Introduction

OPENING REMARKS

The main goal of quantitative research and derivatives libraries is not valuation but risk: what amounts of what assets we must trade to hedge the market risk or the regulatory adjustment of a derivatives portfolio. It is a major achievement of financial theory to have demonstrated that hedge ratios correspond to differential sensitivities. The goal is therefore to produce fast and accurate differentials of the valuation function. The determination and implementation of the valuation function is merely a step on the way, since, obviously, a function must be defined before it can be differentiated. We dealt with valuation in the previous part. This part is dedicated to differentiation.

Differentials are traditionally computed in finance by finite differences – or *bumping*: bump market inputs one by one and repeat valuation every time. This procedure may take unreasonable time, since valuation is repeated as many times as we have market inputs. *The complexity of the differentiation is linear in the number of differentials*. This number is typically large, several thousands for a derivatives book or the xVA of a netting set. It is not viable when the valuation is expensive, as is the case with regulatory calculations, and Monte-Carlo simulations in general. In recent years, the financial industry adopted the much more efficient algorithmic adjoint differentiation (AAD), which computes all sensitivities in constant time.

Despite being vastly superior and widely adopted, AAD is still largely misunderstood. This part explains AAD and its implementation, both

in a general context and for the differentiation of Monte-Carlo simulations.

As promised, the publication comes with a professional implementation in C++, which can be found in our online repository. The files AAD.cpp, AAD.h, AADExpr.h, AADNode.h, AADNumber.h, and AADTape.h form a self-contained implementation of AAD,[1] including advanced constructs like memory management and expression template acceleration. The code is explained and commented in detail in the following chapters.

AAD is arguably the strongest addition to finance of the past decade. It changed the way we design and develop derivatives systems and gave us the means to compute the full risk of the xVA on a large netting set within minutes on a laptop. Without AAD, such calculations had to be conducted overnight in data centers.

For example, the final tests of our implementation of simulations in Dupire's model from Chapter 7 took 1,801 inputs, 1,800 local volatilities and one spot, and returned the price of a 3y barrier option. With our parallel implementation, it took 350 ms with 500,000 paths and 156 steps with Sobol numbers. If we differentiated this calculation with finite differences, we would compute the sensitivity to the spot (delta) and the 1,080 active volatilities[2] (what Dupire calls “microbucket” vega) by repeating valuation 1,081 times, each time bumping one input by a small amount and keeping the other inputs constant. Differentiation time is *linear* in the number of inputs. It would take around 380s or six and a half minutes to complete.[3]

With AAD, we obtain the same results in *half a second*.

In the case of a large xVA, valuation may take minutes, even with all the theoretical, numerical, and computational improvements explained in [10], [80], [7], [30], [81], [31] and the optimized simulations of Part II. In addition to credit, xVA typically depends on many thou-

sands of market variables: all the underlying asset prices, as well as rate, spread, and other curves and all the implied volatility surfaces for all the currencies and assets in the netting set. The computation of the risk could take days.

With AAD, it takes minutes on a laptop, without loss of accuracy. On the contrary, if anything, AAD differentials are *more* accurate. AAD achieves this by computing all sensitivities analytically and *in constant time*. That differentiation is analytic is often put forward, but it makes little difference in practice. AAD is all about the constant time. In theory, the constant time is about that of three evaluations. Empirically, the number varies, depending on the valuation algorithm, the quality of its instrumentation, and the performance of the AAD library. Such speed is unprecedented since FDM. Its impact on the derivatives industry is substantial, profound, and irreversible. In addition to the obvious application to risk sensitivities, AAD is applied, for example, to speed up calibrations or compute regulatory amounts like FRTB; see [82].

Like finite differences, AAD is noninvasive. This means that developers only write the calculation code, and the algorithm produces its differentials automatically. Automatic differentiation is usually presented in two flavors: forward and backward. Forward auto-differentiation computes differentials analytically but in linear time, like bumping, and usually marginally slower. It is therefore mostly an interesting curiosity without practical relevance in our field. We coded forward AD in production systems in the early 2010s and never used it. In this book, we skip it altogether and focus on the automatic *backward* auto-differentiation, which is the one that produces differentials in constant time.

AAD is also sometimes advocated as a means to compute *better* differentials: AAD differentials are analytic, hence, deemed more accurate. While this may be correct in a small number of specific situations, in most cases of practical relevance, differentials produced with AAD or

bumping are almost indiscernible.[4] In most cases, AAD produces similar differentials but it produces them faster. AAD therefore doesn't resolve all the problems with differentiation; it only resolves the problem of *speed*. AAD cannot, for instance, compute the differentials of discontinuous functions any better than bumping. In particular, AAD cannot differentiate code through control flow ("if this then that" statements). It cannot compute the sensitivities of discontinuous payoffs, like digitals or barriers. Evidently, such functions are not differentiable and AAD should not be expected to perform an impossible task. Bump risk over discontinuous functions is known to be extremely unstable. AAD risk over such functions ignores control flow. Both are wrong. Risk over discontinuous functions is not resolved with AAD; it is resolved by the approximation of those discontinuous functions by close continuous ones, a technique known as *smoothing*. We investigate smoothing in a general context in our talk [77] and our publication [11], where we abstract smoothing into *Fuzzy Logic* and produce general smoothing algorithms for scripted payoffs. For an enlightening presentation of digital and barrier smoothing with convolution kernels, see Bergomi's [78].

The origins of AAD can be traced back to 1964 [83]. Adjoint differentiation (AD), the main principle behind AAD, has been applied for a long time in the field of machine learning, under the name "backpropagation." "Backprop," as it is called now, contributed to the recent and spectacular success of deep learning models. It is the constant time calculation of the many derivatives of the loss functions that allows deep artificial neural networks to learn their parameters in a reasonable time. Machine learning, neural networks, and backpropagation are discussed in a vast amount of literature. We are partial to Bishop's excellent [65].

AAD was introduced to finance in 2006 by Giles and Glasserman's award-winning "Smoking Adjoints" [5], followed by pioneering work like Luca Capriotti's [84], [85], [86] or [87]. In the early 2010s, Danske Bank delivered, and publically demonstrated, a large-scale implemen-

tation throughout its production and regulatory systems. Today, AAD is widely considered the only acceptable alternative for the computation of risk sensitivities for xVA and other bank-wide regulatory amounts.

SUMMARY

AAD is the subject of a number of publications like [88], and the author was personally involved in lectures, professional training, workshops, and presentations following the implementation of AAD in Danske Bank's systems. The theory and mathematics of AAD are simple, yet people generally find it hard to wrap their minds around it at first. We hope our introductory Chapters 8 and 9 help with that.

A *basic* implementation is not hard, either. It is based on operator overloading and can be written in languages that support it, like C++.

A *professional* implementation must also provide efficient memory management; otherwise cache and memory load may defeat the benefits of AAD and result in code *slower* than bumping. Chapter 10 delivers an efficient, general-purpose AAD library, including memory management facilities.

Chapter 11 discusses the application and limits of AAD for the differentiation of arbitrary and financial calculation code.

Chapter 12 differentiates the parallel simulation library of Part II.

Chapter 13 introduces the fundamental notion of checkpointing, a crucial idiom for the practical implementation of AAD, and applies it to the problem of risk propagation from local to implied volatilities in the context of Dupire's model.

Chapter 14 extends the AAD library to differentiate not one, but multiple results of a calculation with respect to its inputs, and applies it to

produce an itemized risk report for a portfolio of financial products, in a time virtually constant in the number of risk reports. In reality, AAD is always linear in the dimension of the differentiated result. What we achieve is a small marginal time per additional result in situations of practical relevance, hence the title of the chapter: "multiple differentiation in *almost* constant time."

Chapter 15 introduces a different breed of AAD implementation based on *expression templates*, something initially proposed in [89]. Expression templates may accelerate the production of AAD differentials by a significant factor, depending on the differentiated code, and approach the performance of the manual adjoint differentiation covered in Chapter 8. The differentiation of our simulation library is around twice as fast with expression templates.

A FIRST APPROACH TO ADJOINT DIFFERENTIATION

For a first introduction to adjoint differentiation (AD), we borrow Huge's analogy [90] to the more familiar field of matrix algebra. Consider a calculation represented by the function $F : \mathbb{R}^n \to \mathbb{R}$ that produces a scalar result z out of an input X in dimension n. Assume F may be broken down into a sequence of sub-calculations $G : \mathbb{R}^n \to \mathbb{R}^m$, $K : \mathbb{R}^m \to \mathbb{R}^p$ and $H : \mathbb{R}^p \to \mathbb{R}$, such that:

$$F(X) = H\{K[G(X)]\}$$

where G, K, and H are simple enough that their Jacobians are known analytically and computed without difficulty.

All calculations can be broken down in this manner, although, in general, they consist in not three, but thousands or millions of elementary calculations with derivatives known in closed form. In programming languages with support for *operator overloading*, all calculations can be *automatically* decomposed into elementary operations: additions, multiplications, logarithms, square roots, and so forth, which deriva-

tives are obviously known. Operator overloading allows developers to overload what happens when operators like $+$ or $*$ or functions like $log()$ or $sqrt()$ are invoked. With AAD, operators are overloaded to *record* the sequence of operations.

To evaluate F means evaluate G, K, and H:

$$X \in \mathbb{R}^n \xrightarrow{G} Y \in \mathbb{R}^m \xrightarrow{K} Z \in \mathbb{R}^p \xrightarrow{H} z \in \mathbb{R}$$

in the sequence:

$$\begin{aligned} 1 \quad Y_m &= G(X_n) \\ 2 \quad Z_p &= K(Y_m) \\ 3 \quad z &= H(Z_p) \end{aligned}$$

where lowercase letters are scalars and capital letters are vectors with dimension in subscript. It follows immediately from the chain rule that the Jacobian matrices of F, G, K, and H are related by matrix algebra:

$$\left(\frac{\partial F}{\partial X}\right)^t_{n\times 1} = \left(\frac{\partial G}{\partial X}\right)^t_{n\times m} \left(\frac{\partial K}{\partial Y}\right)^t_{m\times p} \left(\frac{\partial H}{\partial Z}\right)^t_{p\times 1}$$

where the subscripts denote the dimensions of the Jacobian matrices. It is our working assumption that G, K, and H are simple enough that their Jacobians are known and explicit. To find the Jacobian of F, conventional algorithms multiply the Jacobian matrices, in the order of evaluation, left to right:

$$\left(\frac{\partial F}{\partial X}\right)^t_{n\times 1} = \left[\left(\frac{\partial G}{\partial X}\right)^t_{n\times m} \left(\frac{\partial K}{\partial Y}\right)^t_{m\times p}\right]_{n\times p} \left(\frac{\partial H}{\partial Z}\right)^t_{p\times 1}$$

The complexity of the matrix product between brackets is nmp, and its product by the right-hand side vector has complexity np. The complexity of the differentiation is therefore $(m+1)np$.

Because matrix products are associative, we can also accumulate the differentials of F in the *reverse* order:

$$\left(\frac{\partial F}{\partial X}\right)^t_{n\times 1} = \left(\frac{\partial G}{\partial X}\right)^t_{n\times m}\left[\left(\frac{\partial K}{\partial Y}\right)^t_{m\times p}\left(\frac{\partial H}{\partial Z}\right)^t_{p\times 1}\right]_{m\times 1}$$

Now, the right-hand-side term between brackets has complexity mp and results in a vector of dimension m, which product by the left-hand-side matrix is of complexity nm. The complexity of the differentiation of F is therefore $m(n+p)$, one order of magnitude lower.

Differentials accumulate one magnitude faster in reverse order.

It must be so, because F is a scalar function. For this reason, the rightmost Jacobian of the decomposition must be a vector. When we multiply Jacobians right to left, the successive results always collapse into a vector, so all the matrix products are of quadratic complexity instead of cubic.

What traditional differentiation does is expand the *Jacobians* to
from the innermost function to the outermost function :

where it is clear that the differentiation occurs in the order , ,
of the evaluation, or *forward*.

Reverse, or *backward*, differentiation propagates the *adjoints*, the derivatives of to , , and , in this (reverse) order. It starts with the adjoint of :

Next, it computes the adjoint of :

Finally, it produces the adjoint of :

So Jacobians expand forwards, whereas adjoints propagate backwards. The adjoints of scalar calculations accumulate one order of magnitude faster than their Jacobians. To compute differentials in this manner is called *adjoint differentiation* or AD. When adjoints accumulate over a sequence of operations *automatically* extracted from a calculation, it is called *automatic adjoint differentiation* (AAD).

AAD reduces differentiation complexity by an order of magnitude, but only for scalar functions. If (and) were multidimensional functions valued in , AD complexity would be linear in , and slower than conventional differentiation when .

In addition, to compute the Jacobians and , and must be known. So not only the entire sequence of operations in a calculation must be stored in memory, but also all the intermediate results. It is not a problem with three functions, but real-life calculations are decomposed into thousands or millions of elementary operations, imposing a heavy load on RAM and cache.

The practical performance is therefore entirely dependent on how fast operations and results are recorded. AAD is more dependent on implementation than most algorithms. A naive implementation could result in slow differentiation and RAM exhaustion. Efficient memory management is crucial.

The differentiation of multidimensional functions is covered in Chapter 14. Memory management is discussed in deep detail in Chapter 10, where we manage to record operations and results with very little overhead. The techniques of Chapter 13 split differentiations to alleviate RAM and cache footprint. It is, however, impossible to completely eliminate recording overhead, and the number of operations in a calculation is typically large. The techniques of Chapter 15 reduce the number of records by recording entire *expressions* in place of elementary operations.

The next chapter explains adjoint differentiation from a different angle, where it is shown that AD finds the common factors in the expression of differentials, reducing the complexity of their computation. Chapter 9 explains in detail how calculations are automatically split and their operations recorded to accumulate adjoints in the reverse order. Chapters 10 and following deal with the many aspects of a practical implementation.

NOTES

1 With a dependency on gaussians.h for the analytic differentiation of Gaussian functions and blocklist.h for memory management.

2 We have 1,800 local volatilities, out of which 1,080 are between now and 3y, hence active for the 3y barrier.

3 In general, the goal of derivatives risk systems is to compute sensitivities to hedge instruments, not model parameters. Model parameters are computed from hedge instruments, generally by calibration. Sensitivities to hedge instruments can be produced in two ways: directly, by bumping the value of hedge instruments one by one and repeating the whole valuation process, including calibration; or, by computing sensitivities to model parameters first, and using a variety of techniques explained in Chapter 13 to turn them into sensitivities to hedge instruments. The second option is a much better one: it avoids a potentially unstable differentiation through calibration, provides valuable information in terms of sensitivity to model parameters, and offers the possibility to choose different instruments for calibration and risk. But, with a large number of model parameters, it may be prohibitively expensive with bumping; therefore, many risk systems based on bumping implement the first option. We will see that AAD offers a faster alternative to bumping, one that is constant in the number of sensitivities. Hence, an additional benefit of AAD is to permit the implementation of the better option for the production of risk without additional cost (actually, orders of magnitude faster). With AAD, it is best practice to compute sensitivities to model parameters first, and turn them into sensitivities to hedge instruments next. This is exactly the procedure we follow:

Chapter 12 computes sensitivities to model parameters and Chapter 13 explains how to turn them into sensitivities to hedge instruments.

4 Although we will discuss an important counter-example in Chapter 13, where AAD produces a more stable “superbucket” risk.