Structured Reverse Mode Automatic Differentiation in Nested Monte Carlo Simulations

by

An Zhou

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2017

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

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

Abstract

In many practical large scale computational problems, the calculation of partial derivatives of the object function f with respect to input parameters are entailed and the dimension of inputs n is much larger the one of outputs m. The use of reverse mode automatic differentiation (AD) is mostly efficient as it computes the gradient in the same amount of runtime as f regardless of the input dimension n. However, it demands excessive memory. To enjoy the runtime efficiency of reverse mode without paying unaffordable memory, structured reverse mode has been proposed and succeeded in several applications. Due to the fundamental difficulty in automatic structure detection, structured reverse mode has not been fully automated. This thesis, instead of trying to solve to structure detection problem for a completely generic piece of code, is devoted to the analysis and implementation of deploying structured reverse mode to a generic class of problems with a known structure, nested Monte Carlo simulations. We reveal the general structure pattern of Monte Carlo simulations in financial applications. Space/time tradeoff on deploying structured reverse mode are discussed in details and numerical experiments using Variable Annuity program are conducted to corroborate the analysis. Significant memory and runtime reductions are observed. We argue such contribution is important as nested Monte Carlo simulations accommodates several large scale computations in financial services that are crucial in practice.

Acknowledgements

I would like to thank all people who made this possible.

Special thanks to my supervisor Thomas F. Coleman for continuous support and guidance, and generosity of providing the license of the commercial Automatic Differentiation Matlab package: ADMAT 2.0.

Special thanks for Global Risk Institute for supporting the research involving Variable Annuities where part of this thesis is completed.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Background and Motivation

In various large scale computations, the target function is a scalar or of low dimensional with a much higher dimensional input, and derivatives calculation is entailed. In such scenarios, reverse mode automatic differentiation (AD) becomes the unique choice due to its superior runtime efficiency over alternative approaches such as forward mode AD and finite difference. However, reverse mode's excessive memory requirement may impede its deployment for large scale computations. Structured reverse mode, an general approach to relax reverse mode's memory requirement while maintaining reverse mode's celebrated runtime efficiency, has been proposed but yet to be automated. Case dependent studies have proven structured reverse mode's effectiveness, yet there has not been any systematic investigations on its deployment to a generic class of computational problems, nested Monte Carlo simulations. In this thesis, we reveal structures of nested Monte Carlo simulations and research how to best utilize such generic structure pattern to achieve efficient reverse mode computation. Numerical experiments using Variable Annuities are conducted and significant memory reduction are found in accordance with theoretical analysis.

1.1 Why Automatic Differentiation?

1.1.1 Importance and cost of Derivatives

In numerous computational applications, in addition to the function value of a continuous function f, its (partial) derivatives with respect to the inputs are also required.

In optimization where the target function is to be minimized, we use derivatives (gradient and Hessian) to sketch the local profile of the optimized function, guiding the algorithm towards the next iterative point. The vanishing of first order derivatives signals a local extremum and the positive definiteness of the second order derivatives confirms the genuineness of the local minimum. The use of derivatives is ubiquitous in non-linear optimization.

In financial applications, the target function is usually the present value of a financial instrument, whether obtained from PDE approach or Monte Carlo simulation. There we need its derivatives with respect to a list of parameters, such as current price, volatility, interest rate, time etc., to hedge away the corresponding risk exposures they represent. Derivative information is often needed for the sake of risk management and reporting purposes as well.

As crucial as derivatives are among those applications, the calculation of derivatives takes a longer time to run than the original function itself, whose run time may already be prohibitive. To quantify that the cost of computing the derivatives, the following notations are introduced:

Definition 1.1. $\omega(c)$ stands for the runtime of the source code c in a default environment.

Definition 1.2. $\omega(\partial_a c)$ stands for the runtime to calculate the derivatives of the differentiable function f_c, which the source code c represents, using approach a in a default environment.

The phrase 'in a default environment' is crucial as runtime critically depends on the programming languages used, the computational power of the machine and also the software packages deployed.

Definition 1.3. The *Runtime Ratio* of an approach a to calculate the derivatives of the differentiable function f_c that a source code c represents is defined as:

$$T_a(c) \equiv \frac{\omega(\partial_a c)}{\omega(c)} \tag{1.1}$$

Such a ratio measure removes most language and machine dependency. Suppose f_c has n^c scalar inputs and m^c scalar outputs, i.e. $f_c : \mathcal{R}^{n^c} \to \mathcal{R}^{m^c}$. By 'derivatives', we mean the Jacobian $J_{ij} = \frac{\partial z_i}{\partial x_j}, i \in \{1, \cdots, m^c\}, j \in \{1, \cdots, n^c\}, z = f_c(x)$. [1]

[1]Depending on the problem, sometimes only derivatives with respect to a subset of input variables are needed. In such case, we consider the input variables whose derivatives with respect to are not requested to be *parameters* that we pass to the function, rather than part of the 'input'.

Finite difference (FD) calculates approximations to the derivatives. However, FD only calculates derivatives with respect to one input at each run, i.e.:

$$T_{FD}(c) \approx \begin{cases} n^c, & \text{Forward/Backward difference} \\ 2n^c, & \text{Central difference} \end{cases} \tag{1.2}$$

When the dimension of the input is too large–larger than 10^3 for example–using FD to obtain derivatives becomes impractical. One famous example is the stagnation of the artificial neural network field in the 70s after Marvin Minsky and Seymour Papert published *Perceptrons: An Introduction to Computational Geometry* where they 'showed' the training of deep neural networks is practically impossible due to large number of inputs. For instance, modern deep neural networks can have parameters up to 10^7. For complex financial applications such as LIBOR model, $10^4 - 10^6$ inputs scalars of which derivatives are requested are common. When the original function already has a prohibitive runtime, obtaining the derivatives with runtime ratio more than 10^4 is unacceptable. Reverse mode AD rides to the rescue.

1.1.2 Reverse mode Automatic Differentiation

AD calculates the derivatives exactly and automatically[1]. Furthermore, reverse mode AD[2] computes the gradient within constant runtime ratio, regardless of number of inputs.

In Griewank's *On Automatic Differentiation*[3], it reads:

> "...Thus we can conclude that under quite realistic assumptions the evaluation of a gradient requires never more than **five times** the effort of evaluating the underlying function itself..."

It is such guaranteed $O(1)$ runtime ratio scaling that opens the doors for efficient derivative calculations in large scale optimizations[4] and financial instrument pricing[5][6][7]. More precisely:

$$T_{\text{Rev}}(c) = \chi(c) \tag{1.3}$$

where $\chi(c)$ is a factor that depends on the details of c however is upper-bounded as Griewank stated: $\chi \leqslant 5$. The crucial difference with $T_{FD}(c)$ is that it scales with m^c

but not n^c.

We should note that gradient means specifically the derivatives of a scalar function. Indeed, for most applications, target function whose derivatives are of interest is scalar. In optimization, f is scalar simply because only scalar has natural order therefore can be optimized. In finance, the present value of an instrument is a scalar as well.

1.2 Why Structured Reverse Mode?

1.2.1 Bottleneck of Reverse Mode in large scale computation

Reverse mode calculates derivatives with respect to all inputs at once. However, this is at the cost of memory: a typical space/runtime tradeoff. Reverse mode literally evaluates the derivatives of the original function in a backwards fashion which requires saving all the intermediate variables, potentially demanding a much larger memory than the one needed for the original function evaluation.

Definition 1.4. $\sigma(c)$ is the amount of memory needed for executing c, in units of bytes, in a default environment.

Definition 1.5. $\sigma(\partial_a c)$ is the amount of memory needed for computing the derivatives of the function f_c, which the source code c represents, using approach a in a default environment.

Similar to runtime, the size of memory depends on the programming language used[2].

Definition 1.6. The *Space ratio* of an approach a to calculate the derivatives for the differentiable function f_c that a source code c represents is defined as:

$$S_a(c) \equiv \frac{\sigma(\partial_a c)}{\sigma(c)} \tag{1.4}$$

Such a ratio measure largely removes the programming language dependency.

[2]As basic data types are implemented differently and different memory management schemes (static or dynamic) might be used.

The bottleneck for reverse mode is that such space ratio $S_{\text{Rev}}(c)$ as defined above can be excessive, typically scaling linearly with the depth of the original function. Such memory requirement is the bottleneck to apply reverse mode for modern large scale computations, motivating the birth of structured reverse mode.

1.2.2 Structured Reverse Mode

To tackle the extensive space requirements that reverse mode suffers, systematic approaches have been developed [8][9][10]. The space requirements of reverse mode comes from the fact that all intermediate variables during the entire computation need to be saved such that we can propagate the derivatives backwards. To release spaces, only a subset of variables are stored, i.e. checkpoints, which are chosen such that original computational graph can be recovered as we propagate the derivatives. Such method is called 'checkpointing' or 'structured reverse mode', since observation of the structural pattern of f's source code is critical for the approach to be effective.

Checkpointing can significantly relax the space requirements of reverse mode at a marginal cost of runtime and can be deployed recursively. For 1 level of checkpointing, denoted as approach "StrRev", only 1 additional run of f is done. We have:

$$\omega(\partial_{\text{StrRev}}c) = \omega(\partial_{\text{Rev}}c) + \omega(c) \tag{1.5}$$

hence:

$$T_{\text{StrRev}}(c) = T_{\text{Rev}}(c) + 1 = 1 + m^c \cdot \chi(c) \tag{1.6}$$

The additional function run reduces the memory to:

$$\sigma(\partial_{\text{StrRev}}c) \sim \sqrt{\sigma(\partial_{\text{Rev}}c)\sigma(c)} \tag{1.7}$$

hence:

$$S_{\text{StrRev}}(c) \sim \sqrt{S_{\text{Rev}}(c)} \tag{1.8}$$

The bigger picture is: reverse mode obtains great time efficiency at the cost of space. Yet it is an extreme case of time/space tradeoff. Structured reverse mode offers a generalized spectrum of the same tradeoff with naive reverse mode as a special case. What is remarkable is that by increasing T by 1, S is reduced exponentially compared to straight forward reverse mode.

1.3 How to structure Reverse Mode

Structured Reverse Mode seems to be the holy grail for large scale computation: it resolves the memory overflow issue of reverse mode and maintain its powerful runtime efficiency. However, the performance of structured reverse mode depends on what structure is deployed, in other words, how checkpoints are selected. There's no exploitable structure for a generic piece of code and for certain codes, though unlikely a practical one, that the structured reverse mode does not improve over reverse mode itself at all.

People have made progress on algorithms that automatically find optimal positions of checkpoints to truly automate structured reverse mode [11][12][13]. Unfortunately, the automation has yet to achieve full generality. So far, even if a natural checkpointing of c exists, there's no guarantee that the algorithm would find it.

Motivation of this thesis: Instead of focusing on algorithms that can automatically detects code structure, we research on how to best deploy structured reverse mode with a given structure. Such structure is general, and in fact, for any nested simulations in financial applications. Variable Annuities, as a practical example, has the according structure and will be used for numerical experiments.

1.4 Guide to the reader

The thesis is organized in the following way: Chapter I motivates the thesis; Chapter II presents the idea of structured reverse mode automatic differentiation and discusses how different uses of structure affect the tradeoff between space and runtime; Chapter III discusses the use of Monte Carlo simulations in financial applications, its rate of efficiency and its structure; Chapter IV provides the technical background for Variable Annuities program and reveal its structure; Chapter V, the core part of the thesis, is dedicated to the deployment of structured reverse mode in nested simulation with theoretical analysis and numerical experiments; Appendix contains the technical details that are entailed to make the thesis self-contained.

Chapter 2

Structured Reverse Mode

In this chapter, we explain the basic framework and computational aspects of AD, and introduce the idea and mathematics of structured reverse mode using a simple uniform sequential program.

2.1 Understanding Reverse Mode

AD comes from the observation that any code at runtime[1] is an ordered collection of basic analytic operations. The derivatives of the basic operations[2] are easy to compute. If we compute the derivatives for all the basic operations involved and process them as dictated by the *chain rule*, we shall arrive at the exact derivatives. Let's use such a framework to decompose a given source code c.

2.1.1 Decomposition of source code

For an arbitrary given source code c of a differentiable function $f : \mathcal{R}^n \to \mathcal{R}^m$, denote its input as $x \in \mathcal{R}^n$ and output as $z \in \mathcal{R}^m$. The code at runtime can be decomposed

[1]Notice the clause "at runtime" is crucial. Codes typically have loops and conditional expressions. We don't know exactly what part of the code will be executed, how many times that a particular snippet of code would be run, and what are the specific order of execution before completing the code's execution. What we are referring to is the resulting computations of the source code.

[2]Such as add, subtract, multiply, divide, trigonometry functions, exponential, logarithm etc.

chronologically as:

$$\begin{cases} y_1 & = b_1(\text{Arg}_1) = b_1(x) \\ y_2 & = b_2(\text{Arg}_2) \\ & \vdots \\ y_{k-1} & = b_{k-1}(\text{Arg}_{k-1}) \\ z & = b_k(\text{Arg}_k) \end{cases} \tag{2.1}$$

k is the total number of basic operations that f is composed of, which depends on the input x in general. Arg_i stands for the set of *non-constant*[3] arguments given to the elementary function b_i (b *stands for basic*). Every b_i is a basic operation, hence it takes either 1 or 2 arguments.

With the above understanding, we have:

$$\forall i \in \{1, \cdots, k\} \begin{cases} |\text{Arg}_i| \in \{1, 2\} \\ \text{Arg}_i \in \{x, y_1, \cdots, y_{i-1}\} \end{cases} \tag{2.2}$$

[3]We don't consider constants as intermediate variables y. By constant, we mean anything that does not depend on x.

2.1.2 Calculation of Jacobian

When AD is invoked, derivatives of all the basic operations $\{b\}$ are calculated and assigned properly in the following extended Jacobian matrix J_E of $\{b_i\}_{i=1}^k$:

$$J_E \equiv \begin{pmatrix} \frac{\partial b_1}{\partial x} & -I & O & \cdots & \cdots & \cdots & O \\ \frac{\partial b_2}{\partial x} & \frac{\partial b_2}{\partial y_1} & -I & \ddots & \cdots & \cdots & \vdots \\ \frac{\partial b_3}{\partial x} & \frac{\partial b_3}{\partial y_1} & \frac{\partial b_3}{\partial y_2} & -I & \ddots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \frac{\partial b_{k-2}}{\partial x} & \frac{\partial b_{k-2}}{\partial y_1} & \frac{\partial b_{k-2}}{\partial y_2} & \cdots & \frac{\partial b_{k-2}}{\partial y_{k-3}} & -I & O \\ \frac{\partial b_{k-1}}{\partial x} & \frac{\partial b_{k-1}}{\partial y_1} & \frac{\partial b_{k-1}}{\partial y_2} & \cdots & \cdots & \frac{\partial b_{k-1}}{\partial y_{k-2}} & -I \\ \frac{\partial b_k}{\partial x} & \frac{\partial b_k}{\partial y_1} & \frac{\partial b_k}{\partial y_2} & \cdots & \cdots & \cdots & \frac{\partial b_k}{\partial y_{k-1}} \end{pmatrix} \equiv \begin{pmatrix} A & B \\ C & D \end{pmatrix} \tag{2.3}$$

$$\begin{aligned} A &= \left((\tfrac{\partial b_1}{\partial x})^T, \cdots, (\tfrac{\partial b_{k-1}}{\partial x})^T \right)^T \\ B &= \begin{pmatrix} -I & O & \cdots & \cdots & \cdots & O \\ \frac{\partial b_2}{\partial y_1} & -I & \ddots & \cdots & \cdots & \vdots \\ \frac{\partial b_3}{\partial y_1} & \frac{\partial b_3}{\partial y_2} & -I & \ddots & \cdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \frac{\partial b_{k-2}}{\partial y_1} & \frac{\partial b_{k-2}}{\partial y_2} & \cdots & \frac{\partial b_{k-2}}{\partial y_{k-3}} & -I & O \\ \frac{\partial b_{k-1}}{\partial y_1} & \frac{\partial b_{k-1}}{\partial y_2} & \cdots & \cdots & \frac{\partial b_{k-1}}{\partial y_{k-2}} & -I \end{pmatrix} \\ C &= \frac{\partial b_k}{\partial x} \\ D &= \begin{pmatrix} \frac{\partial b_k}{\partial y_1} & \cdots & \frac{\partial b_k}{\partial y_{k-1}} \end{pmatrix} \end{aligned} \tag{2.4}$$

Notice that x, y, $b_{(\cdot)}$ might be vectors hence each block in J_E, A, B, C, D might be matrices. I and O are identity matrix and matrix of all zero of appropriate size. Denoting n_i as y_i's dimension, i.e. b_i's output dimension, $\frac{\partial b_i}{\partial y_j}$ would be a $n_i \times n_j$ matrix. B is lower block triangular and has identity matrix along the diagonals, hence non-singular. B is also sparse for each b takes at most 2 arguments, i.e. each row has no more than 3 non-zero elements. By definition:

$$\begin{cases} Adx + Bd\{y\} &= 0 \\ Cdx + Dd\{y\} &= dz \end{cases} \Rightarrow dz = (C - DB^{-1}A)dx \equiv Jdx \tag{2.5}$$

$$\{y\} = \left(y_1^T, \cdots, y_{k-1}^T \right)^T$$

where we have denoted y in column vectors forms. Such computation is related to the Schur complement of J_E's submatrix. Hence the derivatives are computed as:

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

2.1.3 Intuitive example

Readers that are familiar with reverse mode may skip this section. Consider a simple sequential code c^* that calculates f^* with the following structure:

$$\text{Arg}^*_{i+1} = \begin{cases} \{x\} & i = 0 \\ \{y^*_i\} & \text{otherwise} \end{cases} \tag{2.7}$$

We will use * superscript to indicate reference to the special sequential case of f^*. Due to the structure in Arg^*_i, A^*, b^*, C^*, D^* can be simplified:

$$\begin{aligned} A^* &= \left((\tfrac{\partial b^*_1}{\partial x})^T, O, \cdots, O \right)^T \\ b^* &= \begin{pmatrix} -I & O & \cdots & \cdots & \cdots & O \\ \frac{\partial b^*_2}{\partial y_1} & -I & \ddots & \cdots & \cdots & \vdots \\ O & \frac{\partial b^*_3}{\partial y_2} & -I & \ddots & \cdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \frac{\partial b^*_{k-2}}{\partial y_{k-3}} & -I & O \\ O & \cdots & \cdots & O & \frac{\partial b^*_{k-1}}{\partial y_{k-2}} & -I \end{pmatrix} \\ C^* &= O \\ D^* &= \begin{pmatrix} O, & \cdots, & O, & \frac{\partial b^*_k}{\partial y_{k-1}} \end{pmatrix} \end{aligned} \tag{2.8}$$

Hence the Jacobian can be explicitly calculated as:

$$J^* = C^* - D^*(B^*)^{-1}A^* = \frac{\partial b^*_k}{\partial y^*_{k-1}} \cdot \frac{\partial b^*_{k-1}}{\partial y^*_{k-2}} \cdots \frac{\partial b^*_2}{\partial y^*_1} \cdot \frac{\partial b^*_1}{\partial x} \tag{2.9}$$

as expected. Now how would AD compute the above product for each of the two modes? Reverse mode has a forward sweep and a backward sweep where forward mode only have the forward phase. For the forward phase in both modes, $\frac{\partial b^*_{i+1}}{\partial y^*_i}$ for $\forall i$ would be computed.

Forward mode AD keeps updating $\frac{\partial y_i^*}{\partial x}$ and stores the results. In this example, once $\frac{\partial b_{i+1}^*}{\partial y_i^*}$ is obtained, forward mode left multiplies it to $\frac{\partial b_i^*}{\partial y_{i-1}^*} \cdots \frac{\partial b_1^*}{\partial x}$ then stores the result as $\frac{\partial y_{i+1}^*}{\partial x}$. In contrast, reverse mode simply stores all the $\frac{\partial b_{i+1}^*}{\partial y_i^*}$ and does not do any computation in the forward phase. In the backwards phase, reverse mode computes backwards, i.e. $\frac{\partial b_i^*}{\partial y_{i-1}^*}$ is multiplied to the right of $\frac{\partial b_k^*}{\partial y_{k-1}^*} \cdots \frac{\partial b_{i+1}^*}{\partial y_i^*}$ until the whole product is done. Let's inspect the runtime complexities of both modes.

- **Forward mode**: The above matrix products are time ordered from right to left, which is order the forward mode operates. It first left multiplies $\frac{\partial b_2^*}{\partial y_1^*}$ to $\frac{\partial b_1^*}{\partial x}$. Recalling the complexity of multiplying a $a \times b$ matrix to a $b \times c$ matrix is $O(abc)$, $\frac{\partial b_2^*}{\partial y_1^*}$ is of size $n_2 \times n_1$ and $\frac{\partial b_1^*}{\partial x}$ is of size $n_1 \times n$, such multiplication gives $O(nn_1n_2)$ complexity.

  The resulting matrix $\frac{\partial b_2^*}{\partial y_1^*}\frac{\partial b_1^*}{\partial x}$ is now of size $n_2 \times n$. Then $\frac{\partial b_3^*}{\partial y_2^*}$ is left multiplied to get $\frac{\partial b_3^*}{\partial y_2^*}\frac{\partial b_2^*}{\partial y_1^*}\frac{\partial b_1^*}{\partial x}$ of size $n_3 \times n$ with $O(nn_2n_3)$ operations. So on and so forth.

  Hence overall runtime complexity of forward mode is:

$$\omega(\partial_{\text{For}} c^*) \sim n \cdot [\sum_{i=1}^{k-2} n_i n_{i+1} + n_{k-1} m] \tag{2.10}$$

- **Reverse mode**: As explained, the first computation would be $\frac{\partial b_k^*}{\partial y_{k-1}^*} \cdot \frac{\partial b_{k-1}^*}{\partial y_{k-2}^*}$ which takes $O(mn_{k-1}n_{k-2})$ resulting in a matrix of size $m \times n_{k-2}$. And then $O(mn_{k-2}n_{k-3})$ to a $m \times n_{k-3}$ matrix. So on and so forth.

  Hence overall complexity for reverse mode is:

$$\omega(\partial_{\text{Rev}} c^*) \sim m \cdot [\sum_{i=1}^{k-2} n_i n_{i+1} + n_1 n] \tag{2.11}$$

We see reverse mode and forward mode scales differently. $k \gg 1$, hence $\sum_{i=1}^{k-2} n_i n_{i+1} \gg n_1 n$ and $\sum_{i=1}^{k-2} n_i n_{i+1} \gg n_{k-1} m$, therefore the terms in the square brackets of both runtime

complexity expressions are asymptotically the same. In fact,

$$\omega(c^*) \sim n_1 n + \sum_{i=1}^{k-2} n_i n_{i+1} + n_{k-1} m \tag{2.12}$$

represents the complexity of the function itself (since a step from y_i^* to y_{i+1}^* generally has complexity $n_i n_{i+1}$). Now we can tell why reverse mode is more efficient when the input dimension is large and the output dimension is small. When computing derivatives for f^*, forward mode's computations' matrices always have n^c columns where as reverse mode's computations involves matrices of m^c rows. Other scalings are asymptotically the same. *Forward mode computes the Jacobian matrix one column (each input) at a time while reverse mode computes one row (each output) at a time.*

$$\begin{cases} T_{\text{For}}(c) & \sim n^c \\ T_{\text{Rev}}(c) & \sim m^c \end{cases} \tag{2.13}$$

* superscript is dropped since the above is true in general. Note when a gradient is needed, i.e. $m^c = 1$, reverse mode computes it at once.

2.2 Structuring Reverse Mode

Having observed the runtime behavior, we inspect the space ratio of both modes of AD. In the above example, forward mode takes storage:

$$\sigma(\partial_{\text{For}} c^*) \sim n \max_{i=1}^{k} \{n_i\} + \max_{i=1}^{k} \{\sigma(b_i^*)\} \tag{2.14}$$

since it is keeping $\frac{\partial y_i}{\partial x}$. The first term is negligible and the second part is just:

$$\sigma(c^*) \sim \max_{i=1}^{k} \{\sigma(b_i^*)\} \tag{2.15}$$

Hence the space ratio S of forward mode on c^* is $O(1)$.

$$S_{\text{For}}(c^*) = \frac{\sigma(\partial_{\text{Rev}} c^*)}{\sigma(c^*)} \sim \frac{n \max_{i=1}^{k}\{n_i\} + \max_{i=1}^{k}\{\sigma(b_i^*)\}}{\max_{i=1}^{k}\{\sigma(b_i^*)\}} \sim 1 \tag{2.16}$$

For reverse mode, storage of all the intermediate variables is required. Denote:

Definition 2.1. The *Tape Size* $\bar{\sigma}(c)$ of a source code c is defined as:

$$\bar{\sigma}(c) = \text{Storage needed for all intermediate variables during execution of } c \\ \text{(except those created within basic operations) and the Computational Graph of } c \tag{2.17}$$

Definition 2.2. $\sigma(y)$ is the storage needed to store the variable y in a default environment.

therefore:

$$\sigma(\partial_{\text{Rev}} c) = \bar{\sigma}(c) \tag{2.18}$$

For sequential program f^*, we have:

$$\bar{\sigma}(c^*) = \sum_{i=1}^{k} \bar{\sigma}(b_i^*) \sim \sum_{i=1}^{k} \sigma(b_i^*) \tag{2.19}$$

where we have used the fact that B^* is basic operations hence no intermediate variables are stored and the storage for the type of basic operation have been neglected. Therefore:

$$S_{\text{Rev}}(c^*) = \frac{\sigma(\partial_{\text{Rev}} c^*)}{\sigma(c^*)} = \frac{\bar{\sigma}(c^*)}{\sigma(c^*)} \sim \frac{\sum_{i=1}^{k^*} \sigma(b_i^*)}{\max_{i=1}^{k^*}\{\sigma(b_i^*)\}} \sim k^* \tag{2.20}$$

reverse mode's space ratio scales linearly with the depth of the function k^* whereas the one of forward mode is $O(1)$. The $O(k)$ space ratio may impede the use of reverse mode for large scale computations despite its significant runtime advantages. Let's see how structured reverse mode reduces the undesirable space complexity of reverse mode while maintaining its runtime efficiency.

2.2.1 Basic Idea of Checkpointing

Reverse mode's space complexity comes from storing all intermediate variables that a program have ever created. If such storage is not kept, the derivatives cannot be successfully propagated in the backward phase. However, viewing the whole program as a concatenation of multiple subprograms, divide and conquer can be deployed.

The original idea of multilevel differentiation is found in [8] and further developed in [10]. Only the checkpoint variables, instead of all, are stored in the forward sweep. During

backward sweep, reverse mode is applied to the tape segments between checkpoints one at a time. Obtaining the checkpoints needs one additional evaluation of the original function. Therefore time ratio T only increase by 1 however the space ratio S is reduced *exponentially*, as it turns out for a typical code.

Checkpoints $\{P_i\}_{i=1}^{n_p}$ are essentially an ordered collection of subsets of all intermediate variables $\{y_i\}_{i=1}^{k}$. However, not any ordered collection of subsets of $\{y_i\}_{i=1}^{k}$ qualifies as a set of checkpoints. As the full formulation involves cumbersome notations and is not of importance for the following discussion, we leave it to Appendix B.1 and only states its definition and the qualifying property.

Definition 2.3. Given a code c, its input x, output z and all intermediate variables $\{y_i\}_{i=1}^{k}$ in c's resulting computation, a selection of checkpoints $\{P_i\}_{i=0}^{n_p+1}$ is an ordered collection of subsets of $\{y_i\}_{i=1}^{k}$ s.t.:

$$\begin{cases} P_0 = \{x\} & \\ P_i \subseteq \{y_i\}_{i=1}^{k} & \forall i \in \{1, \cdots, n_p\} \\ P_{n_p+1} = \{z\} & \\ P_i < P_{i+1} & \forall i \in \{1, \cdots, n_p - 1\} \end{cases} \tag{2.21}$$

where $P_1 < P_2$ means $\forall y \in P_1$ and $\forall y' \in P_2$, y appears strictly chronologically earlier than y' in the execution of c.

Definition 2.4. A selection of checkpoints $\{P_i\}_{i=0}^{n_p+1}$ of code c is *admissible* if there exist code $\{W_i\}_{i=1}^{n_p+1}$ s.t. the following ordered execution has exactly the same set of intermediate variables as c:

$$\begin{cases} y_1 & = b_1(\mathrm{Arg}_1) = b_1(x) \\ y_2 & = b_2(\mathrm{Arg}_2) \\ & \vdots \\ y_{k-1} & = b_{k-1}(\mathrm{Arg}_{k-1}) \\ z & = b_k(\mathrm{Arg}_k) \end{cases} \Leftrightarrow \begin{cases} P_1 & = W_1(P_0) = W_1(x) \\ P_2 & = W_2(x, P_1) \\ & \vdots \\ P_{n_p} & = W_{n_p}(x, P_1, \cdots, P_{n_p-1}) \\ z & = P_{n_p+1} = W_{n_p+1}(x, P_1, \cdots, P_{n_p}) \end{cases} \tag{2.22}$$

The set of all y and empty set $\varnothing$ are two trivial admissible selections of checkpoints. All discussions afterwards will be based on an admissible selection of checkpoints. Now we

Algorithm 1 Structured Reverse Mode

function STRREVERSEMODE(x, Code c, Admissible selection of checkpoints $\{P_i\}_{i=1}^{n_p}$)

Evaluate the code c once and store $\{P_i\}_{i=1}^{n_p}$.

Apply reverse mode on code segment W_{n_p+1}. Then on W_{n_p} etc. Until we fully propagate the derivatives to x.

end function

calculate what runtime and space ratio can be achieved for reverse mode. The algorithm of structured reverse mode given $\{P_i\}_{i=1}^{n_p}$ is simply[4]:

For illustration of structured reverse mode, see Appendix B.3. By saving only $\{P_i\}_{i=1}^{n_p}$, space complexity is reduced. Storage comes from two parts[5]: (a) storage of checkpoints (b) storage of intermediate variables during reverse mode on each segment W_i. Hence:

$$\sigma(\partial_{\text{StrRev}}c) = \sum_{i=0}^{n_p} \sigma(P_i) + \max_{i=1}^{n_p+1} \sigma(\partial_{\text{Rev}}W_i) = \sum_{i=0}^{n_p} \sigma(P_i) + \max_{i=1}^{n_p+1} \bar{\sigma}(W_i) \tag{2.23}$$

We see there is tradeoff in the density of checkpoints. The first term would increase if we increase the density and the second term would increase if we decrease the density.

For runtime, we have an additional run of c regardless of the selection of the checkpoints.

$$\omega(\partial_{\text{StrRev}}c) = \omega(c) + \omega(\partial_{\text{Rev}}c)$$

as we mentioned in the first chapter.

2.2.2 A program of uniform checkpoint structure

In previous subsection, we defined what is an admissible selection of checkpoints and calculated the runtime and space of the structured approach. However, all are rather abstract and general, therefore hard to quantify. The objective of this subsection is to understand what checkpoints arrangement yields desirable space and runtime ratio overall. Let's study a uniform program in which quantities of interest can be easily computed therefore several useful insights can be drawn.

[4]The detailed numerical procedure of "derivatives propagation" can be found in Appendix A.

[5]The storage required by the original function evaluation $\sigma(c)$ is neglected since it is negligible for reasonable implementations.

Suppose the code c_0 has been divided by an admissible selection of checkpoints $\{P_i\}_{i=1}^{k-1}$ into subprograms $\{W_i\}_{i=1}^{k}$. c_0 is also'uniform' such that runtime and space computation is simplified:

$$\forall i \in \{1, \cdots, k\} \begin{cases} \omega(W_i) & = \omega_0 \\ \chi(W_i) & = \chi_0 \\ \sigma(W_i) & = \hat{\sigma}_0 \\ \bar{\sigma}(W_i) & = \bar{\sigma}_0 \end{cases} \tag{2.24}$$

$$\forall i \in \{1, \cdots, k-1\} \quad \sigma(P_i) = \sigma_0 \tag{2.25}$$

Since σ_0 is the space to store the input of W: P, $\hat{\sigma}_0$ the space needed during computation of W, and $\bar{\sigma}_0$ is the full tape of W. It follows that:

$$\sigma_0 \leqslant \hat{\sigma}_0 \leqslant \bar{\sigma}_0 \tag{2.26}$$

It also follows that:

$$\begin{cases} \omega(c_0) & = k\omega_0 \\ \omega(\partial_{\text{Rev}} c_0) & = \chi_0 m \cdot k\omega_0 \\ \sigma(c_0) & = \hat{\sigma}_0 \\ \sigma(\partial_{\text{Rev}} c_0) & = k\bar{\sigma}_0 \\ T_{\text{Rev}}(c_0) & = \chi_0 m \\ S_{\text{Rev}}(c_0) & = k\bar{\sigma}_0/\hat{\sigma}_0 \end{cases} \tag{2.27}$$

As inspired by (2.23), there's a tradeoff in the density of checkpoints. Empty set, as an admissible checkpoint selection that leads to naive reverse mode, represents the extreme that minimizes the first term (storage of checkpoints) and reaching the maximum of the second term (storage for reverse mode on subprograms). The set of all y is the other extreme, minimizes the second term (storage for reverse mode on subprograms) and reaching the maximum of the first term (storage of checkpoints). To find a balanced tradeoff without worrying about admissibility, the selection of checkpoints $\{P_i\}_{i=1}^{k-1}$ is further assumed to be Markovian as well[6].

Definition 2.5. An admissible selection of checkpoints is *Markovian* if each of its replicating code segments W_{i+1} only takes x and the previous checkpoint P_i as its input.

[6]It turns out it is not too strong an assumption as such structure is universal in finance applications.

Theorem 2.6. A subset of a Markovian admissible collection of checkpoints is also an Markovian admissible selection of checkpoints.

For proof of Theorem 2.6, see Appendix B.2. In light of Theorem 2.6, any subset $\{Q_i\}_{i=1}^{n_p}$ of $\{P_i\}_{i=1}^{k-1}$ would be admissible. As admissibility is always satisfied, we only need to find an optimal subset $\{Q_i\}_{i=1}^{n_p}$ to minimize our space cost. Suppose $\{Q_i\}_{i=1}^{n_p}$ divides the code c_0 into $\{V_i\}_{i=1}^{n_p+1}$, denoting its indexing of $\{P_i\}_{i=1}^{k-1}$ as q_i:

$$\forall i \in \{0, \cdots, n_p + 1\} \qquad Q_i = P_{q_i} \tag{2.28}$$

The following relation should hold:

$$0 = q_0 < q_1 < \cdots < q_{n_p} < q_{n_p+1} = k \tag{2.29}$$

Hence (2.23) can be rewritten as:

$$\sigma(\partial_{\text{StrRev}} c_0) = \sum_{i=0}^{n_p} \sigma(Q_i) + \max_{i=1}^{n_p+1} \bar{\sigma}(V_i) = (n_p + 1)\sigma_0 + \bar{\sigma}_0 \cdot \max_{i=0}^{n_p}(q_{i+1} - q_i) \tag{2.30}$$

Given a fix number of checkpoints, it is obvious that evenly separated checkpoints are favorable since the first term does not depend on the distribution of Q_i, and the second term can be minimized with equally spaced checkpoints. We assume $k \gg 1$ and $n_p \ll k$ hence the difference between $\text{floor}(\frac{k}{n_p+1})$ and $\frac{k}{n_p+1}$ can be neglected as such difference would be at most a sub-leading term asymptotically.

$$\sigma(\partial_{\text{StrRev}} c_0) \approx (n_p + 1)\sigma_0 + \frac{k}{n_p + 1}\bar{\sigma}_0 \geqslant 2\sqrt{k\sigma_0\bar{\sigma}_0} \tag{2.31}$$

Hence we see that the optimal number of checkpoints are:

$$\hat{n}_p = \sqrt{\frac{\bar{\sigma}_0}{\sigma_0}k} - 1 \sim \sqrt{\frac{\bar{\sigma}_0}{\sigma_0}k} \tag{2.32}$$

$\hat{n}_p$ indeed scales less than k hence our assumption $n_p \ll k$ holds. For later convenience, we denote[7]:

$$S_0 = \sigma_0/\hat{\sigma}_0 \tag{2.33}$$

[7] S_0 is just a quantity that appears frequently in later formula, not a space ratio of the code piece W: it is not even greater than 1 since $\sigma_0 \leqslant \hat{\sigma}_0$.

Hence using structure halves the order of magnitude of space ratio compared to a straightforward reverse mode and only adds 1 to the runtime ratio:

$$\begin{cases} T_{\text{StrRev}}(c_0) & = 1 + T_{\text{Rev}}(c_0) = 1 + \chi_0 m \\ S_{\text{StrRev}}(c_0) & \approx 2\sqrt{k\sigma_0\bar{\sigma}_0}/\hat{\sigma}_0 = 2\sqrt{S_{\text{Rev}}(c_0)S_0} \end{cases} \tag{2.34}$$

providing a space efficient alternative to the bare reverse mode.

Notice the dimensionless combination $\frac{\bar{\sigma}_0}{\sigma_0}$ is a key factor in expressions of $\hat{n}_p$ and space ratio S. As $\bar{\sigma}_0$ is the amount of computation done at each checkpoint, σ_0 is the size of input, it conceptually represents a ratio of "computation per input":

Definition 2.7. The *computation per input* (CPI) ratio $\Omega(c)$ of a code c is defined as:

$$\Omega(c) = \frac{\bar{\sigma}(c)}{\sigma(x_c)} \tag{2.35}$$

where x_c is the input of c.

For W_i, $\frac{\bar{\sigma}_0}{\sigma_0}$ is exactly the CPI ratio:

$$\Omega(W_i) \equiv \Omega_0 = \frac{\bar{\sigma}_0}{\sigma_0} \qquad \forall i \in \{1, \cdots, k\} \tag{2.36}$$

2.2.3 Recursive structure

We see in (2.34) that for the uniform code c_0 of length k, we can apply structured reverse mode to obtain significant improvement of space ratio by marginally increment in runtime ratio. Just as divide and conquer normally works, such structure can be applied recursively.

We relax n_p from being $\hat{n}_p$, and then apply structured reverse mode with n'_p checkpoints again on each piece of tape V_i. Denoting such 2-level structure to be approach Str^{2}Rev, we obtain:

$$\begin{aligned} \sigma(\partial_{\text{Str}^2\text{Rev}} c_0) &\approx (n_p + n'_p + 2)\sigma_0 + \frac{k}{(n_p+1)(n'_p+1)}\bar{\sigma}_0 \geqslant (n_p+1)\sigma_0 + 2\frac{\sqrt{k\sigma_0\bar{\sigma}_0}}{\sqrt{n_p+1}} \\ &\geqslant 3k^{\frac{1}{3}}\sigma_0^{\frac{2}{3}}\bar{\sigma}_0^{\frac{1}{3}} \end{aligned} \tag{2.37}$$

the equal sign is reached when:

$$\begin{cases} \hat{n}_p &= (\frac{\bar{\sigma}_0}{\sigma_0}k)^{\frac{1}{3}} - 1 \sim (\frac{\bar{\sigma}_0}{\sigma_0}k)^{\frac{1}{3}} = (\Omega_0 k)^{\frac{1}{3}} \\ \hat{n}'_p &= \sqrt{\frac{\bar{\sigma}_0}{\sigma_0}\frac{k}{\hat{n}_p+1}} - 1 = \hat{n}_p \sim (\Omega_0 k)^{\frac{1}{3}} \end{cases} \tag{2.38}$$

Interestingly, $\hat{n}'_p = \hat{n}_p$ exactly. The above optimal number of checkpoints achieves the following runtime and space ratio:

$$\begin{cases} T_{\text{Str}^2\text{Rev}}(c_0) &= 1 + T_{\text{StrRev}}(c_0) = 2 + \chi_0 m \\ S_{\text{Str}^2\text{Rev}}(c_0) &\approx 3k^{\frac{1}{3}}(\sigma_0/\hat{\sigma}_0)^{\frac{2}{3}}(\bar{\sigma}_0/\hat{\sigma}_0)^{\frac{1}{3}} = 3S^{\frac{1}{3}}_{\text{Rev}}(c_0)S_0^{\frac{2}{3}} \end{cases} \tag{2.39}$$

We can see a pattern already. As a natural generalization, using structure of level l, we would have:

$$\begin{cases} T_{\text{Str}^l\text{Rev}}(c_0) &= l + \chi_0 m \\ S_{\text{Str}^l\text{Rev}}(c_0) &= (l+1)k^{\frac{1}{l+1}}(\sigma_0/\hat{\sigma}_0)^{\frac{l}{l+1}}(\bar{\sigma}_0/\hat{\sigma}_0)^{\frac{1}{l+1}} = (l+1)S^{\frac{1}{l+1}}_{\text{Rev}}(c_0)S_0^{\frac{l}{l+1}} \\ S_{\text{Str}^l\text{Rev}}(c_0)/S_0 &= (l+1)(S_{\text{Rev}}(c_0)/S_0)^{\frac{1}{l+1}} \end{cases} \tag{2.40}$$

achieved when:

$$n_p^{(1)} = \cdots = n_p^{(l)} \equiv (\frac{\bar{\sigma}_0}{\sigma_0}k)^{\frac{1}{l+1}} - 1 \sim (\frac{\bar{\sigma}_0}{\sigma_0}k)^{\frac{1}{l+1}} = (\Omega_0 k)^{\frac{1}{l+1}} \tag{2.41}$$

The consistency condition changes from $n_p \ll k$ to $\prod_{i=1}^{l} n_p^{(l)} \ll k$, which is satisfied for all $l \in \mathbf{Z}^+$ since the left hand side scales as $k^{l/(l+1)}$, which is less than k.

It might seem meaningless to calculate everything accurately since we are dealing with an artificial example. Also l and the distribution of checkpoint positions represented by n_p are fundamentally discrete no matter how we pretend they can achieve the non-integer optimal value.

Yet there are two general conclusions that we can draw: (1) the optimal arrangement of checkpoints is the one that evenly divides the computation (2) to achieve the best space ratio, the optimal level l is not infinity. When too many level of structures are used, more space are actually needed. To find out the best level depth l, we find the argmin with respect to l by taking derivatives of the In of $S_{\text{Str}^l\text{Rev}}(c_0)/S_0$:

$$0 \equiv \frac{\partial}{\partial l}\text{In}[\frac{S_{\text{Str}^l\text{Rev}}(c_0)}{S_0}] = \frac{\partial}{\partial l}(\text{In}(l+1) + \frac{1}{l+1}\text{In}[\frac{S_{\text{Rev}}(c_0)}{S_0}]) = \frac{1}{l+1} - \frac{1}{(l+1)^2}\text{In}[\frac{S_{\text{Rev}}(c_0)}{S_0}]$$

which we can solve for the optimal $\hat{l}$:

$$\hat{l} = \text{In}[\frac{S_{\text{Rev}}(c_0)}{S_0}] - 1 = \text{In}[\Omega_0 k] - 1 \tag{2.42}$$

where we can realize the following runtime and space ratio:

$$\begin{cases} \hat{T}_{\text{StrRev}}(c_0) & = \text{In}[\Omega_0 k] + \chi_0 m - 1 \sim \ln k \\ \hat{S}_{\text{StrRev}}(c_0) & = S_0 e \text{In}[\Omega_0 k] \sim \ln k \end{cases} \tag{2.43}$$

with:

$$\hat{n}_p = e - 1 \tag{2.44}$$

Somehow e shows up in a funny way. Of course it is not possible to have non-integer valued number of checkpoints, hence $n_p = 2$ is likely the best choice meaning a *tertiary* recursive structure achieves the lowest space ratio.

We conclude that for such code with a uniform Markovian checkpoint structure, recursively structured reverse mode gives a logarithmic scaling in terms of function depth k in both runtime and space ratio, in accord with [10].

2.2.4 Efficient frontier of the runtime/space tradeoff

In analogy to portfolio optimization we consider the tradeoff between expected return and variance, an efficient frontier of space/runtime ratio for our uniform program c_0 can be drawn.

The efficient frontier of space/runtime ratio for c_0 is given by the following formula:

$$\begin{cases} S & = S_0(T - T_0 + 1)(\Omega_0 k)^{1/(T - T_0 + 1)} \\ T_0 & = T_{\text{Rev}}(c_0) = \chi_0 m \end{cases} \tag{2.45}$$

where have use S and T to denote space and runtime ratio respectively.

In Figure 2.1, log scale is used for the space ratio. Points on the curve are the pairs of runtime/space ratio achieved with different levels of structures. As noted, runtime ratio increases linearly with the level of structure however the space ratio does not always

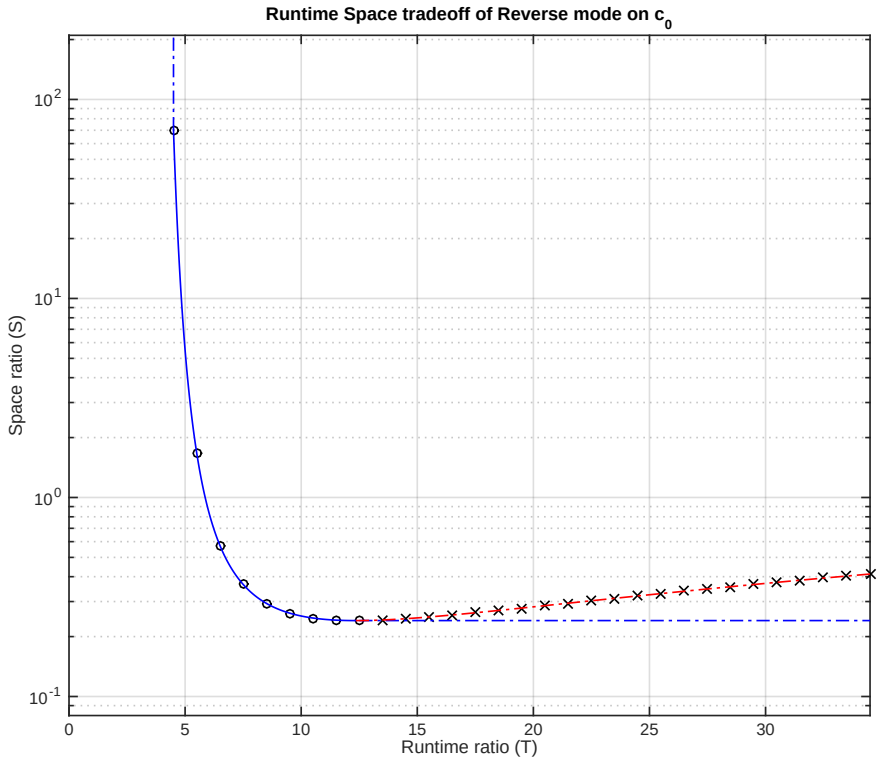

Figure 2.1: Efficient frontier of the runtime ratio and space tradeoff of the uniform code c_0 when $\chi_0 m = 4.5, \Omega_0 k = 7000, S_0 e = 0.01$

decrease. After passing the optimal level depth $\hat{l}$ which is about 8 here (corresponds to Runtime ratio $T = 8 + \chi_0 m = 12.5$), space ratio also goes up when we increase level of structure (represented by crosses on the red dashed line). Hence the right part of the curve is inefficient in both space and time hence is not on the efficient frontier.

This efficient frontier is different from portfolio analysis setting since we are trying to minimize both space and runtime ratio compared to variance minimization coupled with expected return maximization. Our efficient frontier is of finite size as represented by the unfilled circles on the blue solid curve which are essentially from bare reverse mode to structured reverse mode of level $\hat{l}$. We see the bare reverse has an extremely high space ratio, more than 2.5 order of magnitude than the minimal space ratio in the example.

We observe that space ratio can be drastically reduced even when we only have very few layers of structure. The amount of space released per runtime decreases 'exponentially' as we increase the level of structure and would turn negative after $O(\log k)$ level of layers.

2.2.5 Concluding remarks

We have made a few important observations above. However, they are based on our simple example of a program with uniform Markovian admissible checkpoints, one may wonder which ones of them generalize? As we will see in Chapter 3, for Monte Carlo simulations in financial applications, the program always displays a Markovian admissible checkpoint structure. Hence:

(1) Within each level of structure, the checkpoints should separate the amount of computations as evenly as possible.

(2) It is not always beneficial to increase the level of structure. Too many layers of structure are inefficient in both runtime and space and should be avoided.

(3) The first few levels of structure performs best in terms of trading runtime for space reduction. Being easier to implement, just one or two levels could be sufficiently efficient in practice.

We see the choice of structure architecture is not trivial. However, previous discussions are dealing with our simple uniform program example. In practice, even revealing the structure of a given code is non trivial. And it is a lot harder to determine the optimal checkpoints positions when the code segments W contain varied amount of computations. In the following Chapter 3 and Chapter 4, we are going to introduce Monte Carlo simulations and Variable Annuities. With the 'structure' mindset developed in this chapter, we are ready to put their computations into perspectives. In Chapter 5, a general framework of structure design for nested simulations will be formulated.

Chapter 3

Monte Carlo simulation in Finance

This chapter discusses the convergence of Monte Carlo algorithm, and how different approaches of derivatives calculations lead to distinct rate of efficiency. As reverse mode AD calculates exact derivatives efficiently, forward mode AD calculate exact derivative inefficiently, FD calculates approximations inefficiently, their rates of efficiency show hierarchy. Finally, general structure of Monte Carlo simulations in financial applications are revealed.

3.1 Convergence of Monte Carlo

Monte Carlo methods in general are computational algorithms that obtain certain properties of random variables, mostly the mean, by repeated random sampling on them. The convergence of Monte Carlo algorithm on mean estimation is guaranteed by central theorem. If we define the following rate of efficiency, a simple Monte Carlo algorithm would have $\beta_{mc} = 0.5$. See Appendix C.1.

Definition 3.1. The *rate of efficiency* $\beta_a \in (0, \infty]$ of an algorithm a to compute a scalar quantity y is defined via the following asymptotics[1]:

$$|y_a - y| \sim O(M_a^{-\beta_a}) \tag{3.1}$$

where y_a is the approximation of y that approach a computes, and M_a is the complexity of a's runtime.

[1]For simplicity, float point accuracy is considered ∞ precision. For algorithm with $O(1)$ complexity and floating point accuracy, rate of efficiency is set to be ∞ by default.

It can be showed that rate of efficiency of a Monte Carlo deploying approach a to draw samples is:

$$\beta_{mc(a)} = \beta^s_{mc(a)} = \frac{\beta_a}{2\beta_a + 1} \in (0, \frac{1}{2}] \tag{3.2}$$

The rate of efficiency of Monte Carlo is naturally the harmonic mean of a's rate of efficiency β_a and the suppose-to-be rate of efficiency of Monte Carlo $\beta_{mc} = 0.5$. For proof, see Appendix C.2.

Whereas such rate of efficiency is at most 0.5, Monte Carlo's rate of efficiency scaling is excellent for its independence of the problem's dimensionality[2]. In finance, Monte Carlo and PDE are THE[3] two approaches for pricing financial instruments without analytic expressions. PDE scales as:

$$\begin{cases} |x_{pde} - x| & \sim \Delta x^{\gamma_a} \\ M_{pde} & \sim \Delta x^{-d} \\ \beta_{pde} & = \gamma_a / d \end{cases} \tag{3.3}$$

where d is the dimensionality of the problem, γ_a is the convergence rate of the PDE approach, which depends on the time stepping method used and the regularity of the priced instrument. Best situation we have $\gamma_a = 2$ resulting in $\beta_{pde} = 2/d$ which still suffers from the curse of dimensionality despite being superior for $d = 1$. The best for Monte Carlo is $1/2$ hence $d = 4$ is usually considered as a critical dimension beyond which Monte Carlo's efficiency would be higher than PDE in general.

3.2 Convergence of Monte Carlo derivatives

Monte Carlo estimates the mean of a random variable. However in many applications, derivatives of the mean are requested as well. Finite difference provides the most natural approximation and is easy to implement. Automatic differentiation computes the exact derivatives of each sample automatically. Let's compute their rates of efficiency.

[2]Dimensionality might affect implicitly on the complexity of sampling M through β_a since it is harder to draw sample in a higher dimensional space. Yet β_a as a scaling usually is not affected.

[3]Binomial tree or other state tree algorithm can be viewed as special cases of PDE approach.

3.2.1 Finite Difference

We start with central difference since it has the highest accuracy for regular problems. Assuming a $\mathbb{C}^4$ function f that can be computed up to certain level of accuracy using certain algorithm, it is natural to have the following trade-off:

$$\begin{aligned}\partial_\alpha f^*(\alpha_0) &\equiv \frac{f^*(\alpha_0+\Delta\alpha)-f^*(\alpha_0-\Delta\alpha)}{2\Delta\alpha}\\ &= \frac{f(\alpha_0+\Delta\alpha)-f(\alpha_0-\Delta\alpha)}{2\Delta\alpha}+\frac{\epsilon_+-\epsilon_-}{2\Delta\alpha}\\ &= \partial_\alpha f(\alpha_0)+\frac{1}{6}\partial^3_\alpha f(\alpha_0)\Delta\alpha^2+\frac{\epsilon_+-\epsilon_-}{2\Delta\alpha}+\frac{1}{48}[\partial^4_\alpha f(\xi_1)+\partial^4_\alpha f(\xi_2)]\Delta\alpha^3\end{aligned}$$

where we have use $*$ to indicate approximation of the true value and ϵ are the errors of the approximation. α is whose derivatives we are interested in, and we have use central difference of size $\Delta\alpha$. We cannot take $\Delta\alpha$ too small otherwise the third term will blow up even though finite difference approximation is improving (2nd and 4th terms will diminish).

Assume we take a step size $\Delta\alpha$ small enough such that the third order term is negligible:

$$\frac{1}{4!}|\partial^4_\alpha f(\alpha)|\Delta\alpha^4 \ll \frac{1}{3!}|\partial^3_\alpha f(\alpha)|\Delta\alpha^3 \qquad \forall\alpha\in[\alpha_0-\Delta\alpha,\alpha_0+\Delta\alpha] \tag{3.4}$$

Also let's assume the approximation error $\epsilon_\pm$ is of size ϵ_0. Hence our optimal choice of $\Delta\alpha$ and its resulting approximation error of the derivative would be:

$$\begin{cases}\Delta\alpha^* & = (\frac{3\epsilon_0}{|\partial^3_\alpha f(\alpha_0)|})^{\frac{1}{3}}\\ |\partial_\alpha f^*(\alpha_0)-\partial_\alpha f(\alpha_0)| & \geqslant \frac{1}{2}(3\epsilon_0)^{\frac{2}{3}}|\partial^3_\alpha f(\alpha_0)|^{\frac{1}{3}}\sim O(\epsilon_0^{\frac{2}{3}})\end{cases} \tag{3.5}$$

Now suppose our approximation algorithm is Monte Carlo. The above analysis indicates our rate of efficiency using central differences to obtain the derivatives of the mean obtained via Monte Carlo is at best:

$$\beta_{cd\_mc}\leqslant\frac{2}{3}\beta_{mc}\equiv\hat{\beta}_{cd\_mc} \tag{3.6}$$

where *cd* stands for central difference. Similar analysis yields:

$$\beta_{fd\_mc}=\beta_{bd\_mc}\leqslant\frac{1}{2}\beta_{mc}\equiv\hat{\beta}_{fd\_mc}=\hat{\beta}_{bd\_mc} \tag{3.7}$$

where fd and bd stands for forward difference and back difference respectively.

We should be alerted by the $\leqslant$ sign here. It indicates we have to change the finite different size $\Delta\alpha$ as we attempt to increase the accuracy of the algorithm. Take Monte Carlo for example. Suppose we don't scale $\Delta\alpha$ accordingly. If we look back at (3.4), the term $\frac{\epsilon_+ - \epsilon_-}{2\Delta\alpha} \sim \epsilon/\Delta\alpha$ does diminish when we decrease ϵ by drawing more samples. However $\frac{1}{6}\partial^3_\alpha f(\alpha_0)\Delta\alpha^2$ is a fundamental bias that finite difference has, it would not diminish unless we gradually decrease $\Delta\alpha$ when we scale up our algorithm. In otherwise, $\beta = 0$ if we take a fixed $\Delta\alpha$!

Picking the finite difference size $\Delta\alpha$ is tricky. The optimal choice depends on the higher order derivatives of the unknown function f, which is impossible to know in advance and hard to obtain if requested.

3.2.2 Pathwise Finite Difference

Above is a blackbox approach of finite difference. However for Monte Carlo simulation where we have the structure of repeated i.i.d sampling, we have the pathwise approach which takes the finite difference at each sample level before taking the mean to obtain the final result.

We can apply the analysis as in (3.4). At each sample level, our rate of efficiency to compute derivatives change to $\beta'_a \leqslant \frac{2}{3}\beta_a$ for central difference and $\beta''_a \leqslant \frac{1}{2}\beta_a$ for forward/backward difference. Viewing a', which is the finite difference of a's sampling output, as the new sampling algorithm for the Monte Carlo, we arrive at the following rate of efficiency for pathwised finite difference:

$$\begin{cases} \beta_{pcd\_mc} = \beta'_{mc} = \frac{\beta'_a}{2\beta'_a+1} \leqslant \frac{2\beta_a}{4\beta_a+3} \equiv \hat{\beta}_{pcd\_mc} \\ \beta_{pfd\_mc} = \beta_{pbd\_mc} = \beta''_{mc} = \frac{\beta''_a}{2\beta''_a+1} \leqslant \frac{\beta_a}{2(\beta_a+1)} \equiv \hat{\beta}_{pfd\_mc} = \hat{\beta}_{pbd\_mc} \end{cases} \tag{3.8}$$

which improves over their non-pathwised counterpart:

$$\begin{cases} \hat{\beta}_{pcd\_mc} = \frac{2\beta_a}{4\beta_a+3} \geqslant \frac{2\beta_a}{3(2\beta_a+1)} = \frac{2}{3}\beta_{mc} = \hat{\beta}_{cd\_mc} \\ \hat{\beta}_{pfd\_mc} = \hat{\beta}_{pbd\_mc} = \frac{\beta_a}{2(\beta_a+1)} \geqslant \frac{\beta_a}{2(2\beta_a+1))} = \frac{1}{2}\beta_{mc} = \hat{\beta}_{fd\_mc} = \hat{\beta}_{bd\_mc} \end{cases} \tag{3.9}$$

3.2.3 Automatic Differentiation

Automatic Differentiation is naturally pathwise[4]. Since AD computes the exact derivatives, it keeps the rate of efficiency at each sample level[5]. Hence we have:

$$\beta_{ad\_mc} = \beta_{mc} \tag{3.10}$$

which is strictly better than the one of pathwise finite difference.

β_a	$\beta_{fd/bd}$	$\beta_{pfd/pbd}$	β_{cd}	β_{pcd}	β_{ad}	$\beta_{mc(a)}$
∞	1/4 (0.25)	1/2 (0.50)	1/3 (0.33)	1/2 (0.50)	1/2 (0.50)	1/2 (0.50)
2	1/5 (0.20)	1/3 (0.33)	4/15 (0.27)	4/11 (0.36)	2/5 (0.40)	2/5 (0.40)
1	1/6 (0.17)	1/4 (0.25)	2/9 (0.22)	2/7 (0.28)	1/3 (0.33)	1/3 (0.33)
1/2	1/8 (0.12)	1/6 (0.17)	1/6 (0.17)	1/5 (0.20)	1/4 (0.25)	1/4 (0.25)

Table 3.1: Comparison of rates of efficiency of different approaches to calculate derivatives of a Monte Carlo program

Table 3.1 summarizes the rates of efficiency of using different approaches to obtain derivatives of a Monte Carlo simulation. We observe a hierarchy where AD achieve the highest rate of efficiency, the same as the Monte Carlo of the original function; pathwise finite difference performs better than its non-pathwised counterpart; central difference converges faster than forward/backward differences. Yet as finite difference yields at most an approximation, using AD leads to faster convergence.

In this section, we have discussed the convergence properties of Monte Carlo approaches. We see that using finite difference to approximate derivatives of values computed from Monte Carlo simulation can be inefficient and the choice of finite different size is unambiguous. Since the bias of the finite difference approximation is unknown, even a confidence interval of the derivative is hard to draw. AD circumvents such problem by the exact computation of the derivative at sample level. In terms of rate of efficiencies of these algorithms, AD's performance is strictly better than pathwise central difference, the best among the finite difference approach families.

[4]Pathwise in this chapter means in terms of taking derivatives, which is at sampling algorithm level. In Chapter V, pathwise means in terms of computation, which is at implementation level.

[5]Finite difference has inferior performance on accuracy because it is forced to take small step size which inevitably dampens the original accuracy. AD has no such problem.

3.3 Nested simulation in Financial Applications

So far we have discussed Monte Carlo of 1 level of sampling. In financial application, nested simulation is typically required for P&L analysis. We demonstrate its usage in financial applications by examples of option hedging program and then reveal its general structure.

3.3.1 Risk Neutral World simulation

Monte Carlo becomes a pivotal approach to price financial instrument by the establishment of risk neutral measure. It essentially allows one to solving one specific point in the underlying space of PDE by computing an expectation value under the risk neutral measure. One can also justify Monte Carlo to price assets by first using no-arbitrage arguments to establish Black-Scholes type PDE, and then Feynman-Kac formula relates the solution to the PDE to the expectation value of the outputs of underlying stochastic processes[6] which can be computed via Monte Carlo sampling.

Let's demonstrate a simple example of option pricing. Consider an European put option with strike K, maturity T on a stock with current price S_0 and underlying Geometric Brownian Motion of drift μ and volatility σ. The risk free rate is r. In risk neutral world, we have:

$$dS_t = rS_t dt + \sigma S_t dW_t \tag{3.11}$$

where W_t is a Wiener process. For generality, we pretend the exact solution of such SDE is not available and therefore an approximation by discretizing the time dimension is in need. For numerical consistency, we make a change of variable to $X_t \equiv \log S_t$ by Ito's Lemma:

$$\begin{cases} X_t & \equiv \log S_t \\ dX_t & = (r - \frac{\sigma^2}{2})dt + \sigma dW_t \end{cases} \tag{3.12}$$

Then we discretize the stochastic process into N steps with step size $\Delta t = T/N$:

$$\forall i \in \{1, \cdots, M\}, n \in \{1, \cdots, N\} \begin{cases} X_0^{(i)} & = \log S_0 \\ X_n^{(i)} & = X_{n-1}^{(i)} + (r - \frac{\sigma^2}{2})\Delta t + \sigma\sqrt{\Delta t} \cdot Z_n^{(i)} \\ Z_n^{(i)} & \sim N(0,1) \end{cases} \tag{3.13}$$

[6] In the risk neutral measure.

where M is number of Monte Carlo sampling. The upper index is of different samplings where the lower index is of different time grid points. Finally, we compute the discounted expected payoff P to be the value of our option V:

$$V = e^{-rT}\mathbb{E}_Q[P(S)] \approx \frac{e^{-rT}}{M}\sum_{i=1}^{M}\max\{0, K - e^{X_N^{(i)}}\} \tag{3.14}$$

3.3.2 Real World simulation

In previous subsection, we have given an example of pricing a derivative in the risk neutral world. To understand the projected P&L of a hedging program, what we need to simulate is instead, a real world process.

Aside from switching probability measure, updated option greeks are requested at each time step in the real world process which invokes one run of Monte Carlo simulation in the risk neutral world. Let's briefly describe our nested simulation program again using the European put option example.

Suppose only delta hedging with the underlying as hedging instrument is concerned. M_O sampling and N_O time steps are taken in the outer simulation and M_I and N_I for the inner one. $\Delta t_O = T/N_O$, $\Delta t_{I_n} = (T - n\Delta t_O)/N_I$. $X_n^{(i)}$ denotes the log price in the outer simulation: $i \in \{1, \cdots, M_O\}$ is the outer sampling index, $n \in \{0, \cdots, N_O\}$ is the outer time step index. $X_{n,m}^{(i,j)}$ is the log price in the inner simulation where $j \in \{1, \cdots, M_I\}$ is the inner sampling index, $m \in \{0, \cdots, N_I\}$ is the inner time step index. Denote the value of the bank account as $B_n^{(i)}$, number of shares holding in the underlying as $\alpha_n^{(i)}$, and the value of the option as $V_n^{(i)}$. Suppose we use central difference with step size ΔX to calculate delta for hedging, two perturbed versions of $X_{n,m}^{(i,j)}$ are denoted as $X_{n,m,\pm}^{(i,j)}$. The perturbed option value is denoted as $V_{n,\pm}^{(i,j)}$, the original value $V_n^{(i,j)}$ and the projected value

$V_n^{(i)}$. Accordingly, we have:

$$\forall \begin{cases} i \in \{1, \cdots, M_O\} \\ j \in \{1, \cdots, M_I\} \\ n \in \{1, \cdots, N_O\} \\ m \in \{1, \cdots, N_I\} \end{cases} \quad \begin{cases} X_0^{(i)} & = \log S_0 \\ X_{n,0}^{(i,j)} & = X_n^{(i)} \\ X_{n,0,\pm}^{(i,j)} & = X_n^{(i)} \pm \Delta X \\ X_n^{(i)} & = X_{n-1}^{(i)} + (\mu - \frac{\sigma^2}{2})\Delta t_O + \sigma\sqrt{\Delta t_O} \cdot Z_n^{(i)} \\ X_{n,m}^{(i,j)} & = X_{n,m-1}^{(i,j)} + (r - \frac{\sigma^2}{2})\Delta t_{I_n} + \sigma\sqrt{\Delta t_{I_n}} \cdot Z_{n,m}^{(i,j)} \\ X_{n,m,\pm}^{(i,j)} & = X_{n,m-1,\pm}^{(i,j)} + (r - \frac{\sigma^2}{2})\Delta t_{I_n} + \sigma\sqrt{\Delta t_{I_n}} \cdot Z_{n,m,\pm}^{(i,j)} \\ V_n^{(i,j)} & = \max\{0, K - e^{X_{n,N_I}^{(i,j)}}\} \\ V_{n,\pm}^{(i,j)} & = \max\{0, K - e^{X_{n,N_I,\pm}^{(i,j)}}\} \\ V_n^{(i)} & = \frac{1}{M_I}\sum_{k=1}^{M_I} V_n^{(i,k)} e^{-r(T-n\Delta t_O)} \\ \alpha_n^{(i)} & = -\frac{1}{M_I}\sum_{k=1}^{M_I}(V_{n,+}^{(i,k)} - V_{n,-}^{(i,k)})/(e^{X_{n,0,+}^{(i,k)}} - e^{X_{n,0,-}^{(i,k)}}) \\ B_0^{(i)} & = -V_0^{(i)} - \alpha_0^{(i)} S_0 \\ B_n^{(i)} & = B_{n-1}^{(i)} e^{r\Delta t_O} - (\alpha_n^{(i)} - \alpha_{n-1}^{(i)}) e^{X_n^{(i)}} \\ Z_n^{(i)} & \sim N(0,1) \\ Z_{n,m}^{(i,j)} & \sim N(0,1) \\ Z_{n,m,\pm}^{(i,j)} & \sim N(0,1) \end{cases} \tag{3.15}$$

After which we obtain M_O samples of the portfolio relative P&L Π^i

$$\Pi^{(i)} = (V_N^{(i)} + \alpha_N^{(i)} e^{X_N^{(i)}} + B_N^{(i)})/V_0 \tag{3.16}$$

The above seems quite complicated yet it is just hedging a simple European option. We will discuss the structure in the next section.

3.4 General Structure of Monte Carlo simulation in Finance

The Monte Carlo approach in finance is a general framework which can accommodates the valuation of various kinds of financial instruments. Any financial Monte Carlo algorithm has a common structure. For the purpose of applying structured reverse mode, let's reveal these structures.

3.4.1 Single layer Monte Carlo

The first generic structure is the parallel structure of independent sampling of identical distribution as we see in Figure 3.1. The second structure is the sequential path structure as with financial application demonstrated in Figure 3.2.

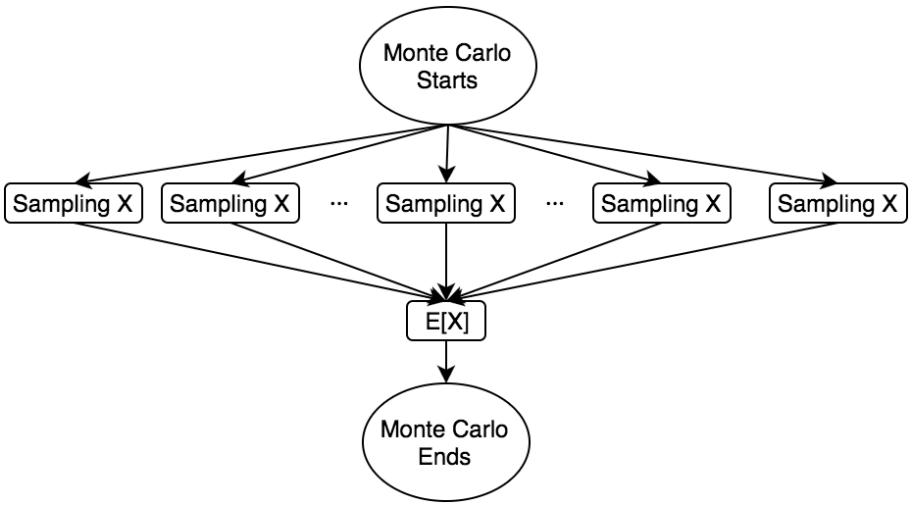

Figure 3.1: Parallel structure of generic Monte Carlo algorithm estimating the mean of random variable X

The structure of independent sampling is perfect for parallelism yet it does not provide any structure for checkpointing at all. In contrast, the sequential path structure is perfect for checkpointing.

As we can see in Figure 3.2, each sampling path looks like the simple uniform sequential linear example code c_0 that we've discussed in depth at the end of the previous chapter. Therefore all our analysis of checkpoint deployment apply.

Nevertheless, the put option program in Figure 3.2 is only an example. One might wonder, how general is such structure? How often do we encounter such structure pattern in real world applications? Is it robust enough to accommodate any marginal changes to the model?

The answer is yes: the abstract structure, as demonstrated in Figure 3.3, with Figure

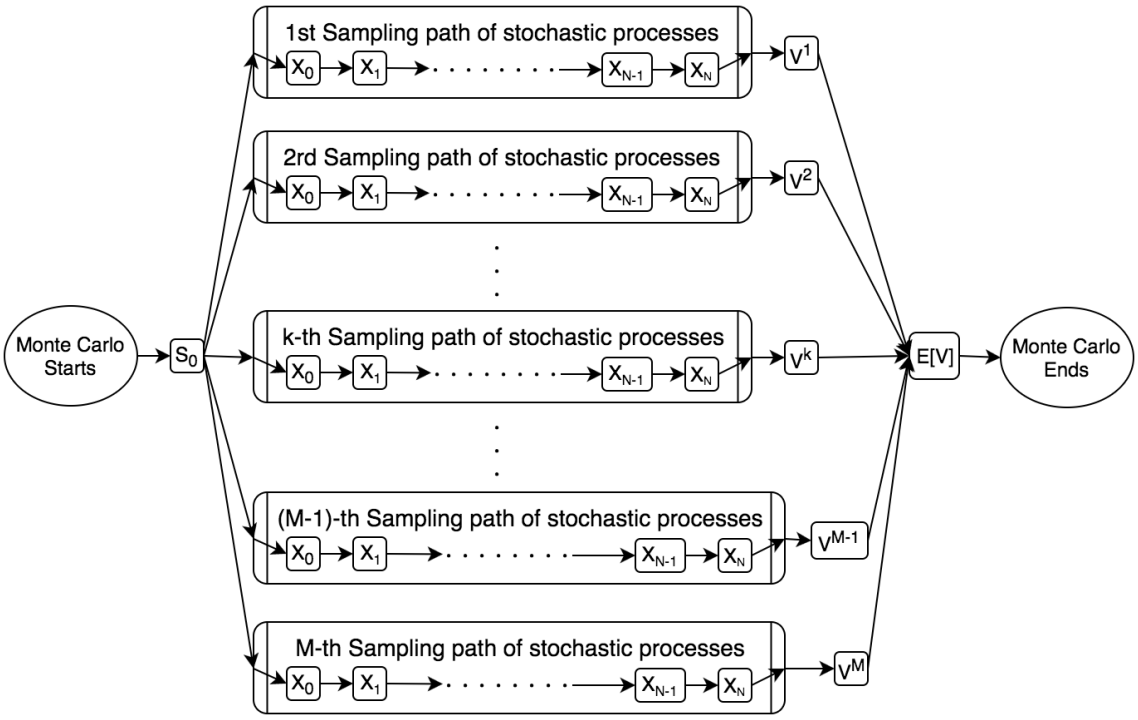

Figure 3.2: Sequential path structure of Monte Carlo using an European put option pricing example

3.2 as a special case, is indispensable in financial applications.

$\Xi_n^{(i)}$ is the complete state variable of the underlying process, whatever it might be, of the i-th sampling at n-th timestep. The checkpoint structure is evident as only state of the previous node Ξ_n along with input X is needed to process to the next node Ξ_{n+1}.

Comparing with Figure 3.2, we can make the following correspondence: the auxiliary variable $X = \mathrm{In}S$ is the state variable Ξ of the underlying processes; the equivalent input X of Figure 3.3 in Figure 3.2 is essentially $X = (S_0, K, T, r, \sigma)$. Parameters K, T, r, σ are not explicitly shown in Figure 3.2 though they participate in each transition between the nodes just as in Figure 3.3.

In such structure, *the set* $\{\Xi_i\}_{i=0}^N$ *is naturally a Markovian admissible selection of checkpoints*, ready for deployment of structured reverse mode. Now we elaborate on why we claim such structure is universal for financial instrument pricing:

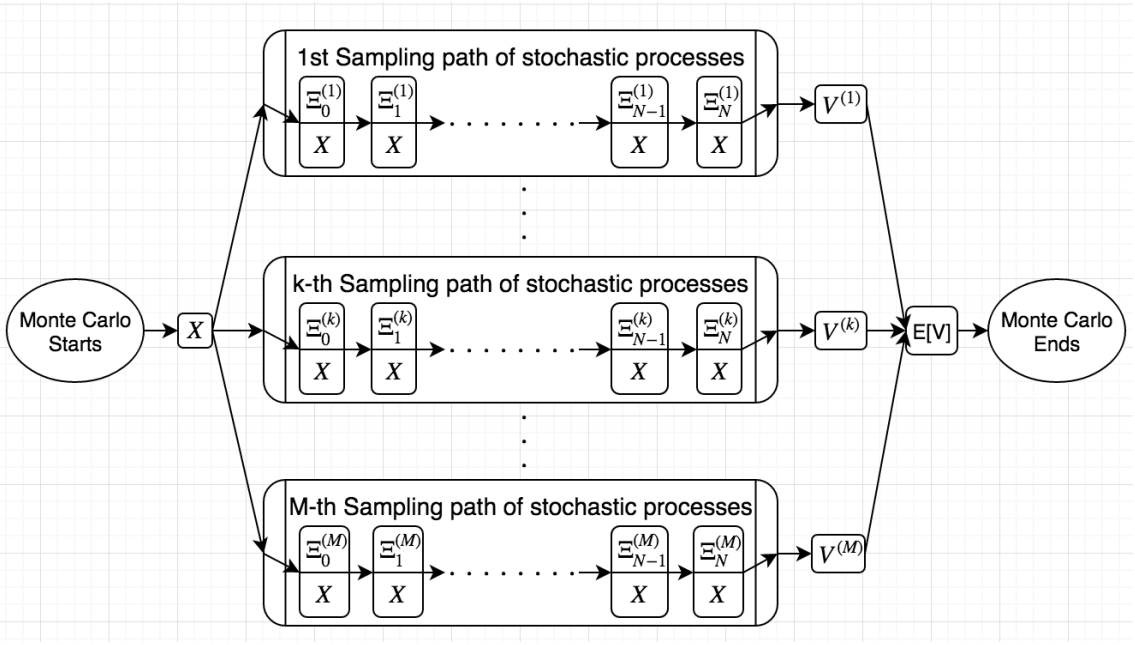

Figure 3.3: Abstract sequential path structure of Monte Carlo in financial applications

(1) **Sequential structure from time stepping**: For any financial instrument, timestepping is an indispensable part of the algorithm. Whatever stochastic processes are, discretization of time dimension is demanded. Whereas step size can be variable in certain algorithms, sequential structure maintains.

(2) ***Augmented Markovian* aspect of practical financial instruments**:

The claim is, for any realistic financial instrument, with a finite dimensional augmentation to the state of the process, the evolution of the augmented state process would become Markovian.

Such claim is obviously true for path independent instruments. For path-dependent instruments, the payoff depends on the history of underlying processes instead of only the final state. However, such path dependency has to be finite dimensional for any realistic financial instrument.

For example, for Asian option, path dependency is via the mean of the underlying, hence is only a 2-dimensional dependency: $(\bar{S}_p, T_p)$–how old the option is T_p and

what the mean $\bar{S}_p$ in the past is. With augmentations of $(\bar{S}_p, T_p)$, total mean can be calculated without further information of the history:

$$\bar{S}_{tot} = \frac{\bar{S}_p T_p + \int_{T_p}^{T} S(t)dt}{T}$$

which determines the final payoff of the option. As for underlying processes of the augmentation:

$$\begin{cases} d\bar{S}_p(t) &= \frac{S(t)-\bar{S}_p(t)}{T_p(t)}dt \\ dT_p(t) &= dt \end{cases} \tag{3.17}$$

For look-back option, path dependency is 1-dimensional: the historical high value $\max_{t \leqslant T_p} S_t$ completes the augmentation. For barrier option, it is a 1-dimensional dependency as well: whether the barrier has been hit or not.

If such path dependency can not be reduced to a finite dimensional features of the history, the payoff of the instrument cannot be robustly defined, which is impossible in practice. Hence, after properly augmenting the state of the underlying processes, we are guaranteed to have such a Markovian sequential path structure.

This concludes our discussion of structure in single layer Monte Carlo.

3.4.2 Nested Monte Carlo

The structure of the nested simulation at each level is the same as the one of single layer Monte Carlo. Schematic flow diagram is drawn in Figure 3.4.

We will discuss how to deploy structured reverse mode to the nested structure in details in Chapter 5. In the next chapter, Variable Annuities is introduced as a practical example of nested simulation with the structure that we've discusses at the end of this chapter.

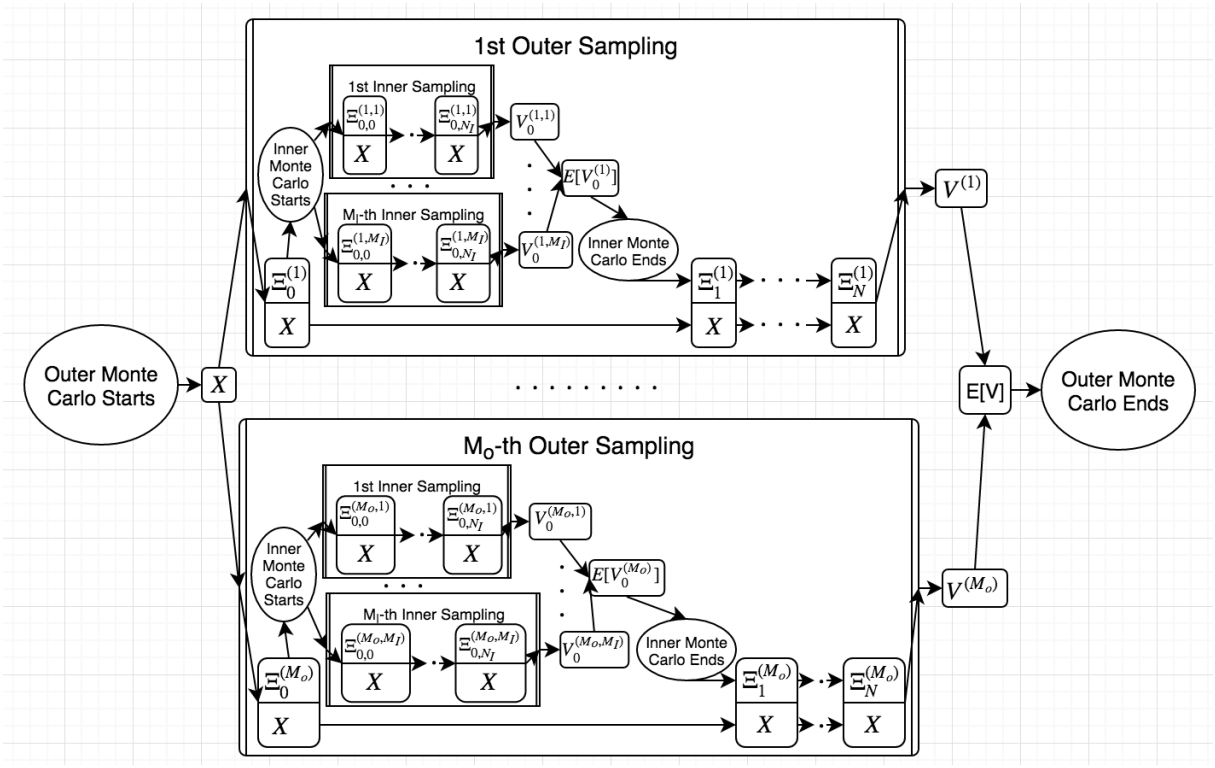

Figure 3.4: Abstract nested Monte Carlo simulation in financial applications

Chapter 4

Variable Annuities

This chapter introduces Variable Annuities (VA) and a program of VA pricing with minimal details. VA is a practical example suited for this thesis's analysis for its following properties: (1) nested Monte Carlo simulations in financial application (2) large scale computation whose runtime is prohibitive (3) derivatives are demanded and target function is scalar with high dimensional input. After full description of its computation, we show how VA's computations can be massage into the general structured template that the previous chapter presents in Figure 3.3.

4.1 General background

Variable Annuities, also known as "segregated funds", are deferred, fund-linked annuity contracts. It is composed of two parts: an investment in an underlying fund that the contract links to, and guarantees on top of the investment. The guarantees are various protections and benefits that an insurance company provides in case of contingencies. Such insurance products realize payoff annually hence the name 'annuities'. It is 'variable' for there are numerous possible combinations of guarantees and multiple forms of additional features on top of the guarantees such as ratchet, roll-up, step-up etc.

Once the contract is established, the policy holder pays the premium to the insurance company, which can be a one time up-front payment, or there can be a pay-out phase that lasts for several years. The premium received is invested in the linked funds and the value of the account will fluctuate with the fund's performance onwards. For the insurance company, each guarantee is an embedded option that the policy holder longs hence

itself shorts. The insurance company benefit from such product by charging an annual management fee on the fund account. The existence of each guarantee increases the priced annual fee to cover its negative payoff to the company. The pricing of the management fee is high enough such that the collected fees are expected to hit certain profitability target besides paying off the contingency claims and low enough to maintain competitiveness in the market.

The guarantees associated with variable annuities are called "GMxB" as self-explained by the names of common guarantees: Guaranteed Minimum Withdrawn Benefit (GMWB), Guaranteed Minimum Accumulation Benefit (GMAB), Guaranteed Minimum Death Benefit (GMDB), Guaranteed Minimum Income Benefit (GMIB) etc. Each guarantee has distinct contingency claims.

As with any embedded options, VA is priced in the risk neutral measure. On top of pricing, the company needs to hedge its exposure to price risk of the underlying funds as well as the interest rate risk exposure since VA contracts have very long duration. Other risk factors may require hedging as well. Such hedging naturally invites the deployment of AD. Since we care about the overall P&L of the portfolio including the hedging practice, nested Monte Carlo simulations are needed where the outer loop simulates real world process in P measure whereas the inner loop operates in risk neutral world Q measure.

4.2 Definition and formulation

For simplicity, we only deal with contracts involving GMWB and GMDB. We further simplify the accumulation phase, if any, to a single up-front premium payment.

4.2.1 Evolution of VA Accounts

We will describe the evolution of all the accounts associated with VA in details in accord with [14]. The linked fund will be referred to as the *underlying*. The account where the premium P is invested into is denoted as A. We denote the price of the linked fund at time t as S_t. A_t stands for the value of account A at year t. Hence $A_0 = P$. Each contract has an annually charged management fee ϕ as a percentage of the account A.

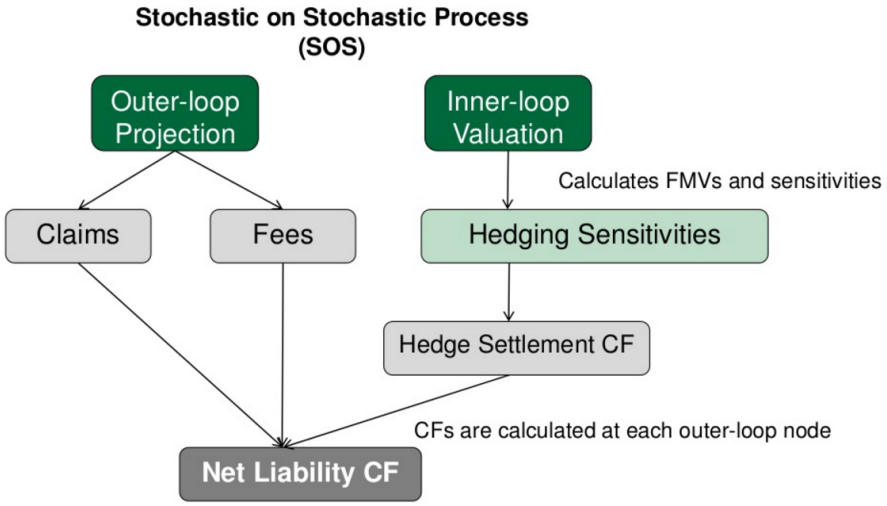

Figure 4.1: High level process of pricing and risk managing VA, credit from Anthony Vaz's Linkedin page's PPT *Complexities of Variable Annuity Management*, page 28

For GMWB, the contact specifies the guaranteed withdrawal amount (usually about 5%-8% of the premium P), which we denoted as G^E. G^E is the maximal allowed withdrawal each year. The maximal total withdrawal amount available at year t is denoted as G_t^W. Initially, $G_0^W = P$ and it decreases as withdraws occur.

GMDB is a money back guarantee upon the death of the insured. We denote the amount of guaranteed death benefit at year t to be G_t^D. Initially $G_0^D = P$ since it is money back guarantee. In the case the contract also has GMWB, G_t^D might decrease such that $G_{t>0}^D < P$ due to pro rata adjustments caused by withdrawals. In the case that $G^D < A$ upon death, the full account value A would be returned, not just the guaranteed death benefit G^D. Precisely, the evolution of A_t, G_t^W, G_t^D are:

- **Initialization:**

$$\begin{cases} A_0^+ &= P \\ G_0^W &= P \\ G^E &= P \cdot x_w \\ G_0^D &= P \end{cases} \tag{4.1}$$

where x_w is the rate of withdrawal as specified on the contract. Here we are modelling a contract with both GMWB and GMDB. If a contract only has GMDB or GMWB,

set $x_w = 0$ or $G_0^D = 0$ would suffice.

There are two periods in any year t. The first period is from t^+ to $(t+1)^-$ where the stock price S changes and consequently the account value A, the policy holder may decease within such period hence a GMDB contingency may occur; the second period is from $(t+1)^-$ to $(t+1)^+$, where the customer can withdraw money from the account if allowed hence a GMWB contingency may occur. Now we model the evolution of the VA account for each period.

- **From t^+ to $(t+1)^-$:**

$$\begin{cases} A_{t+1}^- &= A_t^+ \frac{S_{t+1}}{S_t}(1-\phi) \\ F_{t+1} &= A_t^+ \frac{S_{t+1}}{S_t}\phi \\ D_{t+1} &= \max\{0, G_t^D - A_{t+1}^-\} \end{cases} \tag{4.2}$$

here ϕ is the annualized rate of management fee. We have used F_{t+1} to denote the collected fee at the end of year t; and D_{t+1} denote the amount that the insurance company needs to pay in case of GMDB contingency. We see first the account value changes according to the underlying since it is linked to it, and then a fee is charged. If the policy holder dies, all money left in the account is returned and the company will pay the difference if account value is less than the GMDB.

- **From $(t+1)^-$ to $(t+1)^+$:**

$$\begin{cases} E_{t+1} &= \min\{G_t^W, G^E\} \\ G_{t+1}^W &= G_t^W - E_{t+1} \\ A_{t+1}^+ &= \max\{0, A_{t+1}^- - E_{t+1}\} \\ W_{t+1} &= \max\{0, E_{t+1} - A_{t+1}^-\} \\ G_{t+1}^D &= G_t^D \cdot \frac{A_{t+1}^+}{A_{t+1}^-} \end{cases} \tag{4.3}$$

we have used E_{t+1} to denote the amount of withdrawal by the policy holder at end of year t; W_{t+1} to denote the amount that the company needs to pay in case of GMWB contingency.

$$E_{t+1} = (A_{t+1}^- - A_{t+1}^+) + W_{t+1}$$

We see the withdraw is either covered by the account or the insurance company. Such identity is easy to prove using expression of A_{t+1}^+ and W_{t+1} as well the identities: $\min\{A, B\} + \max\{A, B\} = A + B$ and $-\max\{\cdot\} = \min\{-\cdot\}$.

In (4.3), the first equation says the withdrawal cannot exceed annual limit G^E as well as the total limit G_t^W. Second equation simply says the allowed maximal total withdrawal G^W decreases as the policy holder withdraws. Since $E_{t+1} \leqslant G_t^W$, $G_{t+1}^W \geqslant 0$ is always non-negative. The third equation says that the account value decreases as the policy holder withdraw E_{t+1} yet it cannot go negative. The fourth equation calculates the amount of GMWB contingency. The last equation is the so called pro rate adjustment of GMDB. Essentially the GMDB shrinks proportionally depending on the percentage of the withdrawal.

We see that the first equation here assumes that the customer withdraws the maximal allowed amount each year. For justification, see Appendix D.1.

- **End of contract: maturity**

  The above fully specifies the dynamics of variable annuity account with GMWB and GMDB until the maturity of the contract. Each contract has a maturity T after which the contract expires. At maturity, the account value is returned back to the customer. Some other guarantee offers certain benefits at maturity however GMDB and GMWB does not.

4.2.2 Calculation of variable annuities P&L

The profit comes from management fee, which is recorded in variable F_t, the loss comes from guarantee benefit paid to the policy holder, which is recorded in variable D_t, W_t. Weighting them by appropriate discount factors, we get the sum as the present value of the contract.

$$PV = \sum_{t=1}^{T} (F_t \mathbf{1}_{t-1}^A - D_t \mathbf{1}_{t-1}^D - W_t \mathbf{1}_t^A) disc_t \tag{4.4}$$

where $\mathbf{1}_t^D$ is a binary random variable that is 1 if the policy holder dies within t-th year (implying he/she is alive up to the beginning of t-th year). Similarly, $\mathbf{1}_t^A$ is a binary random variable that is 1 if the policy holder survives up to t-th year. Both variable depends on aliveness of the policy holder. $disc_t$ is the discount factor at year t. The interpretation is: fee F_t is always charged as long as the policy holder is alive at the beginning of t-th year; depending on whether the policy holder dies within t-th year, death or withdrawal benefit might incur.

Three sources of stochasticity/uncertainty are present: (1) Underlying processes, the stock price and interest rate etc. (2) The aliveness of the policyholder, such could be viewed as a two state Markov chain with death as the absorbing state (3) The behavior of the policy holder. (1) can be straight forwardly modelled and sampled via Monte Carlo. (3) has been simplified under maximal withdrawal assumption. For (2), we will simply use its exact mean and variance if needed since we assume the aliveness of the policyholder is independent of all other random variables.

According to common actuarial practice, we assume the mortality rate only depends on gender and current age. We denote the mortality rate as:

$$q_x^g = \text{Probability of a person aged } x \text{ and gender } g \text{ dies in the next year} \tag{4.5}$$

and probability of survival:

$${}_xp_t^g = \text{Probability of a person aged } x \text{ and gender } g \text{ survives the next } t \text{ years} \tag{4.6}$$

We drop the gender upper index afterwards. By definition, we have the following equations due to our Markovian assumption of live-death process:

$$\begin{cases} {}_xp_{t+1} & = {}_xp_t \cdot (1 - q_{x+t}) \\ {}_xp_0 & = 1 \end{cases} \tag{4.7}$$

Essentially, (1) the probability of surviving to the beginning of next year is the probability of surviving to beginning of the current year times the probability of surviving the current year (2) as the contract is still valid at $t = 0$, the policy holder must be alive hence ${}_xp_0 = 1$.

Since the live-death process is just a 0-1 binary random variable at each step:

$$\forall n \in \mathcal{Z}^+ \begin{cases} E[(\mathbf{1}_t^A)^n] & = {}_xp_t \\ E[(\mathbf{1}_t^D)^n] & = {}_xp_t \cdot q_{x+t} \end{cases} \tag{4.8}$$

when we have used the fact that for 0-1 binary variable $\mathbf{1}^n = \mathbf{1}$. Now we can take the expectation value over the policy holder's live-death status as:

$$PV = \sum_{t=0}^{T-1} {}_xp_t[F_{t+1} - D_{t+1}q_{x+t} - W_{t+1}(1 - q_{x+t})]disc_{t+1} \tag{4.9}$$

The discount factor will be given by the underlying interest rate process. With all described, the pricing computation of the VA contract is complete.

4.2.3 Hedging of Variable Annuities

Given the significant size of the portfolio, any risk factors should be properly hedged. As noted, we have at least price risk and interest rate risk to hedge. Indeed, in practice, delta-rho hedge is usually deployed. Second order hedge is rarely exercised hence *first order AD suffice for VA program*. For details of VA hedging, see Appendix D.2.

As insurance companies are regulated, capital/reserve needs to be calculated besides pricing. For details regarding regulations, see Appendix D.3.

4.3 Computational aspects of VA program

After describing the VA program in details, we go back to the computational schemes of the thesis.

4.3.1 Computing cost

We have mentioned that VA's runtime is prohibitive, let's make a qualitative estimate. The following parameters can be typical in practice: (1) A portfolio of $N = 10^5$ contracts (2) Maturity $T = 25$ years (3) Annual timestepping $\Delta t = 1$ year is used (4) Inner Monte Carlo sampling $M_I = 1000$ (5) Outer Monte Carlo sampling $M_O = 1000$. (5) $c = 30$ flops per contract per time step[1]. As result, $cNM_IM_O(T/\Delta t)^2/2 \approx 10^{15}$ flops are estimated for one run of nested Monte Carlo. Using a GHz (10^9 flops per second) machine, we need on the scale of 10^6 seconds, which is on the scale of week. If we want to price the contract, i.e. find out the appropriate annual management fee ϕ instead of calculating P&L with given management fee, more than one run would be required by iterative algorithms. The VA program is indeed computationally intense.

4.3.2 Structure of computation

We show that the VA program possesses the structure in Figure 3.4.

[1] 30 is estimated by ∼8 flops (Operations involving $A^+, A^-, D, E, W, GW, GD$) with a factor of ∼4 from the extra calculation of derivatives.

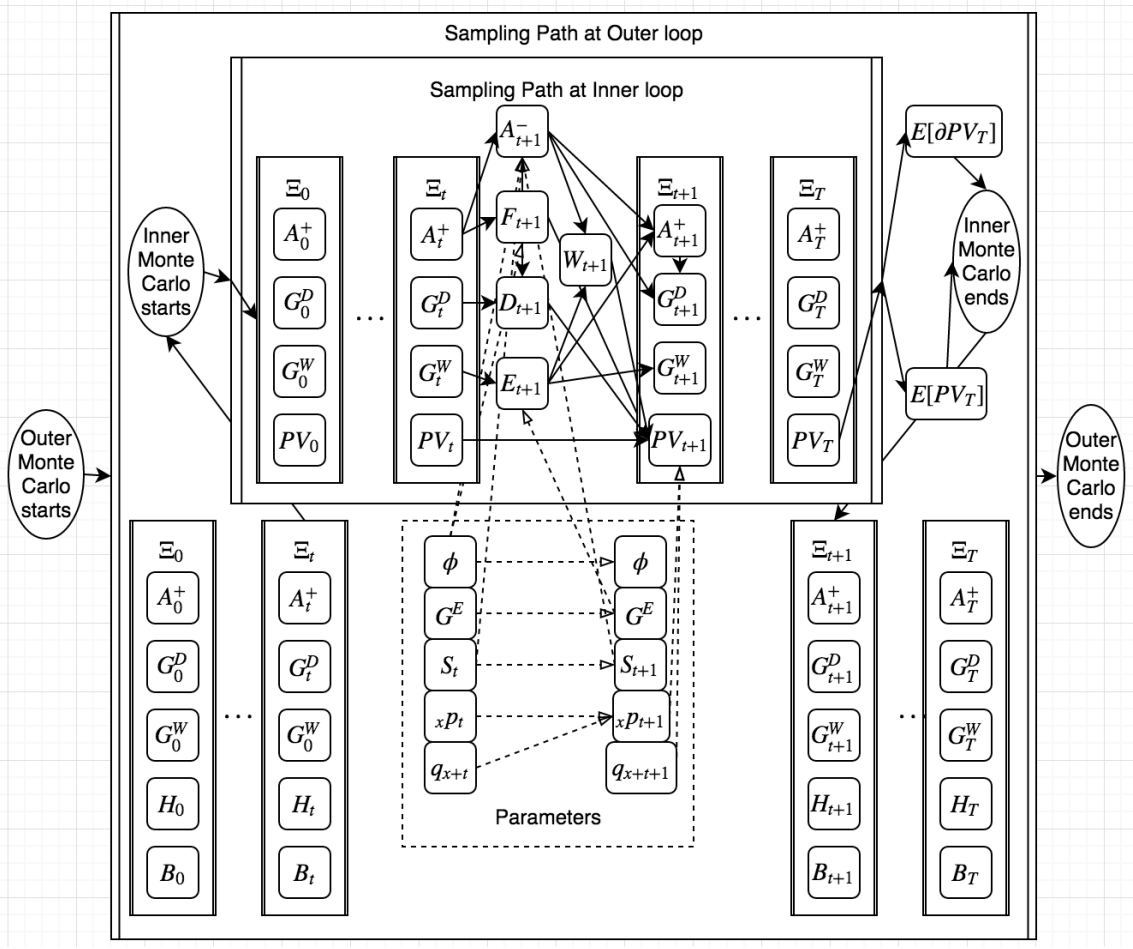

Figure 4.2: Wrapping up state vectors and parameters of the VA program

For VA, the state vector at inner loop $\Xi_{n,m}$ would be (A^+, G^W, G^D, PV) where PV is the cummulative present value of VA contract in risk neutral world; the state vector at outer loop Ξ_n is (A^+, G^W, G^D, H, B) where H are the holdings in the hedging instruments, B is the bank account. After completion of the inner loop Monte Carlo, the present value of VA is computed, its Greek with respect to relevant risk factors are computed at the same time by AD. All are feedback to the outer loop node and the state vector at the outer loop is iterated.

We conclude that VA program does follow the general structure as in Figure 3.4.

4.4 Concluding remarks

From detailed descriptions in this chapter, we observe the VA program has the following properties indeed (1) nested Monte Carlo simulations in financial application (2) large scale computation whose runtime is prohibitive (3) first order derivatives are demanded and target function is scalar with high dimensional input. Therefore, a matching target for deploying structured reverse mode AD. Furthermore, VA's computing structure does fit into our general framework as demonstrated in Figure 3.4.

In the next chapter, detailed analysis on deploying structured reverse mode AD on nest simulations will be discussed. VA program will serve the numerical experiments in the next chapter.

Chapter 5

Structured Reverse Mode in nested simulations

In Chapter 2, we have introduced the fundamentals of structured reverse mode and drawn useful insights from a simple example; in Chapter 3 and 4, we have revealed the general structures in single layered and nested Monte Carlo simulation as well as its application to VA. Now we are ready to combine everything together and thoroughly analyze various approaches to deploy structured reverse mode to nested simulation. Numerical results from VA would be used to corroborate the theoretical analysis.

5.1 Utilization of structure

At the end of Chapter 2, the exact optimal structure has been computed on a clean simple example: the optimal number of checkpoints as each level, the optimal number of level for memory reduction, even the Euler number e shows up. Such accuracy is not necessary nor practical[1]. We focus on straight forward implementation, i.e. taking the natural checkpoint structure in Monte Carlo simulation in financial application as in Figure 3.4 without

[1]Code segments between the elementary Markovian checkpoints might vary in their computational burden, in which case finding the optimal checkpoints distribution is non-trivial and only offers marginal improvement over the straight forward checkpoints structure. In addition, the optimal checkpoint position depends on the memory requirement of all code segments, which is not known in advance, therefore impossible to implement in advance.

finer selection.

The most important aspects of structured reverse mode are the revelation and efficient utilization of structure. The revelation of structure has been covered in Chapter 3 for Monte Carlo simulations in financial applications, however efficient utilization of the revealed structure is non-trivial and counter-intuitive for nested simulations as we will see in this section.

5.1.1 Merit of Structure

Definition 5.1. Merit of Structure α_a for a code c using a structured reverse mode approach a is defined as:

$$\alpha_a = \frac{\sigma(\partial_{\text{Rev}} c)}{\sigma(\partial_a c)} \tag{5.1}$$

Larger α_a is, more beneficial to use structured reverse mode a. If $\alpha \approx 1$, then the space improvement using structure is poor.

For a Monte Carlo simulation with M sampling and state vector Ξ, we have the following admissible checkpoints selection:

$$c^{(i)} = \begin{cases} \Xi_1^{(i)} & = W_1(X) \\ \Xi_2^{(i)} & = W_2(X, \Xi_1^{(i)}) \\ & \vdots \\ \Xi_L^{(i)} & = W_L(X, \Xi_{L-1}^{(i)}) \\ V^{(i)} & = W_{L+1}(X, \Xi_L^{(i)}) \end{cases}$$

$$z = \frac{1}{M} \sum_{i=1}^{M} V^{(i)}$$

where L is the length of each individual path, typically the number of time steps. c_i stands for the code of each sampling. Since the state vectors are in the same form, we denote:

$$\sigma(\Xi_n^i) = \sigma_0 \quad \forall i \in \{1, \cdots, M\}, n \in \{1, \cdots, L\} \tag{5.2}$$

Now recall the fundamental equation (2.23):

$$\sigma(\partial_{\text{StrRev}} c_i) = \sum_{i=1}^{L} \sigma(\Xi_i) + \max_{i=1}^{L+1} \bar{\sigma}(W_i) = L\sigma_0 + \max_{i=1}^{L+1} \bar{\sigma}(W_i) \tag{5.3}$$

whereas the bare reverse mode costs memory of:

$$\sigma(\partial_{\text{Rev}} c_i) = \sum_{i=1}^{L+1} \bar{\sigma}(W_i) \tag{5.4}$$

$\bar{\sigma}(W_i)$ might depend on index i, yet asymptotically, $\sum_{i=1}^{L+1} \bar{\sigma}(W_i) \sim L \cdot \max_{i=1}^{L+1} \bar{\sigma}(W_i)$. Hence we have the following *merit of structure* for straightforward implementation:

$$\alpha_{\text{str}} = \frac{\sum_{i=1}^{L+1} \bar{\sigma}(W_i)}{L\sigma_0 + \max_{i=1}^{L+1} \bar{\sigma}(W_i)} = \frac{\sum_{i=1}^{L+1} \Omega(W_i)}{L + \max_{i=1}^{L+1} \Omega(W_i)} \sim \frac{L\Omega}{L+\Omega} \leqslant \min\{L, \Omega\} \tag{5.5}$$

We see that α_{str} is greater when both the length of the sequential path (timestepping) L and W's typical CPI ratio[2] Ω increase, and is upper bounded by them. Number of time steps L is usually large, 10^3 for example, as is required by accuracy. Therefore W's CPI ratio Ω becomes crucial in structured reverse mode's efficacy.

Notice that we have assumed all computations are done in pathwise fashion hence the number of sampling M does no show up. As such parallelism seems trivial to be explored for single layered Monte Carlo, it is not for nested simulations.

5.1.2 Nested Simulations

Denote the number of outer loop sampling as M_o and the one of inner M_I. The length of the outer loop sampling path being L, the length of inner loop sampling path at time step n at outer loop being L_n. The function L_n depends on specific applications and as we will see below, affects whether or not it is worth diving into the deepest structure in the simulation. We will discuss the merit of structure for all possible combinations of approaches.

Pathwise[1]-Structure[1]

Nested simulation can be viewed as single layered simulation with another single layered simulation as the sampling drawing algorithm.

[2]For definition of CPI ratio, see 2.35.

$$c^{(i)} = \begin{cases} \Xi_1^{(i)} & = W_1(X) \\ \Xi_2^{(i)} & = W_2(X, \Xi_1^{(i)}) \\ & \vdots \\ \Xi_L^{(i)} & = W_L(X, \Xi_{L-1}^{(i)}) \\ V^{(i)} & = W_{L+1}(X, \Xi_L^{(i)}) \end{cases} \qquad c_n^{(i,j)} = \begin{cases} \Xi_{n,1}^{(i,j)} & = W_{n,1}(X, \Xi_n^{(i)}) \\ \Xi_{n,2}^{(i,j)} & = W_{n,2}(X, \Xi_{n,1}^{(i,j)}) \\ & \vdots \\ \Xi_{n,L}^{(i,j)} & = W_{n,L_n}(X, \Xi_{n,L_n-1}^{(i,j)}) \\ V_n^{(i,j)} & = W_{n,L_n+1}(X, \Xi_{n,L_n}^{(i,j)}) \end{cases}$$

$$z = \frac{1}{M_o} \sum_{j=1}^{M_o} V^{(i)} \qquad \Xi_{n+1}^{(i)} = \frac{1}{M_I} \sum_{j=1}^{M_I} V_n^{(i,j)} \tag{5.6}$$

Hence previous analysis apply. $W_{(\cdot)}$ as the sample drawer for the outer loop, are single layered simulation now. Hence $\Omega(W_{(\cdot)})$ at least has a factor of M_I, which is 10^3 at minimal. Therefore we expect the merit of structure for structuring the outer loop $\alpha_{\text{str}} \gg 1$. Therefore structuring the outer Monte Carlo simulation should be preferred. To be more precise, denote:

$$\Omega(W_{n,m}) = \Omega_0 \quad \forall n \in \{1, \cdots, L\}, m \in \{1, \cdots, L_n + 1\} \tag{5.7}$$

It follows that:

$$\Omega(W_n) = \sum_{j=1}^{M_I} \sum_{m=1}^{L_n} \Omega_0 = M_I L_n \Omega_0 \tag{5.8}$$

L_n depends on specific applications. However since $\Omega(\cdot) \geqslant 1$, $\Omega(W_n) \geqslant M_I L_n \gg L_n \geqslant 1$, according to (5.5):

$$\alpha_{\text{path}^1\text{str}} = M_o \frac{\sum_{i=1}^{L} \Omega(W_i)}{L + \max_{i=1}^{L} \Omega(W_i)} = M_o \frac{M_I \Omega_0 \sum_{i=1}^{L} L_n}{L + M_I \Omega_0 \max_{i=1}^{L} \{L_n\}} \leqslant M_o L \tag{5.9}$$

The M_o factor in front is due to pathwising the outer loop. We observe though CPI ratio contains the factor M_I, the merit of structure of plain structured reverse mode at outer loop does not. This is unsatisfactory because M_I is typically large. A natural idea would be to see the merit of structure when both inner loop and outer loop are structured.

Pathwise^2-Structure^2

Assume pathwise on both levels. At inner loop, we store checkpoint variables $\Xi_{n,m}^{(i,j)}$ costing $L_n \sigma_0$ space. The reverse mode on $W_{(\cdot,\cdot)}$ requires $\Omega_0 \sigma_0$ storage; at outer loop, we store

checkpoint variables $\Xi_n^{(i)}$, costing $L\sigma_0$ space. The reverse mode on $W_{(\cdot)}$ is the structured reverse mode on $W_{(\cdot,\cdot)}$ hence no more space is counted. Therefore:

$$\sigma(\partial_{\text{path}^2\text{str}^2} c) = (L + \max_{n=1}^{L}\{L_n\})\sigma_0 + \Omega_0\sigma_0 \tag{5.10}$$

$$\alpha_{\text{path}^2\text{str}^2} = M_o M_I \frac{\Omega_0 \sum_{i=1}^{L} L_n}{L + \max_{n=1}^{L}\{L_n\} + \Omega_0} \tag{5.11}$$

We see the merit of structure of two layers of structure is at least on the order of $M_o M_I$, offering a considerable improvement over Pathwise[1]-Structure[1]. Yet is it due to the pathwise or structure in the inner loop?

Pathwise[2]-Structure[1]

If we only use the outer loop structure but pathwise at both levels, we would need $L\sigma_0$ to store the outer loop checkpoints, and $\sigma_0\Omega_0 \max_{n=1}^{L} L_n$ instead of $M_I\sigma_0\Omega_0 \max_{n=1}^{L}\{L_n\}$ for use of reverse mode.

$$\sigma(\partial_{\text{path}^2\text{str}} c) = (L + \Omega_0 \max_{n=1}^{L}\{L_n\})\sigma_0 \tag{5.12}$$

$$\alpha_{\text{path}^2\text{str}} = M_o M_I \frac{\Omega_0 \sum_{i=1}^{L} L_n}{L + \Omega_0 \max_{n=1}^{L}\{L_n\}} \tag{5.13}$$

Such merit of structure is at the scale of $\sim M_o M_I$ as well. We still observe a strict improvement over Pathwise[1]-Structure[1]. While Pathwise[2]-Structure[2] seems to be a strict improvement over Pathwise[2]-Structure[1], such improvement could be marginal depending on L_n's behavior. Moreover, as a 2-level structure, Pathwise[2]-Structure[2] needs one additional run of the original function and is more complex to implement.

Now let's observe the interaction between structure and pathwise approaches.

Pathwise[1]

Only pathwise the outer loop brings a factor of M_o improvement over naive reverse mode:

$$\alpha_{\text{path}^1} = M_o \tag{5.14}$$

Pathwise^2

One might think pathwising both loops simply brings factor of $M_o M_I$ improvement yet it is not true. Since no structure is used, the reverse mode cannot really pathwise the inner loop since the path structure at the outer loop is given to reverse mode as a blackbox and the inner pathwise structure lives in each node of the path inside the blackbox. As a result, pathwising the computation at inner loop does save space for original function valuation, yet the reverse mode is still recording all the results for the inner loop at the same time, giving a merit of structure at M_o, no improvement over Pathwise^1 at all.

$$\alpha_{\text{path}^2} = M_o \tag{5.15}$$

5.1.3 Application to VA contract pricing

With general insights from previous section, we apply the analysis to VA contract pricing. Consider annual timestepping, hence the path length of the inner loop L_n would be:

$$L_n = L - n + 1 \tag{5.16}$$

as there are less and less year to project later in the future. The CPI ratio Ω_0 for VA at inner loop is $O(1)$, about $2 \sim 3$. Effectively $\Omega_0 \ll L$.

Plugging in we have the merit of structure for VA contract pricing as summarize in Table 5.1. Notice that since each contract is priced separately hence we do not include the number of contract N factor here as it is irrelevant.

We see that Pathwise^2-Structured^2 and Pathwise^2-Structured^1 have similar performance since the CPI factor Ω is not large enough for $\Omega/2 \gg 1$. Taking the longer runtime and more complex implementation of Pathwise^2-Structured^2 into account, we conclude Pathwise^2-Structured would be the best choice in terms of runtime and space trade off for VA program.

5.1.4 Important Points

A few observations are in order based on the space counting of the doubly nested VA problem:

Approach	Checkpoints	ReverseAD	Overall	Merit
Straight forward	0	$M_oM_I\Omega L^2/2$	$M_oM_I\Omega L^2/2$	1
Structured[1]	M_oL	$M_oM_I\Omega L$	$M_oM_I\Omega L$	$L/2$
Structured[1,2]	M_oM_IL	$M_oM_I\Omega$	$M_oM_I(\Omega+L)$	$\approx \Omega L/2$
Pathwise[1]	0	$M_I\Omega L^2/2$	$M_I\Omega L^2/2$	M_o
Pathwise[1]-Structured[1]	L	$M_I\Omega L$	$M_I\Omega L$	$M_oL/2$
Pathwise[1]-Structured[1,2]	M_IL	$M_I\Omega$	$M_I(\Omega+L)$	$\approx M_o\Omega L/2$
Pathwise[2]	0	$M_oM_I\Omega L^2/2$	$M_oM_I\Omega L^2/2$	1
Pathwise[2]-Structured[1]	M_oL	$M_o\Omega L$	$M_o\Omega L$	$M_IL/2$
Pathwise[2]-Structured[1,2]	$2M_oL$	$M_o\Omega$	$M_o(\Omega+2L)$	$\approx M_I\Omega L/4$
Pathwise[1,2]	0	$M_I\Omega L^2/2$	$M_I\Omega L^2/2$	M_o
Pathwise[1,2]-Structured[1]	L	ΩL	ΩL	$M_oM_IL/2$
Pathwise[1,2]-Structured[1,2]	$2L$	Ω	$\Omega+2L$	$\approx M_oM_I\Omega L/4$

Table 5.1: Approaches and their space requirement decomposition for VA problem. All space are in units of σ_0, the space needed to store 1 state vector of the problem. $M_o, M_I, L \gg 1$ is assumed and subleading term is neglected. $L \gg \Omega \geqslant 1$.

(1) Without structure at the outer loop, we wouldn't be able to exploit the pathwise structure in the inner loop at all. As we can see, Pathwise[2] requires the same amount of space as with Pathwise. Hence only using pathwise implementation in nested Monte Carlo cannot even exploit all levels of *parallel path* structure but only the first one, emphasizing the importance of *sequential* structure utilization.

(2) With structure at the outer loop, the path structure of the inner loop is only partially exploited: Pathwise-Structured's checkpoints memory does not have M_I factor, the reverse AD memory stills scales with M_I. Hence it is always better to fully exploit path structure to the deepest level and it is only possible when the second deepest level has been structured.

(3) It is not always better to apply structure to every level of nested Monte Carlo. As we can see the space complexity of the Pathwise[2]-Structured[2] is not significantly reduced from Pathwise[2]-Structured. Whether or not sequential structure should be used at the last level of nested Monte Carlo needs discretion while it is generally beneficial to apply sequential structure up to the second to last level.

5.2 Implementation and Experiments

We now move to implementation of structured reverse mode and numerical experiments on VA.

5.2.1 Template for Structured Reverse mode AD

We choose Matlab as implementation environment and use ADMAT 2.0 as our AD tools.

The pseudo code for the following matlab program can be found in Appendix E. Random seeding is not important in computational aspects as we are able to complete all analysis without mentioning it at all, yet it is crucial to handle them properly for implementation. One reason being that AD essentially can only handle deterministic algorithms or algorithms with given pseudo randomness. In addition, correct pathwise differentiation for Monte Carlo simulations cannot work at all without explicitly keeping track of random seeds.

We have showed the code template for Structured[1] approach in Listing 5.1. In the implementation, the first argument is the input x. The second argument W is the matrix associated with the reverse mode[3]. The third argument *Para* is a struct that contains all the inputs that we do not require derivatives on. Random seeds and other "global information" are also stored in and referenced from Para.

The structure pattern is encoded within the code and is inherited from the input as well. The fourth input *node_ I* is essentially the function handle of $W_{(\cdot,\cdot)}$ in Equation (5.6) "node transition for the Inner loop". The fifth input *init_ O* is the function handle of W_1 that initialize the state vector for the outer loop at the beginning of the sampling path "Initialization of outer loop state vector". The sixth input *transit_ O* is the function handle of $W_{>1}$ that transit the state vector from one to the next one at the outer loop "transition process at the Outer loop". The last input N is the number of time steps/length of the sequential path at the outer loop.

Listing 5.1: Structured[1] reverse mode AD template

[3]The dimension of W is $m \times d_W$ where m is the dimension of output z and $d_W \leqslant m$. In case of scalar output, W=1.

```
function [f,Df,tapeSize] = AD_str_vec(x,W,Para,node_I,init_O,transit_O,N)
%% Obtain size
dimX = numel(x);
[~,dimW] = size(W);
%% Forward Sweep
% Preallocate for seeds
Seeds_O(N+1) = rng;     % Not N+2 since no emit_O

Seeds_O(1) = rng;
Para.Seed = Seeds_O(1);

y_O = cell(1,N+1);  % Outer loop checkpoints
y_O{1} = feval(init_O,x,Para);
for i = 1:N
    % Outer loop action
    Seeds_O(i+1) = rng;
    Para.Seed = Seeds_O(i+1);
    Para.t = i-1;
    % Para.Path_Index_O is already set
    y_O{i+1} = feval(transit_O,[x;y_O{i}],Para);
end
%% Backward Sweep
%% 1. Reverse mode at last step: Initialize w, vf

% W: N+1(dimf) * dimW;

f = zeros(N+1,1);
w = cellmat(1,N+1,1,1); % dimNode * dimW matrix each
Df = zeros(dimX,dimW);

% Keep track of maximum tape!
tapeSize = 0;

for t = N:-1:0
    Para.Seed = Seeds_O(t+1);
    Para.t = t; % Pass current time to Para
    [f(t+1),JT,tapeSize] = Reverse_Combo([x;y_O{t+1}],node_I,1,1,Para,
       tapeSize);
```

```
    Df = Df + JT(1:dimX,:) * W(t+1,:);
    w{t+1} = JT(dimX+1:end,:);
end

%% 2. Propagate derivative back

v = cellmat(1,N+1,1,1); % dimNode * dimW matrix each

% Note that j = t+1 in the below loop.
for j = N:-1:1
    w{j+1} = w{j+1} + v{j+1};
    Para.Seed = Seeds_O(j+1);
    Para.t = j-1;
    [~,JT,tapeSize] = Reverse_Combo([x;y_O{j}],transit_O,size(y_O{j+1},1),w
        {j+1},Para,tapeSize);
    Df = Df + JT(1:dimX,:) * W(j,:);   % AD contribution of x
    v{j} = v{j} + JT(dimX+1:end,:);    % AD contribution of y_O{j}
end

%% 3. Handle the initial step

w{1} = w{1} + v{1};
Para.Seed = Seeds_O(1);

[~,dvf] = Reverse_Combo(x,init_O,size(y_O{1},1),w{1},Para);

Df = Df + dvf;
```

Various approaches introduced in previous section have been implemented to corroborate the analysis. We leave the explicit code of the templates in Appendix E.

5.2.2 VA program

Since we do not have real data of Variable Annuity contracts, we artificially generate contracts and policy holders assuming a reasonable distribution of all properties.

Evolution of VA account

The evolution of VA block is plotted in Figure 5.1.

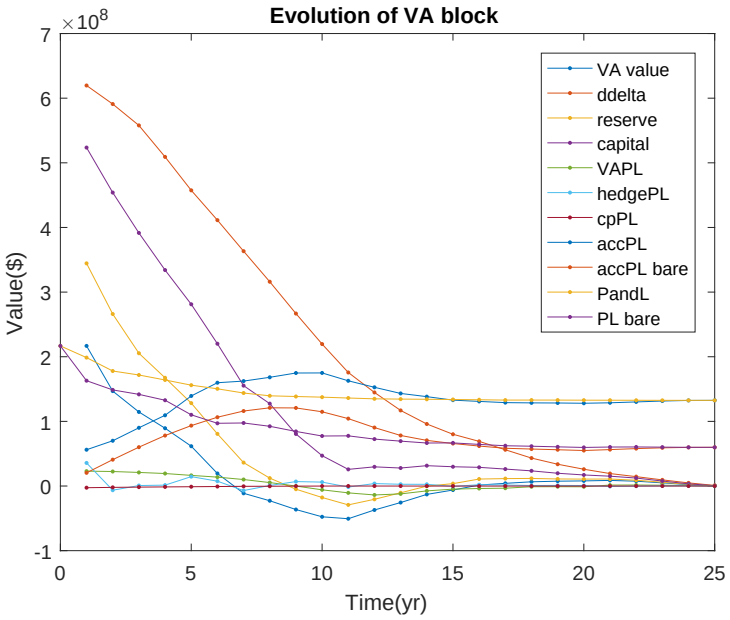

Figure 5.1: Demonstration of the evolution of VA block

Among all the details, we observe the highest two curves are the projected value of VA and the dollar delta of VA. It decays linearly since the policy holder is withdrawing the same amount of money annually by assumption.

Effects of Hedging

The hedged and unhedged P&L of a VA program are compared in Figure 5.2. The plotted lines in the upper part is the accumulative present value P&L trajectory of the whole VA block. We can see the hedged P&L is much less volatile than the unhedged P&L. The lower part of the figure is plotting annual P&L of the VA block, the hedge and together. We see the hedge position shows strong negative correlation with the value of VA block P&L resulting in a overall P&L of much smaller volatility.

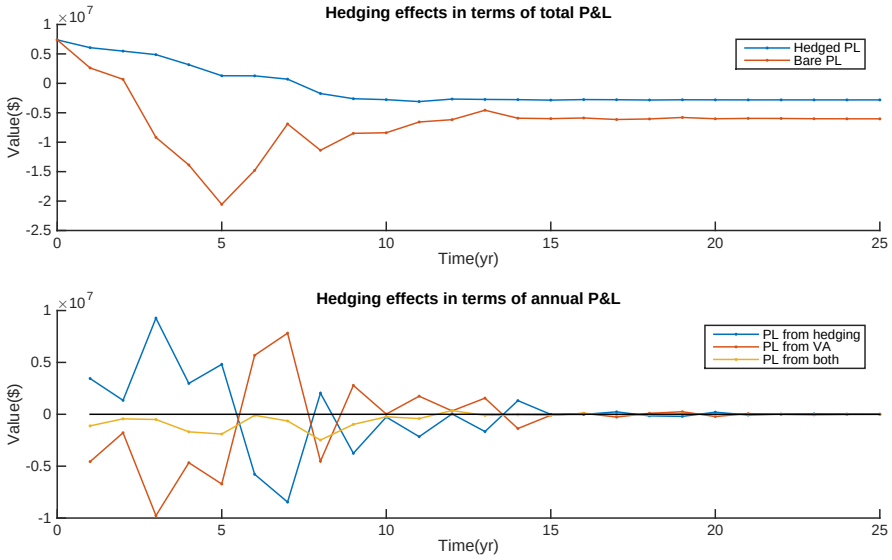

Figure 5.2: Demonstration of hedging effect on VA product

5.2.3 Numerical results

Reverse mode versus forward mode

As shown in the Chapter 1, for a scalar function, reverse mode's runtime ratio is independent of the input size whereas forward mode's runtime ratio scales linearly with the input size. The loglog plot in Figure 5.3 verifies such scaling behavior.

The crossover in the experiment happens when the input size is about 10^2. Hence it verifies the motivation to use reverse mode in large scale computation.

Merit of Structure

The comparison between different approaches on the same nested simulation is presented in Table 5.2. We see the structured reverse mode not only offers promised memory reduction in accord with theory, it runs faster as well due to less intensive usage of non-local memory. It is worth noting that the VA program in Table 5.2 is tiny as it only has 5 sampling at outer loops, 1000 sampling at inner loop and 2 contracts in total. Yet it already requires

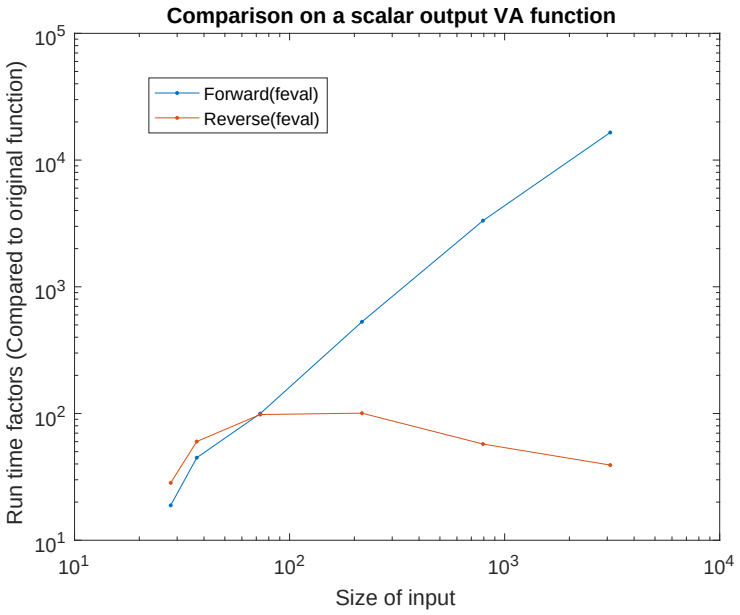

Figure 5.3: Runtime ratio comparison between forward mode and reverse mode on a scalar function using VA program

Approach	Runtime	Memory	Merit(exp)	Merit(Theory)
Naive	1.01 min	2.37 GB	1	1
Struct[1]	27.3 s	1.08 GB	2.2*	12.5
Path[1]	59.1 s	487 MB	5	5
Path[1]-Struct[1]	10.1 s	19.4 MB	125.04	125
Path[1,2]-Struct[1]	14.6 min*	19.0 KB	1.31×10^5	1.25×10^5

Table 5.2: Comparison of runtime and memory of different approaches on the same VA program. Maturity $T = 25$, i.e. length of path $L = 25$. $M_o = 5$, $M_I = 1000$, $N = 2$.

more than 2GB memory due to its remarkable complexities.

We see some merit of structure of numerical results are quite different from theory. We note that tracking of memory usage of the particular AD package used in Matlab environment is tricky. The theoretic merit value is not precise and is rather with an order of magnitude accurary.

5.3 Conclusion and Future works

In this thesis, we revealed the general structure of Monte Carlo simulations in financial applications and investigated all possible deployments of structured reverse mode utilizing the structure.

Monte Carlo simulations in financial applications have the following universal structure: (1) Individual sampling structure, of which can be utilized to parallelize computation (2) Each sampling is a sequential time stepping path. (3) Depending on the instrument in question, specific form of state vector could be extracted and naturally serves as a Markovian admissible selection of checkpoints.

For nested Monte Carlo simulation, general observations on deploying of structured reverse mode have been made: (1) It is always beneficial to parallelize the individual sampling *and* structure the sequential timestepping computation at outer loop. Without structuring the outer loop, the individual sampling structure at the inner loop is enclosed in blackbox and cannot be exploited. (2) Parallelizing both sampling loops and structuring the outer loop is enough to provide a factor of $M_o M_I$ reduction over the space ratio of bare reverse mode. It is the best approach among one level structured reverse mode AD. (3) Parallelizing both sampling loops and structuring both loops may further reduce space ratio however only marginally. As for Variable Annuities, it does not reduce memory requirement yet has longer runtime for being 2-level structure. (4) Excessive use of memory might lead to longer runtime. Structured reverse mode outperforms bare reverse mode by smaller memory *and* shorter runtime as shown in numerical experiments.

Hence the final conclusion for deploying structured reverse mode AD to nested Monte Carlo simulation is: to structure the outer loop and pathwise both loops. Extract the state vectors for the underlying financial instrument first and then identify the fully wrapped node transition processes (timestepping). Then structured reverse mode can be used: the pseudo code, algorithm, and working implementation of template can be found in this thesis.

Future works could be devoted to more fine tuned optimal checkpoint selection. Though we have discussed the optimal checkpoint selections in Chapter 2, we are only using the natural checkpoints in the final chapter. It is true that such optimal choice depends on parameters that are not known before the program runs, yet it is nevertheless possible to

extract such information, such as Ω_0, by completing a single inner loop simulation and then adjust accordingly. In principle, assuming the universal structure of nested Monte Carlo simulation in finance, full automation of optimal checkpoint selection is achievable. However complex the actual code might be, wrapped full automation can leads to significant ease for users with less knowledge of AD, making the runtime/space efficiency power of structured reverse mode more accessible.

References

[1] Andreas Griewank. *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, 2000, second edition 2008.

[2] Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7, 1970.

[3] Andreas Griewank. On automatic differentiation. In *Mathematical Programming - Recent Developments and Applications*, pages 83–108, Amsterdam, 1989. Kluwer Academic publishers.

[4] Christian H. Bischof, Ali Bouaricha, Peyvand M. Khademi, and Jorge J. More. Computing gradients in large-scale optimization using automatic differentiation. *INFORMS Journal on Computing*, 9(2):185–194, 1997.

[5] Luca Capriotti and Michael B Giles. Algorithmic differentiation: adjoint greeks made easy. *Available at SSRN 1801522*, 2011.

[6] Luca Capriotti, Shinghoi Jacky Lee, and Matthew Peacock. Real time counterparty credit risk management in monte carlo. *Risk Magazine (in press)*, 2011.

[7] Luca Capriotti. Fast greeks by algorithmic differentiation. *Available at SSRN 1619626*, 2010.

[8] Yu.M. Volin and G.M. Ostrovskii. Automatic computation of derivatives with the use of the multilevel differentiating technique - 1. algorithmic basis. *Computers & Mathematics with Applications*, 11(11):1099 – 1114, 1985.

[9] A. Griewank and A. Walther. Revolve - reverse or adjoint mode of computational differentiation. *Transaction on Mathematical Software*, 26(1), 2000.

[10] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992.

[11] Thomas F. Coleman and Jorge J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20(1):187–209, February 1983.

[12] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What color is your jacobian? graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005.

[13] Christian H. Bischof, Peyvand M. Khademi, Ali Buaricha, and Carle Alan. Efficient computation of gradients and jacobians by dynamic exploitation of sparsity in automatic differentiation. *Optimization Methods and Software*, 7(1):1–39, 1996.

[14] Daniel Bauer, Alexander Kling, and Jochen Russ. A universal pricing framework for guaranteed minimum benefits in variable annuities. *ASTIN Bulletin: The Journal of the International Actuarial Association*, 38(02):621–651, 2008.

[15] Wei Xu, Xi Chen, and Thomas F. Coleman. The efficient application of automatic differentiation for computing gradients in financial applications. *Journal of Computational Finance*, 19(3):71–96, 2016.

[16] Parsiad Azimzadeh, Peter A. Forsyth, and Kenneth R. Vetzal. *Hedging Costs for Variable Annuities Under Regime-Switching*, pages 133–166. Springer US, Boston, MA, 2014.

[17] Guojun Gan. Application of data clustering and machine learning in variable annuity valuation. *Insurance: Mathematics and Economics*, 53(3):795 – 801, 2013.

[18] Guojun Gan and X. Sheldon Lin. Valuation of large variable annuity portfolios under nested simulation: A functional data approach. *Insurance: Mathematics and Economics*, 62:138 – 150, 2015.

[19] Thomas F. Coleman and Gudbjorn F. Jonsson. The efficient computation of structured gradients using automatic differentiation. *SIAM Journal on Scientific Computing*, 20(4):1430–1437, 1999.

[20] Arun Verma. *Structured Automatic Differentiation*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1998.

[21] A.Griewank. Some bounds on the complexity of gradients, jacobians, and hessians. In P.M. Pardalos, editor, *Complexity in Nonlinear Optimization*, pages 128–161. World Scientific publishers, 1993.

[22] T. Coleman and W. Xu. *Automatic Differentiation in MATLAB Using ADMAT with Applications*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2016.

[23] Thomas F. Coleman and Wei Xu. Fast (structured) newton computations. *SIAM Journal on Scientific Computing*, 31(2):1175–1191, 2009.

APPENDICES

Appendix A

Calculation of derivatives in structured reverse mode

Structured reverse mode only differs from vanilla reverse mode in the dealing with intermediate variables and memory, they propagate the derivatives in the same way.

Suppose the code c is decomposed as follows:

$$\begin{cases} y_1 &= W_1(x) \\ y_2 &= W_2(x, y_1) \\ &\vdots \\ y_k &= W_k(x, y_1, \cdots, y_{k-1}) \\ z &= W_{k+1}(x, y_1, \cdots, y_k) \end{cases} \tag{A.1}$$

where y could be an admissible selection of checkpoints and W is the wrapped code segment, or y is the intermediate variables and W is basic operations. For definition of admissible selection of checkpoints see Section 2.2.1. Similar to Equation (2.3), we have

the following extended Jacobian:

$$J_E = \begin{pmatrix} \frac{\partial W_1}{\partial x} & -I & O & \cdots & \cdots & \cdots & O \\ \frac{\partial W_2}{\partial x} & \frac{\partial W_2}{\partial y_1} & -I & \ddots & \cdots & \cdots & \vdots \\ \frac{\partial W_3}{\partial x} & \frac{\partial W_3}{\partial y_1} & \frac{\partial W_3}{\partial y_2} & -I & \ddots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \frac{\partial W_{k-2}}{\partial x} & \frac{\partial W_{k-2}}{\partial y_1} & \frac{\partial W_{k-2}}{\partial y_2} & \cdots & \frac{\partial W_{k-2}}{\partial y_{k-3}} & -I & O \\ \frac{\partial W_{k-1}}{\partial x} & \frac{\partial W_{k-1}}{\partial y_1} & \frac{\partial W_{k-1}}{\partial y_2} & \cdots & \cdots & \frac{\partial W_{k-1}}{\partial y_{k-2}} & -I \\ \frac{\partial W_k}{\partial x} & \frac{\partial W_k}{\partial y_1} & \frac{\partial W_k}{\partial y_2} & \cdots & \cdots & \cdots & \frac{\partial W_k}{\partial y_{k-1}} \end{pmatrix}$$

$$\equiv \begin{pmatrix} J_x^1 & -I & O & \cdots & \cdots & \cdots & O \\ J_x^2 & J_{y_1}^2 & -I & \ddots & \cdots & \cdots & \vdots \\ J_x^3 & J_{y_1}^3 & J_{y_2}^3 & -I & \ddots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ J_x^{k-2} & J_{y_1}^{k-2} & J_{y_2}^{k-2} & \cdots & J_{y_{k-3}}^{k-2} & -I & O \\ J_x^{k-1} & J_{y_1}^{k-1} & J_{y_2}^{k-1} & \cdots & \cdots & J_{y_{k-2}}^{k-1} & -I \\ J_x^k & J_{y_1}^k & J_{y_2}^k & \cdots & \cdots & \cdots & J_{y_{k-1}}^k \end{pmatrix} \tag{A.2}$$

Reverse mode AD has the input of a W matrix, and the output it computes is the product $J^T W$ where J is the Jacobian matrix of the target function. Such call to bare reverse mode is denoted as:

$$[z, J^T W] = \text{reverseAD}(x, c, W) \tag{A.3}$$

where c is the code to differentiated. Hence we have the following algorithm [15]:

Note in the comments we have related the intermediate variables $\nabla f_{(\cdot)}, \nabla c_{(\cdot)}$ to extended Jacobian J_E defined above. The computation completed is essentially $J = C - DB^{-1}A$ where the inverse of the lower triangular matrix B is implicitly done in the double loop.

Algorithm 2 Structured Reverse Mode

function STRREVERSEAD(x, $\{W_i\}_{i=1}^{k+1}$, W)
  k = $\#W$-1
  for i = 1 to k **do**
    $y_i = W_i(x, y_1, \cdots, y_{i-1})$
  end for
  $[z, J^T W]$ = reverseAD($\{x, y_1, \cdots, y_k\}, W_{k+1}, W$)
  $[\nabla f, \nabla f_1, \cdots, \nabla f_k] = J^T W$ ▷ $\nabla f = (J_x^k)^T W, \nabla f_i = (J_{y_i}^k)^T W$
  for i = 1 to k **do**
    $v_i = 0$
  end for
  for i = k to 1 **do**
    $w_i = \nabla f_j + v_i$
    $[\sim, J^T W]$ = reverseAD($\{x, y_1, \cdots, y_{i-1}\}, W_i, w_i$)
    $[\nabla c^i, \nabla c_1^i, \cdots, \nabla c_{i-1}^i] = J^T W$ ▷ $\nabla c^i = (J_x^i)^T w_i, \nabla c_j^i = (J_{y_j}^i)^T w_i$
    $\nabla f = \nabla f + \nabla c^i$
    for j = 1 to i-1 **do**
      $v_j = v_j + \nabla c_j^i$
    end for
  end for ▷ Memory of ∇c^i and ∇c_j^i is released at the end of each iteration
  return $[z, \nabla f]$
end function

Appendix B

Properties of checkpoints selection

B.1 Admissibility of checkpoints selection

To mathematically formulate the above statement, we first define checkpoints as a selection of $\{y\}$ denoted as $\{P_i\}_{i=1}^{n_p}$:

$$\begin{aligned} P_i \equiv y_{p_i} \qquad p_i \in \{1, \cdots, k\} \qquad \forall i \in \{1, \cdots, n_p\} \\ p_0 \equiv 0 < p_1 < p_2 < \cdots < p_{n_{p-1}} < p_{n_p} < p_{n_p+1} \equiv k \end{aligned} \tag{B.1}$$

To be consistent, we should also have:

$$\begin{cases} P_0 \equiv x \\ P_{n_p+1} \equiv z \end{cases} \tag{B.2}$$

And we denote all the other $\{y\}$ using the following notation:

$$\forall i \in \{0, \cdots, n_p\} \begin{cases} I_j^{(i)} & \equiv y_{p_i+j}, \forall j \in \{1, \cdots, l_i\} \\ l_i & = p_{i+1} - p_i \end{cases} \tag{B.3}$$

namely, $\{I_j^i\}_{j=1}^{l_i-1}$ are the intermediate variables between checkpoint P_i and P_{i+1}. If we order all the variables chronologically, we would get the following:

$$x = P_0 \to I_1^{(0)} \cdots \to I_{l_0-1}^{(0)} \to P_1 \cdots \to P_{n_p} \to I_1^{(n_p)} \cdots \to I_{l_{n_p}-1}^{(n_p)} \to P_{n_p+1} = z \tag{B.4}$$

We reformulate our program f as follows:

$$\begin{cases} P_1 = b_{p_1}(\text{Arg}_1) & \equiv W_1(\text{Parg}_1) = W_1(x) \\ P_2 = b_{p_2}(\text{Arg}_2) & \equiv W_2(\text{Parg}_2) \\ & \vdots \\ P_{n_p} = b_{p_{n_p}}(\text{Arg}_{n_p}) & \equiv W_{n_p}(\text{Parg}_{n_p}) \\ P_{n_p+1} = b_{n_p+1}(\text{Arg}_{n_p+1}) & \equiv W_{n_p+1}(\text{Parg}_{n_p+1}) \\ \Leftrightarrow z = b_k(\text{Arg}_k) & \equiv W_{n_p+1}(\text{Parg}_{n_p+1}) \end{cases} \tag{B.5}$$

where we basically wrap up each piece of the computational tape from P_i to P_{i+1} into a non-elementary function W_{i+1}, and putting all the code segments $\{W\}$ in order recovers the original code c:

$$c \Leftrightarrow W_1 + W_2 + \cdots + W_{n_p} \quad \forall \text{checkpoint selection} \{P_i\}_{i=1}^{n_p} \tag{B.6}$$

b and Arg follows the same notation as in previous chapters; Parg stands for *principal arguments* which we define as follows:

Definition B.1. Given a code c and a selection of checkpoints $\{P_i\}_{i=0}^{n_p+1}$, the respective *principle arguments* are defined as follows:

$$\text{Parg}_i = (\cup_{j=p_{i-1}+1}^{p_i} \text{Arg}_j) - (\cup_{j=p_{i-1}+1}^{p_i} \{y_j\}) \quad \forall i \in \{1, \cdots, n_p + 1\} \tag{B.7}$$

We see that the principal argument Parg_i is a set difference. The set being subtracted is a collection of all arguments that have appeared in code segment W_i, and the set subtracted by is just all the intermediate variables from P_{i-1} to P_i. Hence its meaning is clear: W_i's dependencies previous to P_{i-1}. If the checkpoints constitute a fully encapsulated division of the code, we should expect the following to hold:

$$\text{Parg}_{i+1} \subseteq \{P_0, \cdots, P_i\} \quad \forall i \in \{0, \cdots, n_p\} \tag{B.8}$$

Hence we have:

Definition B.2. A selection of checkpoints $\{P_i\}_{i=1}^{n_p}$ for code c is called *admissible* when either of following equivalent conditions are satisfied $\forall i \in \{0, \cdots, n_p\}$:

$$\begin{aligned} &(1) \text{Parg}_{i+1} \subseteq \{P_0, \cdots, P_i\} \\ &(2) \forall j \in \{1, \cdots, l_i\}, \text{Arg}_{p_i+j} \subseteq \{P_0, \cdots, P_i, I_1^{(i)}, \cdots, I_{j-1}^{(i)}\} \end{aligned} \tag{B.9}$$

B.2 Proof of Theorem 2.6

Here we prove the following theorem:

Theorem 2.6. A subset of a Markovian admissible collection of checkpoints is also an Markovian admissible selection of checkpoints.

Proof: Denote $\{P_i\}_{i=1}^{n_p}$ as the basis Markovian admissible checkpoint selection. We prove in the following that any subset $\{Q_i\}_{i=1}^{n_q} \subseteq \{P_i\}_{i=1}^{n_p}$ is also a Markovian admissible checkpoint selection.

We denote $\{Q_i\}_{i=1}^{n_q}$'s indexing of $\{P_i\}_{i=1}^{n_p}$ by $\{q_i\}_{i=1}^{n_q}$:

$$\begin{aligned} Q_i \equiv P_{q_i} \qquad q_i \in \{1, \cdots, n_p\} \qquad \forall i \in \{1, \cdots, n_q\} \\ q_0 \equiv 0 < q_1 < q_2 < \cdots < q_{n_{q-1}} < q_{n_q} < q_{n_q+1} \equiv n_p + 1 \end{aligned} \tag{B.10}$$

Since $\{P_i\}_{i=1}^{n_p}$ is Markovian, the code c are equivalent to the following computation.

$$c = \begin{cases} P_1 & = W_1(P_0) = W_1(x) \\ P_2 & = W_2(x, P_1) \\ & \vdots \\ P_{n_p} & = W_{n_p}(x, P_{n_p-1}) \\ z & = P_{n_p+1} = W_{n_p+1}(x, P_{n_p}) \end{cases} \tag{B.11}$$

By constructing the following replicating code segment $\{V_i\}_{i=1}^{n_q+1}$:

$$\forall i \in \{0, \cdots, n_q\} \quad V_{i+1}(x, Q_i) = \begin{cases} P_{q_i+1} & = W_{q_i+1}(x, Q_i) = W_{q_i+1}(x, P_{q_i}) \\ P_{q_i+2} & = W_{q_i+2}(x, P_{q_i+1}) \\ & \vdots \\ P_{q_{i+1}-1} & = W_{q_{i+1}-1}(x, P_{q_{i+1}-2}) \\ Q_{i+1} & = P_{q_{i+1}} = W_{q_{i+1}}(x, P_{q_{i+1}}) \end{cases} \tag{B.12}$$

We prove $\{Q_i\}_{i=1}^{n_q}$ is indeed a Markovian admissible checkpoint selection:

$$c = \begin{cases} Q_1 & = V_1(Q_0) = V_1(x) \\ Q_2 & = V_2(x, Q_1) \\ & \vdots \\ Q_{n_q} & = V_{n_q}(x, Q_{n_q-1}) \\ z & = Q_{n_q+1} = V_{n_q+1}(x, Q_{n_q}) \end{cases} \tag{B.13}$$

Q.E.D.

B.3 Illustration of Structured Reverse Mode

Given an admissible selection of checkpoints, we've illustrated bare reverse mode in Figure B.1 and structured reverse mode in Figure B.2. Those variables in grey color either haven't been computed, or has been computed yet is no longer stored. When the direction of the arrow gets flipped, it indicates that the derivative has been back propagated over this point using the algorithm in Appendix A. 6 snapshots are shown in Figure B.1:

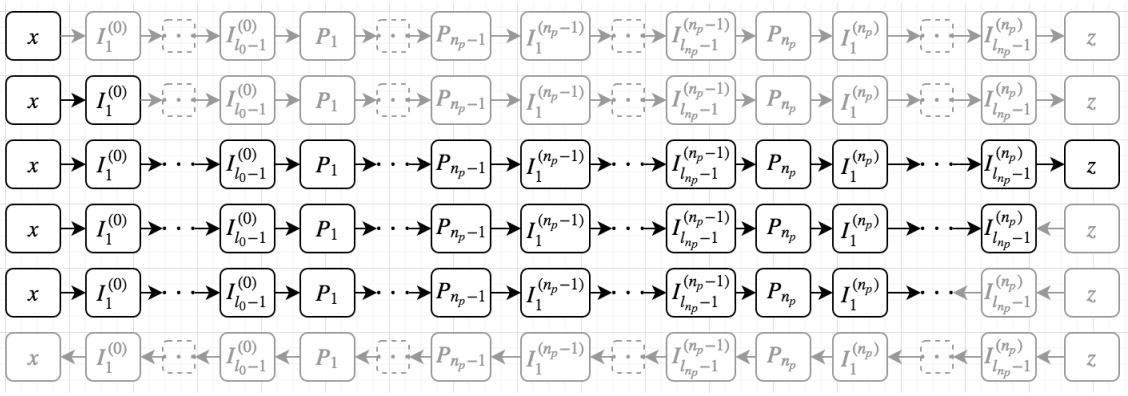

Figure B.1: Demonstration of Naive Reverse Mode: see text for detailed explanation.

(1) Code starts to run

(2) One computational step is done

(3) Computation is done all the way to the output, forward sweep is complete, all intermediate variables are kept, backward sweep will start

(4) Derivatives are propagated for one computational step

(5) Derivatives are propagated for another computational step

(6) The full derivatives have been propagated back to the input, reverse mode is complete

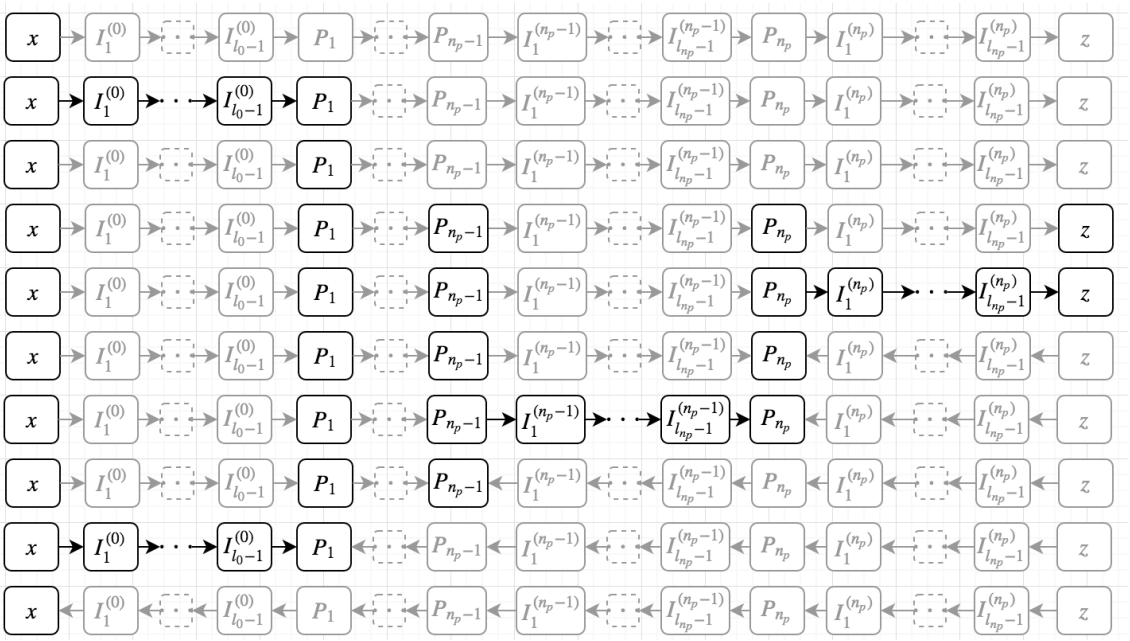

Figure B.2: Demonstration of Structured Reverse Mode: see text for detailed explanation.

10 snapshots are shown in Figure B.2:

(1) Code starts to run

(2) Computation is done up to P_1

(3) Intermediate variables between x and P_1 are no longer stored. Occupied memory are released

(4) Similar processes are done until we reached the output[1]. We are left with x, z and all the checkpoints $\{P_i\}_{i=1}^{n_p}$.

[1]There is actually no need to drop intermediate variables between P_{n_p} and z since we immediately need them in the next step. Nevertheless such action clearly distinguishes forward phase and backward phase.

(5) The code segment from P_{n_p} to z is reconstructed

(6) Reverse mode is applied on the code segment between P_{n_p} and z, derivatives are propagated to P_{n_p}

(7) The code segment from P_{n_p-1} to P_{n_p} is reconstructed

(8) Reverse mode is applied on the code segment between P_{n_p-1} and P_{n_p}, derivatives are propagated to P_{n_p-1}

(9) Similar processes are repeated and derivatives have reached P_1. Code segment between x and P_1 is reconstructed.

(10) Reverse mode is applied on the code segment between x and P_1, derivatives are propagated to input x. Structured Reverse Mode is complete.

Appendix C

Technical details of Monte Carlo Convergence

C.1 Justification of mean estimation

Monte Carlo algorithms that aim to estimate expectation values are justified by the central limit theorem. We first define what does it mean to converge in distribution:

Definition C.1. A series of one dimensional random variable $\{X_i\}_{i\in\mathcal{Z}^+}$ *converge in distribution* to a cumulative distribution function $\Phi(\cdot)$ when the following equation holds:

$$\lim_{n\to\infty} \Pr[X_n \leqslant z] = \Phi(z) \qquad \forall z \in \mathcal{R} \tag{C.1}$$

which is denoted as $X \xrightarrow{d} \phi$ where ϕ is the partial distribution function of Φ.

Then we can state the central limit theorem:

Theorem C.2. Suppose $\{X_i\}_{i\in\mathcal{Z}^+}$ is a sequence of i.i.d. random variables with $E[X] = \mu$ and $\mathrm{Var}[X] = \sigma^2 < \infty$. Then we have:

$$\sqrt{n}\left([\frac{1}{n}\sum_{i=1}^{n} X_i] - \mu\right) \xrightarrow{d} N(0, \sigma^2) \tag{C.2}$$

Using Monte Carlo, one can construct $\{\bar{X}_i\}_{i\in\mathcal{Z}^+}$:

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i \tag{C.3}$$

Therefore by central limit theorem:

$$\bar{X}_n \sim N(\mu, (\frac{\sigma}{\sqrt{n}})^2) \Leftrightarrow p(\bar{X}_n|\mu,\sigma) \approx N(\mu, (\frac{\sigma}{\sqrt{n}})^2) \tag{C.4}$$

Hence we can infer the unknown value μ to have the following posterior:

$$\mu \sim N(\bar{X}_n, (\frac{\sigma}{\sqrt{n}})^2) \Leftrightarrow p(\mu|\bar{X}_n,\sigma) \approx N(\bar{X}_n, (\frac{\sigma}{\sqrt{n}})^2) \tag{C.5}$$

if we have a uniform priori belief on μ.

C.2 Convergence of Cascaded Mean estimation

We refine the definition of rate of efficiency as follows:

Definition C.3. The *rate of efficiency* $\beta_a \in (0,\infty]$ of an algorithm a to compute a scalar quantity y is defined via the following asymptotics[1]:

$$|y_a - y| \sim O(M_a^{-\beta_a}) \Leftrightarrow \begin{cases} |E[y_a - y]| & \sim M_a^{-\beta_a^e} \\ \text{std}[y_a - y] & \sim M_a^{-\beta_a^s} \\ \beta_a & = \min\{\beta_a^e, \beta_a^s\} \end{cases} \tag{C.6}$$

where y_a is the approximation of y that approach a computes, and M_a is the complexity of a's runtime.

It is β_a^e and β_a^s that determines the rate of efficiency for cascaded algorithms as we will see below. Superscript ^{e} stands for 'expectation', ^{s} stands for 'standard deviation'. For deterministic algorithm, $\text{std}[y_a - y] \sim 0$ hence $\beta_a^s = \infty$, $\beta_a = \beta_a^e$. Rate of efficiency not being ∞ implies the existence of a precision-runtime tradeoff. Table C.1 gives examples of rate of efficiency:

[1]For simplicity, float point accuracy is considered ∞ precision. For algorithm with $O(1)$ complexity and floating point accuracy, rate of efficiency is set to be ∞ by default.

Algorithm	β^e	β^s	β
Sampling $\mathbf{1}_{x^2+y^2\leqslant 1}$ where $x, y \sim U[0,1]$	∞	∞	∞
Monte Carlo estimation of π	∞	$1/2$	$1/2$
Sampling lognormal variable by timestepping	1	∞	1
Monte Carlo calculation of an European option	$1/3$	$1/3$	$1/3$
1D PDE with implicit timestepping	1	∞	1
1D PDE with Crank-Nicolson timestepping	2	∞	2

Table C.1: Algorithm examples with their rate of efficiencies. 1st and 3rd algorithm are essentially the sampling algorithm of the 2nd and 4th Monte Carlo algorithms.

If the sampling process of the Monte Carlo has $O(1)$ complexity and is error-free, then overall rate of convergence is $\beta^e = \infty, \beta = \beta^s = 0.5$. Otherwise, the rate of efficiency would be lower.

Suppose we are using Monte Carlo sampling of M times on the random variable X. Each sample $x_a^{(i)}$ of X is given by algorithm a as an estimator of the true sample $x^{(i)}$.

$$\forall i \in \{1, \cdots, M\} \begin{cases} x_a^{(i)} & = x^{(i)} + \epsilon_a^{(i)} \\ |\epsilon_a^{(i)}| & \sim M_a^{-\beta_a} \end{cases} \tag{C.7}$$

The second condition means the following:

$$|\epsilon_a^{(i)}| \sim M_a^{-\beta_a} \Leftrightarrow \begin{cases} |E[\epsilon_a]| & \sim M_a^{-\beta_a^e} \\ \mathrm{std}[\epsilon_a] & \sim M_a^{-\beta_a^s} \\ \beta_a & = \min\{\beta_a^e, \beta_a^s\} \end{cases} \tag{C.8}$$

The Monte Carlo estimator is:

$$\bar{x}_{mc(a)} \equiv \frac{1}{M}\sum_{i=1}^{M} x_a^{(i)} = \frac{1}{M}\sum_{i=1}^{M} x^{(i)} + \frac{1}{M}\sum_{i=1}^{M} \epsilon_a^{(i)} \tag{C.9}$$

where $mc(a)$ stands for the Monte Carlo algorithm using approach a to draw samples. Therefore, overall error would be:

$$\bar{x}_{mc(a)} - E[X] = \frac{1}{M}\sum_{i=1}^{M} x^{(i)} - E[X] + \frac{1}{M}\sum_{i=1}^{M} \epsilon_a^{(i)} \tag{C.10}$$

Central limit theorem applies to both terms. The first term would be normally distributed at mean 0 with standard deviation $O(M^{-\frac{1}{2}})$. For the second term, the scale of its mean is given by a's rate of efficiency β_a^e, while the scale of its standard deviation is given by a's rate of efficiency β_a^s and further reduced by $O(M^{-\frac{1}{2}})$. Correlation between two terms is not important if any[2].

$$\begin{cases} |E[\bar{x}_{mc(a)} - E[X]]| & \sim O(M_a^{-\beta_a^e}) \\ \text{std}[\bar{x}_{mc(a)} - E[X]] & \sim O(\max\{M^{-\frac{1}{2}}, M^{-\frac{1}{2}} M_a^{-\beta_a^s}\}) = O(M^{-\frac{1}{2}}) \end{cases} \tag{C.11}$$

A consistent choice of M_a would be:

$$\hat{M}_a \sim M^{1/2\beta_a^e} \tag{C.12}$$

since scaling lower affects accuracy while higher does not offer improvement. Such optimal choice of M_a leads to:

$$\begin{cases} M_{mc(a)} & \sim M \cdot \hat{M}_a \sim M^{1+\frac{1}{2\beta_a^e}} \\ |E[\bar{x}_{mc(a)} - E[X]]| & \sim \begin{cases} 0, & \text{if } \beta_a^e = \infty \\ M^{-\frac{1}{2}}, & \text{otherwise} \end{cases} \sim M_{mc(a)}^{-\beta_{mc(a)}^e} \\ \text{std}[\bar{x}_{mc(a)} - E[X]] & \sim M^{-\frac{1}{2}} \sim M_{mc(a)}^{-\beta_{mc(a)}^s} \end{cases} \tag{C.13}$$

Hence overall rate of efficiency of a Monte Carlo deploying a to draw samples reads:

$$\beta_{mc(a)}^e = \begin{cases} \infty, & \text{if } \beta_a^e = \infty \\ \frac{\beta_a^e}{2\beta_a^e + 1}, & \text{otherwise} \end{cases} \tag{C.14}$$

$$\beta_{mc(a)} = \beta_{mc(a)}^s = \frac{\beta_a^e}{2\beta_a^e + 1} \in (0, \frac{1}{2}] \tag{C.15}$$

The rate of efficiency of Monte Carlo is naturally the harmonic mean of a's rate of efficiency β_a^e and the suppose-to-be rate of efficiency of Monte Carlo $\beta_{mc}^* = 0.5$. This completes the proof of (3.2).

The following observations can be made. If a is an unbiased stochastic algorithm, i.e. $\beta_a^e = \infty$, we always have $\beta_{mc} = 0.5$ regardless of β_a^s, even if $\beta_a^s < 0.5$. If a is a second order deterministic algorithm, i.e. $\beta_a^e = 2$, then $\beta_{mc} = 0.4$. If a is a linearly converging algorithm, i.e. $\beta_a^e = 1$, we have $\beta_{mc} = 1/3$. If a is a biased Monte Carlo with trivial sampling, i.e. $\beta_a = 0.5$, then we only have $\beta_{mc} = 1/4$.

[2]The two terms might be correlated depending on whether or not ϵ_a correlates with x_a. Yet such correlation does not matter for asymptotic calculations of rate of efficiencies.

Appendix D

Details of Variable Annuities

D.1 Modelling of policy holder behavior

The policyholder withdraws the maximal amount allowed each year is a reasonable possibility not a necessity. A quant may say the rigorous treatment should be to incorporate customer behavior into the model. However it would introduce much complication and is not suited for Monte Carlo setting. Besides there are no natural ways to model customer behavior since there're no rationales to assume they behave rationally or perform the optimal strategy. "(1) Coming up with certain utility function (2) use dynamic programming to construct the optimal strategy as governed by the utility (3) use the optimal strategy as an input to the pricing model" might be one theoretical approach. However, the utility function approach is a theoretical convenience after all. The practical approach would be to statistically model one or several representative policy holders using internal data. Yet in literature, researchers have found that under reasonable utility functions, there's little difference in the contract pricing between assuming policy holders with optimal strategy and assuming maximal allowed withdrawal strategy each year. On line 518-520, it reads "...This suggests that for a large family of parameters, the policyholder withdraws at nearly the contract rate...."[16]. Hence we will use this convenient approximation of customer behavior: withdrawing contract rate each year.

Besides deciding how much to withdrawal each year, a policy holder might also consider revoking the whole contract, which is called "full surrender"[1], where the contract is termi-

[1]Partial surrender means withdrawing above maximal allowed amount in which case certain penalty will be charged.

nated and all money in the investment account is retrieved. Such behavior is captured in practice using lapse curve, a statistical measure of the percentage of contracts that ends prematurely each year.

D.2 Practical considerations of VA hedging

D.2.1 On second order hedging

Second order hedging on VA is not practically desirable for the following reasons:

(1) The payoff of VA has lots of piecewise linear part due to ratchet, roll-up, step-up feature. For such function, taylor expansion around each point has a very limited effective radius. In other words, second order derivatives are not well defined and discontinuous at many points. Hence second order method might not be effective as it is supposed to be.

(2) The VA itself is already computationally intensive. It would be way too expensive to compute the second order derivative of VA present value with reasonable accuracy as first order derivatives are already incurring too much computational burden.

(3) Second order hedging requires additional hedging instruments which might not be available or is just too expensive. For delta-rho hedge, it is common to short[2] stock/index (the underlying) future to hedge price risk and interest rate swap to handle interest rate risk.

The reason that shorting future is preferred over longing put option is for being cheaper. Longing put option is perfect for VA hedging in the sense that it provides full protection if the underlying plunges, at the mean time, it only minimally affects the potential gain when the underlying performs well, i.e. it hedges only the loss but not the profit. Nevertheless, futures are cheaper (per delta exposure), simple, standard and are chosen by practitioners.

Neither future nor interest rate swap provide gamma exposure. If second order method is used, more expensive and complicated instruments have to be involved.

[2]To compensate the put-option like guarantee payoff that the company shorts

D.2.2 The computational cost of hedging

Hedging introduces colossal complexities into the pricing of VA:

(1) **Single layer Monte Carlo to Nested Monte Carlo**: Without hedging, a single layer Monte Carlo in risk neutral world and another single layer Monte Carlo in real world for projected cash flows suffice for pricing. When hedging is involved, at each node of real world projection, a risk neutral pricing to decide how much hedging instruments to be rebalanced is demanded, promoting single layer Monte Carlo to nested Monte Carlo, which doubles the magnitude of runtime, not to mention the cost of obtaining accurate derivatives.

(2) **Time stepping needs to much denser in principle**: In practice, price risk hedging usually requires daily rebalancing at minimal. However, annual time stepping is already enough for the original VA pricing framework since the annuity realizes payoff annually. To incorporate hedging fully, one has no choice but to match the time stepping to whatever hedging requires, if hedging accuracy is required and the computational burden could be afforded.

D.3 Other indispensable details of VA program

Besides hedging off exposures, insurance companies are also regulated to have capital and reserve depending on the risk involved in the business. AG43 reserve and C-3 phase II capital are the common requirements. Quantities such as the projected cash flow at each year at the 30% worst scenario, for example, would be something important for capital and reserve specification.

Since runtime is always excessive if all details are included for VA, people has proposed in literatures [17][18] to first use Monte Carlo to do full pricing on certain representative contracts, which is selected by appropriate unsupervised learning algorithms, then use such data to train a regression machine in supervised way, and finally take the prediction from the learnt regression machine for all other contracts as approximated results. Such approach could significantly reduce the runtime with unguaranteed yet reasonable accuracy. The selection of representative contracts is also done in practice.

Appendix E

Codes and Templates

E.1 Pseudo code templates

E.1.1 General

Assumed the reverse mode AD has the following input and output:

Algorithm 3 Straight forward reverse mode AD

function ADMAT_RVS(x, $f(\cdot)$, W)
...Compute $y = f(x)$ and $v = W^T J$, but not $J = f_x(x)$...
return $[y, v]$
end function

Pathwise Monte Carlo can be completed by the following template.

The structured reverse mode assumes the initialise, transit, and emit code structure.

E.1.2 Markovian Process

The set up of the previous algorithm has a general path dependency. For a Markovian path, we have the simplified version.

Algorithm 4 Pathwise AD for Monte Carlo Simulation

function MC_REVERSE_AD(x, init($\cdot$), Node_Process($\cdot, \{\cdot\}$), emit($\cdot, \{\cdot\}$), W, M, N)
  $y = 0$
  $v = 0$
  for i=1 to M **do**
    $[yc, vc]$ = ADMAT_str(x, init($\cdot$), Node_Process($\cdot, \{\cdot\}$), emit($\cdot, \{\cdot\}$), W, N)
    $y = y + yc$
    $v = v + vc$
  end for
  return $[y, v]/M$
end function

Algorithm 5 Node structured reverse mode AD for a single Monte Carlo path

function ADMAT_STR(x, init($\cdot$), transit($\cdot, \{\cdot\}$), emit($\cdot, \{\cdot\}$), W, $Nsteps$)
dimX = #Elements of x
dimW = #Columns of W
dimF = #Rows of W
seeds = Nsteps+1 $\times$ 1 vector of random number generator (RNG) status
seeds(1) = rng ▷ Record initial RNG status
$y0$ = init(x)
dimNode = #Elements of $y0$
y = dimNode $\times$ Nsteps+1 matrix (each column a node data)
$y(:, 1) = y0$
for i = 1 to Nsteps **do**
seeds(i+1) = rng ▷ Record intermediate RNG status
$y(:, i+1)$ = transit(x, $\{y(:, 1), \cdots, y(:, i)\}$)
end for
w = Nsteps+1 $\times$ 1 vector of dimNode $\times$ dimW matrix
$[f, vf, w(1), \cdots, w(end)]$ = ADMAT_RVS($x, y(:, 1), \cdots, y(:, end)$, emit($\cdot, \{\cdot\}$), W)
v = Nsteps+1 $\times$ 1 vector of dimNode $\times$ dimW matrix of zero
dv = Nsteps $\times$ 1 vector of dimNode $\times$ dimW matrix of zero
for j = Nsteps to 1 **do**
$w(j+1) = w(j+1) + v(j+1)$
rng(seeds(j+1)) ▷ Restore RNG status
$[\sim, dvf, dv(1), \cdots, dv(j)]$ = ADMAT_RVS($x, y(:, 1), \cdots, y(:, j)$,transit($\cdot, \{\cdot\}$), $w(j)$)
$vf = vf + dvf$
for i = 1 to j **do**
$v(i) = v(i) + dv(i)$
end for
end for
$w(1) = w(1) + v(1)$
rng(seeds(1))
$[\sim, dvf]$ = ADMAT_RVS(x, init($\cdot$), $w(1)$)
$vf = vf + dvf$
return $[f, vf]$
end function

Algorithm 6 Node structured reverse mode AD for a Markovian Monte Carlo path

function ADMAT_STR(x, init$(\cdot)$, transit$(\cdot,\cdot)$, emit$(\cdot,\cdot)$, W, $Nsteps$)
dimX = #Elements of x
dimW = #Columns of W
dimF = #Rows of W
seeds = Nsteps+1 $\times$ 1 vector of random number generator (RNG) status
seeds(1) = rng ▷ Record initial RNG status
$y0$ = init(x)
dimNode = #Elements of $y0$
y = dimNode $\times$ Nsteps+1 matrix (each column a node data)
$y(:,1) = y0$
for i = 1 to Nsteps **do**
seeds(i+1) = rng ▷ Record intermediate RNG status
$y(:,i+1)$ = transit$(x, y(:,i))$
end for
w = Nsteps+1 $\times$ 1 vector of dimNode $\times$ dimW matrix of zero
$[f, vf, w(end)]$ = ADMAT_RVS$(x, y(:,end)$, emit$(\cdot,\cdot)$, $W)$
v = Nsteps+1 $\times$ 1 vector of dimNode $\times$ dimW matrix of zero
dv = dimNode $\times$ dimW matrix of zero
for j = Nsteps to 1 **do**
$w(j+1) = w(j+1) + v(j+1)$
rng(seeds(j+1)) ▷ Restore RNG status
$[\sim, dvf, dv]$ = ADMAT_RVS$(x, y(:,j)$, transit$(\cdot,\cdot)$, $w(j))$
$vf = vf + dvf$
$v(j) = v(j) + dv$
end for
$w(1) = w(1) + v(1)$
rng(seeds(1))
$[\sim, dvf]$ = ADMAT_RVS$(x$, init$(\cdot)$, $w(1))$
$vf = vf + dvf$
return $[f, vf]$
end function