;
22    using namespace aadc;
23    using namespace LBFGSpp;
24
25    ////////////////////////////////////////////////////////////
26    ////////////////////////////////////////////////////////////
27
28    class Function_2P {
29
30      typedef double Time;
31
32      typedef __m256d mmType;
```

```
33
34      public:
35
36      //Creation of a stucture to store all variables for each European
37      //Option like, strike price, interest rate, asset price, observed
38      //price and expectation price.
39      template<class vtype>
40      struct EurOption{
41
42        EurOption(const Time maturity_, const vtype strike_
43          , const double vol_, const double rate_
44          , const int paths_) : maturity(maturity_), strike(strike_)
45          , init_vol(vol_) , init_rate(rate_), paths(paths_) {}
46
47        int paths;
48        double grad, init_vol, init_rate, obs_price, total_price, variance;
49        ScalarVector payoffs = ScalarVector(paths, 0.);
50        vtype strike;
51        Time maturity;
52
53      };
54
55      ////////////////////////////////////////////////////////
56
57      //Initialization of an empty vector of European Options.
58      std::vector<EurOption<idouble>> eos;
59
60      ////////////////////////////////////////////////////////
61
62      //This function represents the objective funtion
63      //G = (1/m)* sum_{i=0}^{m} (Ey_i - C_i).
64      inline double loss() {
65
66        double sum(0.);
67
68        for (int i = 0; i < number_options; i++) {
69
70          sum += std::pow(eos[i].total_price - eos[i].obs_price, 2);
71
72        }
73
74        return sum/number_options;
75
76      }
77
78      ////////////////////////////////////////////////////////
79
80      //Stores the interest rate for each European Option.
81      template<class vtype, class vType>
82      class ValuesObs{
83
84        public:
85
86        ValuesObs (const vType& _vals
87          , const ScalarVector& _observations) : vals(_vals)
88          , observations(_observations) {}
```

```
89
90          vtype value (const double& obs) {
91
92            int index = interpolatedIndex(obs);
93            return vals[index];
94
95          }
96
97          int interpolatedIndex (const double& obs) {
98
99            auto i_t = std::lower_bound(
100             observations.begin(), observations.end(), obs
101           );
102           int index_t = std::distance(observations.begin(), i_t);
103           if (index_t == observations.size()) index_t--;
104           return index_t;
105
106         }
107
108         public:
109
110         ScalarVector observations;
111         vType vals;
112
113       };
114
115       ////////////////////////////////////////////////////////
116
117       //Calculates the asset price for a specific period of time for
118       //a random Monte Carlo path.
119       template<class vtype>
120       vtype simulateAssetOneStep(
121         const vtype current_value
122         , Time current_t
123         , Time next_t
124         , ValuesObs<idouble,iVector>& rate_obj
125         , ValuesObs<idouble,iVector>& vol_obj
126         , const vtype& random_sample
127         , int& obs
128       ) {
129
130         //knot of interpolation of the Piecewise constant curve
131         //of volatilities
132         vtype vol = vol_obj.value(obs);
133         vtype rate = rate_obj.value(obs);
134         double dt = (next_t-current_t);
135         //Formula for the calculation of the asset at a certain period of time
136         vtype next_value = current_value * (
137           std::exp(
138             (-vol*vol / 2 + rate)*dt + vol * std::sqrt(dt) * random_sample
139           )
140         );
141
142         return next_value;
143
144       }
145
```

```
146       ////////////////////////////////////////////////////////////
147
148       //Calculates the payoffs, (S(t_i) - K)^+, for all periods of time t_i.
149       void onePathPricing(
150         idouble asset
151         , ValuesObs<idouble,iVector>& rate_obj
152         , ValuesObs<idouble,iVector>& vol_obj
153         , const iVector& random_samples
154         , iVector& payoffs
155         , ScalarVector& maturities
156       ) {
157
158         int it_options = 0, t_i = 0;
159         while (it_options < number_options) {
160
161           t_i = 0;
162
163           while (t_i < time_list.size()-1) {
164
165             //Call options formula, (S(t_i) - K)^+ = (S(t_i) - K, 0)
166             asset = simulateAssetOneStep(
167               asset, time_list[t_i], time_list[t_i+1], rate_obj, vol_obj
168               , random_samples[t_i], it_options
169             );
170
171             if (eos[it_options].maturity == time_list[t_i+1]) {
172
173               payoffs[it_options] = std::max(
174                 asset - eos[it_options].strike, 0.
175               );
176               t_i = time_list.size();
177               it_options++;
178
179             } else t_i++;
180
181           }
182
183         }
184
185       }
186
187       ////////////////////////////////////////////////////////////
188
189       Function_2P () {
190
191         iVector aad_vol, aad_random_samples, aad_rate, aad_obs;
192         idouble aad_asset(init_asset);
193
194         for (int i = 0; i < number_options; i++) {
195
196           eos.push_back(
197             EurOption<idouble>(
198               maturities[i+1], strike_list[i], init_vol_list[i]
199               , init_rate_list[i], paths_per_thread
200             )
201           );
202
```

```
203         }
204
205         aad_vol.resize(number_options), aad_rate.resize(number_options)
206         , aad_random_samples.resize(number_options+1);
207
208         aad_funcs.startRecording();
209         // Mark vector of random variables as input only.
210         //No adjoints for them.
211         markVectorAsInput(random_arg, aad_random_samples, false);
212
213         // Mark vector of random variables as input,
214         //but are adjoints for them.
215         markVectorAsInput(vol_arg, aad_vol, true);
216
217         // Marks the value as imput but the derivative is not required.
218         asset_arg = aad_asset.markAsInput();
219
220         // Marks the value as imput but the derivative is not required.
221         markVectorAsInput(rate_arg, aad_rate, true);
222
223         ValuesObs<idouble,iVector> rate_obj(aad_rate, maturities);
224
225         // Creates a Piecewise linear curve of volatilities.
226         ValuesObs<idouble,iVector> vol_obj(aad_vol, maturities);
227
228         iVector payoffs(number_options);
229
230         // Calls the funtion to calculate the payoffs for a
231         //Monte Carlo path only.
232         onePathPricing(
233           aad_asset, rate_obj, vol_obj, aad_random_samples, payoffs
234           , maturities
235         );
236
237         // All variables are subjected to differentiation.
238         markVectorAsOutput(payoffs_args, payoffs);
239
240         aad_funcs.stopRecording();
241
242       }
243
244       ////////////////////////////////////////////////////////////
245
246       //Prints the results and the corresponding variables for
247       //all European Options.
248       void Print_Results(
249         const VectorXd& x_vol
250         , const VectorXd& x_rate
251         , bool result
252         , const int niter
253       ) {
254
255         for (int i = 0; i < number_options; i++){
256
257           if (result == true) {
258
259             std::cout << "\n----------------------------------\nObs "
```

```
260             << maturities[i] <<"\n";
261
262           } else {
263
264             std::cout << "\n-----------------------------------\nExp Obs "
265             << maturities[i] <<"\n";
266
267           }
268           std::cout << "-----------------------------------\n";
269           std::cout << "Strike = " << eos[i].strike << " / Matur = "
270             << eos[i].maturity << " / ";
271           //It prints the obseved prices if the "result" is true or prints
272           //the expectation prices if the "result" is false.
273           if (result == true) {
274
275             std::cout << "Obser Price = " << eos[i].obs_price << "\n";
276
277           } else {
278
279             std::cout << "Expect Price = " << eos[i].total_price << "\n";
280
281           }
282           std::cout << "Vol = " << x_vol[i]
283             << " / Rate = " << x_rate[i] << "\n";
284
285         }
286
287         //Prints the result of "G" in loss().
288         if (result == false) std::cout << "\nLoss  = " << loss() << "\n\n";
289
290       }
291
292       ////////////////////////////////////////////////////////////
293
294       //Prints the results and the corresponding variables for
295       //all European Options.
296       void Print_Gradient() {
297
298         //Prints the gradient of "G", which is represented in loss().
299         std::cout << "-----------------------------------\nGradient ("
300           << word << ") = ( ";
301         for (int i = 0; i < number_options-1; i++) {
302
303           std::cout << eos[i].grad << ", ";
304
305         }
306         std::cout << eos[number_options-1].grad
307           << ")\n-----------------------------------\n\n";
308         //Prints the standard deviation and variance for each
309         //member of the gradient.
310         for (int i = 0; i < number_options; i++) {
311
312           std::cout << "Standard Deviation of the "
313             << eos[i].maturity <<" member of the gradient = "
314             << sqrt(eos[i].variance/num_mc_paths)
315             << "\nVariance of the "<< eos[i].maturity
316             << " member of the gradient = "
```

```
317             << eos[i].variance/num_mc_paths << "\n\n";
318
319         }
320         std::cout << "-----------------------------------\n";
321
322       }
323
324       ////////////////////////////////////////////////////////////
325
326       //It calculates the average value.
327
328       inline double Medium(double& value, double& size){
329
330         return (value/size);
331
332       }
333
334       ////////////////////////////////////////////////////////////
335
336       //It calculates the adjoint value that it's going to
337       //be used for reverse algorithm.
338
339       inline double Adjoint(
340         const int it_options
341         , ScalarVector& size_adjoint
342         , ScalarVector& adjoint_prod
343         , double& adjoint
344       ) {
345
346         size_adjoint[it_options] += 1;
347         adjoint_prod[it_options] += adjoint;
348         return Medium(
349           adjoint_prod[it_options], size_adjoint[it_options]
350         );
351
352       }
353
354       ////////////////////////////////////////////////////////////
355
356       //This function is to calcute the expectations or the gradient,
357       //depending on the variable "step", performing the Monte Carlo
358       //simulation using parallezation for a faster performance.
359
360       void ThreadFuction(
361         const VectorXd& x_vector
362         , const bool step
363         , ScalarMatrix& grad_vector
364         , ScalarVector& price
365         , ScalarVector& grad_total
366       ) {
367
368         //Initialization of certain variables as well as the
369         //Normal Distribution, N(0, 1).
370         ScalarVector prices(number_options, 0.)
371           , vol_vec(number_options, 0.)
372           , rate_vec(number_options, 0.)
373           , grad_vega(number_options, 0.)
```

```
    , grad_rho(number_options, 0.)
    , adjoint_prod(number_options, 0.)
    , size_adjoint(number_options, 0.);

ScalarMatrix grad_vector_vega(number_options, paths_per_thread, 0.)
    , grad_vector_rho(number_options, paths_per_thread, 0.);

AVXVector<mmType> mm_price(number_options, mmSetConst<mmType>(0))
    , mm_grad_vega(number_options, mmSetConst<mmType>(0))
    , mm_grad_rho(number_options, mmSetConst<mmType>(0))
    , randoms(number_options+1);

std::uniform_real_distribution<> normal_distrib(0.0, 1.0);
gen.seed(2*17+31);

if (word == "Vega") {

    for (int i = 0; i < number_options; i++) vol_vec[i] = x_vector[i];
    rate_vec = init_rate_list;

} else {

    for (int i = 0; i < number_options; i++) rate_vec[i] = x_vector[i];
    vol_vec = init_vol_list;

}

//This instruction creates the execution context, this is just a
//tape of mmType variables.
ws = aad_funcs.createWorkSpace();

for (int mc_i = 0; mc_i < paths_per_thread; ++mc_i) {

    for (int i = 0; i < number_options+1; i++) {

        for (int c = 0; c < AVXsize; c++) {

            toDblPtr(randoms[i])[c] = normal_distrib(gen);

        }

    }

    //All variables that are marked as Input, have to be
    //inicialized first before calling the forward algorithm
    setAVXVector(*ws, random_arg, randoms);
    setScalarVectorToAVX(*ws, vol_arg, vol_vec);
    setScalarVectorToAVX(*ws, rate_arg, rate_vec);
    ws->setVal(asset_arg, mmSetConst<mmType>(init_asset));

    //This instruction call the Forward algorithm with
    //preallocated execution context ws.
    (aad_funcs).forward(*ws);
    //It resets all adjoints previouslly created.
    ws->resetDiff();

    //if "step" == true, then it will only calculate the
```

```
431           //observed prices or the expected prices.
432           if (step == true) {
433
434             for (int i = 0; i < number_options; i++) {
435
436               //ws->val(...) will contain the final value of the G.
437               mm_price[i] = aadc::mmAdd(
438                 mm_price[i], ws->val(payoffs_args[i])
439               );
440
441             }
442
443             //if "step" == false, then it will calculate the gradient of G
444           } else {
445
446             for (int i = 0; i < number_options; i++){
447
448               //ws->val(...) will contain the final value of the G.
449               mm_price[i] = aadc::mmAdd(
450                 mm_price[i], ws->val(payoffs_args[i])
451               );
452
453               eos[i].payoffs[mc_i] = 0.;
454               //it represents payoffs for each Monte_carlo simulation
455               eos[i].payoffs[mc_i] = aadc::mmSum(
456                 ws->val(payoffs_args[i])
457               )/(AVXsize);
458
459             }
460
461             if (mc_i > 0) {
462
463               for (int i = 0; i < number_options; i++){
464
465                 double value = eos[i].payoffs[mc_i-1];
466                 double adjoint = Adjoint(
467                   i, size_adjoint, adjoint_prod, valu
468                 );
469                 ws->setDiff(
470                   payoffs_args[i], (
471                     (adjoint-eos[i].obs_price)*2
472                   )/number_options
473                 );
474
475               }
476
477               //This execution calls the backward AD algorithm.
478               (aad_funcs).reverse(*ws);
479
480               for (int i = 0; i < number_options; i++) {
481
482                 //The instruction ws->diff(...) will contain
483                 //the derivative of the gradient of Vega.
484                 mm_grad_vega[i] = aadc::mmAdd(
485                   ws->diff(vol_arg[i]), mm_grad_vega[i]
486                 );
487                 grad_vector_vega[i][mc_i] = (aadc::mmSum(
```

```
488                   ws->diff(vol_arg[i])
489                 ))/(AVXsize);
490
491                 //The instruction ws->diff(...) will contain
492                 //the derivative of the gradient of Rho.
493                 mm_grad_rho[i] = aadc::mmAdd(
494                   ws->diff(rate_arg[i]), mm_grad_rho[i]
495                 );
496                 grad_vector_rho[i][mc_i] = (aadc::mmSum(
497                   ws->diff(rate_arg[i])
498                 ))/(AVXsize);
499
500               }
501
502             }
503
504           }
505
506         }
507
508         if (step == true) {
509
510           //Sums all the mmType variables.
511           for (int i = 0; i < number_options; i++) {
512
513             price[i] = aadc::mmSum(mm_price[i])/num_mc_paths;
514
515           }
516
517         } else {
518
519           for (int i = 0; i < number_options; i++) {
520
521             price[i] = aadc::mmSum(mm_price[i])/num_mc_paths;
522             if (word == "Vega") {
523
524               grad_total[i] = (aadc::mmSum(
525                 mm_grad_vega[i]
526               ))/((num_mc_paths-1));
527
528             } else {
529
530               grad_total[i] = (aadc::mmSum(
531                 mm_grad_rho[i]
532               ))/((num_mc_paths-1));
533
534             }
535
536           }
537
538           if (word == "Vega") grad_vector = grad_vector_vega;
539           else grad_vector = grad_vector_rho;
540
541         }
542
543       }
544
```

```
545       ////////////////////////////////////////////////////////
546
547       //This function is used to calculate the observed prices
548       //since, this European Options are fictional.
549       void operator()(const VectorXd& x, const std::string word_) {
550
551         word = word_;
552
553         ScalarVector price_obs(number_options, 0.)
554         , grad_total(number_options, 0.);
555
556         ScalarMatrix grad_vector(number_options, paths_per_thread, 0.);
557
558         //With "true", it will calculate only the observed prices.
559         ThreadFuction(x, true, grad_vector, price_obs, grad_total);
560
561         //It will store the results for the observed prices
562         //to each European Option.
563         for (int i = 0; i < number_options; i++) {
564
565           eos[i].obs_price = price_obs[i];
566
567         }
568
569       }
570
571       ////////////////////////////////////////////////////////
572
573       double operator()(const VectorXd& x, VectorXd& grad) {
574
575         ScalarVector exp_price(number_options, 0.)
576         , grad_total(number_options, 0.);
577
578         ScalarMatrix grad_vector(number_options, paths_per_thread-1, 0.);
579
580         //With "false", it will calculate only the gradient of G.
581         ThreadFuction(x, false, grad_vector, exp_price, grad_total);
582
583         //It will store the results and calculate the variance for
584         //each member of the gradients.
585         for (int i = 0; i < number_options; i++) {
586
587           eos[i].total_price = exp_price[i];
588           eos[i].grad = grad_total[i];
589           grad[i] = grad_total[i];
590           eos[i].variance = 0.;
591
592           //This instruction calculates the variance of each member
593           //of the gradient.
594           for (int mc = 0; mc < grad_vector.cols(); mc++) {
595
596             eos[i].variance += std::pow(
597               grad_vector[i][mc] - eos[i].grad, 2
598             );
599
600           }
601
```

```
602           eos[i].variance /= num_mc_paths;
603
604         }
605
606         double _f = loss();
607
608         return _f;
609
610       }
611
612       private:
613
614       //This instance initializes the AADC library
615       aadc::AADCFunctions<mmType> aad_funcs;
616
617       std::shared_ptr<aadc::AADCWorkSpace<mmType> > ws;
618
619       aadc::VectorArg random_arg, vol_arg, rate_arg;
620
621       aadc::AADCArgument asset_arg;
622
623       aadc::VectorRes payoffs_args;
624
625       int num_mc_paths = 100000000 //Number of Monte-Carlo simulations
626         , AVXsize = aadc::mmSize<mmType>()
627         , iterations //Number of iterations used
628         , paths_per_thread = num_mc_paths / AVXsize
629         , number_options = 5;
630
631       double init_asset = 100.;
632
633       //Intial volatilities for each European option
634       ScalarVector init_vol_list = {0.005, 0.004, 0.0045, 0.003, 0.0035}
635         , init_rate_list = {0.1, 0.11, 0.09, 0.12, 0.13}
636         , maturities = {0, 1, 2, 3, 4, 5};
637
638       std::vector<Time> time_list = {0, 1, 2, 3, 4, 5};
639
640       std::mt19937_64 gen;
641
642       iVector strike_list = {95., 105., 115., 90., 100.};
643
644       std::string word;
645
646     };
647
648     ////////////////////////////////////////////////////////
649
650     void new_program_2()
651     {
652
653       //Volatilities for the observed prices.
654       ScalarVector init_vol_list = {0.005, 0.004, 0.0045, 0.003, 0.0035}
655         , init_rate_list = {0.1, 0.11, 0.09, 0.12, 0.13};
656
657       int number_options = 5;
658
```

```
659         //Gradiente with respect to volatility
660         VectorXd x_vol = VectorXd::Zero(number_options)
661         , x_rate = VectorXd::Zero(number_options)
662         , x = VectorXd::Zero(number_options)
663         , grad = VectorXd::Zero(number_options);
664
665         std::vector<std::string> greeks = {"Vega", "Rho"};
666
667         double f;
668
669         for (int it = 0; it < number_options; it++) {
670
671           x_vol[it] = init_vol_list[it];
672           x_rate[it] = init_rate_list[it];
673
674         }
675
676         Function_2P fun;
677
678         for (int i = 0; i < greeks.size(); i++){
679
680           std::cout << "\n->" << greeks[i] << ":\n";
681
682           if (greeks[i] == "Vega") x = x_vol;
683           else x = x_rate;
684
685           //Calculates the observed prices and prints
686           fun(x, greeks[i]);
687           fun.Print_Results(x_vol, x_rate, true, 0);
688
689           // Initial guess
690           if (greeks[i] == "Vega") {
691
692             for (int i = 0; i < number_options; i++) {
693
694               x[i] *= (1+0.2*(double(std::rand()) / RAND_MAX));
695
696             }
697
698           } else {
699
700             for (int i = 0; i < number_options; i++) {
701
702               x[i] *= (1+0.002*(double(std::rand()) / RAND_MAX));
703
704             }
705
706           }
707           //Calculates the Expectation prices and the first gradient
708           //of G then it prints.
709           auto base_start = std::chrono::high_resolution_clock::now();
710
711           std::cout << "\n----------------------------------\nMethod 1:";
712           f = fun(x, grad);
713           if (greeks[i] == "Vega") fun.Print_Results(x, x_rate, false, 0);
714           else fun.Print_Results(x_vol, x, false, 0);
715           fun.Print_Gradient();
```

```
716
717         auto base_stop = std::chrono::high_resolution_clock::now();
718         std::chrono::microseconds base_time =
719           std::chrono::duration_cast<std::chrono::microseconds>(
720             base_stop - base_start
721           );
722         std::cout << "Base time (double): = " << base_time.count()
723           << " microseconds\n\n";
724
725       }
726
727     }
728
```

Bibliography

[1] Dmitri Goloubentsev and Evgeny Lakshtanov. Remarks on stochastic automatic adjoint differentiation and financial models calibration. https://arxiv.org/abs/1901.04200, 2019.

[2] José Brito, Andrei Goloubentsev, and Evgeny Goncharov. Automatic adjoint differentiation for special functions involving expectations. https://arxiv.org/abs/2204.05204, 2022.

[3] R. Boudjemaa, M. Cox, A. Forbes, and P. Harris. Automatic differentiation techniques and their application in metrology. *Report to the National Measurement Directorate, Department of Trade and Industry*, 2003.

[4] C. Käbe, J. H. Maruhn, and E. W Sachs. Adjoint-based monte carlo calibration of financial market models. *Finance and Stochastics*, 13(3):351–379, 2009.

[5] A. Griewank and A. Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS) 26*, 1:19–45, 2000.

[6] Autodiff Community. http://www.autodiff.org/?module=Applications, 2021.

[7] Cristian Homescu. Adjoints and automatic (algorithmic) differentiation in computational finance, 2011.

[8] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1):171–190, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[9] Christian P. Fries. Stochastic automatic differentiation: automatic differentiation for monte-carlo simulations. *Quantitative Finance*, 19(6):1043–1059, 2019.

[10] Paul Glasserman. In *Monte Carlo Methods in Financial Engineering*, pages 3–7, 34–37. Springer.

[11] C. C. Margossian. A review of automatic differentiation and its efficient implementation. *Wiley interdisciplinary reviews: data mining and knowledge discovery 9*, 4, 2019. e1305.

[12] European Option. https://www.investopedia.com/terms/e/europeanoption.asp, 2021.

[13] Karl Sigman. Introduction to reducing variance in Monte Carlo simulations. 2007.

[14] Maidanov S. A. Monte carlo european options pricing implementation using various industry library solutions. newyork city, usa: Intel. 10, 2010. e0217730.

[15] Shariq Mohammed and Samar Singh. Pricing options using monte carlo methods. 2014.

[16] *MatLogica's AADC library*, available to try online by clicking "code examples" at their *https://www.matlogica.com*.