Stochastic Automatic Differentiation: Automatic Differentiation for Monte-Carlo Simulations

Christian P. Fries
email@christian-fries.de

June 27, 2017

Version 1.2.0

Abstract

In this paper we re-formulate the automatic differentiation (and in particular, the backward automatic differentiation, also known as adjoint automatic differentiation, AAD) for *random variables*. While this is just a formal re-interpretation it allows to investigate the algorithms in the presence of stochastic operators like expectation, conditional expectation or indicator functions.

We then specify the algorithms to efficiently incorporate non-pathwise operators (like conditional expectation operators). Under a comparably mild assumption it is possible to retain the simplicity of the backward automatic differentiation algorithm in the presence of conditional expectation operators. This simplifies important applications like - in mathematical finance - the application of backward automatic differentiation to the valuation of Bermudan options or calculation of xVA's.

We give the proof for a generalised version of the result. We then discuss in detail how the framework allows to dramatically reduce the memory requirements and improve the performance of a tapeless implementation of automatic differentiation (while the implementation brings advantages similar to "vector AAD" (sometimes called tape compression) for free, it allows improvements beyond this. We present the implementation aspects and show how concepts from object-functional programming, like immutable objects and lazy evaluation enable additional reductions of the memory requirements.

Disclaimer: The views expressed in this work are the personal views of the authors and do not necessarily reflect the views or policies of current or previous employers.
Feedback welcomed at email@christian-fries.de.

Acknowledgment

We are grateful to Amine Chaieb and Marco Noll for stimulating discussions.

Contents

1 Introduction

1.1 Overview

In this section we give a short overview on existing literature and highlight the main contributions of the paper.

Automatic differentiation is a numerical technique to calculate partial derivatives $\frac{\partial}{\partial x_i} f(x_0, \ldots, x_{n-1})$ for functions f which are represented by computer programs which are compositions of functions from a set of elementary functions. Here, the functions are interpreted as functions from $\mathbb{R}^m$ to $\mathbb{R}^d$ (implemented or approximated on the subset of floating point numbers).

The history of the method reaches back to the 1960's. For example, the back-propagation algorithm used in the training of neural networks is effectively an adjoint automatic differentiation.

1.1.1 Forward versus Backward Mode Automatic Differentiation: Computation Time and Memory Requirements

Automatic differentiation can be performed in different modes. Two important modes are forward mode automatic differentiation, which propagates derivatives along the valuation, i.e., from the function arguments to its next intermediate result, and backward mode automatic differentiation, which propagates derivatives backward from the final result to its arguments.

The two come with different complexity. With respect to computation time, if T_f determines the time required to evaluate the function $f : \mathbb{R}^m \to \mathbb{R}^d$, then the computation time for the derivatives is in backward mode is $O(d) \cdot T_f$ and in forward mode $O(m) \cdot T_f$.

With respect to memory, if M_f is the memory requirement for the evaluation of the function f, then the memory requirement for the calculation of the Jacobi matrix in forward mode is $O(d) \cdot M_f$. However, the memory requirement for that calculation in backward mode - using a non-optimized implementation - is $O(T_f) \cdot M_f$. This follows from the need to store the complete computation tree prior to the execution of the backward mode differentiation. Here it is assumed that the size of the computation tree is proportional to the computation time. Hence, reduction of memory requirements is of vital importance in a successful construction of a generic backward differentiation framework. For details see [21].

Automatic Differentiation in Finance

The application of automatic differentiation to problems in mathematical finance or Monte-Carlo simulations in general appears to be straight forward. For an introduction to the backward mode automatic differentiation applied to models in mathematical finance see, e.g., [19, 6, 22]. The method gained more relevance when discussed in the context of credit valuation adjustments, [1, 7].

For these and most application in mathematical finance, backward mode automatic differentiation (aka. adjoint automatic differentiation, AAD) is the method of choice since its computational cost does not scale with the number of input parameters.

Automatic Differentiation applied to Monte-Carlo Simulations

For the application to Monte-Carlo simulation it is often noted that the desired results can be obtained from the pathwise application of automatic differentiation, i.e., by interchanging the automatic differentiation and the Monte-Carlo integration. However, this approach becomes challenging if the Monte-Carlo algorithm uses non-pathwise operators like conditional expectations. Conditional expectation operators occur for example in Bermudan options, where the conditional expectation of the future value is the strike-price for the option exercise, but also in valuation adjustments like CVA.

A specific issue related to the application of the favourable adjoint automatic differentiation to Monte-Carlo simulations is its exorbitant memory requirements.

Automatic Differentiation and Conditional Expectation Operators

With respect to the differentiation of the valuation of a Bermudan option, it is known that a first order derivative of the exercise boundary – and hence a first order derivative of the conditional expectation operator – is not required, since the exercise is optimal, see [25]. However, this property is only given for a limited class of financial products and does not hold in general. In particular, it does not hold for conditional expectations required to calculate a CVA, [2].

It should be also noted that conditional expectations approximated by regressions are endowed with approximation errors which in turn lead to sub-optimal exercise. Hence a differentiation of the exercise boundary and the conditional expectation operator can be of interest to analyse properties of the regression basis functions and verify if the exercise is optimal. See [13] for an example.

If a conditional expectation is approximated by a regression (so called *American Monte-Carlo*) a straight forward application of automatic differentiation to an American Monte-Carlo simulation would result in the differentiation of the regression basis functions. While this can be done, it will result in lengthy calculations and complex computer code. In addition, the presence of a conditional expectation operator represents a challenge to the backward mode automatic differentiation, as one would naturally need to propagate derivatives forward. In [2, 3] a variant of a forward mode algorithmic differentiation is introduced, applying algorithmic differentiation to a class of derivative products involving conditional expectation operators. Although the method is termed "backward differentiation" it is a forward mode algorithmic differentiation and hence its complexity scales with the number of parameters. In [2] the authors also observe that a differentiation of basis functions of the regression approximating the conditional expectation can be avoided under

the (comparably strong) assumption of the absence of path dependency. Using our approach we can generalize and improve this result.

1.1.2 Standard Implementations of (Adjoint) Automatic Differentiation

The two prominent standard implementations for automatic differentiation are source code transformation and operator overloading. In source code transformation a tool is processing the source code of the original program and creating a new program which calculates the derivatives. In operator overloading the operations on real numbers are replaced by operations performing the original calculation *and* a calculation required for the derivatives. For backward mode differentiation the overloaded operators are used to record the operations, also called *tape*.

With respect to these concepts we perform an operator overloading. However, we perform the operator overloading on operators on random variables. That is, we first define random variables as objects with a set of arithmetic operations, then define an automatic differentiation overload of the random variable to record the operators.

1.1.3 Contribution of the Paper

In this paper we take a different approach right from the beginning: we consider the automatic differentiation algorithm applied to random variables. That is, the space of real numbers (or vectors of real numbers) is replaced by the space of random variables and the set of elementary real valued functions is replaced by a set of random operators (mapping from vectors of random variables to a random variable).

This approach opens some new possibilities:

1. We present several possibilities to significantly reduce the memory requirements using properties of the random variables, of the operators and concepts of object-oriented and object-functional programming (like immutable objects, see below). For example, a deterministic random variable requires only a fraction of the storage.

2. For non-path wise linear operators like the (approximation of the) conditional expectation, we do not need to differentiation the approximation (the regression). Instead we perform a regression on the differentiation.

Additionally assuming that the final operator is an expectation - a condition present in possibly all applications of Monte-Carlo simulation - we can derive further improvements:

3. We can consider the adjoint of a non-path wise operator, which allows to derive a true backward mode automatic differentiation of the Monte-Carlo simulation, where the code complexity is minimal.

   (a) For the special case of a conditional expectation operator this greatly simplifies the AAD algorithm. For the application to Bermudan options further details can be found in [13].

4. We can given an exact definition for the differentiation of indicator functions, which are problematic in a path-wise application of automatic differentiation.

In addition to giving the proof of the methods under more general assumptions, the paper has a focus on the implementation aspects and shows how to achieve the necessary reduction of the memory requirements.

1.2 Relevant Concepts from Object-Oriented and Object-Functional Implementation

Considering automatic differentiation for operators on random variables instead of operators on real numbers can be supported by key concepts from object-oriented and object-functional programming. For readers not familiar with these concepts we give a short review of some key concepts and point to their relevance in the implementation of a stochastic automatic differentiation. See [24] for details.

Objects: An object represents an encapsulated data structure . Here, we define objects for random variables. A random variable on a discrete sample space is represented by the vector of values on the sample path. It is this encapsulation which allows to reduce the memory requirements for, e.g., deterministic random variables.

Interfaces: An interface is the specification of applicable methods (or operators). We say that an object implements an interface, if the methods specified by the interface can be performed on the object. By the specifying an interface we can create programs which only rely on the fact that objects - here random variable - provide the specific methods. Here, we may then execute the program using plain value objects or object endowed with additional automatic differentiation code. Interfaces thus generalize the concept of operator overloading and allows for *dependency injection*, where the same algorithmic program can be executed on different types (of objects).

References: Objects are stored in memory and are identified by an unique identifier (this may be the memory address of the object). In case an object needs to be used in different parts of the program it is sufficient to store the identifier. For a real number it makes no difference if we store the number or the identifier.[1] For an object representing thousands of sample points of a random variable, it makes a huge difference.

[1] On a 64 bit system a memory address requires the same storage space as an IEEE 754 double precision floating point number.

Immutable Objects: An immutable object is an object which does not change state after it has been created. In our context it means that the object represents a fixed random variable. Immutability of objects allows to safely use the object identifier in different parts of the program.

Lazy Evaluation: Lazy evaluation is a program design where the evaluation of an expression is delayed until its result is needed. For example, an expression `z = x.add(y)` could just create an object `z` storing references to the objects `x` and `y` and the type of the operation (addition) while the use of, e.g., `print(z)` would finally start the evaluation of `z`. Lazy evaluation benefits from immutability of objects by storing references to them.

1.3 Automatic Differentiation: Review and Notation

To fix our notation we shortly state the basic algorithmic differentiation algorithms.

We consider the automatic differentiation of an algorithm calculating y depending on some inputs $x_0, \dots, x_{n-1}$, i.e.

$$y = f(x_0, \dots, x_{n-1})$$

where y and x_i are random variables.

Let $x_0, \dots, x_{n-1}$ denote given random variables - the independents. For $n \leq m \leq N$ let

$$x_m := f_m(x_{i_1^{(m)}}, \dots, x_{i_{k^{(m)}}^{(m)}})$$

denote intermediate results, where f_m is an operator of $k^{(m)}$ arguments specified by the arguments indices list $(i_1^{(m)}, \dots, i_{k^{(m)}}^{(m)})$.

To de-mystify the notation: we consider the algorithm as performing operations in a sequence. Operations are enumerated by the index m. The result of the m-th operation is x_m. The operation depends on a list of previous results x_i with $i < m$. That is, the operation is a function f_m. For each single operation we note the indices $i_j^{(m)}$ of the arguments of f_m and $k^{(m)}$ is the number of arguments of f_m. This will span a dependency tree or operator tree. The leaf nodes of this tree are the inputs, which we label by $x_0, \dots, x_{n-1}$.

We assume that the operators f_m are from a set of "elementary" operators for which the partial derivatives $\frac{\partial f_m}{x_{i_j^{(m)}}}$ are known. Examples for the $k^{(m)}$-ary operators f_m are

- the unary operators ($k^{(m)} = 1$): exp, log, $\sqrt{\cdot}$,
- the binary operators ($k^{(m)} = 2$): $+$ (add), $-$ (sub), $\cdot$ (mult), $/$ (div),
- ternary operators like $(a, b, c) \mapsto \begin{cases} b & \text{if a > 0} \\ c & \text{else.} \end{cases}$.

It may be advantageous to add operators which are common in the specific application, e.g. the accrual-operator as it is common in mathematical finance: $(a,b,c) \mapsto a \cdot (1.0 + b \cdot c)$ (accruing a over a period c with rate b).

Let $y := x_N$. Then the algorithm of calculating y as a function of $x_0, \ldots, x_{n-1}$ is determined by

- the sequence of operators $f_n, \ldots, f_N$ and
- their argument index list $i_1^{(m)}, \ldots, i_{k^{(m)}}^{(m)}$ for $m = n, \ldots, N$ (i.e., which of the previous results is forming the argument list of f_m.

Note on Independent and Intermediate Values: In what follows it is not necessary to distinguish between an input value and an intermediate value. In fact, an input value is just a constant function $x_i = f_i()$ that does not depend on any other variable ($k^{(m)} = 0$). In [14] an application is considered where the derivative of y with respect to (almost) all intermediate values are relevant.

1.4 Backward Automatic Differentiation

The calculation of the backward automatic differentiation (also known as adjoint automatic differentiation) of y with respect to $x_0, \ldots, x_{n-1}$ is given by the iterative application of the chain rule in the following form:

For all $m = N-1, N-2, \ldots, n$ (iterating backward through the operator list) it is

$$\frac{\partial y}{\partial x_m} = \sum_{l \in I} \frac{\partial y}{\partial x_l} \frac{\partial f_l}{\partial x_m}(x_{i_1^{(l)}}, \ldots, x_{i_{k^{(l)}}^{(l)}}) \tag{1}$$

where I is the set of all indices l such that f_l depends on x_m (note that this implies $l > m$ and $m \in \{i_1^{(l)}, \ldots, i_{k^{(l)}}^{(l)}\}$), with initial condition $\frac{\partial y}{\partial x_N} = \frac{\partial y}{\partial y} = 1$.

The iteration thus requires the calculation of $\frac{\partial f_l}{\partial x_m}(x_{i_1^{(l)}}, \ldots, x_{i_{k^{(l)}}^{(l)}})$ for all operators (nodes) and the backward propagation of these values.

In this formulation we "pull" the derivatives from the dependent nodes ($x_l = f_l$) to the argument node (x_m). This formulation is just more intuitive, because it directly reflects the chain rule. With respect to an implementation it is maybe easier and cleaner to change the ordering of the summation/iteration and "push" the values from the dependent node to the argument notes. In this case we apply:

Backward Automatic Differentiation Algorithm

Initialise $D_N = 1$ and $D_m = 0$ for $m \neq N$.

For all $m = N, N-1, \ldots, n$ (iterating backward through the operator list) and

$$\text{for all } j = 1, \ldots, k^{(m)} \text{ (iterating through the argument list)}$$

$$D_{i_j^{(m)}} \;\to\; D_{i_j^{(m)}} + D_m \cdot \frac{\partial f_m}{\partial x_{i_j^{(m)}}}(x_{i_1^{(m)}}, \ldots, x_{i_{k^{(m)}}^{(m)}}).$$

Then

$$\frac{\partial y}{\partial x_m} \;=\; D_m.$$

This iteration is just replacing the summation in (1) with partial summations in different order.

1.5 Forward Automatic Differentiation

For completeness we give the corresponding forward automatic differentiation:

For all $m = n, n+1, \ldots, N$ (iterating forward through the operator list) and all $i = 0, \ldots, n-1$ (iterating though the independent arguments) set

$$\frac{\partial x_m}{\partial x_i} \;=\; \sum_{l=1}^{k^{(m)}} \frac{\partial x_{i_l^{(m)}}}{\partial x_i} \, \frac{\partial f_m}{\partial x_{i_l^{(m)}}}(x_{i_1^{(m)}}, \ldots, x_{i_{k^{(l)}}^{(m)}}). \tag{2}$$

This is just the chain rule working from inside (the first operation) to the outside (the last operation) for each independent argument.

Literature The literature on automatic differentiation and its application to mathematical finance is large. For a general overview we refer to [22].

2 Stochastic Automatic Differentiation

2.1 Stochastic Automatic Differentiation

Given a filtered probability space $(\Omega, \mathbb{Q}, \{\mathcal{F}_t\})$ we consider the automatic differentiation algorithm applied to random variables X.

Since automatic differentiation is a formal (algebraic) framework, the rules to calculate derivatives remain unchanged and we immediately obtain an algorithm for calculating $\partial Y/\partial X$, where Y and X are random variables.

While we consider the algorithm on the abstract level of random variables, our main application is a Monte-Carlo simulation, where X is a discrete random variable $\omega_k \mapsto X(\omega_k)$, $k = 0, \dots, M-1$ for a given number of paths M.

In this "stochastic" version of automatic differentiation we can then explore the property that certain differentiation operators will result in scalar values (that is, deterministic random variables), e.g., for $Y = f(X, Z) = X + Z$ we have $\partial Y/\partial X = 1$. This allows to dramatically reduce the memory requirements in an implementation. In addition we can take advantage of the fact that we can re-use random variables, i.e., we do not store value-copies but references to them.

2.2 Stochastic Operators

To improve the efficiency of the method, it is important to distinguish two classes of operators: path-wise operators and non path-wise operators. For example: $X + Y$ is a path-wise operator, because $(X + Y)[\omega] := X(\omega) + Y(\omega)$. On the other hand, $\mathrm{E}(X)$ (the expectation operator) is a non-path-wise operator.

2.2.1 Path-wise Operators

A standard automatic differentiation applied to the operators on floating point numbers (e.g. `double`) would naturally allow to calculate $\frac{\partial Y(\omega_i)}{\partial X(\omega_j)}$ where $\omega_i, \omega_j \in \Omega$ denote paths from the (Monte-Carlo) space Ω.

However, for a path-wise operator the differentiation of the cross term $\frac{\partial Y(\omega_i)}{\partial X(\omega_j)}$ is always zero for $i \neq j$. Hence, for a path-wise operator on a discrete space Ω the derivative matrix

$$\left(\frac{\partial Y(\omega_i)}{\partial X(\omega_j)}\right)_{i,j=0,\dots,M-1}$$

is a diagonal matrix and can be stored as a vector and re-interpreted as a random variable $\omega_i \mapsto \frac{\partial Y(\omega_i)}{\partial X(\omega_i)}$, $i = 0, \dots, M-1$.

If all operators are path-wise, all derivatives will be path-wise (act path-wise) and can be stored as random variables. Given a discrete sample space, the algorithm would coincide with a vectorized algorithmic differentiation (except for some additional memory optimizations, see Section 3.1). In order to keep the algorithm efficient, one needs to avoid the differentiation of non-path-wise operators. We

show that this is possible (under mild additional assumptions) for the expectation operator and the conditional expectation operator.

2.2.2 Expectation and Conditional Expectation Operator

So far, the family of operators considered is limited to arithmetic (i.e., path-wise) operations on random variables. We will now consider an algorithm containing the expectation operator (unconditional and conditional). We interpret the (unconditional and conditional) expectation operator as an operator mapping a random variable to a random variable, namely $Y = \mathrm{E}(X)$ is given by $Y(\omega) = \mathrm{E}(X)[\omega]$. In other words, the random variable $\mathrm{E}(X)$ is constant on every path.

The expectation operator is non-path-wise. For example, in finite dimensional space Ω (or likewise in a Monte-Carlo simulation) we have

$$\mathrm{E}(X) \;=\; \frac{1}{M} \sum_{i=0}^{M-1} X(\omega_i).$$

and the derivative

$$\frac{\partial}{\partial X}\mathrm{E}(X) \;=\; \frac{1}{M} \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix}$$

is a full matrix (not a diagonal matrix).

However, the expectation operator is a linear operator. If $Z = f(Y)$ and f is a path-wise operator, we have

$$\frac{\partial}{\partial Y}\mathrm{E}(Z) \;=\; \mathrm{E}(\frac{\partial Z}{\partial Y}).$$

Hence, we can avoid the differentiation of an expectation operator via a small adjustment of the automatic differentiation algorithm: given

$$\begin{aligned} y &:= x_N \\ x_m &:= f_m(x_{i_1^{(m)}}, \dots, x_{i_{k^{(m)}}^{(m)}}) \end{aligned}$$

where the k-th operator is an expectation operator on $x_{i_1^{(k)}}$, i.e., $x_k = f_k(x_{i_1^{(k)}}) = \mathrm{E}(x_{i_1^{(k)}})$. Then:

for all $m = N-1, N-2, \dots, n$ (iterating backward through the operator list) set

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

and for all $m = i_1^{(k)} - 1, i_1^{(k)} - 2, \dots, n$ (iterating backward through the operator list) set

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

where I is the set of all indices l such that f_l depends on x_m (note that this implies $l > m$ and $m \in \{i_1^{(l)}, \dots, i_{k^{(l)}}^{(l)}\}$), with initial condition $\frac{\partial y}{\partial x_N}\big|_{x_k} = \frac{\partial y}{\partial y}\big|_{x_k} = 1$ and $\frac{\partial x_{i_1^{(k)}}}{\partial x_{i_1^{(k)}}} = 1$.

Then

$$\frac{\partial y}{\partial x_m} = \frac{\partial y}{\partial x_m}\Big|_{x_k} + \frac{\partial y}{\partial x_k}\Big|_{x_k} \cdot \mathrm{E}\left(\frac{\partial x_{i_1^{(k)}}}{\partial x_m}\right)$$

with $\frac{\partial x_{i_1^{(k)}}}{\partial x_m} = 0$ for $m > i_1^{(k)}$.

Note that this destroys the elegance of the backward automatic differentiation, since we first have to calculate $\frac{\partial x_{i_1^{(k)}}}{\partial x_m}$ for $m < i_1^{(k)}$ is a separate backward automatic differentiation, then use the result as a "leaf-node" in a second run, where the expectation has to be applied to $\frac{\partial x_{i_1^{(k)}}}{\partial x_m}$. See Figure 1.

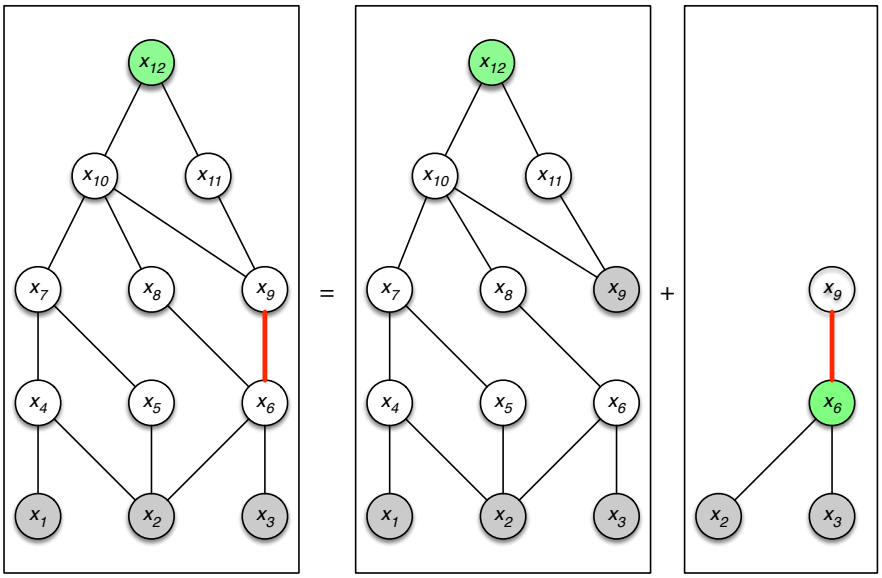

Figure 1: Example for an operator tree calculating x_{12} as a function of x_1, x_2, x_3 where $x_9 = \mathrm{E}(x_6|F)$ is an (conditional) expectation operator. The calculation is decomposed into two independent backward propagations, where the first one is treating x_9 as an independent leaf-node and the second one is calculating x_6 as a function of x_1, x_2, x_3.

2.3 Expected Stochastic Automatic Differentiation

Under a comparably mild assumption the backward automatic differentiation can be simplified greatly. We assume that the last operator f_N is an expectation operator,

that is, we are only interested in the expectation of the Monte-Carlo simulation. Almost all applications in mathematical finance can be written as this.

In this case, we can use that for any two random variables A and B we have

$$\mathrm{E}(A \cdot \mathrm{E}(B)) \;=\; \mathrm{E}(A) \cdot \mathrm{E}(B) \;=\; \mathrm{E}(\mathrm{E}(A) \cdot B). \tag{3}$$

In other words: if we are only interested in an expectation of the backward automatic differentiation, then expectation of the second factor can be replaced by taking expectation on the first factor. From this we can replace

$$\frac{\partial y}{\partial x_m} \;=\; \frac{\partial y}{\partial x_m}\Big|_{x_k} + \frac{\partial y}{\partial x_k}\Big|_{x_k} \cdot \mathrm{E}\left(\frac{\partial x_{i_1^{(k)}}}{\partial x_m}\right)$$

by

$$\frac{\partial y}{\partial x_m} \;\overset{\mathrm{E}}{=}\; \frac{\partial y}{\partial x_m}\Big|_{x_k} + \mathrm{E}\left(\frac{\partial y}{\partial x_k}\Big|_{x_k}\right) \cdot \frac{\partial x_{i_1^{(k)}}}{\partial x_m},$$

were $\overset{\mathrm{E}}{=}$ denotes "equality in expectation", that is $A \overset{\mathrm{E}}{=} B \;:\Leftrightarrow\; \mathrm{E}(A) = \mathrm{E}(B)$.

However, the relation (3) holds for random variables (in discrete space, a vector) only and in general $\frac{\partial x_i}{\partial x_j}$ is an operator (in discrete space, a matrix) and for operators we do not have commutation in general. Note that propagating matrices instead of vectors would likely destroy the efficiency of the algorithmic differentiation.

Our main result, which we will state next, is that we can keep the efficient propagation of random variables, each transformed by either a partial derivative operator or a non-path-wise operator (from a certain class) an retain the simplicity of a (slightly modified) backward automatic differentiation algorithm.

Theorem 1. *(Expected Stochastic Backward Automatic Differentiation) Let $\mathcal{G}$ denote a family of linear operators such that for any random variable A, B and any $G \in \mathcal{G}$ we have*

$$\exists\, G^* \;:\qquad \mathrm{E}\left(A \cdot G(B)\right) \;=\; \mathrm{E}\left(G^*(A) \cdot B\right).$$

Furthermore let the operators f_m be sufficiently regular such that $\frac{\mathrm{d}}{\mathrm{d}x_j}\mathrm{G}\left(f_m\right) = \mathrm{G}\left(\frac{\mathrm{d}f_m}{\mathrm{d}x_j}\right)$.[2] *Then the modified backward automatic differentiation algorithm, where $D_i^{\mathcal{G}}$ are random variables on Ω with:*

Initialise $D_N^{\mathcal{G}} = 1$ and $D_m^{\mathcal{G}} = 0$ for $m \neq N$.

For all $m = N, N-1, \ldots, n$ (iterating backward through the operator list),

[2] Although this assumption may be non-trivial in general, it is trivial if we consider Ω to be a discrete (Monte-Carlo) sampling space.

for all $j = 1, \ldots, k^{(m)}$ *(iterating through the argument list)*

$$D^{\mathcal{G}}_{i^{(m)}_j} \rightarrow \begin{cases} D^{\mathcal{G}}_{i^{(m)}_j} + D^{\mathcal{G}}_m \cdot \frac{\partial f_m}{\partial x_{i^{(m)}_j}}(x_{i^{(m)}_1}, \ldots, x_{i^{(m)}_{k^{(m)}}}) & \text{if } f_m \notin \mathcal{G} \\ D^{\mathcal{G}}_{i^{(m)}_j} + G^* \left(D^{\mathcal{G}}_m \right) & \text{if } f_m = G \in \mathcal{G} \end{cases}$$

gives

$$\mathrm{E}\left(\frac{\partial y}{\partial x_i}\right) = D^{\mathcal{G}}_i.$$

For the proof see Appendix A. The result can be stated as follows: the expectation operator E defines a scalar product on the set of random variables. With respect to this scalar product, G^* is just the adjoint operator for of G. Having a chain of operators we apply them backward by using the adjoint operators.[3]

Remark

If the independent variable x_i is a scalar (identified with a constant random variable), we find

$$\mathrm{E}\left(\frac{\partial y}{\partial x_i}\right) = \mathrm{E}\left(D^{\mathcal{G}}_i\right).$$

This is possible the most usual case as in many application we consider derivatives of expectations of random variables, with respect to deterministic model parameters.

2.3.1 Conditional Expectation Operator

Obviously the previous result also holds for the expectation operator and more general for the conditional expectation operator $\mathrm{E}(\cdot|\mathcal{F}_t)$, since (by the tower law)[4]

$$\mathrm{E}(A \cdot \mathrm{E}(B|\mathcal{F}_t)) = \mathrm{E}(\mathrm{E}(A|\mathcal{F}_t) \cdot \mathrm{E}(B|\mathcal{F}_t)) = \mathrm{E}(\mathrm{E}(A|\mathcal{F}_t) \cdot B). \qquad (4)$$

This allows to apply the algorithm in the presents of an American Monte-Carlo, e.g., where a least-squares-regression is used to estimate the conditional expectation, see [13]. Classical applications of automatic differentiation, see [2, 7], would differentiate the regression algorithm. Note that differentiating an approximation may in general lead to undesired results, since it is not clear if the derivative of the approximation is an approximation of the derivative.[5] A practical example would be the estimation of the conditional expectation via a binning using fixed bins, see [15].

[3] Speaking of matrices, for derivatives, right multi-plication is replaced by left-multiplication, but for path-wise operators f_m the derivative is self-adjoint.

[4] We have $\mathrm{E}(X) = \mathrm{E}(\mathrm{E}(X|\mathcal{F}_t))$ and $\mathrm{E}(A \cdot \mathrm{E}(B|\mathcal{F}_t)|\mathcal{F}_t) = \mathrm{E}(A|\mathcal{F}_t) \cdot \mathrm{E}(B|\mathcal{F}_t)$.

[5] For example (possibly slightly oversimplified): For small x we have that $1 + x$ is a good approximation of $\exp(x)$, and $1 = \frac{\partial}{\partial x}(1 + x)$ is still a comparably good approximation for $\frac{\partial}{\partial x}\exp(x) = \exp(x)$, but $0 = \frac{\partial^2}{\partial x^2}(1 + x)$ is maybe not a good approximation for $\frac{\partial^2}{\partial x^2}\exp(x) = \exp(x)$.

Here the differentiation of the binning would either result in a derivative of zero or in high Monte-Carlo errors due to differentiation of the indication functions.

Instead, here, we can just apply the conditional expectation operator to the derivative of the dependent variable (retaining the original approximation properties of the conditional expectation estimator).

Remark (Avoiding Differentiation of the Filtration) For the conditional expectation operator we assume that

$$\frac{\partial}{\partial x_m}\mathrm{E}\left(x_{i_1^{(k)}}|\mathcal{F}_t\right) = \mathrm{E}\left(\frac{\partial x_{i_1^{(k)}}}{\partial x_m}\Big|\mathcal{F}_t\right), \tag{5}$$

that is, the filtration $\{\mathcal{F}_t\}$ does not depend on our independents. This assumption is fulfilled in typical applications, e.g., in the application to American Monte-Carlo algorithms. For example, if the model generating the random variables is an Itô processes, then the filtration is generated by the Brownian motion. If we simulate the process using a Monte-Carlo simulation, then differentiating the filtration would be as if we differentiate the random numbers, which would neither be required nor make sense (e.g., note that in a finite difference differentiation one keeps random numbers fixed).

Relation to Different Methods for Approximating the Conditional Expectation Operator We like to stress that it is left open how the conditional expectation operator is implemented or approximated and it is an important advantage of the methodology that this is left open. Theorem 1 holds for conditional expectation operators in general and not for their approximations. That is, we may use this result prior to applying any methodology for the approximation of the conditional expectation operator. Hence, (4) and (5) not only allows us to avoid the differentiation of the conditional expectation operator, it also allows us to use a different approximation for the conditional expectation operator on A than that on B in (4).

Using the automatic tracking of the measurability (see [13]), i.e. investigating if in (4) the random variable A is already $\mathcal{F}_t$-measurable, it is possible to just remove the conditional expectation operator.

2.3.2 Indicator Functions

Indicator functions are path-wise operators and very common in financial applications. The payoff of a digital product is an indicator function. The valuation of a Bermudan product involves the composition of an indicator function and a conditional expectation. Other examples for products involving indicator functions are barrier options and TARNs ([26]). The valuation of a CVA includes an indicator function, where the argument is a conditional expectation of a complex portfolio value and the jump size is non-zero (making it different from Bermudan options with an optimal exercise criteria), [4].

Upon the presence of an indicator function, we have to consider

$$\frac{\partial}{\partial X}\mathbf{1}(X>0),$$

where

$$\mathbf{1}(X>0)[\omega] := \begin{cases} 1 & \text{for } X(\omega)>0, \\ 0 & \text{else.} \end{cases}.$$

This differentiation would result in a distribution.

Note again that the automatic differentiation algorithm (or say, the chain rule) applied to algorithms (functions) involving indicator functions, results in linear combinations of differentiations of indicator functions. Hence we have to evaluate linear combinations of (different) expressions of the form

$$A\frac{\partial}{\partial X}\mathbf{1}(X>0),$$

where A is a linear operator.

If we are only interested in the expectation of the final result, it is sufficient to consider

$$\mathrm{E}\left(A\frac{\partial}{\partial X}\mathbf{1}(X>0)\right),$$

which evaluates to

$$\mathrm{E}\left(A\frac{\partial}{\partial X}\mathbf{1}(X>0)\right) = \mathrm{E}\left(A \mid \{X=0\}\right). \tag{6}$$

Although almost trivial, (6) has an important consequence when considering an expected stochastic algorithmic differentiation: it tells us that the differential operator applied the indicator function maps to a conditional expectation operator applied to the adjoint differential A. This is apparent because we can just add an expectation to the right

$$\mathrm{E}\Big(A\frac{\partial}{\partial X}\mathbf{1}(X>0)\Big) = \mathrm{E}\Big(\mathrm{E}\left(A \mid \{X=0\}\right)\Big).$$

that is, using the notation introduced in the beginning of Section 2.3,

$$A\frac{\partial}{\partial X}\mathbf{1}(X>0) \overset{\mathrm{E}}{=} \mathrm{E}\left(A \mid \{X=0\}\right).$$

At this stage it is open how to evaluate this operator, but even if we take a simple finite difference approximation we encounter a striking advantage: the evaluation of (6) can be performed for each discontinuity individually, allowing an important optimization.

The ability to evaluate the expression (6) per operator solves an issue related to classical finite difference to the whole valuation algorithm: the choice of the optimal shift size, see below.

Application to Likelihood Ratio, Proxy Simulation Schemes or Analytic Integration The calculation of accurate sensitivities of Monte-Carlo valuations involving discontinuities is a difficult problem and different methods have been proposed for its solution, [10] provides a excellent overview. A remark is in order how the present approach combines with the methods.

In fact, all these methods may be combined with a stochastic algorithmic differentiation.

In special cases, the Monte-Carlo integral of the indicator function can be replaced by an analytic function. Examples are [23, 27]. In this case the automatic differentiation will just differentiate that analytic function (likely providing a much more accurate result than an application of a finite difference).

Another approach is to use coordinate transformations to remove the dependency of the indicator function on the respective parameters by performing the corresponding transformations on the probability density or the probability measure. Examples are the likelihood ratio method [20], Malliavin methods [5], and proxy simulation schemes [18, 16, 17, 9, 8, 10]. Since all these methods effectively remove the dependency of the discontinuity on the independent parameter, we can apply the stochastic algorithmic differentiation right away, say for example to the proxy scheme, and replace any encounter of $\frac{\partial}{\partial X}\mathbf{1}(X > 0)$ by 0. This is trivial, because these methods ensure that X does not cross the indicator functions boundary. This trick can be used to improve the performance of the algorithm, because we do not need to evaluate the corresponding expression: the stochastic algorithmic differentiation will propagate a scalar 0 though the chain rule, instead of a path-wise derivative.

At the same time, we profit from the advantages of an algorithmic differentiation, e.g., its accuracy on the smooth paths and its performance.

While pure likelihood ratio methods have large Monte-Carlo variance when the underlying functions are smooth, partial proxy schemes and (optimally) localized schemes do not suffer from this defect. However, there is fundamental drawback in the partial proxy simulation methods ([16, 17, 9, 8, 10]): they require a product dependent modification of the simulation. This can be seen as a performance issue. In addition the methods require more in-depth knowledge on the argument of the indicator function: the size of the jump, the location of the jump and the gradient (speed) of the jump boundary. Hence, such methods are often applied to products where such information about the indicator function can be easily extracted from the valuation algorithm (digitals, barriers or TARNs). Note for example that the case where a conditional expectation is part of the argument of the indicator function is much harder (except for the case of product with optimal exercise, where the differentiation of the indicator may be omitted).

Both aspects, the product dependency and the required knowledge of the discontinuity, imposes a heavy constrain when considering portfolio valuations (like xVA). In those applications a simple method may be desirable.

Per Operator Finite Difference Approximation In cases where it is infeasible to calculate $\mathrm{E}\left(A \mid \{X=0\}\right)$ in a more sophisticated way, we may use a classic finite difference approximation. We recover the classical finite-difference approximation on a per-operator basis: The expression (6) can be approximated by

$$\mathrm{E}\left(A \mid \{X=0\}\right) \;=\; \mathrm{E}\left(A \frac{1}{2\delta}\mathbf{1}(|X|<\delta)\right).$$

In other words, if we are only interested in the expectation of the final result, we can approximate

$$\frac{\partial}{\partial X}\mathbf{1}(X>0) \;\overset{\mathrm{E}}{\approx}\; \frac{1}{2\delta}\mathbf{1}(|X|<\delta),$$

Here $\overset{\mathrm{E}}{\approx}$ denotes "approximation in expectation", that is $A \overset{\mathrm{E}}{\approx} B \;:\Leftrightarrow\; \mathrm{E}(A) \approx \mathrm{E}(B)$.

This approximation has an intuitive interpretation. First, we may observe that this approximation agrees with the result of a so-called payoff-smoothing, where the indicator function is approximated by a call spread

$$\mathbf{1}(X>0)[\omega] \;\approx\; \begin{cases} 1 & \text{for } X(\omega) > \delta, \\ \frac{X(\omega)+\delta}{2\delta} & \text{for } |X(\omega)| < \delta, \\ 0 & \text{for } X(\omega) < \delta. \end{cases}$$

But more strikingly, the approximation is just a central finite difference approximation of the derivative. It is

$$\frac{\partial}{\partial X}\mathbf{1}(X>0) \;\overset{\mathrm{E}}{\approx}\; \frac{\mathbf{1}(X+\delta>0) - \mathbf{1}(X-\delta>0)}{2\delta}.$$

In other words, we have locally replaced the automatic differentiation by a (local) finite-difference approximation.

Since our algorithm (e.g., the implementation in [11]) has access to the full random variable X, we can achieve an important improvement in the numerical algorithm: we can choose the δ shift appropriate for the random variable under consideration (see [13] for numerical results using Bermudan digital options).

We like to stress the advantage of this finite difference approximation within an expected stochastic algorithmic differentiation, given that the algorithmic differentiation automatically decomposes the differentiation into individual contributors: while a classical finite difference approximation of $\frac{\partial y}{\partial x_m}$ would propagate a single shift size though the valuation algorithms, here we use the finite difference only at the specific operator.

Further improvement, e.g., in the spirit of localized proxy simulation schemes, are possible and part of future research.

2.4 Generalization to Risk Measures: Automatic Differentation of Expected Shortfall, VaR, Quantiles

The Theorem 1 can be generalized by replacing the conditional expectation operator E by a general projection operator Π and setting the initial value of $D_N^{\mathcal{G}}$ to the associated indicator function $\mathbf{1}_\Pi$, that is $\mathbf{1}_\Pi$ is such that

$$\Pi(A) \;=\; \mathrm{E}(\mathbf{1}_\Pi \cdot A) \qquad \text{for all random variables } A.$$

Then we have

Theorem 2. *(Expected Stochastic Backward Automatic Differentiation) Let $\mathcal{G}$ denote a family of linear operators such that for any random variable A, B and any $G \in \mathcal{G}$ we have*

$$\exists\, G^* \;:\qquad \Pi\left(A \cdot G(B)\right) \;=\; \Pi\left(G^*(A) \cdot B\right).$$

Furthermore let the operators f_m be sufficiently regular such that $\frac{\mathrm{d}}{\mathrm{d}x_j}\mathrm{G}\left(f_m\right) = \mathrm{G}\left(\frac{\mathrm{d}f_m}{\mathrm{d}x_j}\right)$.[6] Then the modified backward automatic differentiation algorithm, where $D_i^{\mathcal{G}}$ are random variables on Ω with:

Initialise $D_N^{\mathcal{G}} = \mathbf{1}_\Pi$ and $D_m^{\mathcal{G}} = 0$ for $m \neq N$.

For all $m = N, N-1, \ldots, n$ (iterating backward through the operator list),

for all $j = 1, \ldots, k^{(m)}$ (iterating through the argument list)

$$D^{\mathcal{G}}_{i_j^{(m)}} \;\to\; \begin{cases} D^{\mathcal{G}}_{i_j^{(m)}} + D_m^{\mathcal{G}} \cdot \frac{\partial f_m}{\partial x_{i_j^{(m)}}}(x_{i_1^{(m)}}, \ldots, x_{i_{k^{(m)}}^{(m)}}) & \text{if } f_m \notin \mathcal{G} \\ D^{\mathcal{G}}_{i_j^{(m)}} + G^*\left(D_m^{\mathcal{G}}\right) & \text{if } f_m = G \in \mathcal{G} \end{cases}$$

gives

$$\Pi\left(\frac{\partial y}{\partial x_i}\right) \;=\; D_i^{\mathcal{G}}.$$

The proof is analogue to the proof of Theorem 1. This generalization allows to use backward automatic differentiations to calculate first order partial derivatives of risk measures. Here it is required to note (or assumed) that the property $y(\omega) > \alpha$ is stable under small perturbations of the independent variables x_i $(i = 0, \ldots, n)$. In this case there is no contribution of possible change of the relevant scenarios ω to the first order derivative coming. This is similar to the corresponding result for Bermudan options in [25].

[6] Although this assumption may be non-trivial in general, it is trivial if we consider Ω to be a discrete (Monte-Carlo) sampling space.

2.5 Further Applications

The Theorem 1 can also be generalized replacing the final operator $\mathrm{E}(\cdot)$ by a conditional expectation $\mathrm{E}(\cdot\,|\mathcal{F}_t)$. Surprisingly, adding some natural assumptions on the structure of the valuation, this can be done by using a single seeding initial value $D_N^{\mathcal{G}} = 1$ and obtain all future conditional expectations at once. This allows to calculate all partial derivatives at all future points in time with a single backward sweep. Important applications inlcude, e.g., the construction of a hedge simulation and the calculation of an MVA. This generalization is elaborated in the forthcoming [14].

2.6 Implementation Design

2.6.1 Backward Automatic Differentiation

The algorithm in Theorem 1 suggests that we have to traverse the list of operators f_m in the strict order $m = N, N-1, \ldots, 0$. However, it is not required to follow this ordering strictly, given that nodes are mutually independent, i.e., x_i, x_j are such that x_i does not depend on x_j and vice versa. Also, it is not necessary to consider node x_k, where the final result x_N does not depend on x_k ("dead ends"). Figure 2 depicts a situation where x_7 is a dead end w.r.t. a calculation of x_{10} as a function of x_1, x_2, x_3.

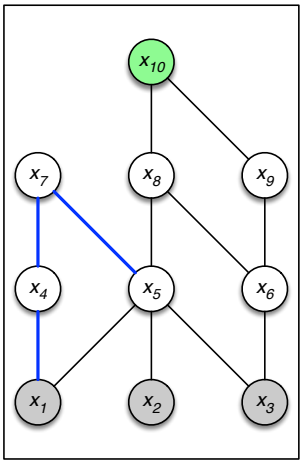

Figure 2: Operator tree with dead end (x_7) in case we are interested in the calculation of the derivative of x_{10} with respect to x_1, x_2, x_3. In this case it is not required to process the node x_4 (since it appears as an argument to x_7 only). Hence the partial derivatives at the blue vertices are not evaluated.

To traverse only relevant nodes, we maintain a list of unprocessed relevant nodes, initialise it with $D_N^* = 1$, then repeatedly taking the node with the highest index from this list, apply the backward propagation, add it's arguments to the list and remove the processed node. This algorithm is easily implemented using a Java `TreeMap` for $k \mapsto D_k^*$, a map which maintains an ordering on the keys.

The algorithm then reads

```
public Map<Long, RandomVariableInterface> getGradient() {

  // The map maintaining the derivatives id -> derivative
  Map<Long, RandomVariableInterface> derivatives = new HashMap<>();
  // Put derivative of this node w.r.t. itself
  derivatives.put(getID(), new RandomVariable(1.0));

  // The set maintaining the independents. Note: TreeMap is maintaining a sorting on the keys.
  TreeMap<Long, OperatorTreeNode> independents = new TreeMap<>();
  // Initialize with root node
```

```
  independents.put(getID(), this.getOperatorTreeNode());

  while(independents.size() > 0) {
    // Get and remove node with the highest id in independents
    Map.Entry<Long, OperatorTreeNode> independentEntry = independents.
        pollLastEntry();
    Long id = independentEntry.getKey();
    OperatorTreeNode independent = independentEntry.getValue();

    // Process this node (node with highest id in independents)
    List<OperatorTreeNode> arguments = independent.arguments;
    if(arguments != null && arguments.size() > 0) {
      // Node has arguments: Propagate derivative to arguments.
      independent.propagateDerivativesFromResultToArgument(derivatives);
      // Remove id of this node from derivatives - keep only leaf nodes.
      derivatives.remove(id);

      // Add all non leaf node arguments to the list of independents
      for(OperatorTreeNode argument : arguments) {
        // If an argument is null, it is a (non-differentiable) constant.
        if(argument != null) independents.put(argument.id, argument);
      }
    }
  }

  return derivatives;
}
```

Remark The modification of the backward automatic differentiation algorithm to traverse only relevant argument nodes improves the performance (w.r.t. CPU time) significantly. In Section 4.2 we give an example where the calculation of the vega in a LIBOR market model shows such a performance improvement depending on the financial product (e.g., the maturity of a Caplet).

2.6.2 Automatic Differentiation of (Conditional) Expectation Operator

With the reformulation of the automatic differentiation in the presence of conditional expectation operators, we are able to implement the methodology with a minimum of code complexity: we require only two additional lines of code: In [12] all random variable objects of a Monte-Carlo simulation are implementing a common interface `RandomVariableInterface`. This interface offers methods for arithmetic operators like `add`, `sub`, `mult`, `exp`, but also an operator

```
RandomVariableInterface getConditionalExpectation(
    ConditionalExpectationEstimatorInterface estimator);
```

The default implementation of this method is just

```
default RandomVariableInterface getConditionalExpectation(
    ConditionalExpectationEstimatorInterface estimator)
{
  return estimator.getConditionalExpectation(this);
}
```

In other words: all conditional expectations of random variables have to be called through this method - which enables us to track conditional expectation in the operator tree.

To enable adjoint automatic differentiation we inject a special implementation of `RandomVariableInterface` which records the operator tree and can perform the AAD. The implementation of the differential of the operator `getConditionalExpectation` now consists of two parts: First, the partial derivative of the operator with respect to its arguments is set to 1.0:

```
case CONDITIONAL_EXPECTATION:
 return new RandomVariable(1.0);
```

- this implies that for the arguments the conditional expectation is replaced by the identity operator. Second, the backward propagation of the differential contains the additional check for the operator, applying it to the differential if required:

```
if(operatorType == OperatorType.CONDITIONAL_EXPECTATION) {
 ConditionalExpectationEstimatorInterface estimator = (
     ConditionalExpectationEstimatorInterface)operator;
 derivative = estimator.getConditionalExpectation(derivative);
}
```

3 Efficient Implementation of Automatic Differentiation for Monte-Carlo Simulations

3.1 Memory Requirements

The backward automatic differentiation suggests that we need to store the argument values $x_{i_1^{(m)}}, \dots, x_{i_{k(m)}^{(m)}}$ to perform the calculation of $\frac{\partial f_m}{\partial x_{i_j^{(m)}}}$ in the corresponding node - using lazy evaluation - or the value of $\frac{\partial f_m}{\partial x_{i_j^{(m)}}}$ - using direct valuation. Both approaches may have strong implications on the memory requirements, especially if the arguments are random variables on a large discrete sample space. In the end we may have to hold all N intermediate values.

Using immutable objects representing the random variables and storing references to them only if needed, we can reduce the memory requirements. The main concepts are:

- We store only a single scalar value for a deterministic random variable, once we know that it is deterministic. This requires a specific *object* to be defined for random variables.
- We store references to random variables, once we need to store a value which has been previously calculated. This requires that objects are *immutable*.
- We use lazy evaluation to calculate values derived from known objects (rather than storing values).

The following examples illustrate this. Note that, while the examples are to some extend trivial calculations (sums, products, etc.), these calculations are frequently encountered and essentially the building blocks of numerical algorithm. At the end we have all parts to discuss an Euler discretization scheme of a multi-dimensional SDE.

3.2 Examples

We start by discussing basic examples to show how immutable objects and lazy evaluation are used to reduce the memory requirement of an adjoint automatic differentiation.

3.2.1 Example: a big sum

Let us first consider the case of lazy valuation, i.e., where we store all argument values $x_{i_1^{(m)}}, \dots, x_{i_{k(m)}^{(m)}}$ as part of the operator tree.

Let x denote a random variable, which is represented by an object implementing the interface `RandomVariableDifferentiableInterface`. Consider the following algorithm (given in Java pseudo-code):

```
RandomVariableDifferentiableInterface x = new
    RandomVariableDifferentiableAAD(...);
RandomVariableDifferentiableInterface sum = new
    RandomVariableDifferentiableAAD(0.0);
for(int i=0; i<numberOfSummations; i++) {
  sum = sum.add(x);
}
```

that is, we calculate $y = \sum_{i=0}^{N-1} x_0$. In a generic implementation of the backward automatic differentiation the algorithm would store the partial sum $\mathtt{sum}_i$ and x as part of the argument list of the calculation of

$$\mathtt{sum}_{i+1} \ = \ f(\mathtt{sum}_i, \mathtt{x}) = \mathtt{sum}_i + \mathtt{x}.$$

If the implementation of the backward automatic differentiation maintains a list of the operators (here f) and the complete arguments list (here $(\mathtt{sum}_i, x)$), then this implies that the algorithm holds every intermediate result $\mathtt{sum}_i$ for $i = 0, \ldots, n$ and eventually copies of x.

This is inefficient since the intermediate results are not required for the calculation of the partial derivatives. For the partial derivatives we have $\frac{\partial f(x,y)}{\partial x} = 1$ and $\frac{\partial f(x,y)}{\partial y} = 1$.

Obviously in this case it would be more efficient to calculate all partial derivatives along the calculation of the value of an operation and storing that value, i.e., 1. However, an automatic differentiation algorithm implemented on the level of the scalars would store this 1 for each Monte-Carlo sample path. Since we define the algorithm on the level of the random variables, we can utilise the fact that 1 is scalar.

For this example our implementation stores the operator tree with N nodes, each node storing a label for the function f being addition (`add`) and not reference to the arguments, since the partial derivative of the function is always a scalar 1.

To summarize: For the example of a sum we see that the object orientation brings a factor of m in the reduction of the memory requirements, where m refers to the number of paths:

- A (simultaneous) path-wise application of automatic differentiation has a memory requirement of $O(N \cdot m)$ storing the N partial derivatives on the m path. All derivatives are 1. Note that this is in line with the theoretical value of $O(T_f) \cdot M_f$ since $T_f = N$ and $M_f = m$.[7]
- Using a object for random variables, optimizing the storage of the deterministic random variable, our memory requirement will be just $O(N)$.

The next example will show that a lazy valuation of the partial derivatives combined with storing references may be more efficient than a direct valuation.

[7] Here we refer to the result stated in Section 1.1.1 where M_f refers to the memory requirement of the valuation and T_f refers to the computation time requirement of the valuation.

3.2.2 Example: geometric sum

Consider the calculation of the geometric sum $\sum_{i=0}^{N-1}(x)^i$ via the following algorithm (here `values` is an array of realizations of x):

```
RandomVariableInterface x = new RandomVariableDifferentiableAAD(values);
RandomVariableInterface sum = new RandomVariableDifferentiableAAD(0.0);
for(int i=0; i<numberOfSummations; i++) {
  sum = sum.add(x.pow(i));
}
```

The calculation consist of the operators

$$\mathtt{sum}_{i+1} \;=\; f(\mathtt{sum}_i, \mathtt{x}_i) = \mathtt{sum}_i + \mathtt{x}_i$$

$$\mathtt{x}_i \;=\; g(x, i) = (x)^i$$

For the partial derivatives we have $\frac{\partial f(x,y)}{\partial x} = 1$, $\frac{\partial f(x,y)}{\partial y} = 1$ and $\frac{\partial g(x,y)}{\partial x} = y \cdot x^{y-1}$.

In the case where x is a random variable, the storage of the partial derivatives along the calculation of the values (direct valuation of the partial derivatives) would require to store all intermediate random variables $i \cdot x^{i-1}$. Note that the objects representing $i \cdot x^{i-1}$ are different and non-deterministic random variables. Alternatively, a lazy evaluation would just storing the reference to the operator g and the argument list (x, i), where the second argument is a scalar and x is stored as a reference to a single instance of the object representing x, that is x only consumes storage once, since we store a reference to x and not a copy of x.

For this example, our implementation stores the operator tree with $2N$ nodes, each node storing a label for the operator, either `add` or `pow`. For `pow` we store a *reference* to x and i. The values of x is stored only once.

To summarize, using immutable objects and storing references in a lazy evaluation, we find

- A classic adjoint algorithmic differentiation would require $O(N \cdot m)$ memory storage for the N partial derivatives, each being an m-vector.
- Using a random variable object keeps a reference to the immutable object x, performing lazy evaluation of $\frac{\partial g(x,y)}{\partial x}$ we require $O(N) + O(m)$ memory storage.

3.3 Decoupling of Values and Operator Tree

In the example 3.2.1 it appears to be optimal to perform a direct valuation of the partial derivative and and take advantage of the fact that the partial derivative is a scalar. In the example 3.2.2 it appears to be optimal to perform a lazy evaluation of partial derivative and take advantage of the fact that the argument lists consist of references to a small selection of random variables (objects) and scalars i.

However, we may obtain the optimal storage in both examples using lazy evaluation: In example 3.2.1 we may deduce from the operator that the argument

list is not needed to evaluate the partial derivatives, hence we can perform lazy evaluation of the partial derivative without retaining references to the arguments lists. This avoids storage of the intermediate results.[8]

To implement this, we have to de-couple the representation of the values (which possibly serve as arguments of operators) and the operator tree.

3.3.1 Random Variables

A random variable X in a Monte-Carlo simulation is represented by a vector of samples (realizations) $(X(\omega_k))_{k=0,\dots,M-1}$. We implement a random variable by defining an interface `RandomVariableInterface` which defines the operators available for a random variable, e.g., `add`, `sub`, `mult`, `exp`, etc.

Our implementation of the interface is a class `RandomVariable` which internally stores the Monte-Carlo vector as a `double[]`. However, we provide an alternative storage model (or implementation) for random variables which are deterministic which are then represented by a single `double` which allows effective reduction of the memory requirements.[9]

3.3.2 Operator Tree and Argument Lists

The key step in our implementation design for the automatic differentiation is to decouple the objects referenced by the algorithms from the objects required for the automatic differentiation algorithm. We introduce a class

`RandomVariableDifferentiableAAD`

where objects of this class represent a random variable X. The class has two members, `values` and `operatorTreeNode`:

```
private final RandomVariableInterface values;
private final OperatorTreeNode operatorTreeNode;
```

Here `values` represents the (classical) value of X. The implementation of `RandomVariableInterface` is such that it can represent a scalar (`double`) if X is deterministic or an array (`double[]`) if X is stochastic (samples). This reduces the storage cost.

In addition `operatorTreeNode` defines the operator and the argument list required to calculate X. In the notation of Section 1 an `operatorTreeNode` consist of

- an ID, e.g. the index m,
- the name of the operator f_m,

[8] In so called "managed" languages, like Java, the virtual machine will free the memory used to store an object, once there is no other references held to the object (cyclic references may be detected and removed).

[9] This design is utilized in [12] since it first release (2004).

- the operator tree nodes of the arguments, i.e., the node IDs $(i_1^{(m)}, \dots, i_{k^{(m)}}^{(m)})$ (references to objects of type `OperatorTreeNode`),
- the argument list $(x_{i_1^{(m)}}, \dots, x_{i_{k^{(m)}}^{(m)}})$ (references to objects of type `RandomVariableInterface`, the values)

The important aspect in this design is that we are keeping two different representations of the arguments lists: the argument's `operatorTreeNode` and the argument values.

Since `operatorTreeNode` is a node of a (larger) tree-list, it is likely that other calculations still hold a reference to this node, but it is now possible to remove reference to the possible large storage of `values`, as long as the operators do not require references to `values`. As we have seen in example 3.2.1, some operators do not need any reference to arguments or intermediate results.

Now, if the random variable X is only an intermediate calculation result and becomes unreferenced by the algorithm, and X is not require for the calculation of a partial derivative, then the memory to X will be freed up and the reference from X to `values` and `operatorTreeNode` will be removed, while other object can still hold reference to the `operatorTreeNode` representing X.

A reference implementation of this design can be found in [11].

3.4 Application: Implementation for Vega Calculation of Bermudan Swaptions in a LIBOR Market Model

In Section 4.2 we present some results for the calculation of model vegas of a Bermudan swaption in a LIBOR Market Model. This is a realistic and highly relevant use case as the model features over 10000 independent volatility parameters for which partial derivatives are calculated in a single backward differentiation. In addition the valuation of the Bermudan option comes with several interleaved conditional expectation operators. We focus on implementation details and performance. For the specific issues related to Bermudan options, e.g., the optimality or non-optimality of the exercise boundary see [13].

The code generating the results of Section 4.2 and the complete implementation of the model and the numerical methods are available online at [12, 11]. The results are generated by the unit test `LIBORMarketModelNormalAADSensitivitiesTest` in package `net.finmath.montecarlo.interestrate` in the folder `src/test` in [11].

The building block of the implementation is the AD capable random variable discussed in Section 3.3.1 *and* an implementation of the model and the valuation solely in terms of the interface `RandomVariableInterface`. This allows for a so called *dependency injection*, that is, we can utilize the valuation with the AAD enabled object. The AAD capable random variable is implemented in [11], while the numerical methods, mathematical model and the valuation code is implemented in [12]. The numerical methods make rigorous use of the interface and do not

make any assumption on the specific implementation. This allows the injection of different implementations: a pure valuation (without automatic differentiation), a valuation with a backward mode differentiation, or a valuation with a forward mode differentiation. The implementation at [11] provides backward and forward mode differentiation.

For this the model is discretized using a log-Euler-scheme

$$X(t_{i+1}) \;=\; X(t_i) + \mu(X(t_i)\Delta t_i + \sigma(t_i)\Delta W(t_i), \qquad X(t_0) = X_0,$$

where $X(t_i)$ are vectors of random variables. Here $X(t_0)$ and the $\sigma(t_i)$'s are the independent leaf nodes of the random variable operator tree and $X(t_{i+1})$ are (intermediate) dependent variables. This is implemented in `LIBORMarketModel` (for the function μ) and `ProcessEulerScheme`.

The valuation of the Bermudan swaption then uses a classical backward algorithm - iterating backward over the exercise times - to construct the snell envople from payoffs being functions of the $X(t_{i+1})$'s. The numéraire is also constructed as a function of the $X(t_{i+1})$'s. This is implemented in `LIBORMarketModel` (for the function numéraire) and `BermudanSwaption` for the valuation algorithm.

4 Numerical Results

4.1 Expected AAD with Conditional Expectation Operator

In [13] the expected adjoint algorithmic differentiation, our Theorem 1, is applied to the backward algorithm[10] for the valuation of Bermudan options. We reproduce the test case from [13], calculating the delta of a Bermudan digital option. This product pays

$$1 \qquad \text{if } S(T_i) - K_i > \tilde{U}(T_i) \text{ in } T_i \text{ and if no payout had been occurred before,}$$

where $\tilde{U}(T)$ is the time T value of the future payoffs, for $T_1, \dots, T_n$. Note that $\tilde{U}(T_n) = 0$, such that the last payment is a digital option.

This product is an ideal test-case: the valuation of $\tilde{U}(T_i)$ is a conditional expectation. In addition conditional expectation only appears in the indicator function, such that keeping the exercise boundary (the condition) fixed, would result in a delta of 0. On the other hand, the delta of a digital option payoff is only driven by the movement of the indicator function, since

$$\frac{\mathrm{d}}{\mathrm{d}S_0}\mathrm{E}(\mathbf{1}(f(S(T)) > 0)) = \phi(f^{-1}(0))\frac{\mathrm{df(S)}}{\mathrm{d}S_0},$$

where ϕ is the probability density of S and $\mathbf{1}(\cdot)$ the indicator function. See [15]

We calculate the delta of a Bermudan digital option under a Black-Scholes model ($S_0 = 1.0$, $r = 0.05$, $\sigma = 0.30$ using 1000000 Monte-Carlo paths). We consider an option with $T_1 = 1$, $T_2 = 2$, $T_3 = 3$, $T_4 = 4$, and $K_1 = 0.5$, $K_2 = 0.6$, $K_3 = 0.8$, $K_4 = 1.0$. The implementation of this test case is available in [11] (among many other tests of our method).

The results are depicted in Figure 3.

First, our numerical experiment shows the well known effect that a finite-difference approximation of the derivative is biased for large shift sizes and unstable / unreliable for small shift sizes. Figure 3 depicts the finite difference approximation as red dots with different shift sizes on the x-axis. The finite difference approximation was performed with different Monte-Carlo seeds and we see stable and unbiased results only for shift sizes h in the range 0.01 – 0.1.

We then depict the result of our method in green.[11] Our method reproduces the stable and unbiased result, i.e., gives the correct value. We also use different Monte-Carlo seeds for our methods, but the difference in undistinguishable in the graph.

To illustrate that taking the conditional expectation of the adjoint differential is required, we repeated the calculation without this step (which then gives wrong results): In blue we depict the value of an automatic differentiation if the conditional expectation is omitted, that is G is replaced by the identity Theorem 1, and see that this would give a wrong result. And even worse, if we would keep the exercise boundary fixed, the result would be 0.

[10] The backward algorithm is constructing the snell envelope.
[11] Since there is no shift size, the is no dependency on the shift size and we get a horizontal line.

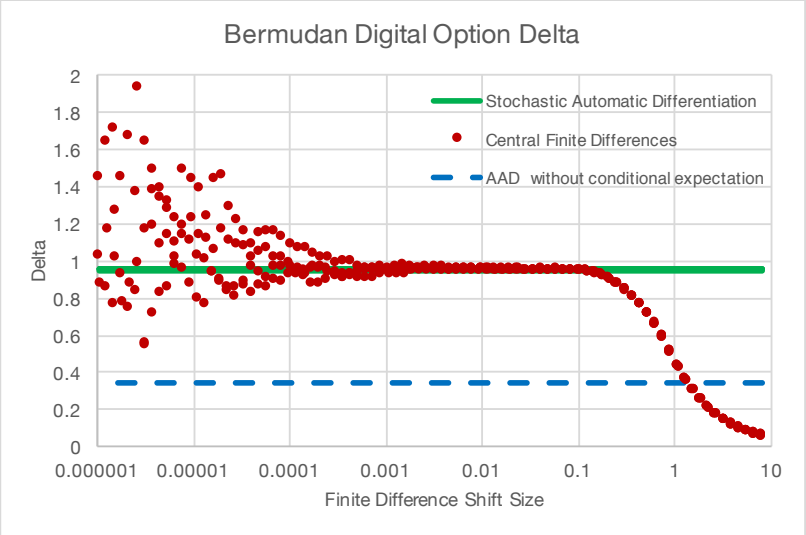

Figure 3: Delta of a Bermudan digital option using finite differences (red) and stochastic AAD. The calculations were repeated with 4 different Monte-Carlo random number seeds (hence there 4 red data series and 4 data series). The finite difference approximation is biased for large shifts size The blue dashed line depicts the value of the delta of the conditional expectation operator is ignored. This generates a bias due to the correlation of derivative and underlying.

4.2 Model Vegas in a LIBOR Market Model

In the following we show the result of the vega calculation in a LIBOR Market Model considering different products. While the test cases also include Bermudan options (i.e., approximations of conditional expectation operators), the test also illustrates the efficiency of the algorithm traversing only relevant nodes.

The LIBOR Market Model uses a 0.125 (1.5 month) time discretization, semi-annual interest rate curve discretization. The model simulates 40 years. We calculate all model vegas.[12] This amounts to 25600 theoretical vegas. Since fixed rates can be ignored we have only 12800 relevant model vegas. However, the actual amount of vegas relevant for a product depends on its maturity.

The test also illustrates the performance increase given by the modification of the algorithm to traverse only relevant operator nodes.

We test different products: Caplets, Swaptions and Bermudan Swaptions. In the test of the Bermudan Swaption we use the expected backward automatic differentiation algorithm of Theorem 1. Table 1 summarizes our results.[13]

LIBOR Market Model: model vega				
product	effective number of vegas	evaluation (plain)	evaluation (AAD)	derivative (AAD)
Caplet maturity 5.0	220	3.4 s	4.0 s	0.6 s
Caplet maturity 30.0	7320	3.8 s	4.1 s	8.3 s
Swaption 30.0 in 10.0	11880	3.2 s	3.6 s	12.9 s
Bermudan Swaption 30.0 in 10.0	12640	3.6 s	3.9 s	14.0 s

Table 1: Calculation times for the calculation of model vegas in a LIBOR Market Model. The calculation time "evaluation (plain)" is the time required for product evaluation using a standard implementation for the `RandomVariableInterface`. This valuation does not provide automatic differentiation. The calculation time "evaluation (AAD)" is the time required for product evaluation using an AAD implementation for the `RandomVariableInterface`, i.e., it includes the overhead to initially build the operator tree. The calculation time "derivative" gives the additional time required for *all* effective model vegas. All times in seconds.

Note that a finite-difference approximation of a single model vega model would

[12] We use a LIBOR Market Model in normal specification

$$\mathrm{d}L(T_j, T_{j+1}; t) \;=\; \mu_j(t)\mathrm{d}t + \sigma_j(t)\mathrm{d}W_j,$$

with $\sigma_j(t) \equiv \sigma_{i,j}$ for $t_i \leq t < t_{i+1}$ where $L(T_j, T_{j+1}; t)$ denotes the forward rate for the period $[T_j, T_{j+1}]$ observed at time t and the model vegas are the partial derivatives with respect to $\sigma_{i,j}$.

[13] The results may be reproduced by running the unit test `LIBORMarketModelNormalAADSensitivitiesTest` in [11].

require the calculation time "evaluation (plain)", e.g., a finite-difference approximation of all model vegas for the Bermudan Swaption would require 14 hours ($3.5\,\mathrm{s} \times 12640 = 49296\,\mathrm{s}$).

4.3 Numerical Results related to Memory Efficient Implementation

We present some numerical results using various implementation of the adjoint algorithmic differentiation.

We compare calculation time and memory requirements for the following algorithms:

- **finite differences**: Classic central finite differences, requiring two valuations for the approximating the derivative.
- **pathwise AAD**: Adjoint algorithmic differentiation implemented on a per path level (that is for the `double`). With lazy derivative valuation and full retaining of argument lists.
- **stochastic AAD (full)**: Stochastic auto differentiation with lazy derivative valuation and full retaining of argument list.
- **efficient stochastic AAD**: Stochastic auto differentiation with lazy derivative valuation and selective retaining of argument list.

4.3.1 Calculation of a Big Sum

We consider the evaluation of $y = \sum_{i=0}^{N-1} x$ and the calculation of $\frac{\partial y}{\partial x}$. Here x is a sample vector of 100000 `double` floating point numbers and $N = 7500$. Note that storing all intermediate results would require $7500 \cdot 100000 \cdot 8$ bytes $= 5722$ MB $= 5.6$ GB of memory.

Big Sum: $y = f(x) = \sum_{i=1}^{N} x$			
algorithm	evaluation	derivative	memory usage
finite difference	3.77 s	8.07 s	1.93 MB
pathwise AAD	6.56 s	6.62 s	5725.14 MB
stochastic AAD (full)	6.89 s	0.03 s	5727.55 MB
efficient stochastic AAD	3.41 s	0.03 s	2.17 MB

Table 2: Calculation times and memory usages for various backward algorithmic differentiation algorithms.

The central finite difference requires two valuations, hence it takes twice the time of a valuation. Since the two valuation runs are subsequent, we require just the amount to store a few random variables.

It is known that the backward algorithmic differentiation of $f : \mathbb{R}^n \mapsto \mathbb{R}$ takes approximately the same computational time as the valuation of f itself. Since both,

the valuation and the backward algorithmic differentiation have some additional overhead, the time is a bit larger than in the case of finite difference. Note that AAD would calculate all n partial derivatives in one sweep. This advantage is not visible in this example, since $n = 1$. Since AAD is performed on the level of the floating point number, storing all intermediate results would require 5 GB of memory.

Using the fact that the partial derivatives of the operator is a scalar and not a vector, we reduce the computational time to 0.03 sec - a fraction of the valuation itself.

Finally, using the fact that the intermediate results are not required in the calculation of the derivative we reduce the memory usage to that of the valuation alone.

4.3.2 Calculation of a Big Geometric Sum

We consider the evaluation of $y = \sum_{i=0}^{N-1} x^i$ and the calculation of $\frac{\partial y}{\partial x}$. Here x is a sample vector of 100000 `double` floating point numbers and $N = 5000$. In this calculation we have the repetitive composition of the operators $x_{2i} := f_{2i}(x) := x^i$ and $x_{2i+1} := f_{2i+1}(x_{2i-1}, x_{2i}) := x_{2i-1} + x_{2i}$ with $x_{-1} := 0$ and $y = x_{2N-1}$. Note that storing all $2N$ intermediate results would require $2 \cdot 5000 \cdot 100000 \cdot 8$ bytes $= 7629$ MB $= 7.5$ GB of memory.

Big Geometric Sum: $y = f(x) = \sum_{i=0}^{N-1} x^i$			
algorithm	evaluation	derivative	memory usage
finite difference	99.68 s	188.18 s	1.96 MB
pathwise AAD	118.53 s	140.81 s	7632.67 MB
stochastic AAD (full)	114.08 s	123.36 s	7632.83 MB
efficient stochastic AAD	94.55 s	97.10 s	3.36 MB

Table 3: Calculation times and memory usages for various backward algorithmic differentiation algorithms.

4.3.3 Calculation of a Sum of Products

We consider the calculation of $y = f(x_0, \dots, x_{n-1}) = \sum_{j=0}^{M-1} \sum_{i=0}^{n-1} c_i x_i$ with $M = 10$ and $n = 100$, where x_i and c_i are random variables represented by vectors of 1000000 `double` floating point numbers. We calculate all $n = 100$ partial derivatives $\frac{\partial y}{\partial x_i}$.

We use two different implementations: For the results in Table 4 we used the elementary operators $(a, b) \mapsto \mathrm{add}(a, b) = a + b$ and $(a, b) \mapsto \mathrm{mult}(a, b) = a \cdot b$. For the results in Table 5 we used the single operator $(a, b, c) \mapsto \mathrm{addProduct}(a, b, c) = a + b \cdot c$.

In this example, the backward automatic differentiation represents a huge improvement compared to the classical finite difference, since 100 partial derivatives

Sum of Products: $y = f(x_0, \ldots, x_{n-1}) = \sum_{j=0}^{M-1} \sum_{i=0}^{n-1} c_i x_i$			
algorithm	evaluation	derivative	memory usage
finite difference	9.86 s	1096.18 s	770.57 MB
pathwise AAD	7.83 s	21.82 s	8392.66 MB
stochastic AAD (full)	14.42 s	10.60 s	8392.59 MB
efficient stochastic AAD	10.57 s	5.69 s	770.74 MB

Table 4: Calculation times and memory usages for various backward algorithmic differentiation algorithms. The function is constructed from the operators `add` and `mult`.

Sum of Products: $y = f(x_0, \ldots, x_{n-1}) = \sum_{j=0}^{M-1} \sum_{i=0}^{n-1} c_i x_i$			
algorithm	evaluation	derivative	memory usage
finite difference	3.12 s	550.09 s	755.31 MB
pathwise AAD	2.74 s	6.68 s	4577.79 MB
stochastic AAD (full)	2.04 s	6.05 s	4577.76 MB
efficient stochastic AAD	3.53 s	6.54 s	770.70 MB

Table 5: Calculation times and memory usages for various backward algorithmic differentiation algorithms. The function is constructed from the operator `addProduct`.

have be calculated.

The efficient stochastic AAD achieves the result with the same low memory usage as the classic finite differences, since the algorithm only needs to keep references to the x_i's for a lazy-evaluation of the derivatives, recognising that c_i are constants.

In the non-optimised stochastic AAD the use of a single operator `addProduct` reduces the memory requirements compared to the use of elementary operators `add` and `mult`. This motivates the introduction of more complex operators which allows to further reduce the memory usage.

Note that this example can be interpreted as the result of an Euler-scheme discretization

$$z_{i+1} = z_i + \sigma_i \Delta W(t_i),$$

where $x_i = \sigma_i$, $c_i = \Delta W(t_i)$.

4.3.4 Calculation of a European-, Asian- and Bermudan-Option

We consider a simple, single-asset Black-Scholes model with a log-Euler-scheme discretization on a time-discretization $0 = t_0 < \ldots < t_n$:

$$S(t_{i+1}) \;=\; S(t_i) \exp\left((r - \frac{1}{2}\sigma^2)\Delta t_i + \sigma \Delta W(t_i) \right), \quad S(t_0) = S_0.$$

For the European option we calculate

$$V(T) \;=\; \frac{\max(S(T)-K,0)}{N(T)}$$

with $T = t_n$, $N(t) = \exp(rT)$. The objects S_0, r, σ, $S(t_i)$, $N(T)$, $\Delta W(t_i)$, $V(T)$ are random variable and we perform differentiation with respect to S_0, r, σ.

The model uses 250000 paths and 200 time steps. The Bermudan option has 50 exercise dates.

Monte-Carlo Valuation of European Option			
algorithm	evaluation	derivative	memory usage
finite difference	4.3 s	19.7 s	775.1 MB
efficient stochastic AAD	4.0 s	1.1 s	1166.2 MB

Table 6: Calculation times and memory usages for finite differences and backward algorithmic differentiation algorithms (caching the model primitives).

Monte-Carlo Valuation of Bermudan Option			
algorithm	evaluation	derivative	memory usage
finite difference	7.9 s	43.9 s	775.2 MB
efficient stochastic AAD	8.5 s	3.3 s	1631.9 MB

Table 7: Calculation times and memory usages for finite differences and backward algorithmic differentiation algorithms (caching the model primitives).

5 Conclusion

In this note we presented the automatic differentiation for random variables.

We presented modifications of the automatic differentiation algorithms which allowed an efficient and easy handling of the expectation and conditional expectation operator. The methods hence allows to apply (backward) automatic differentiation without a direct differentiation of the conditional expectation estimator ([2, 7]). Hence, the methods simplifies and improves the application of (backward) automatic differentiation to prominent applications in mathematical finance, e.g., the valuation of Bermudan options or the calculation of xVA's.

Also, the methods now enables us to further improvements in the treatment of the conditional expectation operator. Given that the conditional expectation operator is approximated by a numerical method, e.g., a regression, we may use different approximations of the conditional expectation operator in the valuation and in the algorithmic differentiation. This can improve the accuracy of the differentiation. For example, it is possible to choose different regression basis functions.

In addition, the setup motivated an implementation design improving performance and memory requirements for automatic differentiation applied to Monte-Carlo simulations, where we could take advantage of the fact that some operators result in deterministic values (reducing the memory requirements). Working with reference to immutable objects representing the random variables allowed to further reduce the memory requirements. Since storage is only required for operator nodes, the framework can be extended by introducing more complex operators, further reducing performance cost and memory requirements.

Results were presented using the Java reference implementation [11].

A Proof of Theorem 1

In Section 2.3 we already illustrated why the modified backward automatic differentiation algorithm gives the desired result by considering an operator f_k being an expectation, conditional expectation or other self-adjoint operator. While it may be clear that the argument given in Section 2.3 serves as an induction step, such that the result holds for situations with arbitrary many operators being, we give the complete induction argument here.

Proof. We proof Theorem 1 via induction over the number of operators in $\mathcal{G}$. For the induction start note that for $\mathcal{G} = \emptyset$ we have (almost) a classical backward automatic differentiation:

Let U denote an arbitrary random variable, then for random variables y and x we have that $\frac{\partial y}{\partial x}U$ is a random variable representing the directional derivative in direction U. For the random variable $z = E(y)$ we have that $\frac{\partial z}{\partial x}U = \mathrm{E}\left(\frac{\partial y}{\partial x}U\right)$. The random variable $\mathrm{E}\left(\frac{\partial y}{\partial x}U\right)$ is constant on every path and taking $U = \mathbf{1}_\omega$ we can identify $\frac{\partial z}{\partial x}$ with a random variable.[14]

From the chain rule, we have that the backward application of the chain rule can be identified with adjoint application of the partial derivative (for matrices this is left multiplication). For the path-wise operators f_m the operator $\frac{\partial f_m}{\partial x_k}$ is self-adjoint (as a matrix, left and right multiplication agree) and operator concatenation agrees with element wise multiplication of the representation random variables $D_{m,k}$. In this case we shortly write $\frac{\partial f_m}{\partial x_k} = D_{m,k}$ as a short notation for $\forall\, U \;:\; \mathrm{E}\left(\frac{\partial f_m}{\partial x_k}U\right) = \mathrm{E}\left(D_{m,k} \cdot U\right)$.

Letting $D_k^{\mathcal{G}} = D_k^{\emptyset} = D_k$ with $D_N = 1$ we find for all random variable U

$$\mathrm{E}\left(\frac{\partial y}{\partial x_i}U\right) \;=\; \mathrm{E}\left(D_i U\right) \;=\; \mathrm{E}\left(D_i^{\mathcal{G}} U\right).$$

Taking $U = \mathbf{1}_\omega$ this gives the result for the case $\mathcal{G} = \emptyset$:

$$\mathrm{E}\left(\frac{\partial y}{\partial x_i}\right) \;=\; D_i^{\mathcal{G}}.$$

To prove the induction step we introduce the following notation: Let

$$D_{m,k} \;:=\; \sum_{\vec{q} \in I_{m,k+1}} \prod_{j=1}^{|\vec{q}|} D^1_{q_j, q_j - 1}$$

where

$$\begin{aligned} I_{m,k} \;&:=\; \left\{(q_1, \ldots, q_p) \,\middle|\, p \in \mathbb{N}, m = q_1 > \ldots > q_p = k\right\}, \\ |\vec{q}| \;&=\; |(q_1, \ldots, q_p)| \;:=\; p \end{aligned}$$

[14] It is this random variable, which is propagated by the algorithm in Theorem 1 and the initial value $\frac{\partial z}{\partial x}$ is the the unit random variable 1.

and

$$D^1_{k,l} := \begin{cases} 0 & \text{if } l \neq i^{(k)}_j \text{ for some } j \\ \frac{\partial f_k}{\partial x_l} & \text{if } l = i^{(k)}_j \text{ for some } j \text{ and } f_k \notin \mathcal{G} \\ f_k & \text{if } l = i^{(k)}_j \text{ for some } j \text{ and } f_k \in \mathcal{G}. \end{cases}$$

Then the chain rule, together with

$$\frac{\partial}{\partial x_j} G\left(f_m\right) = G\left(\frac{\partial f_m}{\partial x_j}\right) \quad \text{for all } G \in \mathcal{G}$$

gives

$$\frac{\partial f_m}{\partial x_k} \;=\; D_{m,k}.$$

Let r be the smallest index with $f_r \in \mathcal{G}$. Set $\mathcal{H} := \mathcal{G} \setminus \{f_r\}$. Let U be an arbitrary random variable. As induction assumption we assume

$$\mathrm{E}\left(\frac{\partial y}{\partial x_k} U\right) \;=\; \mathrm{E}\left(D^{\mathcal{H}}_k \cdot U\right)$$

and show

$$\mathrm{E}\left(\frac{\partial y}{\partial x_k} U\right) \;=\; \mathrm{E}\left(D^{\mathcal{G}}_k \cdot U\right).$$

Let $G := f_r$, then we can factor

$$D_k \;=\; D_{N,k} \;=\; D^{\not r}_{N,k} + D_{N,r+1} \cdot G(D_{r-1,k}), \tag{7}$$

where $D^{\not r}_{N,k}$ is defined like $D_{N,k}$ with $D^1_{r,r-1}$ set to zero (i.e., vectors $\vec{q}$ containing a component r are removed from the summation), i.e., $D^{\not r}_{m,k} = \frac{\partial f_m}{\partial x_k}\big|_{x_r}$. Likewise let $D^{\mathcal{H},\not r}_k$ be defined like $D^{\mathcal{H}}_k$ where $\frac{\partial f_r}{\partial x_{i^r_1}}$ is set to 0. Note that from the induction assumption we also have

$$\forall\, U \;:\qquad \mathrm{E}\left(D^{\not r}_{N,k} U\right) = \mathrm{E}\left(\frac{\partial y}{\partial x_k}\big|_{x_r} U\right) = \mathrm{E}\left(D^{\mathcal{H},\not r}_k U\right),$$

since we only make assumption on the number of operators being in $\mathcal{H}$.

Now, applying expectation to (7) we have with $G = f_r = D_{r,r-1}$ for any random variable U

$$\begin{aligned} \mathrm{E}\left(\frac{\partial y}{\partial x_k} \cdot U\right) \;&=\; \mathrm{E}\left(D_{N,k} \cdot U\right) \;=\; \mathrm{E}\left(D^{\not r}_{N,k} \cdot U + D_{N,r+1} \cdot G(D_{r-1,k} \cdot U)\right) \\ &=\; \mathrm{E}\left(D^{\not r}_k \cdot U\right) + \mathrm{E}\left(D_{N,r+1} \cdot G(D_{r-1,k} \cdot U)\right) \end{aligned}$$

and with the assumption $\mathrm{E}\left(A \cdot G(B)\right) \;=\; \mathrm{E}\left(G^*(A) \cdot B\right)$

$$=\; \mathrm{E}\left(D^{\not r}_k \cdot U\right) + \mathrm{E}\left(G^*(D_{N,r+1}) \cdot D_{r-1,k} \cdot U\right)$$

and from the induction assumption

$$
\begin{aligned}
&= \mathrm{E}\left(D_k^{\mathcal{H},\eta'} \cdot U\right) + \mathrm{E}\left(G^*(D_{N,r+1}) \cdot D_{r-1,k} \cdot U\right) \\
&= \mathrm{E}\left(D_k^{\mathcal{H},\eta'} \cdot U + G^*(D_{N,r+1}) \cdot D_{r-1,k} \cdot U\right) \;=\; \mathrm{E}\left(D_k^{\mathcal{G}} \cdot U\right)
\end{aligned}
$$

□

References

[1] Jesper Andreasen. "CVA on an iPad Mini". In: *Global Derivatives, ICBI, Amsterdam* (2014).

[2] Alexandre Antonov. "Algorithmic Differentiation for Callable Exotics". In: *SSRN* (2017). DOI: 10.2139/ssrn.2839362. URL: http://ssrn.com/abstract=2839362.

[3] Alexandre Antonov, Serguei Issakov, and Andrew McClelland. "Efficient SIMM-MVA Calculations for Callable Exotics". In: *SSRN* (2017).

[4] Alexandre Antonov et al. "PV and XVA Greeks for Callable Exotics by Algorithmic Differentiation". In: *SSRN* (2017). DOI: 10.2139/ssrn.2881992. URL: http://ssrn.com/abstract=2881992.

[5] E. Benhamoud. "Optimal Malliavin weighting function for the computation of the Greeks". In: *Mathematical Finance* 13.1 (2003), pp. 37–53.

[6] Luca Capriotti and Mike Giles. "Algorithmic Differentiation: Adjoint Greeks Made Easy". In: *SSRN* (2011). DOI: 10.2139/ssrn.1801522. URL: http://ssrn.com/abstract=1801522.

[7] Luca Capriotti, Yupeng Jiang, and Andrea Macrina. "AAD and Least Squares Monte Carlo: Fast Bermudan-Style Options and XVA Greeks". In: *SSRN* (2016). DOI: 10.2139/ssrn.2842631. URL: http://ssrn.com/abstract=2842631.

[8] Jiun H. Chan and Mark S. Joshi. "Fast Monte-Carlo Greeks for financial product with discontinuous pay-offs". In: *Mathematical Finance* 23.3 (2013), pp. 459–495.

[9] Jiun H. Chan and Mark S. Joshi. "Minimal partial proxy simulation schemes for generic and robust Monte-Carlo Greeks". In: *Journal of Computational Finance* 15.2 (2011), pp. 77–109.

[10] Jiun H. Chan and Mark S. Joshi. "Optimal limit methods for computing sensitivities of discontinuous integrals including triggerable derivative securities". In: *IIE Transactions* 47.9 (2015), pp. 978–997. DOI: 10.1080/0740817X.2014.998390.

[11] finmath.net. *finmath-lib automatic differentiation extensions: Enabling finmath lib to utilise automatic differentiation algorithms (e.g. AAD)*. URL: http://finmath.net/finmath-lib-automaticdifferentiation-extensions.

[12] finmath.net. *finmath-lib: Mathematical Finance Library: Algorithms and methodologies related to mathematical finance*. URL: http://finmath.net/finmath-lib.

[13] Christian P. Fries. "Automatic Backward Differentiation for American Monte-Carlo Algorithms (Conditional Expectation)". In: *SSRN* (2017). DOI: 10.2139/ssrn.3000822. URL: http://ssrn.com/abstract=3000822.

[14] Christian P. Fries. "Fast Stochastic Forward Sensitivities in Monte-Carlo Simulations using Stochastic Automatic Differentiation (with Applications to Initial Margin Valuation Adjustments (MVA))". In: *SSRN* (2017). DOI: 10.2139/ssrn.3018165. URL: http://ssrn.com/abstract=3018165.

[15] Christian P. Fries. *Mathematical Finance. Theory, Modeling, Implementation*. John Wiley & Sons, 2007. DOI: 10.1002/9780470179789. URL: http://www.christian-fries.de/finmath/book.

[16] Christian P. Fries and Mark S.. Joshi. "Partial proxy simulation schemes for generic and robust Monte Carlo Greeks". In: *Journal of Computational Finance* 11.3 (2008), pp. 79–106.

[17] Christian P. Fries and Mark S.. Joshi. "Perturbation stable conditional analytic pricing scheme for auto-callable products". In: *International Journal of Theoretical and Applied Finance* 14.2 (2011), pp. 197–219.

[18] Christian P. Fries and Joerg Kampen. "Proxy simulation schemes for generic robust Monte Carlo sensitivities, process oriented importance sampling and high accuracy drift approximations". In: *Journal of Computational Finance* 10.2 (2007), pp. 97–128.

[19] Michael Giles and Paul Glasserman. "Smoking adjoints: fast Monte Carlo Greeks". In: *Risk* January 2006 (2006).

[20] Paul Glasserman. *Monte Carlo Methods in Financial Engineering. (Stochastic Modelling and Applied Probability)*. New York: Springer, 2003. ISBN: ISBN 0-387-00451-3. DOI: 10.1007/978-0-387-21617-1.

[21] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*. SIAM, 2008, p. 438. ISBN: 978-0-89871-659-7.

[22] Cristian Homescu. "Adjoints and Automatic (Algorithmic) Differentiation in Computational Finance". In: arXiv:1107.1831v1 (2011). DOI: 10.2139/ssrn.1828503. URL: https://arxiv.org/abs/1107.1831v1.

[23] Mark S. Joshi and Derminder S. Kainth. "Rapid computations of prices and deltas for n-th to default swaps in the Li Model". In: *Quantitative Finance* 4.3 (2004), pp. 266–275.

[24] Tim Peierls et al. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006, p. 424. ISBN: 978-0321349606.

[25] Vladimir Piterbarg. “Computing Deltas of callable LIBOR exotics in forward LIBOR models”. In: *Journal of Computational Finance* 7 (2004), pp. 107–144. DOI: 10.21314/jcf.2004.109.

[26] Vladimir Piterbarg. “TARNs: models, valuation, risk sensitivities”. In: *Wilmott Magazine* 14 (2004), pp. 62–71.

[27] Marius G.. Rott and Christian P. Fries. “Fast and robust Monte Carlo CDO sensitivities and their efficient object oriented implementation”. In: *SSRN* (2004). DOI: 10.2139/ssrn.73256. URL: http://ssrn.com/abstract=73256.

Notes

Suggested Citation

FRIES, CHRISTIAN P.: Stochastic Automatic Differentiation: Automatic Differentiation for Monte-Carlo Simulations. (June, 2017). https://ssrn.com/abstract=2995695 http://www.christian-fries.de/finmath/stochasticautodiff

Classification

Classification: MSC-class: 65C05 (Primary)
ACM-class: G.3; I.6.8.
JEL-class: C15, G13, C63.

Keywords: Automatic Differentiation, Adjoint Automatic Differentiation, Monte Carlo Simulation, American Monte Carlo, Conditional Expectation, Indicator Function, Object Oriented Implementation

45 pages. 3 figures. 7 tables.