Efficient Gradient Computation of Exotic Options using Automatic Differentiation

by

Ka Hei Kathleen Wong

A research paper
presented to the University of Waterloo
in partial fulfillment of the
requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Supervisor: Prof. Thomas F. Coleman

Waterloo, Ontario, Canada, 2017

I hereby declare that I am the sole author of this report. This is a true copy of the report, including any required final revisions, as accepted by my examiners.

I understand that my report may be made electronically available to the public.

Abstract

The focus of this paper is on efficient gradient computations for the following exotic options, namely basket options, Asian options, and best of Asian options. Gradients are essential for portfolio hedging and risk minimization. Instead of using the traditional methods, a handy tool will be employed to compute the derivatives in this paper. Automatic differentiation (AD) is a handy tool in comparison to other traditional methods for derivative computations. It is potentially faster and more precise. Despite the promising advantages of AD, time and space efficiency are the challenges when implementing AD.

There are two objectives in this paper: space efficiency and time efficiency. The first objective is to introduce one of the two types of structure that would significantly reduce the space required for gradient computations. The second objective is to introduce a new version of AD that would reduce the time for derivative computations.

Three experiments and comparison tests were performed. The results show that the memory required for gradient computations was significantly reduced after exploiting the structure. Furthermore, the performance of structured AD is independent of the implementation of the underlying user code. Lastly, we illustrate a new version of AD that is more time efficient than the previous version of AD for gradient computations.

Acknowledgements

I would like to express my sincere thanks to my supervisor, Prof. Thomas F. Coleman, for his patient guidance and generous support during my time in University of Waterloo. I would also like to thank all of the staffs, my professors, and friends in University of Waterloo for the wonderful experience.

Dedication

This is dedicated to my family.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Importance and Applications of Partial Derivatives

Partial derivatives are essential for many multidimensional problems. Many real life problems involve multiple variables as their domains. These practical problems include edge detection in image processing, optimization, fluid viscosity in engineering, and marginal analysis in economics. The focus of this paper is on option pricing in finance.

In finance, an option is a contract which gives the buyer the right, but the not obligation, to buy or sell an underlying asset or instrument at a specific strike price on a specified date, depending on the form of the option [2]. In general, the gradient is the vector of the first partial derivatives of a multidimensional function while the Hessian is defined as the matrix of the second partial derivatives of the function. In the finance industry, gradients and Hessian are referred to as Delta and Gamma, respectively. These two "Greeks" are used to measure sensitivity to parameters such as initial stock price, volatility and risk-free interest rate [3]. They are very important for portfolio hedging and risk minimization. Therefore, it is essential to find an effective method to compute these "Greeks". The focus of this paper is on computing the gradients for the following exotic options, namely basket options, Asian options, and best of Asian options in an efficient manner.

1.2 Challenges and Advantages of Automatic Differentiation

As mentioned in the previous section, partial derivatives are widely used in various applications. Some traditional methods to compute derivatives have been developed. The most common ones are finite differencing, hand coding, and symbolic differentiation. However, there are some disadvantages for each of the methods. While finite differencing is easy to use, it is difficult to choose a good differencing parameter which is crucial for the accuracy of the derivatives [6]. Hand coding is prone to typo errors. It can also be very expensive. Moreover, even it is easy to hand code derivative functions for simple European options, it is actually quite challenging for more complicated exotic options like the ones examined in this paper. Symbolic differentiation appears to be too expensive for "large scale" problems [4].

In this section, the advantages of automatic differentiation will be discussed. Automatic differentiation (AD) is a technology for obtaining derivatives of codes used in scientific computing. AD is a handy tool in comparison to other traditional methods of computing derivatives for the following reasons: significantly faster than finite differencing, easier to achieve an accurate derivative than finite differencing, and more convenient than hand coding [4].

Despite the advantages mentioned above, time and space are challenges when implementing AD. There are two basic modes of AD: forward mode and reverse mode. Both modes of AD use the idea of chain rule. While the forward mode AD executes chain rule straightforwardly, reverse mode first computes and stores all intermediate values on a "tape", then roll back the "tape" from the end to the beginning to obtain the derivatives. While the forward mode seems straightforward to use for "small-scale" problems, it gets more complex for "large-scale" problems and requires more time [4,10]. Since the method for option pricing in this paper is Monte Carlo, which is a "large-scale" problem, the reverse mode is attractive for computing the gradients of our exotic options. One drawback of the reverse mode is that it requires lots of "tape" memory for storing the intermediate operations. The first objective of this paper is to introduce two types of structure that can significantly reduce the space required for gradient computations. The second objective of this paper is to introduce a new version of ADMAT that reduces the time for derivative computations.

1.3 Project Objectives

The main focus of this paper is on efficiently computing the gradients for the following exotic options, namely basket options, Asian options, and best of Asian options. There are two objectives in this paper: space efficiency and time efficiency in computing first derivatives by AD. In order to achieve the first objective using reverse mode AD, this paper introduces two types of structure that can significantly reduce the space required for the gradient computations compared to non-structured problems. There are two common special cases of the general AD structure, namely composite functions (dynamic structure) and generalized partially separable (GPS) structure. In general, the dynamic structure (DS) vertically cuts paths into segments, usually in the unit of timesteps for option pricing, whereas the GPS function horizontally divides simulations into smaller clusters of stochastic paths.

The second objective is achieved by applying the reverse-mode AD with the AD structures. Suppose $f : R^n \to R^1$, f is differentiable. The number of operations (a measure of time) to evaluate a function f at an arbitrary point, given a code for f, is denoted as $\omega(f)$. The gradient computation using reverse-mode AD, at iterate x, requires time proportional to $\omega(f)$. In other words, the time required for gradient computation is proportional to the time required for evaluating the function itself. But both forward-mode AD and finite differencing require time proportional to $n \cdot \omega(f)$. However, the order of work for reverse-mode AD is bounded by $\omega(f)$ only before fast memory is exhausted. The gradient computation will slow down significantly after running out of fast memory [2]. Thus, the space-saving AD structures introduced in the last paragraph are crucial.

In our experiments, we use two packages for differentiating MATLAB functions: ADMAT 2.0 [4] and MADO [7].

As mentioned, Monte Carlo is useful for option pricing in financial applications. Monte Carlo method predicts option values by generating a large number of random paths and taking the average of a large number of terminal values. The more random simulation paths, the more reliable the predicted option value would be. Since the Monte Carlo formula contains the random Brownian term, the option value produced by Monte Carlo is unbiased. This method is widely used in the finance industry because of its simplicity. Better yet, Monte Carlo is a special case of structured AD, i.e. the Jacobian of Monte Carlo exhibits the structure that is required for implementing the structured AD techniques. Therefore, the objective of this project can be achieved by implementing the reverse mode AD on the special case of structured problem (i.e. Monte Carlo in this project). The computational space usage was significantly improved using structured AD compared to the non-structured AD (i.e. plain reverse

AD).

1.4 Paper Overview

The main focus of this paper is on efficiently computing the gradients for basket options, Asian options, and best of Asian options. The GPS structured AD, Monte Carlo, and MADO are used to achieve space and time saving. In Chapter 2, the two types of structured AD, namely dynamic structure (DS) and generalized partially separable structure (GPS), are introduced. The algorithms for gradient computation for the generalized and special case structure are explained. In Chapter 3, we illustrate why the Monte Carlo method can be expressed in the GPS structured problem step-by-step. Moreover, the calculation of option values and gradient computations for each type of the objective exotic options, as well as the algorithms for structured AD, will be discussed in detail. In Chapter 4, the results of memory usage for each type of the options using structured vs. non-structured AD will be shown and compared. The results of time ratio produced by ADMAT 2.0 and MADO will be reported and compared as well. Lastly, we report our conclusions in Chapter 5.

Chapter 2

Types of Structured Automatic Differentiation

2.1 General Automatic Differentiation Structure

As mentioned in the previous chapter, the efficiency of automatic differentiation can be significantly improved by using problem structure. In this section, we describe a general structure and then specialize to our exotic option case [4].

Before discussing the mathematical functions, here is some background of automatic differentiation (AD). In AD, it is assumed that all mathematical functions are defined by a finite set of elementary operations. The elementary operations include unitary operation, i.e. $\sqrt{x}$, $cos(\cdot)$, and binary operations, i.e. $+, -, x, /$ [4]. Therefore, an objective function can be set up to have a structure as shown in (2.1), using these elementary operations. The partitioned extended Jacobian (2.3) of this structure consists of a block lower triangular matrix which is useful for improving computational efficiency. This is what we are referring to as structured AD in this paper.

These structures are used to improve space and time efficiency. For example, in the gradient case (2.7), the block lower triangular structure allows the use of (2.12) to compute the product $J^T\omega$ (i.e. reverse-mode) directly without computing the submatrices explicitly. Therefore, computational space and time can be saved.

Now, we describe the overview of the general approach of structured AD. This overview explains the derivation of (2.1), (2.2), (2.3), and (2.4) shown in this section. First, we set up the objective function to have a structure, using elementary operations, that would lead to a block lower triangular matrix in the partitioned extended

Jacobian. This structure shown in (2.1) is referred to as the general structure. Secondly, we differentiate (2.1) to obtain the entries of the extended Jacobian matrix, J^E (2.2). The differentiation can be done by forward mode, reverse mode, or a combination of both. Then, we partition J^E as shown in (2.3) for the purpose of using the Schur-complement formula later on. Finally, the derivative of the objective function can be obtained using the Schur-complement formula shown in (2.4).

The general form we consider for computing $z = F(x)$, $F(x) : R^n \to R^m$, can be expressed as follows:

$$\left.\begin{array}{rl} \text{Solve for } y_1: & F_1(x, y_1) = 0 \\ \text{Solve for } y_2: & F_2(x, y_1, y_2) = 0 \\ & . \\ & . \\ & . \\ \text{Solve for } y_p: & F_p(x, y_1, y_2, ..., y_p) = 0 \\ \text{Solve for output } z: & z - \overline{F}(x, y_1, y_2, ..., y_p) = 0 \end{array}\right\} \tag{2.1}$$

where F_i and $\overline{F}$ are continuously differentiable vector-valued functions and y_i are the intermediate values for $i = 1 : p$. Note that $z = F(x)$ is the objective function.

Therefore, the extended Jacobian J^E of equation (2.1) can be expressed as follows:

$$J^E = \begin{pmatrix} \frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial y_1} & & & \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial y_1} & \frac{\partial F_2}{\partial y_2} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \frac{\partial F_p}{\partial x} & \frac{\partial F_p}{\partial y_1} & \frac{\partial F_p}{\partial y_2} & \cdots & \frac{\partial F_p}{\partial y_p} \\ \frac{\partial \overline{F}}{\partial x} & \frac{\partial \overline{F}}{\partial y_1} & \frac{\partial \overline{F}}{\partial y_2} & \cdots & \frac{\partial \overline{F}}{\partial y_p} \end{pmatrix} \tag{2.2}$$

In order to gain more insight, the extended Jacobian matrix J^E can be partitioned into four sections. The four partitions will be labeled as A, B, C, and D hereafter. Partition A is composed of $\frac{\partial F_i}{\partial x}$ where $i = 1 : p$ as mentioned above. Partition B is composed of $\frac{\partial F_i}{\partial y_i}$ as there are p intermediate functions and p intermediate values. Lastly, Partition C and D are composed of $\frac{\partial \overline{F}}{\partial x}$ and $\frac{\partial \overline{F}}{\partial y_i}$, respectively. The partitioned extended Jacobian matrix J^E can be expressed as follows:

$$J^E = \left[\begin{array}{c|cccc} J_x^1 & J_{y_1}^1 & & & \\ J_x^2 & J_{y_1}^2 & J_{y_2}^2 & & \\ \vdots & \vdots & \vdots & \ddots & \\ J_x^p & J_{y_1}^p & J_{y_2}^p & \cdots & J_{y_p}^p \\ \hline \overline{J}_x & \overline{J}_{y_1} & \overline{J}_{y_2} & \cdots & \overline{J}_{y_p} \end{array}\right] = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \tag{2.3}$$

Assuming a unique solution to this, B is a nonsingular matrix. Therefore, it is clear that B is block lower triangular. In addition, in many application each $J_{y_p}^p$ is lower triangular.

The Jacobian of F can be obtained using the following Schur-complement formulation [9]:

$$J = C - DB^{-1}A \tag{2.4}$$

As mentioned in the introduction, there are two ways to compute (2.4). The first method is the forward mode:

$$J = C - D[B^{-1}A] \tag{2.5}$$

As shown in (2.3), both B and A can be obtained from top to bottom as the intermediate values are obtained. Since B is block lower triangular and nonsingular, computing $B^{-1}A$ is straightforward. Hence the Jacobian J can be obtained simultaneously from top to bottom as the function F is evaluated. The space required for forward mode is essentially the same as it would required for evaluating F and storing J. The forward mode is relatively easy to implement.

The other method is the reverse mode:

$$J = C - [DB^{-1}]A \tag{2.6}$$

Compared to forward-mode, reverse-mode AD records each intermediate operation of the differentiating function on a "tape", then rolls back from the end of the "tape" to the beginning to obtain the derivative [4]. In order to perform the reverse mode, computing DB^{-1} first, "tape" memory is required to store (2.3). Since A and B would be generated from top to bottom when the function F is evaluated, (2.3) must be stored so that B can be retrieved from bottom to top for the reverse mode operation.

Now, assume the last intermediate function in (2.1) is a continuously differentiable scalar-valued function, the structured form of the objective function $z = f(x)$, $f(x) : R^n \to R$, can be expressed as follows:

$$\left.\begin{array}{rl} \text{Solve for } y_1: & F_1(x) - y_1 = 0 \\ \text{Solve for } y_2: & F_2(x, y_1) - y_2 = 0 \\ & . \\ & . \\ & . \\ \text{Solve for } y_p: & F_p(x, y_1, y_2, ..., y_{p-1}) - y_p = 0 \\ \text{Solve for output } z: & \overline{f}(x, y_1, y_2, ..., y_p) - z = 0 \end{array}\right\} \tag{2.7}$$

Similar to (2.3), the extended Jacobian J^E of (2.7) can be expressed as follows:

$$J^E = \left[\begin{array}{c|cccc} J_x^1 & -I & & & \\ J_x^2 & J_{y_1}^2 & -I & & \\ \vdots & \vdots & \vdots & \ddots & \\ J_x^p & J_{y_1}^p & \vdots & \cdots & -I \\ \hline \nabla\overline{f}_x^T & \nabla\overline{f}_{y_1}^T & \cdots & \cdots & \nabla\overline{f}_{y_p}^T \end{array}\right] = \left[\begin{array}{c|c} A & B \\ \hline \nabla\overline{f}_x^T & \nabla\overline{f}_y^T \end{array}\right] \tag{2.8}$$

Hence, the gradient of f can be obtained using the Schur-complement formula [1]:

$$\nabla f^T = \nabla \overline{f}_x^T - \nabla \overline{f}_y^T B^{-1} A \tag{2.9}$$

Algorithm 2.1: Computing Gradients of General Structured Functions

1. Evaluate $y_i = F_i(x)$, $i = 1, \cdots, p$.

2 Evaluate $z = \overline{f}(x, y_1, \cdots, y_p)$ and apply reverse-mode AD to obtain $\nabla \overline{f}^T = (\nabla \overline{f}_x^T, \nabla \overline{f}_{y_1}^T, \cdots, \nabla \overline{f}_{y_p}^T)$.

3. Gradient Computation with (2.12):
(a) Consider $\nu^T = \nabla \overline{f}_y^T B^{-1} A$. Define vector $\omega^T = (\omega_1^T, \cdots, \omega_p^T)$ which satisfies $\omega^T = \nabla \overline{f}_y^T B^{-1}$.

(b) Set $\nu_i = 0$ for $i = 1 : p$. Let $\nabla f = \nabla \overline{f}_x$.

(c) From bottom to top:
For $j = p, p-1, \cdots, 1$
- Solve $\omega_j = \nabla \overline{f}_{y_j} - \nu_j$
- Evaluate $F_j(x, y_1, \cdots, y_{j-1})$
    - Apply reverse-mode AD with ω_j^T to obtain $\omega_j^T \cdot (A^j, B^j)$.
- Let $\nu_i^T = \nu_i^T + \omega_j^T \cdot B_i^j$ for $i = 1 : j-1$
- Update $\nabla f^T \leftarrow \nabla f^T + \omega_j^T A^j$

Step 3 in Algorithm 2.1 [4], A^j represents J_x^j and B^j represents $(J_{y_1}^j, \cdots, J_{y_{j-1}}^j)$. In order to compute $\nu^T = \nabla \overline{f}_y^T B^{-1} A$, vector $\omega^T = (\omega_1^T, \cdots, \omega_p^T)$ was defined to satisfy $\omega^T = \nabla \overline{f}_y^T B^{-1}$. It can be expressed as follows:

$$\omega^T B = \nabla \overline{f}_y^T \tag{2.10}$$

Transpose (2.10) to obtain:

$$B^T \omega = \nabla \overline{f}_y \tag{2.11}$$

Hence, (2.11) can be expressed as follows:

$$\begin{bmatrix} -I & (J^2_{y_1})^T & (J^3_{y_1})^T & \cdots & (J^{p-1}_{y_1})^T & (J^p_{y_1})^T \\ & -I & (J^3_{y_2})^T & \cdots & \vdots & (J^p_{y_2})^T \\ & & -I & \cdots & \vdots & \vdots \\ & & & \ddots & (J^{p-1}_{y_{p-2}})^T & (J^p_{y_{p-2}})^T \\ & & & & -I & (J^p_{y_{p-1}})^T \\ & & & & & -I \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \vdots \\ \omega_{p-1} \\ \omega_p \end{bmatrix} = \begin{bmatrix} \nabla \overline{f}_{y_1} \\ \nabla \overline{f}_{y_2} \\ \nabla \overline{f}_{y_3} \\ \vdots \\ \nabla \overline{f}_{y_{p-1}} \\ \nabla \overline{f}_{y_p} \end{bmatrix} \tag{2.12}$$

and

$$\nu^T = \nabla \overline{f}_y^T B^{-1} A = \omega^T A = \omega_1^T J_x^1 + \omega_2^T J_x^2 + \cdots + \omega_{p-1}^T J_x^{p-1} + \omega_p^T J_x^p \tag{2.13}$$

Reverse-mode AD computes $B^T\omega$ directly and does not compute B first. Since explicit computation of submatrices $J^i_{y_i}$ is not required, computing the derivative as shown in (2.12) saves computation. The vector ω_j can be obtained from bottom to top, i.e. first solving for ω_p, then solve for ω_{p-1} using ω_p, etc. Furthermore, this structured approach (Algorithm 2.1) is more space efficient than plain reverse-mode AD (unstructured). Although the time required for both approaches are the same, i.e. $\omega(\nabla f) \sim \omega(f)$, structured AD is more space efficient. The space required by a given code to evaluate f is denoted as $\sigma(f)$. Plain reverse-mode AD requires $\sigma \sim \omega(\overline{f}) + \sum_{i=1}^{p} \omega(F_i)$ while structured AD only requires $\sigma \leq \max\{\omega(\overline{f}), \omega(F_i), i = 1, \cdots, p\}$. The effect becomes even more obvious for large-scale problems, i.e. large p. Table 2.1 and 2.2 summarize the time and space required for each of the methods, respectively.

In addition to memory saving, structured reverse-mode AD contributes to time efficiency as well. As mentioned in Section 1.3, reverse-mode AD is faster than forward-mode AD or finite differencing for gradient computations. Forward-mode AD and finite differencing require time proportional to $n \cdot \omega(f)$ while reverse-mode AD only requires time proportional to $\omega(f)$. But the order of work for reverse-mode AD is bounded by $\omega(f)$ only when fast memory is available [1]. The gradient computation will slow down significantly after running out of fast memory. Structured reverse-mode AD reduces memory usage. It allows more gradient computations to be

Methods	Time Required
Finite differencing	$n \cdot \omega(f)$
Forward-mode AD	$n \cdot \omega(f)$
Reverse-mode AD (fast memory only)	$\omega(f)$

Table 2.1: Time Required by Each Method

Methods	Space Required
Plain reverse-mode AD	$\sigma \sim \omega(\overline{f}) + \sum_{i=1}^{p} \omega(F_i)$
Structured reverse-mode AD	$\sigma \leq \max\{\omega(\overline{f}), \omega(F_i), i = 1, \cdots, p\}$

Table 2.2: Space Required by Reverse-mode AD

done using the fast memory compared to plain reverse-mode AD. Thus, structured reverse-mode AD requires less running time than plain reverse-mode AD. That is why structured reverse-mode AD is crucial for both space and time efficiency.

2.2 Composite Functions/Dynamic Structure

There are two common special cases of the general AD structure, namely composite functions and generalized partially separable structure. In general, composite functions vertically cut paths into segments, usually in the unit of timesteps for option pricing, whereas generalized partially separable function horizontally divides simulations into smaller clusters of stochastic paths [3]. Therefore, combining the two structures will provide the highest efficiency in both time and space. Generalized partially separable structure (GPS) will be discussed in Section 2.3. A composite function is a highly recursive function, $f : R^n \to R$, which has the following form:

$$f(x) = \overline{f}(T_q(T_{q-1}(\cdots(T_1(x))\cdots))) \tag{2.14}$$

where $z = f(x)$ is the objective function, T_i $(i = 1 : q)$ is a vector-valued function, and $\overline{f}$ is a continuously differentiable scalar-valued function.

Composite (recursive) function can also be expressed in the following structured form:

$$\left.\begin{array}{rl} \text{Solve for } y_1: & y_1 - T_1(x) = 0 \\ \text{Solve for } y_2: & y_2 - T_2(y_1) = 0 \\ & . \\ & . \\ & . \\ \text{Solve for } y_q: & y_q - T_q(y_{q-1}) = 0 \\ \text{Solve for output } z: & z - \overline{f}(y_q) = 0 \end{array}\right\} \qquad (2.15)$$

where the intermediate vector y_i's, $i = 1 : q$, are of varying dimension and $\overline{f}$ is a continuously differentiable scalar-valued function.

It can be clearly seen that each current intermediate value only depends on the intermediate value immediately before itself. For example, y_2 only depends on y_1, similarly, y_q only depends on y_{q-1}. Since y_i only depends on y_{i-1}, the composite function structure uses less memory than non-structured AD as it does not require space for storing computations before y_{i-1} for the computation of y_i.

The composite function can also be written in a more condensed form:

$$\left.\begin{array}{rl} \text{For } i = 1 : q & \\ \text{Solve for } y_i: & y_i - T_i(y_{i-1}) = 0 \\ \text{Solve for } z: & z - \overline{f}(y_q) = 0 \end{array}\right\} \qquad (2.16)$$

where $y_0 = x$.

Taking a similar approach from Section 2.1, the extended Jacobian J^E of the composite function can be computed and written as follows:

$$J^E = \left[\begin{array}{c|cccc} \frac{\partial T_1}{\partial x} & \frac{\partial T_1}{\partial y_1} & & & \\ \frac{\partial T_2}{\partial x} & \frac{\partial T_2}{\partial y_1} & \frac{\partial T_2}{\partial y_2} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \frac{\partial T_q}{\partial x} & \frac{\partial T_q}{\partial y_1} & \frac{\partial T_q}{\partial y_2} & \cdots & \frac{\partial T_q}{\partial y_q} \\ \hline \frac{\partial \overline{f}}{\partial x} & \frac{\partial \overline{f}}{\partial y_1} & \frac{\partial \overline{f}}{\partial y_2} & \cdots & \frac{\partial \overline{f}}{\partial y_q} \end{array}\right] = \left[\begin{array}{c|cccc} -J_x^1 & I & & & \\ & -J_{y_1}^2 & I & & \\ & & \ddots & \ddots & \\ & & & -J_{y_{q-1}}^q & I \\ \hline & & & & \nabla \overline{f}_{y_q}^T \end{array}\right] \tag{2.17}$$

Note that dynamic system (DS) is a special case of the composite function where each transformation is identical, i.e. $T_i = T$ for $i = 1 : q$.

Figure 2.1 illustrates the block diagram of the composite function (dynamic structure). J_i represents the Jacobian of T_i, $i = 1 : q$. It helps to understand the computation of the composite function shown in (2.15). The block diagram of the composite function can be explained as a timeline. First, we supply the input vector x, function T_1 is evaluated at x and we obtain y_1 as one of the outputs. Also, we can differentiate T_1 to obtain J_1 as the other output. The differentiation can be done by forward mode, reverse mode, or a combination of both. Then, we supply the previous output, y_1, as the new input. Function T_2 is evaluated at y_1 to obtain y_2 as an output. Similarly, we can differentiate T_2 to obtain J_2 as the other output. This process is carried out for $i = 1 : q$. Hence, as explained before (2.16), each current intermediate value of the composite function only depends on the intermediate value immediately before itself.

2.3 Generalized Partially Separable Functions

As mentioned in the previous section, the best performance in terms of time and space efficiency happens when exploiting both of the generalized partially separable (GPS) and composite function (CF) structure. We explore this in the context of Monte Carlo pricing. The use of Monte Carlo for evaluating option pricing will be discussed in Chapter 3. Each path generated by Monte Carlo is independent of other paths. Each function in the GPS structure is independent of other functions.

Assume to evaluate a scalar-valued function, $f : R^n \to R$, can be expressed in the following structured form:

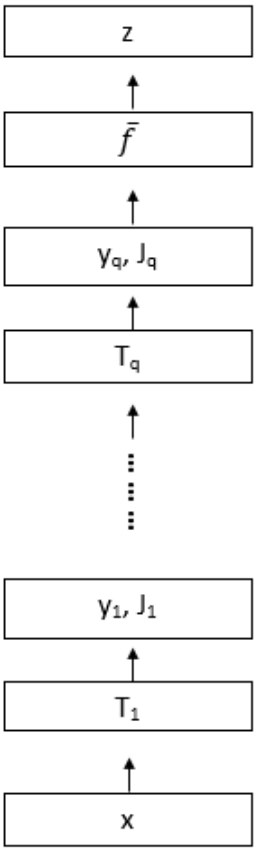

Figure 2.1: Block diagram of composite function (dynamic structure) [1]

$$\left.\begin{array}{rl} \text{Solve for } y_1: & y_1 - T_1(x) = 0 \\ \text{Solve for } y_2: & y_2 - T_2(x) = 0 \\ & . \\ & . \\ & . \\ \text{Solve for } y_p: & y_p - T_p(x) = 0 \\ \text{Solve for output } z: & z - \overline{f}(x, y_1, \cdots, y_p) = 0 \end{array}\right\} \tag{2.18}$$

where $z = f(x)$ is the objective function, T_k $(k = 1 : p)$ is a vector-valued function, and $\overline{f}$ is a continuously differentiable scalar-valued function.

In a more condensed form,

$$\left.\begin{array}{ll} \text{For } k = 1:p & \\ \text{Solve for } x: & y_k - T_k(x) = 0 \\ \text{Solve for } z: & z - \overline{f}(x, y_1, \cdots, y_p) = 0 \end{array}\right\} \tag{2.19}$$

where x is a vector of input variables of function T_k, $k = 1:p$. As shown in (2.18) each intermediate function is independent of the previous intermediate values.

Therefore, the extended Jacobian J^E of this GPS structure can be written as:

$$J^E = \left[\begin{array}{c|ccc} -J_x^1 & I & & \\ -J_x^2 & & I & \\ \vdots & & & \ddots \\ -J_x^p & & & I \\ \hline & \nabla\overline{f}_{y_1}^T & \nabla\overline{f}_{y_2}^T & \cdots \quad \nabla\overline{f}_{y_p}^T \end{array}\right] = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \tag{2.20}$$

where partition B is an identity matrix.

Figure 2.2 illustrates the block diagram of the generalized partial separable (GPS) function. It helps to understand the computation of the GPS function shown in (2.18). The superscript represents the path number and the subscript represents the timestep. Also, p is the total number of paths and q represents the number of timesteps. Note that $p >> q$. Recall Section 2.2, each path in Figure 2.2 represents a composite function (2.14). In other words, Figure 2.2 is made up of p block diagrams of the composite function. The block diagram of the composite function is shown in Figure 2.1. Since each path is independent of other paths, each intermediate function only depends on x. Hence, we can write $y_k - T_k(x) = 0$ for $k = 1:p$ as shown in (2.19). Algorithm 2.2 illustrates the gradient computation using the GPS approach [3].

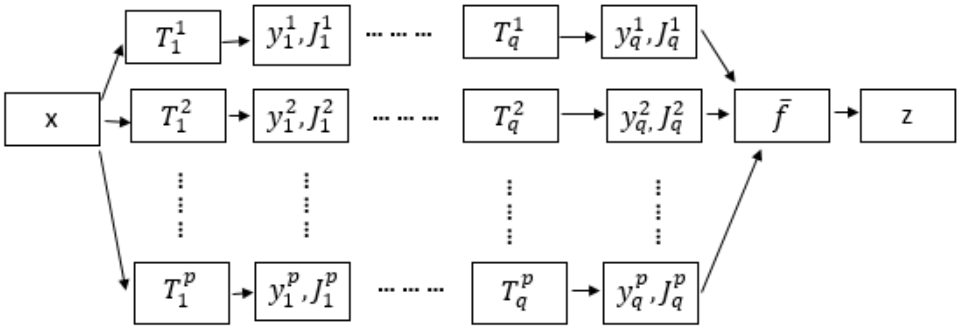

Figure 2.2: Block diagram of generalized partial separable function [4]

Algorithm 2.2: Generalized Partially Structure Functions (GPS)

1. Evaluate $y_k = T_k(x)$, $k = 1, \cdots, p$. (Note: there are p number of paths)

2. Evaluate $\overline{f}(y_1, \cdots, y_p)$ and apply reverse-mode AD to obtain $w_k = \nabla \overline{f}_{yk}$, $k=1, \cdots, p$. (Note: matrix B is the identity matrix)

3. Gradient Computation by reverse-mode AD:
   - Compute $v_k^T = w_k^T J_x^k$, where J_x^k is the Jacobian of $T_k(x)$, $k = 1, \cdots, p$.
   - Set $\nabla f(x) \leftarrow \sum_{k=1}^{p} v_k$.

Chapter 3

Monte Carlo and Types of Options

3.1 Monte Carlo and its Gradient Calculations

As mentioned in the previous chapter, Monte Carlo is useful for option pricing in financial applications. Monte Carlo method predicts option values by generating a large number of random paths and taking the average of a large number of terminal values. The more random simulation paths, the more reliable the predicted option value would be. Since the Monte Carlo formula contains the random Brownian term, Z, the option value produced by Monte Carlo is unbiased. This method is widely used in the financial industry because of its simplicity. When making an investment and evaluating a portfolio, the derivatives are equally important as the option value itself. Hedging analysis is crucial for risk assessment and it is necessary for constructing a portfolio. For instance, the gradient of the option value with respect to its asset price, $\frac{\partial V}{\partial S}$, is used to offset the sensitivity to the underlying up to the first order. Unlike the simple vanilla options, it is not easy to compute the gradients for the types of exotic options mentioned in the next sections. Hence, automatic differentiation becomes very handy for this application.

Monte Carlo exhibits the structure that is required for implementing the structured AD techniques. In this chapter, we introduce the types of exotic options examined in this paper. And we describe the gradient computation of these exotic options using the generalized partially separable structure. Figure 3.1 is an example of the Monte Carlo simulation.

In this section, we demonstrate how the average option value $\widehat{C}$ and its derivative can be estimated in a path-by-path manner [1]. Then we explain how to relate this to the general AD structure (2.7) and simplify (2.7) to obtain the generalized separable

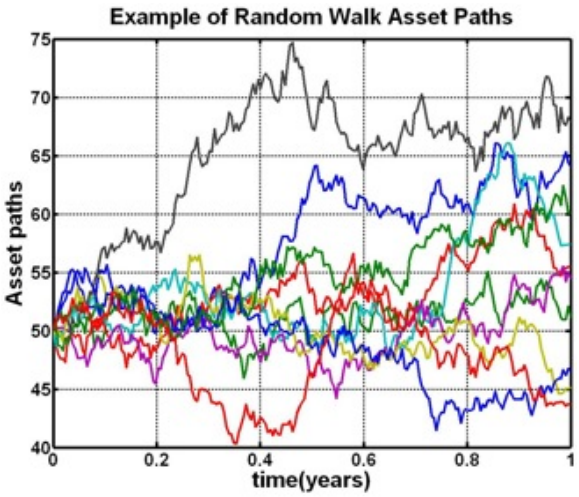

Figure 3.1: An example of the Monte Carlo simulation

structure (GPS) (2.18). First, the basic Monte Carlo formula for option pricing can be expressed as follows:

$$C = E(V(S)) \tag{3.1a}$$

where C is the option value, S is the underlying asset price, $V(S)$ is the payoff function of the underlying asset S at expiry and $E(\cdot)$ is the expectation. Note that the underlying asset price S is a function of variables x, including but not limited to the initial asset price S_0, volatility σ, and risk-free interest rate r, and the random Brownian motion, Z. Therefore, S can be expressed as follows:

$$S = h(x, Z) \tag{3.1b}$$

where x is the vector of initial parameters that can influence the evolution of S. Z is the random innovation that drives the price. Note that C can also be written in its integral form as follows:

$$C = \int (V(S))\rho(Z)dZ = \int V(h(x, Z))\rho(Z)dZ \tag{3.2}$$

where $\rho(Z)$ is the probability density function. Note that $\rho(Z)$ is independent of the deterministic variables. In order to be able to interchange the order of integration and differentiation, certain regularity conditions must be fulfilled by $V(\cdot)$ [3]. Then, we can write:

$$\frac{\partial C}{\partial x} = \frac{\partial}{\partial x} \int V(h(x, Z))\rho(Z)dZ = \int \frac{\partial V(h(x, Z))}{\partial x}\rho(Z)dZ \tag{3.3}$$

Due to (3.1a) and (3.2), (3.3) can be written as follows:

$$\frac{\partial C}{\partial x} = \frac{\partial}{\partial x}E(V(S)) = E(\frac{\partial V(S)}{\partial x}) \tag{3.4}$$

Assume p simulations have been generated, we can estimate the integral with Monte Carlo simulations. First, break down the random Brownian term Z to p discrete vectors Z_k. Each vector corresponds to a simulation path. Then we can write:

$$\frac{\partial \widehat{C}}{\partial x} = \frac{1}{p}\sum_{k=1}^{p} \frac{\partial V(h(x, Z_k))}{\partial x} \tag{3.5}$$

where $\frac{\partial \widehat{C}}{\partial x}$ is the derivative of the objective function and $\frac{\partial V(h(x,Z_k))}{\partial x}$ is the derivative of each path. The process above has shown that the derivative of the objective function can be obtained in a path-by-path manner.

Similarly, the Monte Carlo option value can be written as:

$$\widehat{C} = \frac{1}{p}\sum_{k=1}^{p} V(h(x, Z_k)) = \frac{1}{p}\sum_{k=1}^{p} \widehat{C_k} \tag{3.6}$$

where $\widehat{C_k}$ represents the simulation path corresponding to Z_k.

Now we have shown that the derivatives of the option can be obtained path-by-path. We can discuss how this idea relates to the general structure shown in (2.7) and the generalized partially separable (GPS) structure shown in (2.18). Consider the evaluation of $f(x)$ in (2.7) is the same as the evaluation of $\widehat{C}$. Each

$y_k = F_k(x, y_1, \cdots, y_{k-1})$ where $k = 1 : p$ in (2.7) corresponds to path $\widehat{C_k}$ with $y_k = V(h(x, Z_k))$, $k = 1 : p$. Since each path of Monte Carlo simulations is independent of other paths, vector y_k is independent of other vector y_j, $j < k$ in this case. In other words, y_k only depends on x. Hence, we can write $y_k - F_k(x) = 0$ for $k = 1 : p$ as shown in (2.18). Thus, option pricing in a Monte Carlo framework exhibits the generalized partially separable (GPS) structure shown in (2.18). In this case, $\overline{f}(x, y_1, \cdots, y_p) = \frac{1}{p}\sum_{k=1}^{p} \widehat{C_k}$. Note that both $T_k(x), k = 1 : p$ in (2.18) and $F_i(x), i = 1 : p$ in (2.7) represent intermediate functions and p simulation paths [1]. They are the same in this context. Figure 3.2 illustrates the computation of option values using Monte Carlo where $\widehat{C}_k, k = 1 : p$ is the option value of each path and $\widehat{C}$ is the average option value.

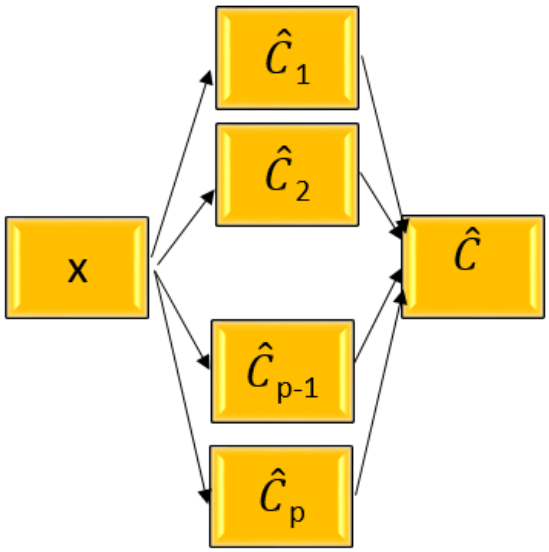

Figure 3.2: Computation of option value using Monte Carlo

3.2 Basket Options

The stochastic differential equation (SDE) for the asset price can be expressed as follows:

$$S^{i+1} = S^i e^{(r-\frac{\sigma^2}{2})\Delta t + \sigma Z\sqrt{\Delta t}} \tag{3.7}$$

where r is risk-free interest rate, σ is volatility, and $Z \sim N(0,1)$ is standard normal.

For simple European options, the payoff functions are defined as follows:

$$V_{call} = max(S(T) - K, 0) \tag{3.8a}$$

$$V_{put} = max(K - S(T), 0) \tag{3.8b}$$

where V represents the option value, K represents the strike price, and $S(T)$ represents the underlying asset price at expiry.

The payoff functions of the exotic options are slightly different as there are usually more than one underlying asset in the portfolio or the average of the asset price is used instead of the price at expiry. For instance, the basket option is an extension of the European option. A call basket option takes the weighted average of a group of M assets and calculates the payoff using the weighted average of asset price, instead of a single asset price, and the strike price [1,8]. The formula can be expressed as follows:

$$V_{call} = max(\sum_{j=1}^{M} w_j S_j(T) - K, 0) \tag{3.9}$$

where K is the strike price, T is the expiry time, $S_j(T)$ is the j^{th} asset price at expiry in the basket, and w_j is the weight for the j^{th} asset.

For a basket option with l baskets, (3.9) can also be written in the matrix form:

$$\begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_l \end{bmatrix} = max\left(\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1M} \\ w_{21} & w_{22} & \cdots & w_{2M} \\ \vdots & \vdots & \cdots & \vdots \\ w_{l1} & w_{l2} & \cdots & w_{lM} \end{bmatrix} \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_M \end{bmatrix} - \begin{bmatrix} K_1 \\ K_2 \\ \vdots \\ K_l \end{bmatrix}, 0 \right) \tag{3.10}$$

where S is a vector with M underlying asset prices, W_{lj} is a matrix with l by j basket weights, V is a vector with l basket option values, and K is a vector with l strike prices. Each row of w represents a basket with different weights for each underlying asset. Figure 3.4 illustrates the computational diagram for one basket. Suppose there are M assets in the basket ($j = 1 : M$) and q timesteps ($i = 1 : q$). $S_{j,0}$ is the initial price of the j^{th} asset, $S_{j,q}$ is the price of the j^{th} asset at expiry, and V is the weighted basket option value.

Algorithm 3.1 illustrates how to compute the gradients for basket options using the generalized partially separable structure (GPS) [1,8]. In general, x represents a vector involving multiple deterministic parameters such as volatility and risk-free interest rates, etc. In this demonstration, x only includes S_0 and volatility. Also, x_j represents the vector of deterministic parameters for the j^{th} asset, $S_{j,0}$ is the initial asset price of the j^{th} asset, and σ_j is the volatility of the j^{th} asset. Note that the vector of parameters is different for each asset, i.e. $x_1 \neq x_2$ and $S_{1,0} \neq S_{2,0}$.

As mentioned in Chapter 2, composite function (dynamic structure) vertically cuts paths into segments, usually in the unit of timesteps for option pricing, whereas generalized partially separable function horizontally divides simulations into smaller clusters of stochastic paths. The terms "by batch" and "by segment" will be referred to as "generalized partially separable structure" and "dynamic structure", respectively, hereafter. Figure 3.3 illustrates that if there are total of eight paths and they are divided into two batches, the solid blue lines represent one of the two batches with batch size equaling to four. Similarly, the dashed black lines represent the other of the two batches with batch size equaling to four.

Figure 3.4 illustrates the computational diagram of the basket option. This is a simplified diagram as it is showing only one path for each asset in the basket. For example, $S_{1,0}$ represents the initial asset price of the first asset and $S_{M,0}$ represents the initial asset price of the M^{th} asset. Similarly, $S_{1,q}$ represents the asset price at expiry of the first asset and $S_{M,0}$ represents the asset price at expiry of the M^{th} asset. The straight arrows represent the evolution of the simulated paths going from initial time point to expiry. The straight arrow is referred to as the forward sweep. We perform forward sweeps to obtain the asset price at expiry. The curved arrow is referred to as the reverse sweep. We perform reverse sweeps via Algorithm 2.2 and 3.1 to obtain the matrices of the extended Jacobian shown in (2.20). These Jacobian matrices, i.e. $J_1, J_2, \cdots, J_M$, are stored and used for computing the gradient using the Schur-complement formula (2.9).

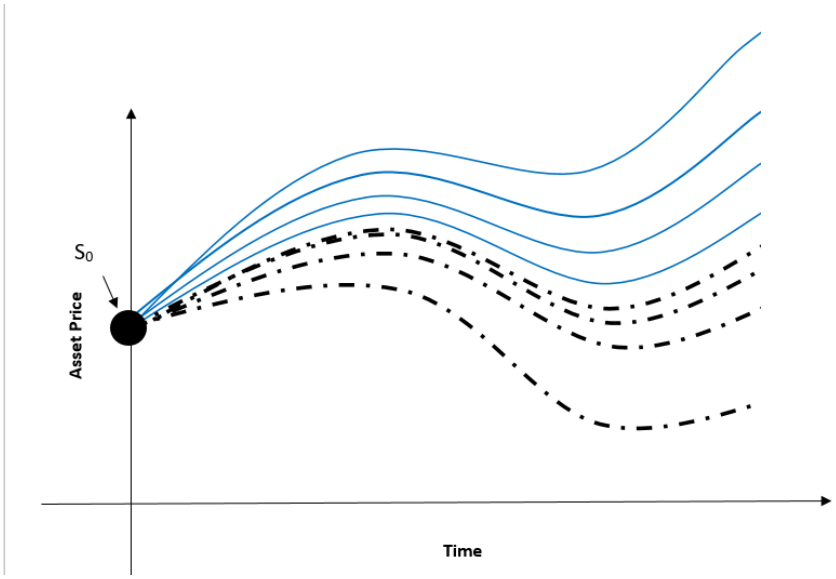

Figure 3.3: Diagram of a Batch

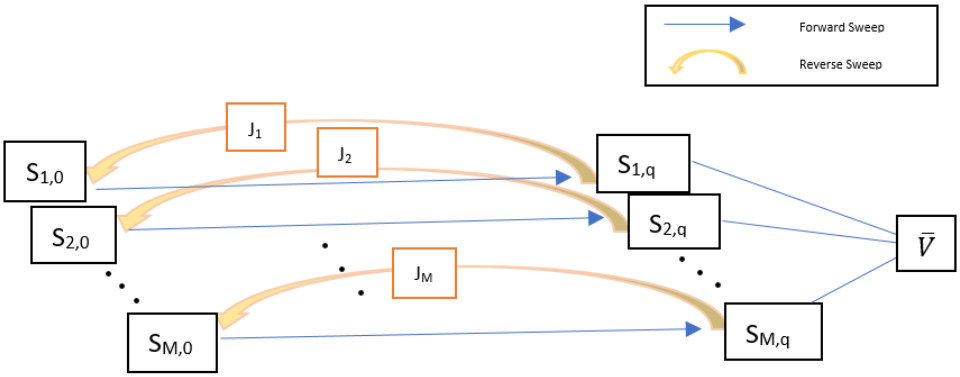

Figure 3.4: Diagram of Basket Options Computation with M Assets

Algorithm 3.1: Computation of Basket Options - By Batch

1. Set number of paths, $k = 1 : p$, for one asset.

2. Set number of timesteps: $i = 1 : q$.

3. For path k_1 to k_h, perform forward sweep until expiry, q.

   • Store all the intermediate values for k_1 to k_h. Perform reverse sweep in one segment to compute the Jacobian matrix $\frac{\partial S_k}{\partial x}$ for path k_1 to k_h.

   • Store the Jacobian matrix and clear all intermediate values of this batch.

   • Repeat for all batches, i.e. k_{h+1} to k_p.

4. Compute the gradients of $\frac{\overline{V}}{x}$ with Algorithm 2.2 using the stored Jacobian matrix.

5. Repeat for all assets, $j = 1 : M$, in the basket.

Note: x only includes S_0 and volatility in this demonstration.

In Algorithm 3.1, the memory usage reduced for the computation is proportional to the number of batches. Since all intermediate values from k_1 to k_h are overwritten by the intermediate values from k_{h+1} to k_p after computing the Jacobian matrix. That is, the intermediate values of the previous batch are overwritten by the intermediate values of the current batch, the intermediate values of the current batch are overwritten by the intermediate values of the next batch, etc. Therefore, only $\frac{1}{2}$ of the memory is required when dividing the total number of paths into two batches. Similarly, only $\frac{1}{t}$ of the memory is required when dividing the total number of paths into t batches.

3.3 Asian Options

As mentioned in the previous section, there are different types of exotic options. The payoff function of an Asian call option [1,2] is defined as follows:

$$V_{call} = max(\frac{1}{q}\sum_{i=1}^{q} S_i - K, 0) \tag{3.11}$$

where S_i is the asset price at time point i. q is the total number of timesteps. K is the strike price. Figure 3.5 illustrates the diagram of the Asian options computation for one path.

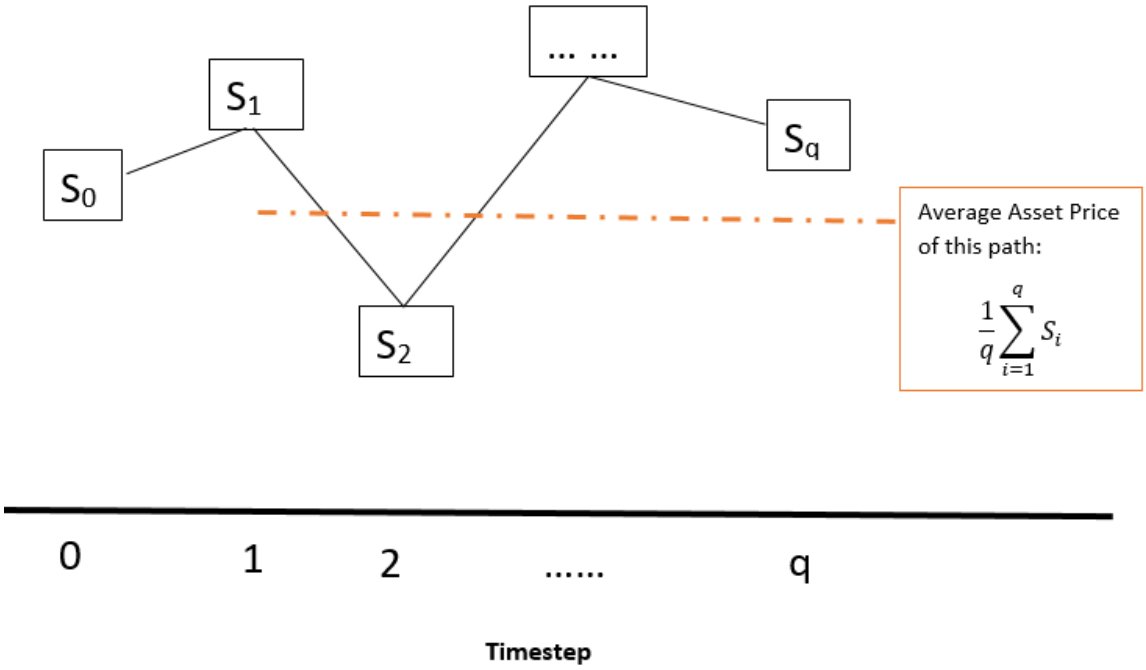

Figure 3.5: Diagram of Asian Options Computation: one of the paths shown

Algorithm 3.2 illustrates how to compute the gradients for Asian options by batch [1,6]. Figure 3.5 illustrates the computational diagram of Asian options for one path. Figure 3.6 illustrates the computational diagram of Asian option's gradient. This is a simplified diagram as it is showing only one path and one asset. For example, S_0 represents the initial asset price of the asset and S_q represents the asset price at expiry of the asset The straight arrows represent the evolution of the simulated paths going from initial time point to expiry. The straight arrow is referred to as the forward sweep. We perform forward sweeps to obtain the asset price at expiry. The curved arrow is referred to as the reverse sweep. We perform reverse sweeps via Algorithm 2.2 and 3.2 to obtain the matrices of the extended Jacobian shown in (2.20). The Jacobian matrix, i.e. $\frac{\partial S_k}{\partial x}$, is stored and used for computing the gradient

using the Schur-complement formula (2.9).

Algorithm 3.2: Computation of Asian Options - By Batch

1. Set number of paths, $k = 1 : p$, for the Asian option.

2. Set number of timesteps, $i = 1 : q$.

3. For path k_1 to k_h, perform forward sweep until expiry, q.

   • Store all the intermediate values for k_1 to k_h. Perform reverse sweep in one segment to compute the Jacobian matrix $\frac{\partial S_k}{\partial x}$ for path k_1 to k_h.

   • Store the Jacobian matrix and clear all intermediate values of this batch.

   • Repeat for all batches, i.e. k_{h+1} to k_p.

4. Compute the gradients of $\frac{\overline{V}}{x}$ with Algorithm 2.2 using the stored Jacobian matrix.

Note: x only includes S_0 and volatility in this demonstration. In general, x can be a vector involving other deterministic parameters such as volatility and risk-free interest rates, etc.

In Algorithm 3.2, the memory usage reduced for the computation is proportional to the number of batches. Since all intermediate values from k_1 to k_h are overwritten by the intermediate values of k_{h+1} to k_p after computing the Jacobian matrix. That is, the intermediate values of the previous batch are overwritten by the intermediate values of the current batch, the intermediate values of the current batch are overwritten by the intermediate values of the next batch, etc. Therefore, only $\frac{1}{2}$ of the memory is required when dividing the total number of paths into two batches. Similarly, only $\frac{1}{t}$ of the memory is required when dividing the total number of paths into t batches.

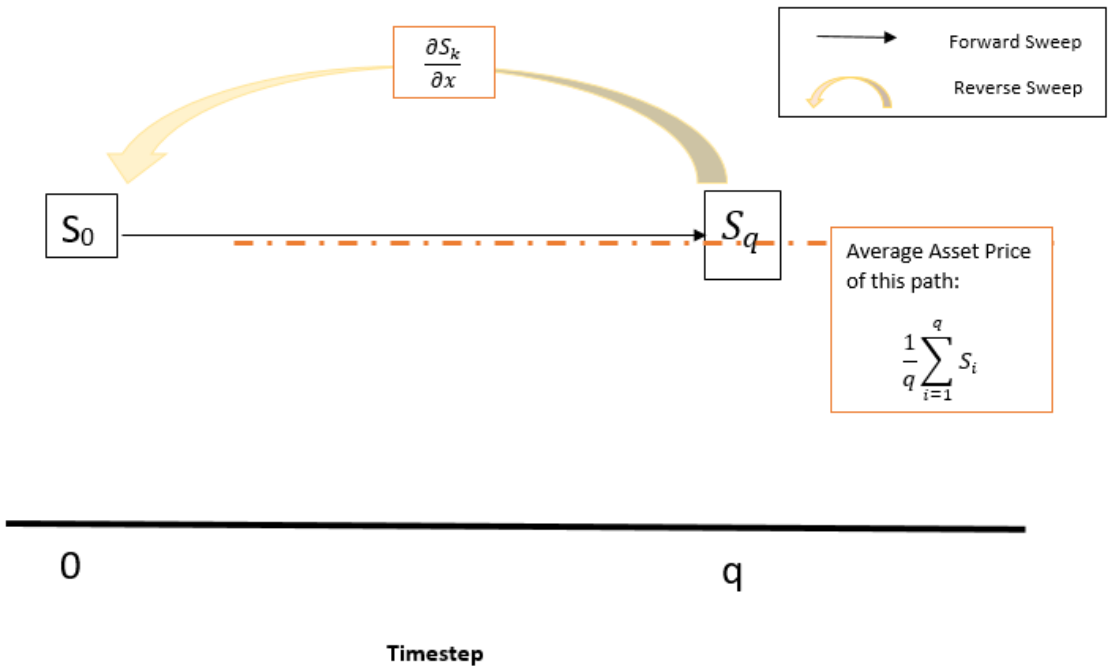

Figure 3.6: Diagram of Asian Options' Gradient Computation

3.4 Best of Asian Options

Suppose there are M assets in the portfolio. The Asian option value of the j^{th} asset is denoted as A_j for $j = 1 : M$. The best of Asian option is obtained by first computing the values of the Asian option and then choosing the maximum value from the vector of Asian option values [1,2]. The mathematical formula can be written as follows:

$$V_{call} = max(A) \tag{3.12}$$

where A can be expressed as:

$$A_j = max(\overline{S}_j - K_j, 0) \tag{3.13a}$$

$$\overline{S_j} = \frac{1}{q}\sum_{i=1}^{q} S_j^i \tag{3.13b}$$

where $\overline{S_j}$ is the average price of the j^{th} asset and K_j is the strike price.

The algorithm for the best of Asian option is the same as the algorithm for the Asian option since the option value of the best of Asian option is simply the maximum value of the Asian option. Refer to Algorithm 3.2 for the gradient computation of best of Asian options using the generalized partially separable (GPS) structure. The Jacobian of the best of Asian option is just a row vector when the regular Asian option produces a diagonal matrix Jacobian. For example:

$$\textbf{Asian Option Values} = \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_m \end{bmatrix} \tag{3.14}$$

$$\textbf{Jacobian of Asian Option} = \begin{bmatrix} J_1 & & & \\ & J_2 & & \\ & & \ddots & \\ & & & J_m \end{bmatrix} \tag{3.15}$$

Assuming V_m is the maximum value in (3.14), then:

$$\textbf{Best of Asian Option Values} = V_m \tag{3.16}$$

$$\textbf{Jacobian of Best of Asian Option} = \begin{bmatrix} 0 & \dots & \dots & 0 & J_m \end{bmatrix} \tag{3.17}$$

Chapter 4

Results of Experiments

4.1 Machine Specification and Experiments Overview

To be consistent, all experiments have been performed on the same machine with the following specifications: Intel Core i5-5200U, 8GB DDR3 L memory, and 1000 GB HDD. The version of Matlab used was R2016b. The AD packages used were ADMAT 2.0 and MADO.

In order to perform the objective comparison tests, two versions of AD algorithm have been developed and tested. The experiments were performed with several combinations of AD algorithm and underlying user code for option pricing. The first experiment was called the HV2.0. It incorporated the ADMAT 2.0 algorithm, a highly vectorized underlying user code, and exploited the GPS structure. The second experiment was called the FL2.0 (ADMAT). It employed the ADMAT 2.0 algorithm, a for loop underlying user code, and exploited the GPS structure. The last experiment was called the MADO. It used the MADO algorithm, a for loop underlying user code, and exploited the GPS structure. The three experiments performed are summarized in Table 4.1.

Three comparison tests were performed. The first comparison test compared the memory usage between HV2.0 structured and HV2.0 unstructured (plain reverse mode). The second comparison test compared the memory usage of FL2.0 structured

Experiment Name	AD algorithm	Underlying User Code	Structure		Mode
HV2.0	ADMAT 2.0	Highly vectorized	GPS	By batch	Reverse Mode
FL2.0 (ADMAT)	ADMAT 2.0	For loop	GPS	By batch or path	Reverse Mode
MADO	MADO	For loop	GPS	By path	Reverse Mode

Table 4.1: Overview of Experiments

versus FL2.0 unstructured. The last comparison test compared the time efficiency between FL2.0 vs MADO. In the subsequent sections, FL2.0 will be referred to as ADMAT.

4.2 Comparison Test 1 Results

In this section, the results of the comparison test between HV2.0 structured and HV2.0 unstructured will be shown and discussed. It was done to test the performance of the AD-use of the generalized partially separable structure (GPS) for basket options, Asian options, and best of Asian options. The setup for the basket option computation was shown in (3.10) where $l = 5$ and $M = 10$. Therefore, there were five weighted basket options in this experiment. Each basket consists of ten underlying assets. Table 4.2 shows the time and memory usage for the calculation of basket option values and their gradients using the Monte Carlo method. The formula for the computation of the Asian options and best of Asian options were in (3.11) and (3.12), respectively. There were ten underlying assets in the portfolio of the Asian options and best of Asian options. Table 4.3 shows the time and memory usage for the calculation of Asian option values and their gradients using the Monte Carlo method. Table 4.4 shows the time and memory required for the calculation of best of Asian option values and their gradients using the Monte Carlo method.

For each of the options, six total Monte Carlo instances were examined, namely 100000, 80000, 60000, 40000, 20000, and 10000. Recall that the GPS structure horizontally divides simulations into smaller clusters of stochastic paths. In this paper, each cluster of paths is referred to as a batch and the number of paths in a batch is referred to as batch size. Five different combinations were tested for the case of 100000 total MC paths. The first combination (1 x 100000) was the Plain Reverse AD mode (also referred to as non-structured AD), where there is only 1 batch and the batch size is the same as the total number of MC paths. This is used as a controlled case for comparison with the structured scenarios. For the structured cases, the following combinations of batch x batch size are tested: 10x10000, 20x5000, 50x2000, and 100x1000.

Before discussing the results, here are the notations for Table 4.2, 4.3, and 4.4. Total MC represents the total Monte Carlo paths generated, NBatch represents the number of batches, batch size represents the number of paths in a single batch. Memory ratio is the ratio between the memory required by plain reverse mode AD and the memory required by the respective structured AD. Consider Table 4.2 as an example, for the case of 80000 total MC paths, the memory required by plain reverse mode AD is 4508.816 MB. For the first structure combination (10 batches x 8000 paths), the memory required is 450.888 MB. Therefore, the memory ratio is 9.9999. For the second structure combination (20 batches x 4000 paths), the memory ratio

is 255.448. Thus, the memory ratio is 19.9994, etc. Similarly, time ratio is the ratio between the time required by plain reverse mode AD and the time required by the respective structured AD.

Table 4.2, 4.3, and 4.4 all show that the memory usage reduces as the number of batches increases. Moreover, the memory usage reduced approximately equals to the number of batches. This phenomena was explained in the paragraph immediately after Algorithm 3.2. In short, the memory usage is reduced because the intermediate information is overwritten after each AD session due to the GPS structure. Take Table 4.2 as an example, for total MC paths of 80000, the memory usage for plain reverse AD was 4508.812 MB and the memory usage for structured AD (100x800) was only 45.096 MB. The memory usage was reduced by 100 times which equals to the number of batches. Similarly, for total MC paths of 20000, the memory usage for plain reverse AD was 1127.208 MB and the memory usage for structured AD (20x1000) was only 56.364 MB. Again, The memory usage was reduced by 20 times which equals to the number of batches. This pattern holds for all total number of MC instances, batch combinations, and all three types of exotic options examined. Table 4.3 and 4.4 show the results for Asian options and best of Asian options, respectively.

Figure 4.1, 4.2, and 4.3 show the trend of memory usage for different total number of MC instances and batch combinations. The x-axis and y-axis represent the number of batches and memory usage in MB, respectively. Each line represents the total number of MC instances. From top to bottom, the top line uses the most memory because it represents the memory usage for 100000 MC instances in total, the second line represents the memory usage for 80000 MC instances in total, and the third line represents the memory usage for 60000 MC instances in total, so on and so forth. The bottom line uses the least memory because it represents the memory usage of 10000 MC instances in total. For all cases of total number of MC instances, all three types of option show the same decreasing trend as the number of batches increases.

Figure 4.4, 4.5, and 4.6 show that all MC instance cases result in the same memory ratio which is proportional to the number of batches regardless of the total number of MC instances.

4.3 Comparison Test 2 Results

In order to show that the performance of structured AD is independent of the implementation of the underlying user code, the results of the comparison test between FL2.0 structured and FL2.0 unstructured will be shown and discussed in this section. This comparison test is similar to comparison test 1. The setup for the basket option

			Plain Reverse AD		Structured AD			
Total MC	NBatch	Batch Size	Memory(MB)	Time(s)	Memory(MB)	Time(s)	Memory Ratio	Time Ratio
	1	100000	5636.02	159.674655			1	1
	10	10000			563.608	125.525408	9.999893543	1.272050476
100000	20	5000			281.808	127.666146	19.99950321	1.250720414
	50	2000			112.728	125.710322	49.99662905	1.270179349
	100	1000			56.484	134.951923	99.7808	1.18319
	1	80000	4508.812	107.7002			1	1
	10	8000			450.888	100.0016	9.9998	1.0770
80000	20	4000			255.476	100.4558	19.9969	1.0721
	50	1600			90.184	101.2327	49.99557	1.0639
	100	800			45.096	102.1406	99.9825	1.0544
	1	60000	3381.612	85.8128			1	1
	10	6000			338.168	86.1482	9.9998	0.9961
60000	20	3000			169.088	88.5867	19.9991	0.9687
	50	1200			67.64	86.6762	49.9942	0.9900
	100	600			33.8844	88.2671	99.8000	0.9722
	1	40000	2254.412	52.051197			1	1
	10	4000			225.452	48.782437	9.999520962	1.067006903
40000	20	2000			112.8	49.995203	19.98592199	1.041123825
	50	800			46.852	49.342586	48.11773243	1.054893981
	100	400			22.688	50.871891	99.36583216	1.023181879
	1	20000	1127.208	25.478715			1	1
	10	2000			112.728	24.466931	9.999361294	1.041353123
20000	20	1000			56.364	25.114317	19.99872259	1.014509572
	50	400			22.62	24.90138	49.83236074	1.02318486
	100	200			11.304	27.414264	99.71762208	0.929396281
	1	10000	563.608	14.314761			1	1
	10	1000			56.364	12.382822	9.999432262	1.156017667
10000	20	500			28.188	12.354023	19.99460763	1.15871251
	50	200			11.336	12.622057	49.7184192	1.134106826
	100	100			5.64	13.152183	99.93049645	1.088394299

Table 4.2: Basket Option with 5 Weighted Basket Options - HV2.0

			Plain Reverse AD		Structured AD			
Total MC	NBatch	Batch Size	Memory(MB)	Time(s)	Memory(MB)	Time(s)	Memory Ratio	Time Ratio
	1	100000	5636.18	142.0514			1	1
	10	10000			563.608	125.3115	10.0002	1.1336
100000	20	5000			281.816	125.0417	19.9995	1.1360
	50	2000			112.724	126.7981	49.9998	1.1203
	100	1000			56.388	126.4218	99.9536	1.1236
	1	80000	4508.816	106.2053			1	1
	10	8000			450.888	97.7114	9.9999	1.0869
80000	20	4000			255.448	97.969	19.9994	1.0841
	50	1600			90.244	99.4338	49.9625	1.0681
	100	800			45.092	99.7190	99.9915	1.0650
	1	60000	3381.608	78.7129			1	1
	10	6000			338.232	73.8844	9.9979	1.0654
60000	20	3000			169.092	73.3636	19.9986	1.0729
	50	1200			67.636	73.9170	49.9972	1.0649
	100	600			33.824	74.1921	99.9766	1.0609
	1	40000	2254.436	49.7433			1	1
	10	4000			225.444	49.7946	10.0000	0.9990
40000	20	2000			112.788	53.5537	19.9883	0.9288
	50	800			45.092	49.4490	49.9964	1.0059
	100	400			28.408	50.0092	79.3592	00.9947
	1	20000	1131.08	24.6128			1	1
	10	2000			112.724	24.6423	10.0341	0.9988
20000	20	1000			56.364	24.9646	20.0674	0.9859
	50	400			22.548	25.0055	50.1632	0.9843
	100	200			11.336	26.8979	99.7777	0.9150
	1	10000	563.608	12.6555			1	1
	10	1000			60.092	12.6814	9.3791	0.9980
10000	20	500			28.252	12.5079	19.9493	1.0118
	50	200			11.272	12.6677	50.0007	0.9990
	100	100			5.636	14.0885	100.0014	0.8983

Table 4.3: Asian Option with 10 Underlying Assets - HV2.0

			Plain Reverse AD		Structured AD			
Total MC	NBatch	Batch Size	Memory(MB)	Time(s)	Memory(MB)	Time(s)	Memory Ratio	Time Ratio
	1	100000	5636.00	138.42			1	1
	10	10000			563.61	120.91	9.9999	1.1448
100000	20	5000			281.86	122.76	19.9961	1.12764
	50	2000			112.73	124.7	49.997	1.11
	100	1000			56.388	124.44	99.951	1.1123
	1	80000	4508.8	102.04			1	1
	10	8000			450.89	97.182	9.9998	1.05
80000	20	4000			225.44	99.065	20.0000	1.03
	50	1600			90.244	98.065	49.962	1.0405
	100	800			45.092	99.235	99.991	1.0282
	1	60000	3381.6	75.499			1	1
	10	6000			338.17	74.569	9.9998	1.1013
60000	20	3000			169.08	73.814	20.000	1.0228
	50	1200			67.7	73.587	49.95	1.026
	100	600			33.888	74.599	99.788	1.0121
	1	40000	2254.4	49.198			1	1
	10	4000			225.51	49.611	9.997	0.9917
40000	20	2000			121.58	49.525	18.543	0.9934
	50	800			45.092	48.899	49.996	1.0061
	100	400			22.612	49.519	99.70	0.9935
	1	20000	1131.3	24.425			1	1
	10	2000			113.65	24.40	9.9542	1.001
20000	20	1000			56.424	25.1157	20.05	0.9725
	50	400			22.608	24.967	50.04	0.9783
	100	200			11.276	27.724	100.33	0.9495
	1	10000	563.61	12.271			1	1
	10	1000			56.428	12.224	9.9881	1.0038
10000	20	500			28.76	12.413	19.597	0.98861
	50	200			11.344	13.719	49.683	0.8945
	100	100			5.64	12.919	99.93	0.9499

Table 4.4: Best of Asian with 10 Underlying Assets - HV2.0

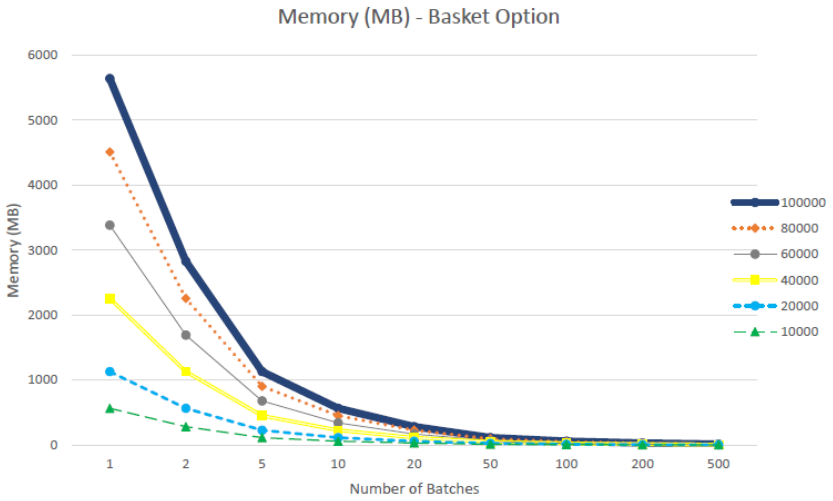

Figure 4.1: Peak Memory: Basket Options' Gradient Computation - HV2.0

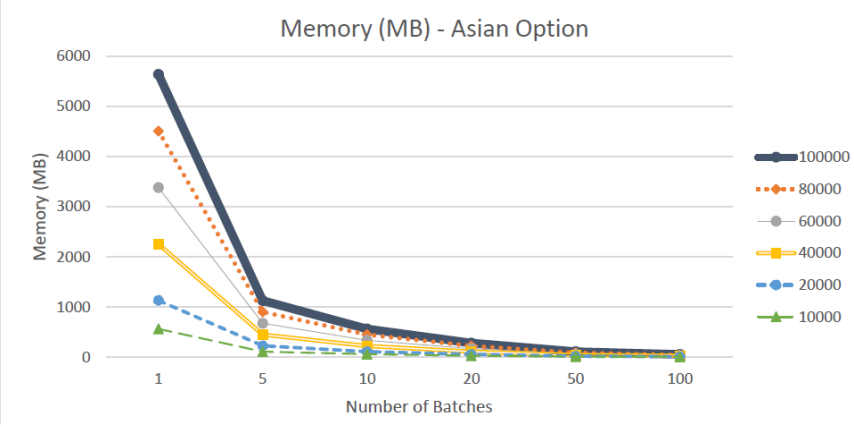

Figure 4.2: Peak Memory: Asian Options' Gradient Computation - HV2.0

computation was shown in (3.10) where $l = 1$ and $M = 1000, 2000, 3000, 4000, 5000$. Therefore, the portfolio of the basket options consists of one weighted basket and 1000-5000 underlying assets. Table 4.5 shows the memory usage for the calculation of basket option values and their gradients using the Monte Carlo method. The formula for the computation of the best of Asian option was in (3.12). The portfolio of the best of Asian options consists of 3000, 4000, and 5000 underlying assets. Table 4.6 shows the memory usage for the calculation of best of Asian option values and their gradients using the Monte Carlo method. Please note that the result for the Asian option is unavailable due to insufficient memory storage of the testing machine.

For each of the options, two batching structures and one unstructured (plain reverse mode) case were examined. The two batching structures are 100 MC paths with 100 batches and 100 MC paths with 50 batches. The plain reverse mode case is

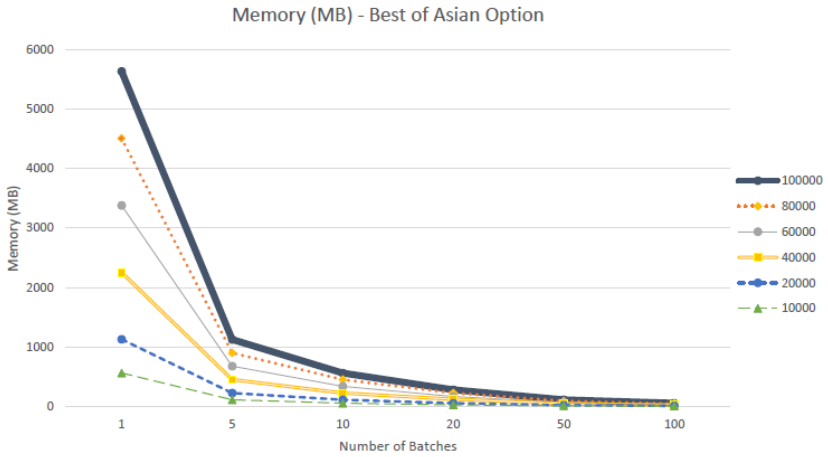

Figure 4.3: Peak Memory: Best of Asian Options' Gradient Computation - HV2.0

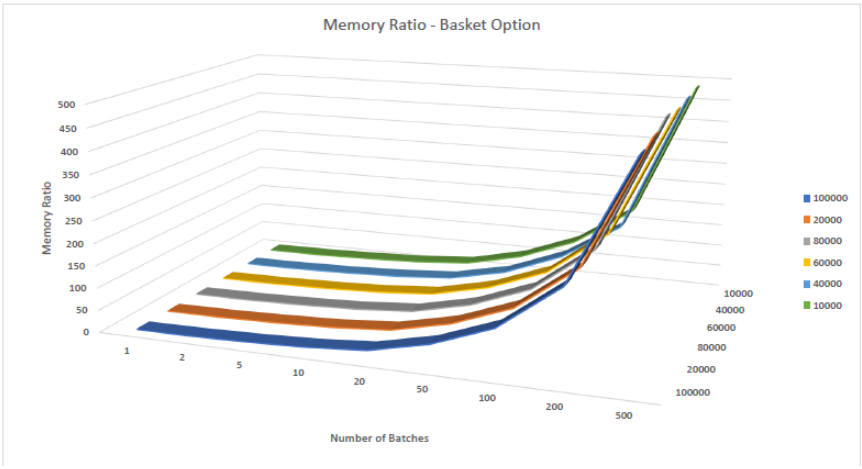

Figure 4.4: Memory Ratio: Basket Options' Gradient Computation - HV2.0

100 MC paths with 1 batch. As mentioned earlier, "batching" refers to as exploiting the GPS structure (i.e. clustering several paths together). In a more precise sense, there is a different meaning of "batches" between HV2.0 and FL2.0. In the highly vectorized case (HV2.0), number of batches is referred to as the size of the vector. In the for loop case (FL2.0), number of batches is referred to as number of loops to run in an AD session.

For the basket options, five number of assets were tested, namely 1000, 2000, 3000, 4000, and 5000. Each number of assets were examined with each of the three "batching" cases mentioned above. For the best of Asian options, three number of assets were tested, namely 3000, 4000, and 5000. Each number of assets were

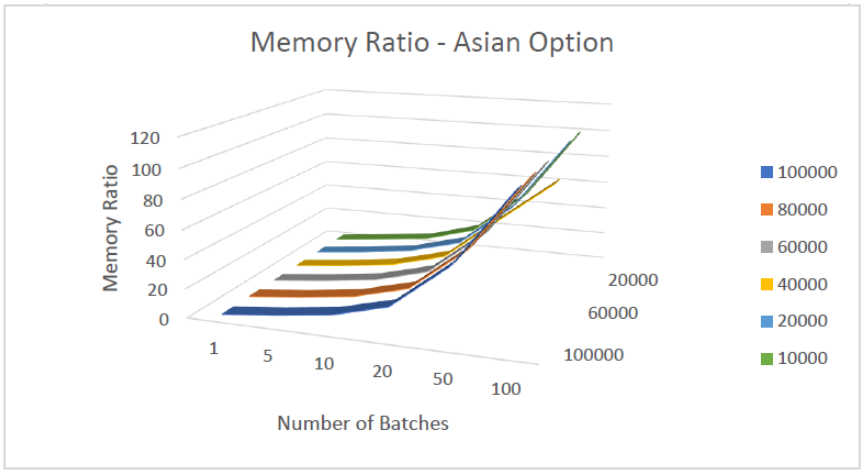

Figure 4.5: Memory Ratio: Asian Options' Gradient Computation - HV2.0

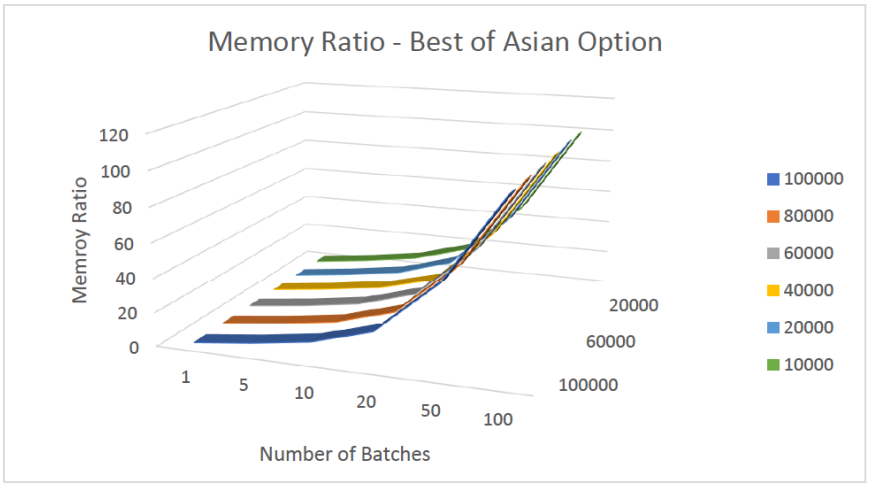

Figure 4.6: Memory Ratio: Best of Asian Options' Gradient Computation - HV2.0

examined with each of the three "batching" cases as well.

Before discussing the results, NAssets represents the number of assets in the portfolio in Table 4.5 and 4.6. Both Table 4.5 and 4.6 show that the memory usage reduces as the number of batches increases. Moreover, the memory usage reduced approximately equals to the number of batches. These results are consistent with the results of comparison test 1 in the last section. In short, the memory usage is reduced because the intermediate information is overwritten after each AD session due to the GPS structure. Consider Table 4.5 as an example, for 1000 assets , the memory usage for plain reverse AD (unstructured: 100 mcpaths and 1 batch) was 1238.5 MB and the memory usage for structured AD (100 mcpaths and 100 batches) was only 12.648 MB. The memory usage was reduced by 100 times which equals to

Memory(MB) - FL2.0 Basket Options			
NAssets	100 mcpaths/100 batches	100 mcpaths/50 batches	100 mcpaths/1 batch (plain)
1000	12.648	20.54	1238.5
2000	25.896	55.512	2444.94
3000	33.932	67.944	3416.4
4000	45.384	90.996	4546.75
5000	57.64	114.904	5672.66

Table 4.5: Memory Usage of FL2.0 structured vs FL2.0 unstructured - Basket Options

the number of batches. This pattern holds for all total number of assets, batching combinations, and both types of exotic options examined. Table 4.6 shows the results for best of Asian options. Comparison test 2 shows consistent results as comparison test 1. Whether the underlying user code is highly vectorized or for loop, the GPS structure still improves the space efficiency. Therefore, the performance of structured AD (GPS) is independent of the implementation of the underlying user code.

Figure 4.7 and 4.8 show the memory usage of different number of assets and batching combinations. The x-axis and y-axis represent the number of assets and memory usage in MB, respectively. The bars with the diagonal line pattern represent the memory usage of the plain reverse mode (unstructured: 100 mcpaths and 1 batch). The solid and dotted-patterned bars represent the memory usage of the following GPS structured combinations: 100 mcpaths x 50 batches and 100 mcpaths x 100 batches, respectively. Both figures reveal that the plain reverse mode requires significantly more space than the other two structured combinations.

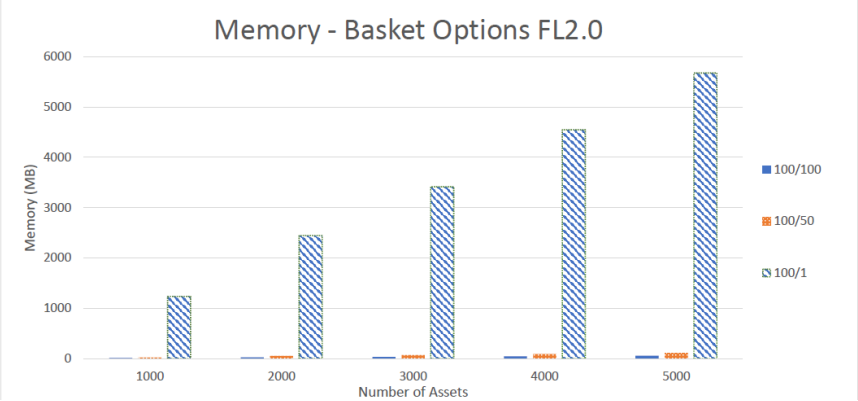

Figure 4.7: Peak Memory: FL2.0 Basket Options

Memory(MB) - FL2.0 Best of Asian Options			
NAssets	100 mcpaths/100 batches	100 mcpaths/50 batches	100 mcpaths/1 batch (plain)
3000	35.204	68.964	3421.56
4000	45.828	91.872	4549.83
5000	58.368	113.432	5685.2

Table 4.6: Memory Usage of FL2.0 structured vs FL2.0 unstructured - Best of Asian Options

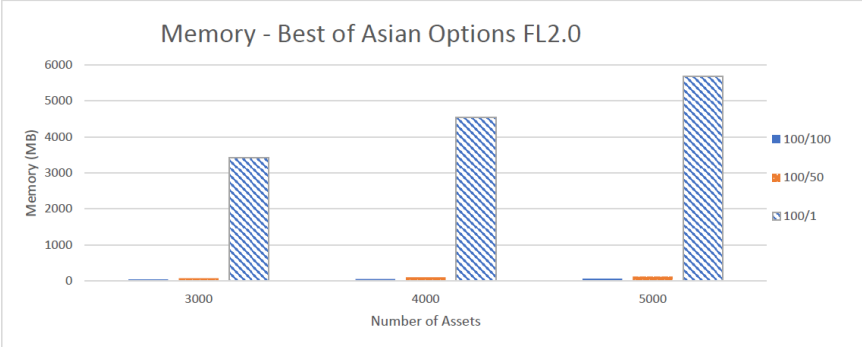

Figure 4.8: Peak Memory: FL2.0 Best of Asian Options

4.4 Comparison Test 3 Results

The focus of this section is on time efficiency. The performance of FL2.0 (ADMAT) and MADO will be examined and compared. The performance of AD as derivative computation is affected by two factors: AD algorithm and the underlying user code. The time required by the user code to evaluate option values is called the primary time. The time required by the AD algorithm for derivative computation is called the derivative time. The time efficiency performance of the AD package is measured by the time ratio between derivative time and primary time. That is, how much more expensive the derivative computation is compared with the user code.

The setup for the basket options computation was shown in (3.10) where $l = 1$ and $M = 10, 100, 500, 1000, 2000, 3000, 4000, 5000$. Table 4.7 shows the time ratio results for the basket options. The formulas for the computation of the Asian options and best of Asian options were presented in (3.11) and (3.12), respectively. The portfolio of the Asian options and best of Asian options also consist of 10, 100, 500, 1000, 2000, 3000, 4000, and 5000 underlying assets. Table 4.8 and 4.9 show the the time ratio results for the Asian options and best of Asian options, respectively.

As mentioned in Section 4.1, FL2.0 will be referred to as ADMAT in this section. NAssets represents the number of assets in the portfolio in Table 4.7, 4.8, and

4.9. For each of the experiments, three total MC paths were examined. All of them exploited the GPS structure and had only 1 path per batch. For instance, 100admat means 100 mc paths with 100 batches, 200admat means 200 mc paths with 200 batches, 300admat means 300 mc paths with 300 batches. Similarly for the experiment MADO, 100mado means 100 mc paths with 100 batches, 200mado means 200 mc paths with 200 batches, and 300mado means 300 mc paths with 300 batches. For each of the experiments, the following combinations (number of assets x mc paths) were tested: 10x100admat, 10x200admat, 10x300admat, 10x100mado, 10x200mado, and 10x300mado, so on and so forth.

Table 4.7, 4.8, and 4.9 show that the time ratios of MADO are significantly less than the corresponding time ratios produced by ADMAT. That means the derivative computation of MADO is less expensive than the derivative computation of ADMAT. In other words, MADO is faster than ADMAT in these examples. The reason is that ADMAT implements AD by function overloading. It is a dynamic approach, i.e. differentiate user code at runtime. On the other hand, MADO implements AD by code generation. It is a static approach, i.e. parsing and processing the computational graph before runtime. Since MADO generates derivative code before runtime, the generated code has less to deal with. Thus, MADO is faster than ADMAT in these examples. Since the time ratios slightly fluctuate, these time ratios are average results of five runs. Consider Table 4.7 as an example, for 10 assets, the time ratio produced by 100admat is 99.59 while the time ratio produced by 100mado is only 19.99. That means ADMAT required 99.59 times more than the primary time to compute its' derivative. And MADO took only 19.99 times more than the primary time to compute its' derivative. In other words, MADO is five times faster than ADMAT for this combination. Similarly, for 1000 assets, the time ratio of 200admat is 15.38 while the time ratio of 200mado is only 4.57. Hence, MADO is roughly three times faster than ADMAT for this combination. In Table 4.8, some time ratios of the Asian options are unavailable due to insufficient memory storage of the test machine. These cells have been labeled as o.f.m. (out of memory).

Figure 4.9, 4.10, and 4.11 show the time ratios for different number of assets and MC paths combinations. The x-axis and y-axis represent the number of assets and time ratio, respectively. The solid lines represent the time ratios produced by MADO. The dotted lines represent the time ratios produced by ADMAT. As shown in all three graphs, ADMAT exhibits a much higher ratio than MADO. Again, that means MADO is faster than ADMAT as an AD package in these examples.

Note that in Figure 4.9 and 4.11, time ratios started out with a large difference between ADMAT and MADO, but they started to merge together to a similar time ratio as number of asset increases. This merging trend is due to the size of the asset vector. The AD overhead becomes less significant as vector becomes larger. Thus, package difference becomes less significant as well. Figure 4.12 and 4.13 are the close up plots of Figure 4.9 and 4.11, respectively. They illustrate the time ratio results for

Time Ratio - Basket Options						
	FL2.0(ADMAT)			MADO		
NAssets	100admat	200admat	300admat	100mado	200mado	300mado
10	99.59	114.05	137.07	19.99	26.41	26.98
100	74.59	78.93	75.95	16.29	15.63	16.1
500	23.57	24.73	23.83	6.16	6.42	6.34
1000	14.59	15.38	14.8	4.54	4.57	4.53
2000	7.92	7.3	8.98	3.09	3.46	3.51
3000	7.19	7.16	7.1	3.37	3.22	3.23
4000	6.09	6.2	6.16	3.34	3.27	3.19
5000	5.59	5.67	5.65	3.15	3.2	3.21

Table 4.7: Time Ratio of FL2.0 (ADMAT) vs MADO - Basket Options

Time Ratio - Asian Options						
	FL2.0(ADMAT)			MADO		
NAssets	100admat	200admat	300admat	100mado	200mado	300mado
10	144.57	135.38	145.47	32.25	28.71	31.39
100	118.99	126.55	128.5	35.36	36.08	37.07
500	321.75	3162.28	309.62	217.65	212.76	204.91
1000	o.f.m.	o.f.m.	o.f.m.	473.18	o.f.m.	o.f.m.
2000	o.f.m.	o.f.m.	o.f.m.	1000.61	o.f.m.	o.f.m.
3000	o.f.m.	o.f.m.	o.f.m.	1762.05	o.f.m.	o.f.m.
4000	o.f.m.	o.f.m.	o.f.m.	2384.38	o.f.m.	o.f.m.
5000	o.f.m.	o.f.m.	o.f.m.	o.f.m.	o.f.m.	o.f.m.

Table 4.8: Time Ratio of FL2.0 (ADMAT) vs MADO - Asian Options

"large-scale" problems, i.e. 1000-5000 assets. Figure 4.14 and 4.15 show the memory usage for the basket options and best of Asian options, respectively. The x-axis and y-axis represent the number of assets and memory usage (MB), respectively. As mentioned, 100admat (solid lines) represents 100 paths with 100 batches using ADMAT. 100mado (dotted lines) represents 100 paths with 100 batches using MADO. Both plots show that MADO requires less space than ADMAT. In other words, MADO is both faster and more space efficient than ADMAT in these examples.

Time Ratio - Best of Asian Options						
	FL2.0(ADMAT)			MADO		
NAssets	100admat	200admat	300admat	100mado	200mado	300mado
10	135.3	145.92	144.83	30.34	30.67	31.3
100	75.28	77.89	76.18	16.87	17.94	17.42
500	20.61	24.93	22.93	5.97	7.05	6.42
1000	14.46	17.06	14.08	4.55	5.11	4.64
2000	9.14	9.34	9.07	3.77	3.8	3.71
3000	8.38	7.35	7.49	4.17	3.58	3.63
4000	6.63	6.76	6.63	3.74	3.84	3.78
5000	6.06	6.1	6.12	3.69	3.75	3.75

Table 4.9: Time Ratio of FL2.0 (ADMAT) vs MADO - Best of Options

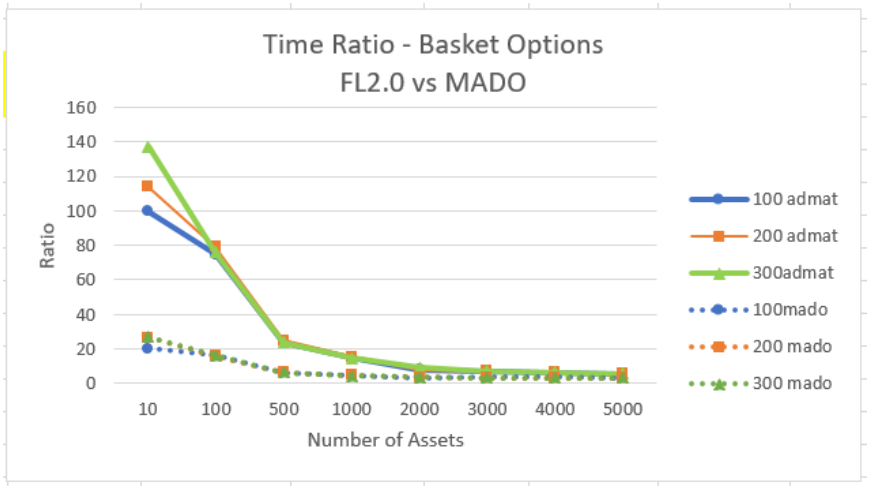

Figure 4.9: Time Ratio: FL2.0 (ADMAT) vs MADO - Basket Options

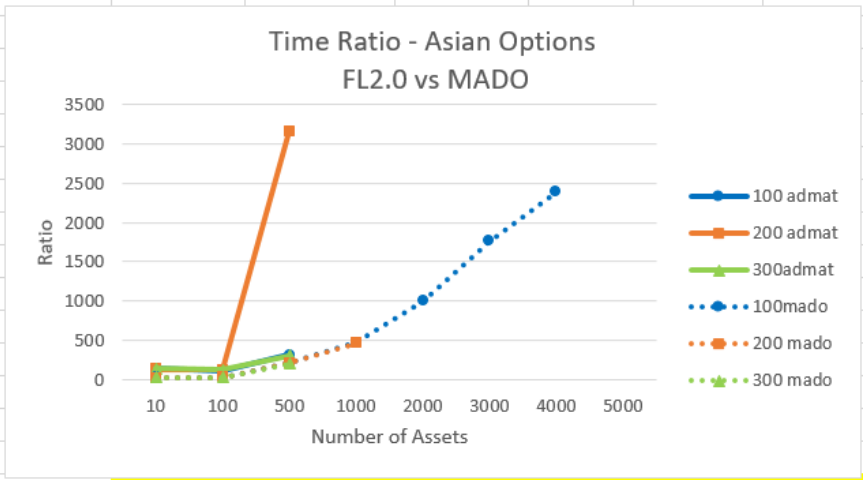

Figure 4.10: Time Ratio: FL2.0 (ADMAT) vs MADO - Asian Options

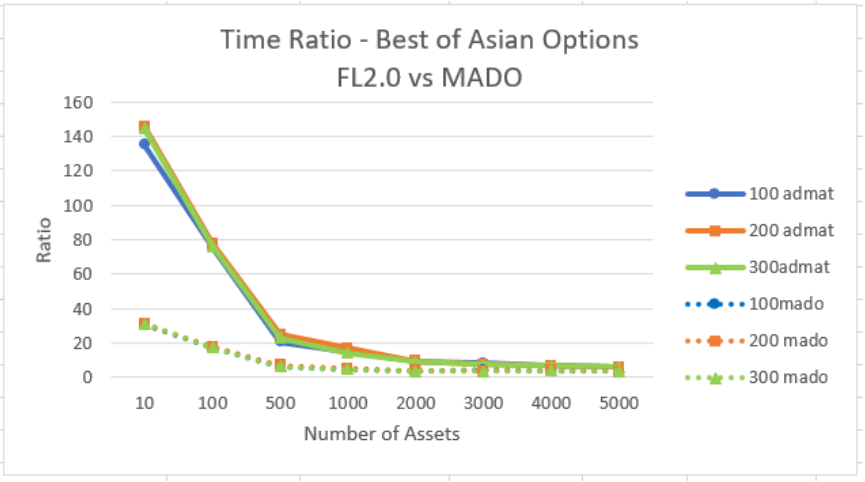

Figure 4.11: Time Ratio: FL2.0 (ADMAT) vs MADO - Best of Asian Options

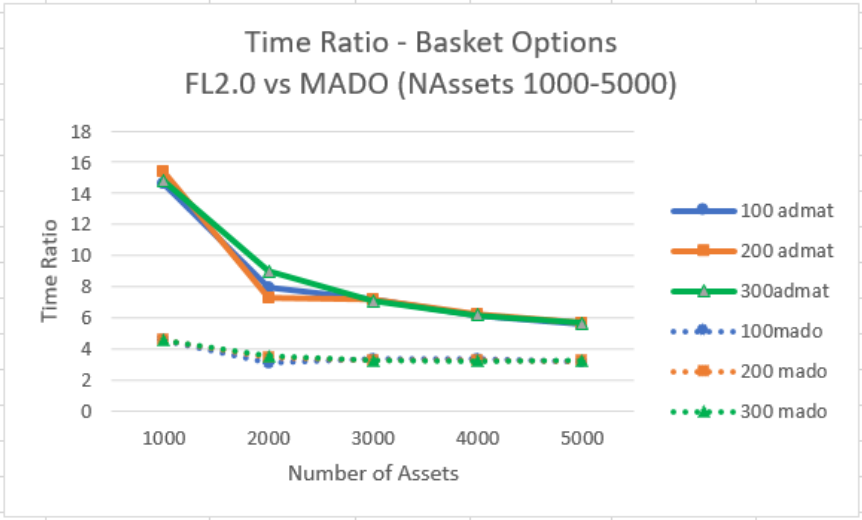

Figure 4.12: Time Ratio NAssets 1000-5000: FL2.0 (ADMAT) vs MADO - Basket Options

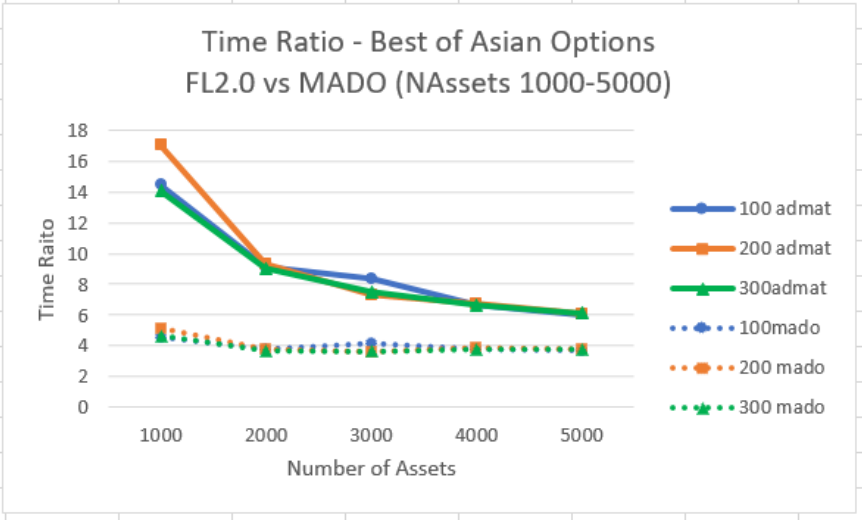

Figure 4.13: Time Ratio NAssets 1000-5000: FL2.0 (ADMAT) vs MADO - Best of Asian Options

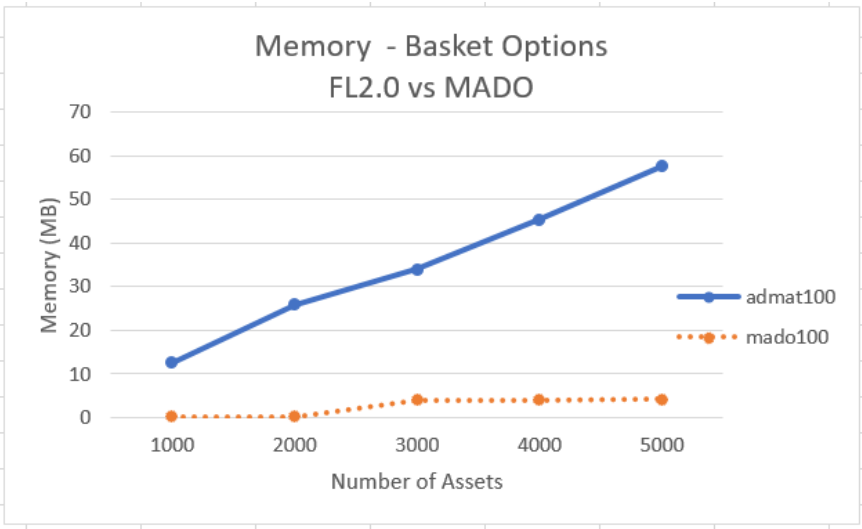

Figure 4.14: Memory: FL2.0 (ADMAT) vs MADO - Basket Options

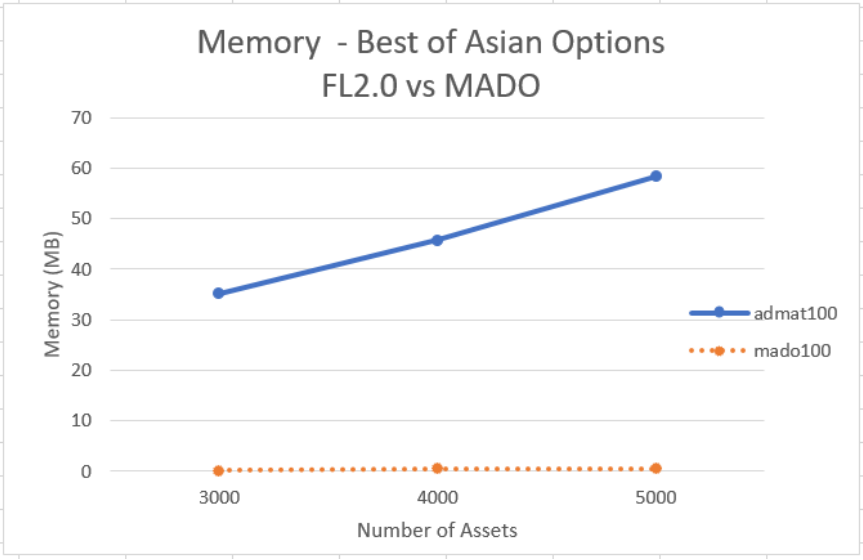

Figure 4.15: Memory: FL2.0 (ADMAT) vs MADO - Best of Asian Options

Chapter 5

Conclusion

In summary, structured AD is an efficient tool for derivative computations. Three experiments and comparison tests were done to show the effectiveness of structured AD for determining Greeks of exotic options in a Monte Carlo framework. The first experiment was called the HV2.0. It incorporated the ADMAT 2.0 algorithm, a highly vectorized underlying user code, and exploited the GPS structure. The second experiment was called the FL2.0 (ADMAT). It employed the ADMAT 2.0 algorithm, a for loop underlying user code, and exploited the GPS structure. The last experiment was called the MADO. It used the MADO algorithm, a for loop underlying user code, and exploited the GPS structure.

Three comparison tests were performed. The first comparison test compared the memory usage between HV2.0 structured and HV2.0 unstructured (plain reverse mode). The results show that the memory required for the gradient computations was significantly reduced after exploiting the GPS structure. Furthermore, the memory usage reduced is proportional to the number of batches. It is because the GPS structure allows the intermediate information to be overwritten after each AD session.

The second comparison test compared the memory usage of FL2.0 structured versus FL2.0 unstructured. This comparison test is similar to the first comparison test. The only difference is that HV2.0 employs a highly vectorized user code while FL2.0 uses a for loop user code. Comparison test 2 shows consistent results as comparison test 1. Whether the underlying code is highly vectorized or for loop, the GPS structure still improves the space efficiency. Thus, the performance of structured AD is independent of the implementation of the underlying user code.

The last comparison test compared the time efficiency between FL2.0 (ADMAT) and MADO. The results show that the time ratios of MADO are significantly less than the corresponding time ratios produced by ADMAT. That is, the derivative computation of MADO is faster than that of ADMAT in these examples.

References

[1] Xi Chen, Wei Xu, and Thomas F. Coleman. *The Efficient Application of Automatic Differentiation for Computing Gradients in Financial Applications*. J. Comput. Finance, 2014.

[2] Thomas F. Coleman and G.F. Jonsson. *The efficient computation of structured gradients using automatic differentiation*. SIAM Journal on Scientific Computing, Vol. 20, 1999, pp. 1430-1437.

[3] Thomas F. Coleman and Wei Xu. *Fast (Structured) Newton Computations*. SIAM Journal on Scientific Computing, 31.2, 2008. pp. 1175-1191.

[4] Thomas F. Coleman and Wei Xu. *Automatic Differentiation in MATLAB Using ADMAT with Applications*. SIAM, Philadelphia, Pennsylvania, 2016.

[5] Michael Giles and Paul Glasserman. *Computation Methods: Smoking Adjoints: Fast Monte Carlo Greeks*. Risk-London-Risk Magazine Limited, 19.1, 2006, pp. 92.

[6] John C. Hull. *Options, Futures and Other Derivatives*. Prentice Hall, Philadelphia, Pennsylvania, 6th edition, 2005.

[7] Wanqi Li. *MADO Editor User Manual*. http://github.com/vanchi7/mado-editor(poundsign)generate-derivative-code, 2017.

[8] Pablo Olivares and Alexander Alvarez. *Pricing Basket Options by Polynomial Approximations*. Journal of Applied Mathematics, 2016, Article ID 9747394.

[9] W. Murray P.E. Gill and M.H. Wright. *Practical Optimization*. Academic Press, New York, 1982.

[10] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.