ALGORITHMIC DIFFERENTIATION IN FINANCE EXPLAINED

Marc Henrard

FINANCIAL ENGINEERING EXPLAINED

Financial Engineering Explained

Series Editor
Wim Schoutens
Department of Mathematics
Katholieke Universiteit Leuven,
Heverlee, Belgium

Financial Engineering Explained is a series of concise, practical guides to modern finance, focusing on key, technical areas of risk management and asset pricing. Written for practitioners, researchers and students, the series discusses a range of topics in a non-mathematical but highly intuitive way. Each self-contained volume is dedicated to a specific topic and offers a thorough introduction with all the necessary depth, but without too much technical ballast. Where applicable, theory is illustrated with real world examples, with special attention to the numerical implementation.

Titles in the series:
Equity Derivatives Explained, Mohamed Bouzoubaa
The Greeks and Hedging Explained, Peter Leoni
Smile Pricing Explained, Peter Austing
Financial Engineering with Copulas Explained, Matthias Scherer and Jan-Frederik Mai
Interest Rates Explained Volume 1, Jörg Kienitz

Forthcoming titles:
Interest Rates Explained Volume 2, Jörg Kienitz
Submissions: Wim Schoutens - wim@schoutens.be

More information about this series at
http://www.springer.com/series/14984

Marc Henrard

Algorithmic Differentiation in Finance Explained

palgrave
macmillan

Marc Henrard
Advisory Partner,
Open Gamma
London, United Kingdom

ISBN 978-3-319-53978-2 ISBN 978-3-319-53979-9(eBook)
DOI 10.1007/978-3-319-53979-9

Library of Congress Control Number: 2017941212

Printed on acid-free paper

This Palgrave Macmillan imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Contents

List of Figures

List of Tables

Preface

The book's title originally proposed to the editor was "Forty-nine shades of Algorithmic Differentiation"; a title that he duly refused as not fitting the series guideline very well and potentially distracting the target audience. The proposed title was not only a marketing exercise; there may not be as many as forty-nine variations of Algorithmic Differentiation (AD) presented in the book, nevertheless it is one of the goals of the book to present as many variations, shades or faces of the subject as possible, and not a one-size-fit-all approach. While exaggerating the number of shades, why not going to fifty? I'm expecting the readers to implement their own version of the technique, so for each reader it will be "*fifty shades of AD*." I also removed one from the number in the title to avoid confusion between AD and grey; AD is a very colorful subject!

Algorithmic Differentiation (AD) has been used in engineering and computer science for a long time. The term *Algorithmic Differentiation* can be explained as "the art of calculating the differentiation of functions with a computer."

When I first heard about it, it sounded to me like some kind of dark art or black magic where you get plenty of results from your computer for (almost) free. As a beginner in finance I was told that "there is no free lunch." Certainly this technique could not be applied to finance.

I nevertheless scrutinized the technique carefully, having heard about it several times from serious quants in serious conferences. What I discovered is that at the same time, it is not magic – it is mathematics –, it is not free – you have to invest in development time – and it appears like magic and (almost) free at run time.

Six years later, I have invested a lot of time understanding the technique and developing quantitative finance libraries which implement AD. Now I ask myself: "Why is not everybody in finance using AD?" I don't have an answer to this last question! But I'm very biased in this. I'm like a newly converted faithful to a religion, I don't understand how I was not a believer in the past and I don't understand how others don't believe.

It took me a while to write this book. Between my first article on the subject in 2010 and this book, more than 6 years have passed.

Why it took me so long?

> Chacun sa méthode … Moi, je travaille en dormant et la solution de tous les problèmes, je la trouve en rêvant. (Personal translation: Each his own method … Myself, I work sleeping and the solution to all problems, I find it dreaming).
>
> *Drôle de drame (1937) – Marcel Carné*

This means a lot of working nights to dream all those pages.

Obviously I cannot write something without referring to my previous book: *Interest Rate Modelling in the Multi-curve Framework: Foundations, Evolution, and Implementation* Henrard (2014). Most of the examples in this book are related to interest rate modeling and the multi-curve framework.

In my previous book I tried to follow the steps of giants in the art of relevant and irrelevant quotes. For this book, I changed the style. Each chapter starts with a couple of sentences summarizing some of the chapter's discoveries, like in old detective fictions. Those of the readers who know me personally, may have notice my zealous collection of (paper) books and the personal library to collect them. That collection contains a lot of detective fictions.

This book grew from seminars, lectures and training I presented on Algorithmic Differentiation. The first of those seminars was probably a "pizza seminar" at OpenGamma in March 2012. Pizza have disappeared from OpenGamma diet since, but seminars are still going on … Training are also continuing, at conference workshops or in-house in some banks, and I see more and more interest in the AD subject. Professional finance magazines, like *Risk* and *Structured Product*, have proposed articles on the subject – see for example Sherif (2015) and Davidson (2015) – and interviewed me for them.

This book has benefitted of the valuable feedback from numerous people in the financial "quant" community, among them Wim Schoetens, Luca Capriotti, Yupeng Jiang, and Andrea Macrina

In the book I use the term "we" with the general and vague meaning of the author, the finance community and the readers. The terms "I," "me" or "my" are used with the precise meaning of *the author personally with its opinions and biases*. The term "I" should be used as a warning sign that the sentence contains opinions and maybe not only facts. You have been warned!

The book was written astride 2014 and 2015. Among my resolutions for 2015 was to be more clear and direct in expressing my opinions. You may find some vapor of that resolution in the book. Some opinions may appear strong or lacking nuance. It is for clarity sake, not to be rude to others opinions. Even if I have spent some time in England, I have not learned the art of *understatement* yet!

Enjoy!

Marc Henrard
London, Brussels and Wépion – September 2016

> *Supplied with cleverness of every imaginable type,*
> *Man ventures once towards evil, and then towards good.*
>
> *Sophocles, Chorus in Antigone*

Use Algorithmic Differentiation for the good!

Bibliography

Davidson, C. (2015). Structured products desks join the AAD revolution. *Structured Products*, 11(7):14–18.

Henrard, M. (2014). *Interest Rate Modelling in the Multi-curve Framework: Foundations, Evolution and Implementation*. Applied Quantitative Finance. London:Palgrave Macmillan. ISBN: 978-1-137-37465-3.

Sherif, N. (2015). Chips off the menu. *Risk Magazine*, 27(1):12–17.

La lecture des bons livres est comme une conversation avec les plus honnêtes gens des siècles passés qui en ont été les auteurs, et même une conversation étudiée, en laquelle ils ne nous découvrent que les meilleures de leurs pensées.

René Descartes
Discours de la méthode (1637)

Chapter 1
Introduction

Where the characters are introduced – Where the non-characters are released from duty – Where the adjoint come on stage – The code to the code.

1.1 Differentiation in Finance

In quantitative finance, computing the price of financial instruments is only part of the game; most of the (computer) time is spent in calculating the first order derivatives of the price with respect to the different inputs, the so-called deltas or *greeks*. This does not include only the simple greeks in a Black-Scholes formula, like Delta, Vega or Rho but more importantly, and more costly in computation time, the bucketed deltas to interest rate or credit curves and vega to interpolated volatility surfaces or cubes. The bucketed delta is usually more computationally costly as the multiple curves in a multi-curve framework are often made of 20–100 points and there are 20–100 deltas to compute.

The computation of function derivatives is not restricted to finance. The problem has important application in engineering in general. The Algorithmic Differentiation (AD) techniques have been used in different fields for a long time. The techniques comes in two modes: *standard*, *forward* or *tangent* and *adjoint* or *backward*. Both modes have their advantages and their drawbacks. The most useful one in finance is the adjoint mode; the reason for this is explained below.

In the computation of derivatives, two aspects have to be taken into account: precision and speed. The AD is an answer to both concerns; it is the proverbial stone with which two birds can be killed. With AD, the results are computed to machine precision level; the only approximations are the ones of processing doubles, there is no theoretical approximation. The speed is in general very good, with *all* derivatives and the function value computed in a time equivalent to the time required to compute 3–4 function values. This time is for *all* the derivatives of the function as one block, even if there are hundreds of them.

The algorithmic differentiation-like methods were popularized in finance by Giles and Glasserman (2006). The method used was not pure Algorithmic Differentiation method, and they are called in some places algebraic adjoint approaches. The method was applied to Monte Carlo computation for the Libor Market Model; those methods are usually computational time intensive. The method was extended to Bermudan style derivatives by Leclerc et al. (1999) and further extended in

M. Henrard, *Algorithmic Differentiation in Finance Explained*,
Financial Engineering Explained, DOI 10.1007/978-3-319-53979-9_1

Denson and Joshi (2009a) and Denson and Joshi (2009b). Similar methods are used for Monte Carlo-based calibration in Kaebe et al. (2009).

The first uses of fully fledged Algorithmic Differentiation in finance, include the application to correlation risk in credit models in Capriotti and Giles (2010), to Monte Carlo credit risk in Capriotti et al. (2011), to gamma of CDO tranches in Joshi and Yang (2010). In two different papers, Capriotti (2011) and Homescu (2011) describe the AD general technique and its uses in finance. The review paper Capriotti and Giles (2012) describes several of those applications. More recent results also include application to second order greeks in Capriotti (2015), to PDE in Capriotti et al. (2015) and to least square Monte Carlo in Capriotti et al. (2016). The acronym "AAD" to describes Adjoint Algorithmic Differentiation was apparently first used in Capriotti (2011) and Capriotti and Giles (2012).

This book does not present all the computer science theoretical results regarding the method. For the theoretical results, we refer to Griewank and Walther (2008) and Naumann and du Toit (2014). Those references go a lot deeper on the computer science theoretical aspects of the technique.

In particular in this book, we will not prove theoretical results regarding the efficiency of the method. We will show the efficiency through practical examples and implementations in quantitative finance. The goal is a pragmatic implementation, from very simple cash flow discounting to model calibration, with some stops at Black-Scholes formula, SABR smile and multi-curve calibration.

The direct application of AD in finance is to the computation of first order risks. For that reason I will use the name of *Risk Manager* for the business end users of the libraries developed. It does not mean that the title Risk Manager is printed on their business card, but that they have an interest in the risks of the financial instruments. This category includes traders, quantitative analysts, risk managers, portfolio managers, model validators, senior management, supervisors, etc. With that definition in mind, the trader is probably the risk manager which has the most interest in AD as he probably requires speed more than others. On the other side I will call *developers* or computer scientists the people that write the code in the libraries. But again, their business card can indicate a completely different title, like quantitative analyst, model validator, risk manager, developer, architect, etc.

1.2 Standard and Adjoint: A Starter

AD comes in two modes: *standard*, *forward* or *tangent* and *adjoint* or *backward*. Those modes refer to the way the derivatives are computed.

The starting point of our presentation is always that the algorithm to compute the value of the function has been developed and implemented in some code. From there we want to write efficient ways to compute the derivatives of the value function with respect to the inputs. The design of the initial algorithm for the value itself is not discussed in this book.

The *standard* mode starts from the inputs and computes the derivatives with respect to the inputs going through the code in the same order as the value algorithm

to finish with the derivatives of the output with respect to the inputs. This seems natural and would probably be the first implementation one would think of. If we first apply function g and then function f to the inputs a to get the output $z = f(g(a))$, the first approach to derivative computation is to compute $Dg(a)$, then $Df(g(a))$ and combine[1] them to get $D(f \circ g)(a)$. This approach put the emphasis on a: we want to compute the derivatives of everything with respect to a.

The other way to look at this problem is to view the problem as computing the derivative of the output. The output is z; obviously we known the derivative of z with respect to z, it is 1. Let go back one step and look at the previous operation, which is f. The function f is applied to an intermediary result, let's call it b for the moment. We have $z = f(b)$. We can compute the derivative of z with respect to b easily, there is only one step: $Dz = Df(b)$. But b itself is the result of a previous operation: $b = g(a)$. We are not finished yet, we still have to compose with $Dg(a)$ to obtain $Dz = Df(b).Dg(a)$.

Obviously the two approaches provide the same answer with the same computation complexity in the very simple example above with only one input and only one output. But what happens with multi-dimensional inputs and outputs? If you have one input and several outputs, the forward mode is easy: compute the derivative with respect to the unique input and for each line of code in the valuation algorithm add a line of code with the derivative. Several output for the value leads to several outputs for the derivatives, but the number of line of code is one to one between value and derivatives.

What if we have several inputs and one output? At each line of code, as we progress forward through the code, we have to add *several* line of code, one for each input. The computation time of the derivative algorithm will be of the order of magnitude of the number of inputs multiplied by the computation time of the value. In term of computational complexity it does not seems different from a one-side finite difference approach – the finite difference approach is presented in Section 1.5 –, there is one extra computation required for each input.

What is the more likely situation in finance? The standard computation is one present value (for each instrument) with the value depending of numerous inputs: interest rate curves, credit spreads, forex rates, etc. the usual problem is a "numerous inputs and one output" type of question. The usual finance case does not seems to be improved, from a computation time perspective, with respect to one-side finite difference by the use of forward AD.

In the case of the adjoint version, we start from the end, i.e. from the one dimensional present value output and at each step going backward, we ask what is the derivative of the (unique) output with respect to the intermediary value we are computing at this line. So for each line of code in the value algorithm, there is one new line of code in the derivatives algorithm. The code may be slightly more complex, but in term of order of magnitude it is the same for the value and all the derivatives.

[1] The mathematics required for the theoretical part of AD are described in Chapter 2. The notation used are described in Appendix A.

Now, this is a real game changer in finance. If we can compute the derivatives, all the derivatives – and this can be as many as 100 or 1,000 derivatives – in roughly the same time as the value, we are saving a huge computation time. Suppose we are computing the present value of a portfolio with 100 inputs (interest rate curves and forex) in one minute; if we want all the 100 sensitivities, just give us two minutes more. That is different from the standard answer: the portfolio present value is computed in one minute, if you want all the sensitivities, please come back in one hour. Risk management is not any more a lot more expansive in CPU term than simple present value computation.

The explanation above is an introduction to why we focus to the adjoint version of the algorithmic differentiation in this book. More details about the standard and adjoint modes will be given in the next chapter called "The principles of Algorithmic Differentiation."

1.3 Why Exact Derivative?

The goal of Algorithmic Differentiation is to compute the derivatives of a function, in the mathematical sense of the term, as precisely and fast as possible using a computer. A remark I have often heard in this context is the following: "I don't want to compute the derivative, I want the bump and recompute number with precisely one basis point shift to take convexity into account." Even if I understand the origin of the remark, I disagree strongly that it makes sense and it is relevant.

I understand that the exact derivatives multiplied by one basis point is not the same as an actual movement of one basis point. The derivative is the slope in the first order development and is just that, it is the precise slope of the first order development. If you want the exact value of function, the derivative will provide an approximation, an indication of the true result. Obviously if you know that the market will move by one basis point and exactly one basis point you don't want to compute the exact derivative, you want to compute the exact profit. But if you know that the market will move by exactly plus one basis point, what is the point of risk management? You want to compute the profit, fine, but not the sensitivity to the unknown movement of a quantity for which you just pretended to know its actual movement. Is the one basis point bump and recompute computation better than the exact derivative? Yes it is, when you know that the market will move by exactly plus one basis point. If the movement is minus one basis point, the result will be worst. So if you know that the movement is exactly plus one basis point, you have an easy way to get rich. Your problem is not to know what your risk is, exactly, it is "How rich I'm? Exactly!"

The same argument could be said for larger bumps where the argument is "one basis point does not really hurt my profit," only 5 basis point does, so I want a bump and recompute with 5 basis points. But 5 basis points in which direction? In the direction adverse to my portfolio. Ok, I see what you want now. Actually you have a number in mind; a currency amount that hurts you and you want to see if it can be reached by your position. What you are looking for is not sensitivity, it is VaR. Give

me you probability tolerance, and I will give you a VaR for your portfolio and you can compare it with your hurting figure.

In this context, I have heard similar remarks than the one above but with a twist, like "I don't want to compute the derivative, I want the symmetrical bump and recompute number with precisely five basis point shift." The five basis point is to take into account the expected absolute shift of the market – in the same sense as in expected shortfall –, the symmetrical is to take into account the uncertainty of the direction of the market or "to be more precise." The symmetrical approach will make it more precise in the sense it is closer to the *exact derivative* but not that it is more precise to estimate your potential loss or gain. So again the proposed methodology is not in line with the stated goal.

From my point of view, I have never heard a convincing business argument for not computing the exact derivatives but an approximated bump and recompute ones. Maybe there is one, but you will need to find someone else – or an older and wiser me – to tell you about it.

1.4 Code in This Book

In the book, in several places some code is used as examples of the different concepts described. The code examples are written in Java. The main reason why I have used Java is that it is the programming language which I'm using on a daily basis and for which I have written quantitative finance libraries which are open source, giving reader the access to full size production grade libraries that implement the ideas described in the book. There is no other fundamental reason for it. Some may even say that this is not a very good choice and a language like C++, with operator overload, would be better, specially for Automatic Algorithmic Differentiation (see Chapter 4). I selected a language that, to my opinion, make it easy to transmit the ideas I want to describe and in which some quantitative finance libraries are written. There is talk about introducing operator overloading in Java, but it will have to wait for Java 10 at best. If this happen, I may revisit my code.

Moreover the code proposed in the book should be viewed as pseudo-code. All the important ingredients to make it work are there, but some details, like punctuation, programming language syntax, types may be slightly off when it shortens the code or make it easier to read. There is certainly no implicit claim that the pseudo-code will even compile as provided. If this warning saddens you, you may want to glance at Section 1.7.

1.5 What Is Not in This Book: Finite Difference

The most used approximation in finance for the computation of first order derivatives of scalar functions are the *forward difference*

$$Df(a) = \frac{f(a+\epsilon) - f(a)}{\epsilon} + O(\epsilon).$$

and *backward difference*

$$Df(a) = \frac{f(a) - f(a - \epsilon)}{\epsilon} + O(\epsilon).$$

These two methods are also known as one-sided *finite difference* or *bump and recompute*.

In term of computational complexity, there is one computation for the initial value, plus one similar computation for the value at each shifted input. This means a complexity of two for a one dimensional function and complexity $1 + n$ for a n-inputs/1-output function.

Take a very simple function, like $f(x) = \exp(x+x^2)$ and look at the resulting graph in Fig. 1.1. The X-axis is in logarithmic scale to be able to see the impact of shifts with different order of magnitude. You can see in the center of the graph that the error between the difference ratio described above and the exact value of the derivative appears for shift larger than 10^{-4} but disappears for shifts smaller than 10^{-5}. For the larger shifts, the higher order terms of the function disturb the computation of the exact differentiation. In this case the higher orders have a significant second order part and the forward and backward finite difference errors have similar sizes but with different signs.

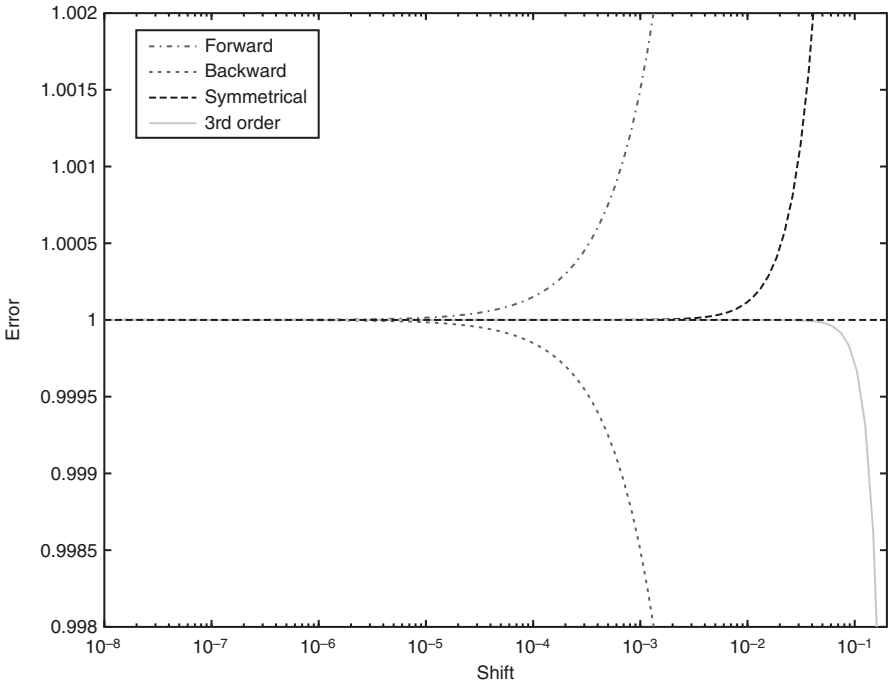

Fig. 1.1 Comparison of the convergence for different finite difference approximation of the first order derivative

To deal with higher order term impact, higher order schemes with better convergence properties can be used. A scheme of order 2 is given by the *symmetrical* (or *centered* or *two sided*) finite difference approximation

$$Df(a) = \frac{f(a+\epsilon) - f(a-\epsilon)}{2\epsilon} + O(\epsilon^2).$$

As can be seen from the same figure, with this approach the convergence happen for larger shifts, and for the example function it is for shifts below 10^{-2}. The computational cost of this approach is one computation for the initial value and two computations for the values with shifted inputs. The total complexity is $1 + 2 * n$ for an n-input function.

One can go one step further and use a scheme of *order four* given by

$$Df(a) = \frac{-f(a+2\epsilon) + 8f(a+\epsilon) - 8f(a-\epsilon) + f(a-2\epsilon)}{12\epsilon} + O(\epsilon^4).$$

As can be seen from the same figure, with this approach the convergence happen for even larger shifts, and for the example function it is for shifts below 10^{-1}. The computational cost of this approach is one computation time for the initial value and four computations for the values with shifted inputs. The total complexity is $1+4*n$ for a n-input function.

We have proposed above four different approaches to the computation of the derivatives of a function using finite difference. As you can guess from the title of the book, none of them is an approach we recommend through this book. Nevertheless it is useful to review them quickly in term of convergence.

Note also that in several places, the one sided finite difference approximation is confused with the real derivative. This is in particular the case in the recent policy framework on Fundamental Review of the Trading book by the Basel Committee on Banking Supervision (2014). Hopefully the Committee will correct this in the final version of the regulation following the advices in my answer to their request for comments. There is no reason to force the industry to use sub-optimal methods when better approaches are available. Central planning and regulators hampering improvements (and the sales of this book!) is unfortunately not unheard off.

The most talked about convergence in place describing the convergence of the approximation methods is below which level of shift ϵ will the approximation be good enough. This is the question we have analyzed above. The function used in the graph is $\exp(x + x^2)$. There is nothing special about this function. The exponential is often used in curve description and $\exp(x^2)$ is related to the normal distribution density which is the base of the diffusion models; the function used is not alien to finance.

From the graph, it is clear that the higher order approximations provide a better approximation of the derivative. A quick answer could be that the first two approximations are good below 10^{-4}, the symmetric one below 10^{-2} and the fourth order one below 10^{-1}. This represents a gain of two orders of magnitude in the shift for

the symmetrical approach and three orders for the higher order one. Those gains in precision have to be balanced with the computation cost. The first two approaches have a cost of one computation for each derivative, the symmetrical one has a cost of two and the fourth order one has a cost of four. This looks like a standard trade-off between quality and cost.

But the view of the question at this stage is very one-sided. We have looked at the "right half" of the graph: how large can the shift be and still provides a decent answer? We should look also at the left side of it: how small can the shift be and still provides a decent answer?

The formulas above and their order of convergence are true in theory, i.e. true for infinitely precise arithmetic. In practice, in computers, the precision is limited. Depending of the machine, language, and other details the precision will differ but will be finite in all cases. The limit process involved in the derivative computation by differential ratio is a limit of the style "0/0"; both the numerator and the denominator converge to 0. At some stage, for very small shifts, the figures computed will be indistinguishable by the computer from 0 in its finite precision. The only number left will be the quasi random last bits of the finite precision binary information. The computed ratio becomes of the style "random/random." No meaningful information is left in the process.

This can be seen in Fig. 1.2. The left side of the graph is the computed derivative approximation using the same approximation schemes and letting the ϵ becoming very small. At around 10^{-11} some noise appears and at 10^{-13}, there is more noise than information. This is the case for all the finite difference schemes, even for the higher order ones. The levels creating the problems are the same for all approaches; there is no advantage for the higher order schemes.

Looking at the graph, one may think that this is not a problem, that there is a very wide range of shift values for which all the schemes propose a good enough approximation, just pick one of them. In the above case any shift between 10^{-10} and 10^{-6} is good enough for all practical purposes.

This is certainly true here, with the hindsight of the graph and having run the process hundreds of times for different values. But the answer to the acceptable range question will change for each function; depending of the higher order derivative size, the right size of the range may change dramatically. Close to the money and close to expiry, options have a very high gamma and the above picture on the right would be quite different for them.

The function we used to draw the graph is implemented as an explicit function depending on very easy primitives (exponential and square). In a lot of cases the function we are interested in in finance will be implemented through some more complex numerical schemes. It can involve solving a root-finding problem (like curve calibration or swaption pricing in Hull-White model) or an heavy numerical scheme like numerical integration, PDE solving, trees or Monte Carlo. In that case the boundary between acceptable and unacceptable on the left part of the graph could be a lot more to the right. There is no guarantee that there is any space between the left side of the acceptable range and the right side of it. Even if there is such a range, there is no method a priori to guess where it is. If for each computation,

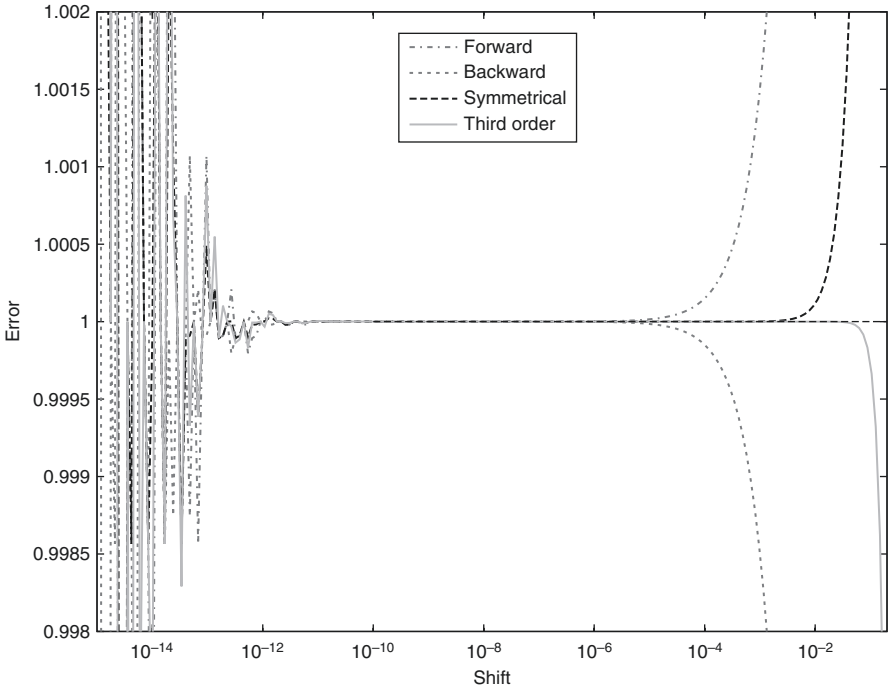

Fig. 1.2 Comparison of the convergence for different finite difference approximations of the first order derivative. Including the "very small shifts" part of the graph

one has to run tens of similar computation, which different shifts, to assess if the initial computation is good enough, any derivative computation would be even more CPU intensive. The problem of finite difference derivatives with trees is described in Pelsser and Vorst (1994) and Henrard (2006). For that numerical technique the short answer is that there is no value of the shift for which a simple bump-and-recompute provides a good result.

We now give an example of what is happening at the computer internal representation level. The goal of this example is to show in the simplest example possible that the behavior described above is not related to the particular function we have used; even the simplest of the functions will display the same problems[2]. We take the example of a bill with a face value of one million currency units, denoted N and three days to maturity. The bill is quoted on a discounted basis, i.e. the value of the bill is $N \times (1 - \delta R)$ with δ the accrual factor and R the rate. We want to compute the sensitivity of the value with respect to a movement of the rate by one basis point. To simplify the matter further, suppose that the current rate is 0 and the day count

[2] The example below is inspired by (Naumann 2012, Section 1.3)

convention is "ACT/300"[3]. In symbols, we have $f(x) = N - x$ and we want to compute the derivative of f at $x = 0$.

We compute the derivative by forward finite difference, with $\epsilon = 0.1$, i.e. we compute

$$\frac{(1,000,000 - 0.1) - 1,000,000}{0.1}.$$

We do this computation using `float` representation of real numbers. According to the IEEE 754 standard, the representation of `float` uses 32 bits[4] , with 1 bit for the sign, 8 bits for the exponent and 23 bits for the fraction or mantissa.

The representation of the different numbers involved is given in Listing 1.1. We have represented in the five lines the numbers N, ϵ, $N - \epsilon$, $(N - \epsilon) - N$ and $((N - \epsilon) - N)/\epsilon$.

Listing 1.1 Internal representation of `float` numbers

```
Theoretical  S-EXPONANT-MANTISSA.............   Actual
1000000.000  0-10010010-11101000010010000000000   1000000.000
      0.100  0-01111011-10011001100110011001101         0.100
 999999.900  0-10010010-11101000010001111111110    999999.875
     -0.100  1-01111100-00000000000000000000000        -0.125
      1.000  0-01111111-01000000000000000000000         1.250
```

Already for the extremely simple case the finite difference in finite precision arithmetic gives a result which is far off with respect to the actual numbers. In this example the base function f requires only one operation, one of the simplest operation for a computer, a subtraction. For a reasonable shift of 0.1, the computed number is 1.25, this is 25% off with respect to the correct number which is 1.

Nevertheless, in a lot of places, the finite difference approach seems to be the state of the art in finance. A lot of derivatives, sensitivities or greeks are computed, in a lot of financial institutions, using a finite difference scheme as described above. Usually without really checking if the approximation is acceptable or not. Maybe twisting the shift to be larger or smaller here and there to get more meaningful numbers.

As you know, or at least as you hoped by buying this book, it is possible to do a lot better and a lot faster. Practical recipes on how to do it in quantitative finance are the *raison d'être* of the book you have in hands.

The comparison in term of performance between algorithmic differentiation and other methods will often be against one-side finite difference (forward or backward). The results will be theoretical for the finite difference. On top of the computation

[3] This is an invented convention which is used to simplify the example.

[4] See for example http://en.wikipedia.org/wiki/Single-precision_floating-point_format.

of the function itself with shifted input, the finite difference requires some data manipulation, in particular it includes copying the input vector to shift one component. For very simple functions, this can take a non-negligible amount of time. In most of our comparisons we ignore those extra-costs. Only in the very first case, with results in Table 2.4, do we include the actual implementation numbers.

In this introduction we used a one dimensional function and computed one derivative. In practice, function used in finance have several inputs, their number often ranging to the hundreds. The computation indicated above should be done for each input, in the finite difference approach, each input to the computation should be bumped individually. The computation time growths linearly with the number of inputs.

1.6 What Is Not in This Book: Higher Order

In the book we discuss the computation of first order derivatives. One may say that in theory, the second order derivative is simply the first order derivative applied to the function implementing the first oder derivative and nothing more should be said. The practice is unfortunately quite different.

Let's start with the dimensions. We said that when we have a several-inputs-and-one-output function, the most efficient approach is the adjoint mode and when the function has one input and several outputs, the most efficient approach is the forward mode. In general in finance, we have a several-inputs-and-one-output function. The derivative function of that function is a several-inputs-and-several-output function; one output for each input of the original function. For those n-input and n-output function, the computation time improvement of AD with respect to finite difference is reduce to nothing and is even not an advantage anymore. The finite difference computation time will be roughly $n + 1$ times, while the *forward AD* method will be around $1.5 \times n$ times and the *adjoint AD* method around $3 \times n$. So the speed advantage is lost for the second order derivatives.

Then there is the fact that the first order derivative is "additive" in the sense that you can compute several part of it separately and add them later. In finance, you can for example compute the "delta" of options in one function and the "vega" in another. This is not the most efficient from a computational point of view but represent the usual business decomposition of risk in several parts. For the second order, this is not true anymore; you cannot compute part of the second order derivatives in one place and another part in a second place, you would be missing the crosses. In financial terms you can compute the cross-gamma somewhere and the cross-volga somewhere else, but you are them missing the crosses between delta and vega.

In Chapter 4, we will discuss AD by operator overloading. The results are obtained by adding information to the doubles used for the computations. This means that the first order derivative code does not exists anywhere, but only the procedures are mocked in memory to achieve the result. The situation is not the same between the original code and the one used for the first order derivatives. We cannot apply the same technique a second time.

For those reason, the technique I use the most often in practice is to compute first order derivatives using AAD and to compute the second order derivatives, when needed, using finite difference on the first order derivatives. We still gain a lot of time and precision with respect to a pure finite difference. On the other side, developing a full second order machinery without a real gain, certainly from a computation time point of view, would require a lot of development time.

1.7 What Is Not In This Book: Code

In the book we propose some (pseudo-code) extract to give practical details of some implementations. The listings do not contain the full code, only enough information to understand what the code does. Nevertheless the full code of a lot of examples is available under an open source license. There is not even the need to type the code from the example, you can download it and run it directly. The code is available from GitHub at the address

`https://github.com/marc-henrard/algorithmic-differentiation-book`

The code proposed also contains an implementation of automatic Algorithmic Differentiation using forward mode and adjoint mode. The implementation is probably not production ready in the sense it is not really an operator overloading implementation but an "operators replaced by methods" implementation. Also the implementation does not propose an optimization of memory allocation; the memory requirement for numerically intensive computation can be a problem in traditional automatic AD implementation. The implementation proposed is nevertheless certainly good enough to understand the advantages and drawbacks of the methods and play with different implementation solutions. The performances results of my automatic AD implementation are in line with advertised results from third party close source implementations.

In some places in the book, for more complex cases, in particular curve sensitivity, curve calibration and model calibration, I refer to OpenGamma's *Strata* library to which I'm one of the contributors. The Strata library is also available open source under the same Apache 2.0 license[5]. My goal in the AD library is to show in action the different implementation techniques, not to develop a full quantitative finance library. The Strata library has a different goal, which is to develop a full-size quantitative finance library using one version of AD as one of its underlying techniques. The examples based on Strata are also available under an open source license at

`https://github.com/marc-henrard/analysis`

That repository contains examples related to this book and also other analysis and in particular code related to posts on my blog.

[5] It is available on github at the address `https://github.com/OpenGamma/Strata`.

Bibliography

Basel Committee on Banking Supervision. (2014). Fundamental review of the trading book: outstanding issues. Consultative document, Basel Committee on Banking Supervision.

Capriotti, L. (2011). Fast greeks by algorithmic differentiation. *The Journal of Computational Finance*, 14(3):3–35.

Capriotti, L. (2015). Likelihood ratio method and algorithmic differentiation: Fast second order greeks. *Algorithmic Finance*, 4, 4:81–87.

Capriotti, L. and Giles, M. (2010). Fast correlation Greeks by adjoint algorithmic differentiation. *Risk*, 23(3):79–83.

Capriotti, L. and Giles, M. (2012). Algorithmic differentiation: Adjoint Greeks made easy. *Risk*, 24(9):96–102.

Capriotti, L., Lee, J., and Peacock, M. (2011). Real time counterparty credit risk management in Monte Carlo. *Risk*, 24:86–90.

Capriotti, L., Jiang, Y., and Macrina, A. (2015). Real-time risk management: An AAD-PDE approach. *Journal of Financial Engineering*, 2(4).

Capriotti, L., Jiang, Y., and Macrina, A. (2016). AAD and least squares Monte Carlo: Fast Bermudan-style options and XVA greeks. Working Paper Series 2842631, SSRN.

Denson, N. and Joshi, M. (2009a). Fast and accurate Greeks for the Libor Market Model. *Journal of Computational Finance*, 14(4):115–140.

Denson, N. and Joshi, M. (2009b). Flaming logs. *Wilmott Journal*, 1:5–6.

Giles, M. and Glasserman, P. (2006). Smoking adjoints: Fast Monte Carlo Greeks. *Risk*, 19:88–92.

Griewank, A. and Walther, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd edn. Philadelphia:SIAM.

Henrard, M. (2006). A semi-explicit approach to Canary swaptions in HJM one-factor model. *Applied Mathematical Finance*, 13(1):1–18. Preprint available at IDEAS: http://ideas.repec.org/p/wpa/wuwpfi/0310008.html.

Homescu, C. (2011). Adjoints and automatic (algorithmic) differentiation in computational finance. Working Paper Series 1828503, SSRN.

Joshi, M. and Yang, C. (2010). Fast gamma computations for CDO tranches. Working Paper Series 1689348, SSRN.

Kaebe, C., Maruhn, J., and Sachs, E. (2009). Adjoint based Monte Carlo calibration of financial market models. *Finance and Stochastics*, 13(3):351–379.

Leclerc, M., Liang, Q., and Schneider, I. (1999). Fast Monte Carlo Bermudan greeks. *Risk*, 12(7):84–88.

Naumann, U. (2012). *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.* Philadelphia:SIAM.

Naumann, U. and du Toit, J. (2014). Adjoint algorithmic differentiation tool support for typical numerical patterns in computational finance. Technical report, RWTH Aachen University.

Pelsser, A. and Vorst, T. (1994). The binomial model and the greeks. *The Journal of Derivatives*, Spring, (13):45–49.

Chapter 2
The Principles of Algorithmic Differentiation

Divide et impera (Divide and conquer): the art of composition. – Where we learn that the world is made of assignments – Two modes: move forward or move backward – Branches

2.1 Derivative

The starting point of everything is obviously the definition of *derivative*, the notion we are planning to compute through Algorithmic Differentiation (AD).

Definition 2.1 (Derivative) A function $f : \mathbb{R}^m \to \mathbb{R}^n; x \mapsto f(x)$ is said to be *differentiable* at a point $x_0 \in \mathbb{R}^m$ if f is defined on that point and there exist a linear function $Df(x_0) : \mathbb{R}^m \to \mathbb{R}^n$ such that

$$\lim_{\epsilon \to 0; \epsilon \in \mathbb{R}^m} \frac{f(x_0 + \epsilon) - (f(x_0) + Df(x_0)(\epsilon))}{|\epsilon|} = 0.$$

The linear function $Df(x_0)$ is called the *derivative* of f in x_0.

We will represent element of $\mathbb{R}^n$ by column vectors. The linear functions from $\mathbb{R}^m$ to $\mathbb{R}^n$ will be represented by matrices of dimension m (lines) $\times n$ (columns). Through the text, we will not distinguish between elements of $\mathbb{R}^n$ and column vectors or between linear functions and matrices; the context should be clear enough.

When a function $f : \mathbb{R}^m \to \mathbb{R}$ is differentiable at a point x_0, we can define its partial derivatives $D_i f(x_0)$ $(1 \leq i \leq m)$ as the coefficient of the derivative in the i-th direction. Let e_i be the base vector of $\mathbb{R}^m$ in the i-th dimension, i.e. the vector of $\mathbb{R}^m$ with all components equal to 0 except the i-th one equal to 1. The partial derivative $D_i f(x_0) \in \mathbb{R}$ is defined by $Df(x_0)(e_i)$. In particular we have

$$D_i f(x_0) = \lim_{\epsilon \to 0; \epsilon \in \mathbb{R}} \frac{f(x_0 + \epsilon e_i) - f(x_0)}{\epsilon}. \tag{2.1}$$

By abuse of language, like for matrices, we will use the notation $Df(x_0)$ indifferently for the linear function and the $\mathbb{R}^m$ vector with components $(D_i f(x_0))_{1 \leq i \leq m}$. The context will, in general, be enough to clarify which representation we use.

M. Henrard, *Algorithmic Differentiation in Finance Explained*,
Financial Engineering Explained, DOI 10.1007/978-3-319-53979-9_2

The computation of the (partial) derivatives through one-sided finite difference quotient, as described in Section 1.5, is approximating the limit in Eq. 2.1, by the value of the ratio at a fixed value ϵ.

2.2 Composition

The main theoretical concept underlying Algorithmic Differentiation is the notion of *composition*. A long task can be divided in shorter subtasks combined one behind the other. This idea of combining one thing after another is described mathematically by the concept of composition.

Definition 2.2 (Composition) Let $g : \mathbb{R}^m \to \mathbb{R}^n; x \mapsto g(x)$ and $f : \mathbb{R}^n \to \mathbb{R}^p; y \mapsto f(y)$ be two functions. The function called f *compounded with* g or f *after* g and denoted $f \circ g$, is the function $\mathbb{R}^m \to \mathbb{R}^p$ with

$$(f \circ g)(x) = f(g(x)).$$

This is the same idea as the division of labor. Different tasks are performed by different workers or different functions. Combined specialized functions – or workers, or methods, or micro-services – provide efficiency gains.

One of the main tasks of quantitative finance libraries is to perform the computation of single output functions applied to financial instruments and market data. Throughout the book we use the present value case in the descriptions as it is the most common case in practice, but it is also valid for par rate, currency exposure, price, etc. The use of present value is for convenience in the description, but it does not change the technical part of the analysis.

We suppose that the task of computing this single output method has been implemented using a long list of specialized functions or methods like f and g above. The functions can, and, in practice, will, have several inputs. How to write the algorithm to compute the present value is not the problem that we propose to solve through this book; the reader is expected to have solved that problem already and we suppose that he knows how to implement the value part.

The next task, and the center of this book, is the computation of the sensitivities of those present values to the multiple inputs. In finance those sensitivities are often called the *greeks*. The greeks can be a simple one dimensional number, like the delta of an option to its unique underlying, or multi-dimensional results like the bucketed sensitivities to market quotes in interest rate instruments. The latter being more demanding in computational time and the center of interest here.

It means that the task of this book is to obtain for each function, not only the computation of the value it is supposed to compute, but also of its derivatives with respect to the inputs. If the Black function, providing the option price from the forward, volatility, strike and time to expiry is implemented, we also want to have the derivatives of the price with respect to those four inputs implemented. In the greeks jargon, they are called delta, vega, adjoint delta and theta.

The principles to achieve those implementations are described in this chapter. The building block of this approach is the mathematical result below[1].

Theorem 2.1 (Chain rule) Let $g : \mathbb{R}^m \to \mathbb{R}^n; x \mapsto g(x)$ and $f : \mathbb{R}^n \to \mathbb{R}^p; y \mapsto f(y)$ be two functions. If g is differentiable in x_0 and f is differentiable in $g(x_0)$, then the composition is differentiable in x_0. The derivative of the composition is given by

$$D(f \circ g)(x_0) = Df(g(x_0)) \cdot Dg(x_0). \tag{2.2}$$

In Formula (2.2), the operation $\cdot$ is the composition if we use the linear function representation and is the matrix multiplication if we use the matrix representation.

The differentiation operator transforms a composition of function into the composition of the linear functions representing the derivatives. The composition of linear functions can be computed through the product of the matrices representing the functions.

The result on the derivative of a composition of functions may look benign, with simply a circle on the left hand side of the equality replaced by a small dot on the right hand side. But that simple one liner is the core of AD; understanding that one line and its impact on quantitative finance libraries is the only goal of this book. This one line is a tremendously good news for library's architects!

The summary of the impact is that the differentiation part of the library can be written without changing the architecture put in place for the valuation. The composition of functions is transformed into a composition of linear operators, which can be written as a product of matrices. Any architecture that has been established and is efficient for computing the values by splitting the general task in subtasks can be used without change for the computation of the derivatives. The method achieving the different tasks will have to be complemented by new methods to achieve the differentiation of those tasks, but the general architecture can stay unchanged.

As one good news never comes alone, the second good news is the product part. The composition of function at the value level is converted in a product of matrices at the derivative level. The product of matrices is, at the coefficient levels, a combination of sums and products of real numbers. Those real numbers are represented in computers by `double`. It turns out that processors are very efficient at computing sums and products of doubles. We can thus expect that there will be a very hardware-efficient way to implement the derivative computation procedure deduced from the composition formula.

Now it is time for a little bit of bad news. Up to now we have said that the computation of derivatives can be done without changing the architecture and can probably be done very efficiently. The bad news is that it has to be done; it will not

[1] We describe the results without all the rigor that would be used in a mathematical text. In particular, we do not indicate which exact class of functions will satisfy the results. The functions are supposed regular enough to satisfy the different results, without indicating exactly what we mean by regular enough. More precise statement are given in Appendix A.

appear magically. It is not only a figure of speech that "something has to be done" but that to have it working everything has to be done.

When you read the theorem again, you can see that the final result requires the derivatives of each of the composing functions. If you want to implement this technique for your library you have to start from the very bottom and work your way up. There is no way to get the benefits for a couple of time consuming pricing methods at the top of your library and ignore the rest. All the low level pieces on which those complex pricing methods are based need to be implemented in an AD compatible way, with their derivatives. In the quantitative finance case, this often mean that interpolation schemes, discount factors, Black formula, forward curves, and other building block methods have to be looked at in all details before we start looking at numerical integration, option pricing, Monte Carlo or other numerically demanding tasks.

The Algorithmic Differentiation approach is really a *philosophy* based on the division of labor. Each small task is performed by an expert method providing the value and all its derivatives with respect to its inputs. As much as it was emphasized in the previous paragraph that AD is a bottom up approach that requires all the methods to provide the derivatives, as much there is nobody looking under the hood to see how the task is performed. Each subtask can provide the derivatives using its own technique. For some subtask, one may choose to use either a finite difference method as described in the introduction, or an adjoint AD implementation, or a standard AD implementation, or a direct analytical formula. There is no restriction on how each part of the list of tasks is working internally, as long as it provides the required result in an efficient way. An Algorithmic Differentiation philosophy could be implemented with only finite difference at the lowest level but product of derivatives at each level above. There are as many shades of AD as there are AD users. The *Principles of Algorithmic Differentiation* chapter provides to the user the black and the white; it is up to him to get the correct shade of grey that fits his taste and his requirements.

This remark is quite important in term of implementation. If a financial instrument appears only once in the balance sheet of a bank and there is no incentive to improve the computation speed for it, its derivatives can be computed by a simple finite difference approach. This will not impact the general functioning of the library. The derivatives are provided, maybe inefficiently, but they are there and the global process can be achieved. For example when implementing a new interpolation mechanism for a yield curve, one may chose to focus on the results first without looking at the computational efficiency by implementing the derivatives of the interpolation using a generic finite difference implementation. If the results are good from a financial perspective, one may come back to the details of the interpolation derivatives latter. The internal implementation may be changed but it should remain transparent to the developer. Each method has a value implementation and a value and derivatives implementation. Those methods can be refined internally but its availability to the outside world and its signature remain the same.

2.3 Algorithm: Assignment

We now enter into the internal working of the Algorithmic Differentiation principles. We start discussing code writing.

The starting point is the computation of a function $f : \mathbb{R}^{p_a} \to \mathbb{R}$;

$$a \mapsto z = f(a). \tag{2.3}$$

The function inputs are $a = a[0 : p_a - 1]$ of dimension p_a. We use a mixture of Matlab and Java notation for the vectors, with $a[i : j]$ being the vector of dimension $j - i + 1$ and components a_k for $k = i, \ldots, j$.

The vector has to be thought of as the market quotes used to calibrate curves or as the inputs used to price options, like forward, volatility – single value or full surface –, strike, bond price, etc. The output of the function is z of dimension 1. The main application in finance is for the present value, the price, the implied volatility, the spread or the par rate.

As a starting point we suppose that the program that represents the algorithm to compute f is a simple linear program with one assignment for each line of code and nothing else. This approach is called Single Assignment Code (SAC) in Naumann (2012). We will introduce other potential elements in the code later. In particular ifs, loops and equation solving.

As the starting point of the analysis of Algorithmic Differentiation, we suppose that the algorithm for the function itself is implemented. It is not the goal of this book to explain how to implement the algorithms for the functions themselves.

All the intermediary values used in the program will be denoted b (with different indices). The algorithm starts with the inputs a and goes to z through a lot of bs. The new variables are denoted by $b[j]$ with j starting at 1 and going up to p_b. There are p_b intermediary variables (or parameters) b in the program. The number p_b is also the number of lines of code. At each line of code, the new variable $b[j]$, which depends potentially on all the previous variables $b[k]$ with $k < j$ and the initial inputs a, is created.

The simplified version of an algorithm is represented in Table 2.1. Note that we initialize the variables $b[j]$ $(-p_a + 1 < j < 0)$ with the inputs values. This is to unify the notation of the Algorithmic Differentiation implementation later. Each line of code perform the computation of a function g_j. That function can be an elementary arithmetic computation, like $b_1 + b_2$, a mathematical function, like $\exp(b_3)$, or another method already implemented in the library for which the value and derivative versions are available.

Table 2.1 The generic single assignment code for a function computing a single value

Initialization	$[j = -p_a + 1 : 0]$	$b[j]$	=	$a[j + p_a - 1]$
Algorithm	$[j = 1 : p_b]$	$b[j]$	=	$g_j(b[-p_a + 1 : j - 1])$
Output		z	=	$b[p_b]$

2.3.1 Standard Algorithmic Differentiation

Standard Algorithmic Differentiation is also called *Forward Algorithmic Differentiation* or *Tangent Algorithmic Differentiation*.

Our goal is to compute $\partial z/\partial a_i$. We achieve this by computing for each j ($-p_a+1 \leq j \leq p_b$) the value

$$\dot{b}[j,i] = \frac{\partial}{\partial a[i]} b[j].$$

Note that the derivative is denoted by a dot ($\dot{}$) on the variable and $\dot{b}[j,i]$ is the derivative of $b[j]$ itself with respect to some other variable, the inputs $a[i]$. The summary of the standard AD algorithm is presented in Table 2.2. The starting point is the easiest part. For $j = -p_a + 1 : 0$, the derivative of $b[j]$ with respect to $a[i]$ is simply the derivative of $a[i]$ with respect to itself – which is 1 – for $j = -i$ and 0 for $j \neq -i$. This is the starting point of a recursive algorithm. The starting part is an identity matrix: $\dot{b}[j,i] = \delta_{i,-j}$.

From there we read the code in the forward order and use the derivative of a composition p_a times for each line of code. Each intermediary variable $b[j]$ uses only the variables that have been populated in the previous lines of code in the implementation of the algorithm. The derivatives $\dot{b}[j,i]$ are given by

$$\begin{aligned}\dot{b}[j,i] = &\frac{\partial}{\partial a[i]} b[j] = \sum_{k=-p_a+1}^{j-1} \frac{\partial}{\partial b[k]} b[j] \cdot \frac{\partial}{\partial a[i]} b[k] \\ &= \sum_{k=-p_a+1}^{j-1} \frac{\partial}{\partial b[k]} g_j(b[-p_a+1:j-1]) \cdot \dot{b}[k,i].\end{aligned}$$

From the recursive approach, the values $\dot{b}[k,i]$ for $k < j$ are already known. The derivatives of the functions g_k can be obtained from the implementation of algorithmic

Table 2.2 The generic code for a function computing a single value and its standard algorithmic differentiation pseudo-code

Initialization	$[j = -p_a + 1 : 0]$	$b[j]$	$= a[j + p_a - 1]$
Algorithm	$[j = 1 : p_b]$	$b[j]$	$= g_j(b[-p_a + 1 : j - 1])$
Output		z	$= b[p_b]$
Initialization	$[j = -p_a + 1 : 0][i = 0 : p_a - 1]$	$\dot{b}[j,i]$	$= \delta_{i,-j}$
Algorithm	$[j = 1 : p_b][i = 0 : p_a - 1]$	$\dot{b}[j,i]$	$= \sum_{k=-p_a+1}^{j-1} \frac{\partial}{\partial b_k} b_j \frac{\partial}{\partial a_i} b_k$
			$= \sum_{k=-p_a+1}^{j-1} \frac{\partial}{\partial b_k} g_j \dot{b}[k,i]$
Output	$[i = 0 : p_a - 1]$	$\frac{\partial}{\partial a_i} z$	$= \dot{b}[p_b,i]$

differentiation that we have supposed to have been done for all the functions g_j. Using that approach one can recursively obtain the values $\dot{b}[j,i]$ up to $\dot{b}[p_b,i]$ going through $j = -p_a + 1$ to p_b. The numbers $\dot{b}[p_b,i]$ are equal to the derivatives of $z = b[p_b]$ with respect to a_i ($i = 0,\ldots,p_a - 1$). This concludes the algorithm for the computation of $\partial z/\partial a_i$.

The requirements for such an implementation is that all the intermediary functions g_j have a derivative version. The Algorithmic Differentiation approach is a bottom-up approach: it can be implemented for an algorithm only if all the components "below it," all the components entering into the composition have already been implemented.

There is no way to take the most complex function in a library and try to implement algorithmic differentiation there if it has not been implemented fully for all the functions below. It is not a miraculous method that can be applied in complex situations and ignored in simpler, more elementary parts of a library. You cannot have a solid quantitative finance library if you have not spend enough time and efforts on the foundations.

As a first practical example of the theoretical description above, we implement the algorithm for a very simple function. The function has no special financial meaning. It is a four inputs one output function using elementary analytical functions.

The function (with 4 inputs) is:

$$z = \cos\left(a_0 + \exp(a_1)\right)\left(\sin(a_2) + \cos(a_3)\right) + (a_1)^{\frac{3}{2}} + a_3.$$

The code of one algorithm to compute that simple function is given in Listing 2.1. As indicated in Section 1.7, the full code for the examples is available in an open source format on Github. From now on, we don't repeat the advertisement for the open source code and we simply give the name of the class in which the code can be found. The code corresponding to Listing 2.1 can be found in the class `AdStarter`.

Listing 2.1 Simple function code

```
double f(double[] a) {
  double b1 = a[0] + Math.exp(a[1]);
  double b2 = Math.sin(a[2]) + Math.cos(a[3]);
  double b3 = Math.pow(a[1], 1.5d) + a[3];
  double b4 = Math.cos(b1) * b2 + b3;
  return b4;
}
```

The code for the manual Standard Algorithmic Differentiation version is given in Listing 2.2 and in the method `f_Sad` of the class `AdStarter`. Note that the output of the original function is a double while the output of the AD version is an object containing a double and an array representing the derivatives of the

result with respect to the different inputs. The description of the object, called `DoubleDerivatives` can be found in Listing 2.3.

Listing 2.2 Standard Algorithmic Differentiation version of the starter function

```
static public DoubleDerivatives f_Sad(double[] a) {
    // Forward sweep - function
    double b1 = a[0] + Math.exp(a[1]);
    double b2 = Math.sin(a[2]) + Math.cos(a[3]);
    double b3 = Math.pow(a[1], 1.5d) + a[3];
    double b4 = Math.cos(b1) * b2 + b3;
    // Forward sweep - derivatives
    int nbA = a.length;
    double[] b1Dot = new double[nbA];
    b1Dot[0] = 1.0;
    b1Dot[1] = Math.exp(a[1]);
    double[] b2Dot = new double[nbA];
    b2Dot[2] = Math.cos(a[2]);
    b2Dot[3] = - Math.sin(a[3]);
    double[] b3Dot = new double[nbA];
    b3Dot[1] = 1.5d * Math.sqrt(a[1]);
    b3Dot[3] = 1.0d;
    double[] b4Dot = new double[nbA];
    for(int loopa = 0; loopa < nbA ; loopa++) {
      b4Dot[loopa] = b2 * - Math.sin(b1) * b1Dot[loopa] +
          Math.cos(b1) * b2Dot[loopa] + 1.0d * b3Dot[loopa];
    }
    return new DoubleDerivatives(b4, b4Dot);
  }
```

Listing 2.3 Object containing a double and its derivatives with respect to the inputs

```
public class DoubleDerivatives {
  private final double value;
  private final double[] derivatives;

  public DoubleDerivatives(double value, double[] derivatives) {
    this.value = value;
    this.derivatives = derivatives;
  }
  ...
}
```

The code itself does not require lengthy explanations. For each variable `bx`, a derivative vector is created with the dimension of the inputs, which is 4 and denoted `nbA` in the code. When created, the vector is filled with 0, which is the default in Java. Then we look at each of the next lines and see how they depend on the inputs. The first three lines depend only on the inputs and the derivatives of standard functions are used. The fourth variable depends of the previous variables.

Its derivatives depends on the four inputs; they are computed in a loop on the dimension of the inputs.

The performance of the different implementations in term of computation time is discussed at the end of the section and summarized in Table 2.4. The performance is analyzed in the test class `AdStarterAnalysis`.

2.3.2 Adjoint Algorithmic Differentiation

Adjoint Algorithmic Differentiation is also called *Reverse Algorithmic Differentiation*.

Our goal is to compute $\partial z/\partial a_i$. We achieve this by computing for each intermediary variable j ($-p_a + 1 \leq j \leq p_b$) the value

$$\bar{b}[j] = \frac{\partial}{\partial b[j]} z.$$

Note that the derivative is denoted by a "bar" (¯) on the variable and $\bar{b}[j]$ is the derivative of the output with respect to $b[j]$. This is a non-standard notation in the sense that the variable in the name is not the variable that is differentiated but the variable with respect to which the derivative is taken. It is important to switch perception between the forward and reverse approach. What is fixed in the reverse approach is the output; we always compute the derivative of the same variable, the output; it is not required to repeat that information at each step.

The summary of the algorithm is presented in Table 2.3. The starting point of the algorithm is the easy part. For $j = p_b$, the derivative of z with respect to b_j is simply the derivative of z with respect to itself, which is 1. This is the starting point of a recursive algorithm.

Table 2.3 The generic code for a function computing a single value and its adjoint algorithmic differentiation code

Init	$[j = -p_a + 1 : 0]$	$b[j]$	=	$a[j + p_a - 1]$
Algorithm	$[j = 1 : p_b]$	$b[j]$	=	$g_j(b[-p_a + 1 : j - 1])$
Value		z	=	$b[p_b]$
Value		$\bar{z}$	=	1.0
Value		$\bar{b}[p_b]$	=	1.0
Algorithm	$[j = p_b - 1 : -1 : -p_a]$	$\bar{b}[j]$	=	$\sum_{k=j+1}^{p_b} \frac{\partial}{\partial b_k} z \frac{\partial}{\partial b_j} b_k$
			=	$\sum_{k=j+1}^{p_b} \bar{b}[k] \frac{\partial}{\partial b_j} g_k$
Init	$[i = 0 : p_a - 1]$	$\frac{\partial}{\partial a_i} z$	=	$\bar{b}[i - p_a + 1]$

From there we read the code in the reverse order and uses the derivative of a composition. Each intermediary variable $b[j]$ is used only in the lines of code that follow in the computation. The derivative $\bar{b}[j]$ is given by

$$\bar{b}[j] = \frac{\partial}{\partial b_j} z = \sum_{k=j+1}^{p_b} \frac{\partial}{\partial b_k} z \cdot \frac{\partial}{\partial b_j} b_k = \sum_{k=j+1}^{p_b} \bar{b}[k] \cdot \frac{\partial}{\partial b_j} g_j(b[-p_a + 1 : k-1]). \tag{2.4}$$

From the recursive approach, the values $\bar{b}[k]$ for $k > j$ are already known. The derivatives of the functions g_k can be obtained from the implementation of algorithmic differentiation for the building blocks functions g_k or as known mathematical functions. Using that approach one can recursively obtain the values $\bar{b}[j]$ down to $\bar{b}[l]$ for $l = -p_a + 1, \ldots, 0$. Those numbers are equal to the derivatives of z with respect to the inputs a_i ($i = 0, \ldots, p_a - 1$). This concludes the algorithm for the adjoint mode of AD.

The version of adjoint Algorithmic Differentiation for the simple function described above is displayed in Listing 2.4 and implemented in the method `f_Aad` of the class `AdStarter`.

Listing 2.4 Adjoint Algorithmic differentiation version of the function

```
static public DoubleDerivatives f_Aad(double[] a) {
  // Forward sweep - function
  double b1 = a[0] + Math.exp(a[1]);
  double b2 = Math.sin(a[2]) + Math.cos(a[3]);
  double b3 = Math.pow(a[1], 1.5d) + a[3];
  double b4 = Math.cos(b1) * b2 + b3;
  // Backward sweep - derivatives
  double[] aBar = new double[a.length];
  double b4Bar = 1.0;
  double b3Bar = 1.0d * b4Bar;
  double b2Bar = Math.cos(b1) * b4Bar + 0.0 * b3Bar;
  double b1Bar = b2 * - Math.sin(b1) * b4Bar
    + 0.0 * b3Bar + 0.0 * b2Bar;
  aBar[3] = 1.0 * b3Bar - Math.sin(a[3]) * b2Bar + 0.0 * b1Bar;
  aBar[2] = Math.cos(a[2]) * b2Bar;
  aBar[1] = 1.5d * Math.sqrt(a[1]) * b3Bar
    + Math.exp(a[1]) * b1Bar;
  aBar[0] = 1.0 * b1Bar;
  return new DoubleDerivatives(b4, aBar);
}
```

The dimension of the returned vector $\bar{a}$ is the same as the one of a. The function returns the final value `b4` and returns its derivatives in `aBar`. The value `0.0 * bxBar` should be added in a lot of places for the aBar computations. Here we ignore those terms that appear in the theoretical algorithm but do not add any value in practice.

Table 2.4 Computation time for the example function and four derivatives

Repetitions	Computation	Time	Ratio
100,000	Function	110	1.00
100,000	Function + FD forward	575	5.23
100,000	Function + FD backward	575	5.23
100,000	Function + FD Symmetrical	915	8.32
100,000	Function + FD 4th order	1830	16.64
100,000	Function + SAD	340	3.09
100,000	Function + SAD Optimized	195	1.77
100,000	Function + AAD	220	2.00
100,000	Function + AAD Optimized	165	1.50

Figures for time in milliseconds. Times measured on the author's laptop. Each repetition is with five different data sets so that there is actually 500,000 repetitions.

Note that the mathematical derivative of exp, sin and cos are known and already implemented. In a full library implementation, one is not supposed to know what the derivatives of the other methods are; all methods should have a relevant derivative version available.

What is the performance of those different implementations? The estimation of the computation times are provided in Table 2.4. First we compute the derivatives using different finite difference schemes. This allows us to check that our implementation of the AD versions are correct – unit tests are important in a library – or at least that they are correct within a certain error margin due to the systematic error from the finite difference approach. It allows us also to compare the performance on a practical case and see if our above theoretical discussion related to the finite difference performance is realistic. The performance is analyzed in the test `derivativesPerformance` of the class `AdStarterAnalysis`. Before the run used to produce the table, the Java JVM is *warmed-up*, allowing it to find the hotspots and JIT these. This will be the case for all the performance tests discussed in the book.

For the finite difference versions, the performance is in line with our description. The one-sided schemes, the forward and the backward finite difference, take slightly more than $1 + n = 5$ times the CPU time used for the value computation. For the two-sided scheme the computation time is slightly more than $2 * n = 8$ and for the fourth order scheme, the computation time is slightly more than $4 * n = 16$. As can be seen from the actual numbers, the finite difference take a little bit more time than the theoretical description. This is the time required to manipulate the data before starting the repricing; in particular the input data vector needs to be copied and shifted by the finite difference shift.

If we look at the different Algorithmic Differentiation versions, we see that the results are in line with promises. The direct adjoint implementation takes roughly two times the value computation time for computation of the value and the four derivatives. The standard implementation is a little bit slower with a ratio above three. This result for standard algorithmic differentiation is not typical; usually the ratio is closer to the number of inputs plus 1 (5 in our case). This improved performance

is due to the very simplistic nature of our example. For advanced case, in particular the Black option price function discussed in the next chapter, will show that this is not a general result.

We have introduced two new version of the AD implementation. In one of them, the adjoint version code has been "optimized" in a very simple way. The code reuses as much as possible the numbers already computed. In particular, the `Math.exp(a[1])` is used twice in the first code. Computing it once and storing the value for the second usage saves some computation time. Similarly, storing `Math.cos(b1)` helps. We also store `Math.pow(a[1], 1.5d)` and notice that `Math.sqrt(a[1])` is equal to `Math.pow(a[1], 1.5d) / a[1]`. The division being faster than the square root, we gain a little bit more time. In total, with those small improvements, the computation time ratio is now around 1.50. It means it is more efficient to compute *the value and all four derivatives* by AD than to compute *only one derivative* by finite difference. Even if the code is used to extract only one derivative, the adjoint AD version is more efficient that one-sided finite difference, not to mention more precise. This is a feature we will notice repeatedly; you don't need to require all derivatives of a function to take advantage of AD. In a lot of cases, if you need the derivatives to only one or two input, it is more efficient to compute all derivatives – maybe hundreds of them – by AD and keep only the one required than to compute only the one you need by finite difference.

Obviously this example can be seen a "cheating," there is no full library involved, no call to lower level functions with their own AD implementation. In the next chapter, we will introduce more complex situation in financial applications.

2.4 Algorithm: Branches

Algorithmic Differentiation deals with derivatives. It is expected that the function to which the mechanism is applied is itself differentiable. If not the result is undefined. This is the occasion to introduce a warning: *Algorithmic Differentiation creates the derivative of the code executed, not a derivative of the function.* It is important to remember that in the execution, one set of instructions is executed. It is the derivative of that set of instructions that is naturally obtained by AD, not the derivative of the abstract function that is approximated by the code. To parody the saying about "*winners writing history*" one could say that the "*the winning branch is writing the code*" or anticipating on the automatic Algorithmic Differentiation of Chapter 4, "*the winning branch is writing the tape*."

One extreme case would be the implementation of the logarithm function with a huge lookup table (remember those logarithm tables?). It means that in all cases the result returned will be one of the constant of the table. What is the derivative of a constant? It is 0. Such an implementation of the function with a un-careful use of AD would lead to the practical result that the derivative of the logarithm is always 0!

One case where this division of a function algorithm into different sub-algorithms executed under different circumstances appears explicitly, is the presence of

Table 2.5 The generic code for a function computing a single value with an *if* statement

Initialization	$[j = -n+1:0]$	$b[j]$	=	$a[j+n-1]$
Algorithm	$[j = 1:m]$	$b[j]$	=	$g_j(b[-n+1:j-1])$
				if $f(b[-n+1:m])\{$
	$[j = m+1:p_b]$	$b[j]$	=	$\quad g_j(b[-n+1:j-1])$
				$\}$ else $\{$
	$[j = m+1:\tilde{p}_b]$	$b[j]$	=	$\quad \tilde{g}_j(b[-n+1:j-1])\}$
Value		z	=	$b[p_b]$ or $b[\tilde{p}_b]$

Table 2.6 The generic code for a function computing a single value and its algorithmic differentiation code

Initialization	$[j = -n+1:0]$	$b[j]$	=	$a[j+n-1]$
Algorithm	$[j = 1:m]$	$b[j]$	=	$g_j(b[-n+1:j-1])$
				if $f(b[-n+1:m])\{$
	$[j = m+1:p_b]$	$b[j]$	=	$\quad g_j(b[-n+1:j-1])$
				$\}$ else $\{$
	$[j = m+1:\tilde{p}_b]$	$b[j]$	=	$\quad \tilde{g}_j(b[-n+1:j-1])\}$
Value		z	=	$b[p_b]$ or $b[\tilde{p}_b]$
Value		$\bar{z}$	=	1.0
Value	$\bar{b}[p_b]$ or $\bar{b}[\tilde{p}_b]$		=	1.0
				if $f(b[-n+1:m])\{$
Algorithm	$[j = p_b-1:-1:-p_a]$	$\bar{b}[j]$	=	$\sum_{k=j+1}^{p_b} \bar{b}[k]\frac{\partial}{\partial b_j} g_k$
				$\}$ else $\{$
	$[j = \tilde{p}_b-1:-1:-p_a]$	$\bar{b}[j]$	=	$\sum_{k=j+1}^{\tilde{p}_b} \bar{b}[k]\frac{\partial}{\partial b_j} g_k\}$
Init	$[i = 0:p_a]$	$\frac{\partial}{\partial a_i} z$	=	$\bar{b}[i-p_a+1]$

branches or *ifs* in the code. Suppose that the code is now something like the one described in Table 2.5 with f a function of $n+j$ double with boolean value.

The resulting AAD code is given in Table 2.6. The branching on the main algorithm is repeated in the backward sweep.

At this stage, one has to be careful with the result's meaning. Suppose that the boolean function denote f in the table is of the type $b[m] > 0$. This will lead to non-differentiable function at the point where $b[m] = 0$ in most of the cases. To be correct, if we are in the case $b[m] = 0$, both branches should be computed and the results of the value itself and its derivatives should be compared. Only if both the values and the derivatives of the different branches are equal, should a derivative result be returned. In practice this is not done. The code is executed, going through one and only one of the branches and the result is returned. It is expected that it is the responsibility of the algorithm writer and not the code writer to verify that this is the case. The actual differentiability of the function resulting from the algorithm is not checked at the code level, it is assume to take place before, in the design phase.

There are cases where the branching, even with only one very simple branch, can be nasty. Take the case where the condition is of the type $b[m] = 0$ and the first branch is something like $b[m+1] = g_m(b[-n+1 : m]) = 0$. This is not an exceptional case, and we will encounter it in our SABR example in the next chapter. If you apply the AD algorithm directly, you get

$$\bar{b}[m-1] = \bar{b}[m]\frac{\partial}{\partial b_{m-1}} g_m = \bar{b}[m]0 = 0$$
$$(j < m-1) \quad \bar{b}[j] = \sum_{k=j+1}^{p_b} \bar{b}[k]\frac{\partial}{\partial b_j} g_k = \sum_{k=j+1}^{p_b} 0\frac{\partial}{\partial b_j} g_k = 0.$$

From that very simple branch, where the value is a hardcoded constant when the branching condition is met for one specific value, we have contaminated the rest of the algorithm. All the derivatives computed from that branch will be 0.

When you look at it from a theoretical point of view, this is the correct result. We act like if the branches did not exist; if we take one branch, only the code in that branch exists. The final result is a constant. The derivative of a constant is always zero, whatever the inputs are. The AD implementation for that branch is a complex mechanism to produce outputs with value 0. It is very difficult to escape from that trap in an automatic way. Except from the algorithm designer knowing the problem and creating a local solution, we don't know a way to solve this. In this case, the local solution would be something like

$$b[m+1] = c \times b[m]$$

with c the derivative of the output with respect to the input $b[m]$ computed by the algorithm designer outside of the AD framework. In that case, the output will be the same: $b[m+1] = 0$, but the derivative will be different, it will be

$$\bar{b}[m] = c \times \bar{b}[m+1] = c.$$

2.5 Algorithm: Loops

Loops, through *for* or *while*, are almost a non-event for Algorithmic Differentiation. If the code in the loop is composed of single assignments, one can unwind the loop to create a longer list of assignments. The assignments may not be written separately from a code perspective, but they are still executed consecutively as if they were different lines of the program.

As the adjoint mode of AD goes through the code in reverse order, the loops have to be also read in the reverse order. A loop that reads as **for** `(int i = 0; i < N; i++)` in the function computation will probably read as **for** `(int i = N-1; i >= 0; i--)` in the reverse AD code.

Moreover, one has to be careful with the variables that change value through the loop. The safest approach is often to change from a single variable that changes value through the loop to an array, with each element assigned only once. This is more in line with the Single Assignment Code presentation used in this chapter.

Bibliography

Naumann, U. (2012). *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.* Philadelphia:SIAM.

Chapter 3
Application to Finance

Knowledge of finance and mathematics helps for programming – A single point is not enough to know about derivatives – Function spaces are of infinite dimension – Black magic with Black formula.

It seems there is no easy starting point for reviewing Algorithmic Differentiation in finance. Each function, especially in the simpler one, seems to be an exception with a particular trick more than a general direct application of the methodology.

Instead of trying to find a convoluted way to introduce a simple application that comes directly from the general principles, we introduce a couple of application to finance. For each of them the relevant pricing approaches probably appear in the early chapters of quantitative finance books.

We will look at the Black-Scholes formula and discover that knowing finance and mathematics helps also for Algorithmic Differentiation. Next we move to SABR implied volatility formula to remember that the derivative of a function at a given point does not depend on the point alone but also on its neighborhood. Further, we pass by cash flow discounting to be reminded that function spaces are of infinite dimension which are not easily represented in computers and finish with Monte Carlo simulations that are maybe numerically complex but look straightforward from an AD implementation perspective.

3.1 Black Formula

We start with the Black and Scholes formula, or more exactly the version in term of forward price or rate often referred to as Black (1976). We could have started with the Bachelier (1900) formula, if we were to respect the historical order, but from a popularity perspective, Black formula seems a better starting point. The formula can be summarized in a couple of lines. The inputs are the underlying's forward price or rate F, the (implied) volatility σ, the numeraire P, the strike price or rate K and the time to expiry θ. The formula is different for calls and puts. In the formulas, we use $\omega = 1$ for a call and $\omega = -1$ for a put. The formula is based on the quantities

$$d_{\pm} = \frac{\ln\left(\frac{F}{K}\right) \pm \frac{1}{2}\sigma^2\theta}{\sigma\sqrt{\theta}}.$$

M. Henrard, *Algorithmic Differentiation in Finance Explained*,
Financial Engineering Explained, DOI 10.1007/978-3-319-53979-9_3

The value d_+ is the boundary between exercise and non-exercise regions in the underlying random normal variable – after change of measure to the numeraire measure. The present value is given by

$$\text{Black}(F, \sigma, P, K, \theta) = \omega P \left(FN(\omega d_+) - KN(\omega d_-) \right) \tag{3.1}$$

where N is the cumulative standard normal distribution function.

The implementation of such formula is very simple and direct and is represented in Listing 3.1. The implementation code for the Black function can be found in the class `BlackFormula` of the library associated to the book. The implementation does not check for correctness of the inputs – at least the forward, volatility and strike should be positive – as should good production code, but the implementation contains all the numerical ingredients of the formula.

Listing 3.1 Simple implementation of the Black 76 formula

```
double price(double forward, double volatility,
    double numeraire, double strike, double expiry,
    boolean isCall) {
  double omega = isCall ? 1.0d : -1.0d;
  double periodVolatility = volatility * Math.sqrt(expiry);
  double dPlus = Math.log(forward / strike) / periodVolatility
    + 0.5d * periodVolatility;
  double dMinus = dPlus - periodVolatility;
  double nPlus = NORMAL.getCDF(omega * dPlus);
  double nMinus = NORMAL.getCDF(omega * dMinus);
  double price = numeraire * omega
    * (forward * nPlus - strike * nMinus);
  return price;
}
```

Looking quickly at the code, it appears to fit perfectly with our description of SAC code and that nothing more can be said. Let's start with the first part. Yes, it fits perfectly the applicability criterion. The standard and adjoint AD code can be written directly. As the standard mode is not the most efficient in finance, we don't comment on it any more. As a reference, the code is nevertheless available in the code repository in the class mentioned above.

The output of the AD methods are the price and the derivative of the price with respect to the inputs. Like in the previous chapter, the output of the methods are `DoubleDerivatives` objects. The Listing with the description of the object was provided in Listing 2.3.

The adjoint algorithmic differentiation code is provided in Listing 3.2. The new output for the AD adjoint mode provides the derivative with respect to the five inputs – forward, volatility, numeraire, strike, and expiry – in the order they are listed in the method signature.

Listing 3.2 Direct implementation of the Black 76 formula in adjoint mode

```
DoubleDerivatives price_Aad(double forward, double volatility,
    double numeraire, double strike, double expiry,
    boolean isCall) {
  // Forward sweep - function - see above
  ...
  // Backward sweep - derivatives
  double priceBar = 1.0;
  double nMinusBar = numeraire * omega * -strike * priceBar;
  double nPlusBar = numeraire * omega * forward * priceBar;
  double dMinusBar = NORMAL.getPDF(omega * dMinus)
    * omega * nMinusBar;
  double dPlusBar = 1.0d * dMinusBar +
    NORMAL.getPDF(omega * dPlus) * omega * nPlusBar;
  double periodVolatilityBar = -1.0d * dMinusBar +
    (-Math.log(forward / strike) /
    (periodVolatility * periodVolatility) + 0.5d) * dPlusBar;
  double[] inputBar = new double[5];
  inputBar[4] = volatility * 0.5 / Math.sqrt(expiry)
    * periodVolatilityBar;
  inputBar[3] = -1.0d / strike / periodVolatility * dPlusBar
    + numeraire * omega * -nMinus * priceBar;
  inputBar[2] = omega * (forward * nPlus - strike * nMinus)
    * priceBar;
  inputBar[1] = Math.sqrt(expiry) * periodVolatilityBar;
  inputBar[0] = 1.0d / forward / periodVolatility * dPlusBar
    + numeraire * omega * nPlus * priceBar;
  return new DoubleDerivatives(price, inputBar);
}
```

If you run that code and put a break-point before returning the result, you will notice that for any input, the value of the variable `dPlusBar` is always 0. That is where the subject matter expert eye can make a difference with respect to an automatic differentiation – see Chapter 4 for more on automatic differentiation, including for the Black formula. What is the d_+ variable? This is the number obtained by solving the equation indicating for which value of the underlying normal random variable, the strike is achieved. Take the example of a call. If the option is exercised at a lower value, we exercise in some case with a loss; if the option is exercised only for a higher value, we do not take full advantage of the option and do not replicate its full value and we have a 0 profit while we could have a positive one. In conclusion, the value d_+ is the optimal value at which the exercise boundary should be placed. Optimal for a one dimensional function in the interior of its domain means – subject to differentiability – that the derivative is 0. This is exactly the impact of this theoretical result that we see in practice. The derivative of the price – the output – with respect to the variable d_+ is always 0; locally the final result is not sensitive to changes in d_+. That derivative, in our code implementation, is the variable `dPlusBar`.

With our subject matter expert knowledge applied to the above code, we can remove the variable `dPlusBar` from the code and remove all the code referring to it, replacing it by 0. This required human eyes and subject expertise; it is very difficult to reproduce in an automated way. Implementing those changes, we obtain the code of Listing 3.3.

Listing 3.3 Implementation of the Black 76 formula in adjoint mode with code optimization

```
DoubleDerivatives price_Aad_Optimized(double forward,
    double volatility, double numeraire, double strike,
    double expiry, boolean isCall) {
  // Forward sweep - function - see above
  ...
  // Backward sweep - derivatives
  double priceBar = 1.0;
  double nMinusBar = numeraire * omega * -strike * priceBar;
  double dMinusBar = NORMAL.getPDF(omega * dMinus)
    * omega * nMinusBar;
  double periodVolatilityBar = -1.0d * dMinusBar;
  double[] inputBar = new double[5];
  inputBar[4] = volatility * 0.5 / sqrtExpiry
    * periodVolatilityBar;
  inputBar[3] = numeraire * omega * -nMinus * priceBar;
  inputBar[2] = omega * (forward * nPlus - strike * nMinus)
    * priceBar;
  inputBar[1] = sqrtExpiry * periodVolatilityBar;
  inputBar[0] = numeraire * omega * nPlus * priceBar;
  return new DoubleDerivatives(price, inputBar);
}
```

With a little bit of thinking, we have been able to reduce the number of lines of the code. Does that reduction translate into a significant impact on the performance? The data to answer to this question is displayed in Table 3.1.

We now move to the performance analysis of the different implementations. The Black function and the three AD implementations described above are run one million times each. The AD implementations provide the value itself and the five derivatives as outputs. The first implementation is a standard AD version. The code is not detailed here but can be found in the open source code repository associated to the book. The two next implementations are the adjoint AD versions described above. Note that in the code, there are two implementations of the Black function and its AD versions: one called `BlackFunction` where the *Apache Commons Mathematics Library*[1] is used as underlying mathematical library and a second one called `BlackFormula2` where the *Colt mathematics library*[2] is used. The mathematical library is used for the cumulative normal distribution function and the

[1] http://commons.apache.org/proper/commons-math/
[2] https://dst.lbl.gov/ACSSoftware/colt/index.html

Table 3.1 Computation time for different implementations of the Black function and five derivatives (with respect to forward, volatility, numeraire, strike, and expiry)

Repetitions	Computation	Time	Ratio
100,000	Black function	75	1.00
100,000	Black function + SAD	230	3.07
100,000	Black function + AAD	180	2.40
100,000	Black function + AAD Optimized	125	1.67

Figures in milliseconds. Each repetition is with 5 data sets and call/put so that there is actually 1,000,000 repetitions.

density function. The figures reported in Table 3.1 are the one obtained with the Colt version.

The first point to notice is that for all versions, the performance ratio is below or close to three. A finite difference implementation, even the fastest one, would have a ratio of at least 6. Any AD implementation improves on any finite difference version. Now you may point out that what is often required is not all the sensitivities, but only a limited number of them. In the case of the Black formula, you may be interested only in the price and the hedging ratio for delta hedging. Note that computing those numbers requires two pricing, this is a computation time ratio of 2. The optimized adjoint version has a ration of 1.67! Even for the case you need only one sensitivity, computing all of them by AD is faster than only one by finite difference.

The optimized adjoint version improves significantly on the non-optimized adjoint version, with the computation time ratio decreasing from 2.40 to 1.67. This may appear as a magical one-off trick. It is actually a very generic technique that we will make more formal when we discuss calibration in later chapters. Moreover, a lot of explicit pricing mechanism in quantitative finance are based on Black-like formulas. The same improvement will be welcome when we discuss SABR pricing by implied volatility in the next section. It also apply to swaption pricing in Hull-White one factor model (see Henrard 2003), valuation of quality option for bond futures, and other cases where an exercise boundary plays a role.

3.2 SABR

The SABR acronym stands for Stochastic Alpha Beta Rho model (see Hagan et al. (2002)). This is a stochastic volatility model often used in interest rate modeling. The parameters of the model are the starting volatility level α, the elasticity coefficient β, the correlation ρ between the forward Brownian motion W_t^1 and the volatility Brownian motions W_t^2 and the volatility of volatility ν.

The equations for the model are

$$\begin{aligned} dS_t &= \alpha_t (S_t)^\beta dW_t^1 \\ d\alpha_t &= \nu \alpha_t dW_t^2 \end{aligned}$$

with covariance $[W^1, W^2]_t = \rho t$.

A special feature of one of the implementation of the model makes it very suitable as a starting point of AD analysis. In the Hagan et al. (2002) implementation, the price of a call or put in this model is approximated by a Black formula, as described in the previous section, with an implied volatility provided by the formula below. The option strike is denoted K, the expiry θ, and the current forward F. The implied volatility is given by

$$\sigma(F,\alpha,\beta,\rho,\nu,K,\theta) = \frac{\alpha}{(FK)^{(1-\beta)/2}\left(1+\frac{(1-\beta)^2}{24}\ln^2\frac{F}{K}+\frac{(1-\beta)^4}{1920}\ln^4\frac{F}{K}\right)}\frac{z}{x(z)} \left(1+\left(\frac{(1-\beta)^2}{24}\frac{\alpha^2}{(FK)^{1-\beta}}+\frac{1}{4}\frac{\rho\beta\nu\alpha}{(FK)^{(1-\beta)/2}}+\frac{2-3\rho^2}{24}\nu^2\right)\theta\right) \tag{3.2}$$

where

$$z = \frac{\nu}{\alpha}(FK)^{(1-\beta)/2}\ln\frac{F}{K}$$

and

$$x(z) = \ln\left(\frac{\sqrt{1-2\rho z+z^2}+z-\rho}{1-\rho}\right).$$

The pricing formula is written by using an implied volatility approach. The number computed above has to be used as implied volatility for the Black formula from the previous section. It is an example of "wrong number in the wrong formula" to obtain the correct price. Independently of what is wrong and what is correct in that approach, it is a perfect example of composition on which the AD framework is based and for which it excels. The pricing formula in SABR is

$$\text{SABR}(F,\alpha,\beta,\rho,\nu,P,K,\theta) = \text{Black}(F,\sigma(F,\alpha,\beta,\rho,\nu,K,\theta),P,K,\theta).$$

The implementation of the formula (3.2) is straightforward and reproduced in Listing 3.4. The main point of attention I want to emphasize here is the ratio $z/x(z)$ when z is close to 0, this is when the forward F is close to the strike K. The ratio is of the type "0/0" and converge to 1 at that point. The singularity at that point has no financial significance, it is just an artifact on the way the approximation is obtained and the function should be replaced by its limit for that singular point.

As mentioned in the Principles chapter, one has to be careful about branching as the theoretical differentiability is not guaranteed on the branch boundaries. Here the branching is very special as it is a branching for only one point, when $z = 0$ – in practice when z is close to 0. Actually there is more trouble as the limit in $z = 0$ is of the type "0/0." The phenomenons on numerical instability we discussed in

Listing 3.4 Implementation of the SABR implied volatility approximation

```
private static final double Z_RANGE = 1.0E-6;
double volatility(double forward, double alpha, double beta,
      double rho, double nu, double strike, double expiry) {
  double beta1 = 1.0d - beta;
  double fKbeta = Math.pow(forward * strike, 0.5 * beta1);
  double logfK = Math.log(forward / strike);
  double z = nu / alpha * fKbeta * logfK;
  double zxz;
  double xz = 0.0d;
  double sqz = 0.0d;
  if(Math.abs(z) < Z_RANGE) { // z~0, 1st order approx. for x/x(z)
    zxz = 1.0d - 0.5 * z * rho;
  } else {
    sqz = Math.sqrt(1.0d - 2.0d * rho * z + z * z);
    xz = Math.log((sqz + z - rho) / ( 1.0d - rho));
    zxz = z / xz;
  }
  double beta24 = beta1 * beta1 / 24.0d;
  double beta1920 = beta1 * beta1 * beta1 * beta1 / 1920.0d;
  double logfK2 = logfK *logfK;
  double factor11 = beta24 * logfK2;
  double factor12 = beta1920 * logfK2 * logfK2;
  double num1 = (1 + factor11 + factor12);
  double factor1 = alpha / (fKbeta * num1);
  double factor31 = beta24 * alpha * alpha / (fKbeta * fKbeta);
  double factor32 = 0.25d * rho * beta * nu * alpha / fKbeta;
  double factor33 = (2.0d - 3.0d * rho * rho) / 24.0d * nu * nu;
  double factor3 = 1 + (factor31 + factor32 + factor33) * expiry;
  return factor1 * zxz * factor3;
}
```

Section 1.5 applies here also. It is not enough to simply branch for the value 0, we need a protection around it to avoid a picture like the one provided in Fig. 1.2. In our code, we have selected – arbitrarily – the range to be `Z_RANGE` = 1.0E-6. In that range the value will not be computed by the $z/x(z)$ formula but by an explicit approximation.

Unfortunately this introduces another problem. If we simply fill the range with the constant 1.0, which is the limit in 0, we have a branch for which the function $z/x(z)$ is a constant function of z and thus its derivative is 0. We get an incorrect derivative at one of the most important point, the at-the-money (ATM) strike. A way around this is to not only incorporate the value 1.0 at that point but also its first order approximation. In the branch we use the function $1 - \rho/2z$ as an approximation of the value of $z/x(z)$. In that way we have the correct first order derivative with respect to z at that point. There will still be a slight jump at the branching in z=1.0E-6, but at least the shape is correct to the first order.

With this branching in mind, the code for the adjoint AD version is slightly more complex that initially hoped for but still very manageable. The branching will appear also in the backward sweep, but for the rest there is nothing special in this code. The highlights of the code are provided in Listing 3.5. As usual the full code is provided in the open source repository associated to the book.

Listing 3.5 Implementation of the SABR implied volatility approximation: adjoint AD version

```
DoubleDerivatives volatility_Aad(double forward, double alpha,
    double beta, double rho, double nu, double strike,
    double expiry) {
  // Forward sweep - function - see above
  // Backward sweep - derivatives
  double volatilityBar = 1.0d;
  double factor3Bar = factor1 * zxz * volatilityBar;
  ...
  double zBar;
  double xzBar = 0.0d;
  double sqzBar = 0.0d;
  if(Math.abs(z) < Z_RANGE) {
    zBar = 0.5 * rho * zxzBar;
  } else {
    xzBar = -z / (xz * xz) * zxzBar;
    sqzBar = xzBar / (sqz + z - rho);
    zBar = zxzBar / xz;
    zBar += xzBar / (sqz + z - rho);
    zBar += (-rho + z) / sqz * sqzBar;
  }
  double logfKBar = nu / alpha * fKbeta * zBar;
  ...
  inputBar[5] += 0.5 * beta1 * fKbeta / strike * fKbetaBar;
  inputBar[5] += -logfKBar / strike;
  inputBar[6] += (factor31 + factor32 + factor33) * factor3Bar;
  return new DoubleDerivatives(volatility, inputBar);
}
```

All the discussion above concerns the implied volatility part. We now have to price the option through composition. This is done in Listing 3.6. We have directly provided the adjoint AD version. The value part is made of two lines, the computation of the implied volatility in the first and the computation of the price based on the Black formula using the volatility as an input in the second.

The adjoint AD is almost as simple, the volatility derivative is provided by the adjoint AD version of the Black formula price. Then the sensitivities to the different inputs are collected from the derivatives of the two functions used in the pricing. When the composition is used not only for each line of code but for the call to intermediary methods, the derivatives are not derivatives of elementary functions anymore, but the derivatives provided by the AD versions of the lower level methods.

Listing 3.6 Implementation of the SABR price through implied volatility

```
DoubleDerivatives price_Aad(double forward, double alpha,
    double beta, double rho, double nu, double numeraire,
    double strike, double expiry, boolean isCall) {
  DoubleDerivatives volatility = SabrVolatilityFormula
    .volatility_Aad(forward, alpha, beta, rho, nu,
    strike, expiry);
  DoubleDerivatives price = BlackFormula.price_Aad_Optimized(
    forward, volatility.value(), numeraire, strike,
    expiry, isCall);
  double priceBar = 1.0d;
  double volatilityBar = price.derivatives()[1];
  double[] inputBar = new double[8];
  inputBar[7] += price.derivatives()[4] * priceBar;
  inputBar[7] += volatility.derivatives()[6] * volatilityBar;
  inputBar[6] += price.derivatives()[3] * priceBar;
  inputBar[6] += volatility.derivatives()[5] * volatilityBar;
  inputBar[5] += price.derivatives()[2] * priceBar;
  inputBar[4] += volatility.derivatives()[4] * volatilityBar;
  inputBar[3] += volatility.derivatives()[3] * volatilityBar;
  inputBar[2] += volatility.derivatives()[2] * volatilityBar;
  inputBar[1] += volatility.derivatives()[1] * volatilityBar;
  inputBar[0] += price.derivatives()[0] * priceBar;
  inputBar[0] += volatility.derivatives()[0] * volatilityBar;
  return new DoubleDerivatives(price.value(), inputBar);
}
```

Note that at this stage, when we are running the code of the external function – SABR price in our case we don't need to know what happened in the internal computation of the underlying code. How the Black formula price is computed is irrelevant, only the final results of that method – the value and its derivatives – and the memory space to hold them is relevant. This will not be the case for the automatic AD by operator overloading described in Chapter 4. The standard implementation in that case uses a *tape* that records every elementary operation, including those inside the lower level methods. The recorded information is reused at a later stage when the derivatives are computed. In some sense the tape creates an in memory explicit version of the computation as a SAC list. The memory impact of automatic adjoint AD is one of its main drawbacks.

Like for the Black formula, we can check the performance of the different implementations of the SABR price with approximated implied volatility. The number of inputs is now eight. The results are presented in Table 3.2. The results were obtained by running the analysis code `SabrFormulaAnalysis`.

Again the performance is quite impressive with the computation of the value and the 8 derivatives taking less than twice the computation time of one value. As for Black formula, computing one sensitivity by finite difference would be slower than computing all the 8 sensitivities by AD.

Table 3.2 Computation time for the SABR price and eight derivatives (with respect to forward, alpha, beta, rho, nu, numeraire, strike, and expiry)

Repetitions	Computation	Time	Ratio
100,000	SABR price	240	1.00
100,000	SABR price + AAD Optimized	440	1.83

Figures in milliseconds. Each repetition is with 5 data sets and call/put so that there is actually 1,000,000 repetitions.

Note that in practice this formula is often used with interest rate products. It means that the inputs of the above formula, like the forward rate F and the numeraire P are themselves output of previous methods taking 10–100 inputs. In practice the saving are compounded with the saving described in the next section. This accumulation of saving multipliers make the technique extremely efficient, even for vanilla interest rate products like swaptions.

3.3 Coupon Sensitivities

Where we encounter sensitivities that are financially useful but ADly complex to extract, sensitivities that have weak financial meaning and ADly complex to extract and finally sensitivities that are financially useful and directly available in all AD implementations.

In this section, we analyze interest methods related to present value and its derivatives for interest rate swaps. In the finance jargon, the derivatives are often called *sensitivities*, we use that name in this section. The sensitivities are to the curves – discounting and forward – used to value the swaps. The meaning of sensitivities to curves is explained below. The instruments we analyze are composed of fixed coupons and Ibor coupons. We work in a multi-curve framework as described in Henrard (2014). Here we only sketch the main financial problem and the important definitions. Readers unfamiliar with the multi-curve framework fascinating world are referred to the above reference for more details.

The challenges would not be very different in a simpler one curve framework, which was the main interest rate framework for derivatives before the crisis, but we prefer to present a more recent framework. In term of efficiency, the AD implementation appears even more spectacular in the multi-curve framework as there are more inputs and more sensitivities to compute. In a typical multi-curve framework for one currency, there would be four or five curves, each calibrated to 10–25 instruments. A typical simple vanilla single currency interest rate book will exhibit sensitivities to 50–125 inputs.

The first challenge when working with curves is the dimensionality. In the theoretical description of our function by Eq. 2.3 and the code description in Table 2.1, all the functions have a finite number of arguments. From a theoretical point of view a curve $f : [0, T] \to \mathbb{R}; t \mapsto f(t)$, like the one used to represent the discount factors, is an element of an infinite dimensional space, the space of functions from an interval

to $\mathbb{R}$. Obviously the element of the space is represented in the code by a finite number of data that are used to construct the element of the infinite dimensional space. The usual representation of a curve is through the nodes of an interpolated curve or the parameters of a functional curve. But formally the problem we are facing involves a function from a finite number of data to an element of an infinite dimension space composed with a function from the infinite dimension space to the one dimensional present value. We will need some mechanism to deal with this infinite dimensional space.

Before developing the algorithm differentiation challenges further, we describe the notations and main features of the multi-curve framework. Let $P_X^D(v)$ be the discount factor to the valuation time in currency X for a payment at time v. The present value of a fixed coupon paying in time v a rate r on a notional N with accrual factor δ is given by

$$\text{PV} = NP_X^D(v)\delta r.$$

The discount factor function is a function $P_X^D : [0, T] \to \mathbb{R}; t \mapsto P_X^D(t)$.

We call a j-Ibor floating coupon a financial instrument which pays at the end of the underlying period the Ibor rate set at the start of the period. The details of the instrument are as follow. The rate is set or fixed at a date t_0 for the period $[u, v](t_0 \leq u < v)$ of length equal to the tenor of the Ibor index j, at the end date v the amount paid in currency X is the Ibor fixing $I_X^j(t_0)$ multiplied by the conventional accrual factor δ for the period. The lag between t_0 and u is called the *spot lag* or *settlement lag*. The difference between u and v is the tenor of the index j. All periods and accrual factors should be calculated according to the day count, business day convention, calendar and end-of-month rule appropriate to the relevant Ibor index.

The present value of the Ibor coupon paying in v the rate fixed in t_0 for the period $[u, v]$ on the notional N for an accrual factor δ is given by

$$\text{PV} = NP_X^D(v)\delta F^j(t_0, u, v)$$

where $F^j(t_0, u, v)$ is called the forward rate for index j viewed from the valuation date. The forward rate function is a function $F^j : [0, T] \to \mathbb{R}; t \mapsto F^j(t, u(t), v(t))$. The dates u and v are duplicate information, as they can be inferred from t and the convention of the index j. They are usually provided in literature related to interest rate for the reader convenience but could be simply ignored. Looking at the multi-curve foundations, the function depends only of the fixing date and is thus a one dimensional function.

We have described the two main ingredients of an interest rate swaps: the fixed coupons and the Ibor coupons. A full swap is simply a combination of those two types of coupons, one type on each leg. Its total present value is obtain as the sum of the present values of the different coupons. The "sum" part does not cause any problem for the AD framework.

Before starting the implementation, and in particular to review how to work around the infinite dimension problem in practice, we have to ask ourself "What are the derivatives (sensitivities) that a risk manger would like to see in such circumstances?" I can see three different, and non-exclusive, answers to that question.

The first possible answer is that we would like to see the sensitivity to each discount factor and forward rate. This would be the sensitivity for each discount factor at time v at which there is a payment and for each forward rate at time t_0 at which there is a fixing. This is where the infinite dimensional question enter into play. You cannot create a vector of all possible times and fill the sensitivity to the relevant ones with a number and leave the rest at 0. There is no exhaustive list of times for which one may want to compute this, a priori any time between 0 and T is a possible payment or fixing time.

One can claim that in practice, the fixing and payment dates are only precise to the date level, so the values are not any real number in the interval $[0, T]$ but in a discrete subset of it representing each day in the period. Even if we take only good business days into account, for a 50-year curve, this means a little bit more than 12,500 potential times. Would you like to carry around an array of that size when you have only a couple of fixings or payments? This would probably be a huge cost in memory management. Moreover the argument may be valid for payment time, but when we will move to options, there is a clear expiry time and one would like to model this down to the hours and minutes. An option with 23h00 to go is not the same as an option one hour after expiry! Often the implied volatilities used with Black-like formulas are time dependent and there a full continuous time dimension is required. The mechanism we put in place to deal with the infinite dimension of the interest rate curve – or finite with very large dimension – will be useful in other circumstances later.

In conclusion, due to the very large discrete potential times or due to the really continuous feature of the time, we may want to represent the sensitivities not through an array of fixed length. We have to move away from the simplified framework of the AD we have worked with until now and accept that some information is not simply represented by a double or an array of them. Our suggestion in the case of interest rate sensitivities is to use lists, i.e. a collection without an a priori size. But list of what? In one of our implementations[3], we have used the objects `ZeroRateSensitivity` for sensitivities associated to discount factors and `IborRateSensitivity` for sensitivities associated to forward Ibor rates.

By tradition, instead of representing the sensitivity in term of sensitivity to the discount factor, it is represented as the sensitivity to the zero-coupon rate $r(t)$

[3] The implementation is the *Strata* framework developed by OpenGamma. The code is available under an OpenSource license at https://github.com/OpenGamma/Strata.

associated to it by $P_X^D(t) = \exp(-r(t)t)$. The sensitivity object for one cash flow is represented in Listing 3.7[4].

Listing 3.7 Object for sensitivities to discount factors

```
public class ZeroRateSensitivity implements PointSensitivity {
  Currency ccyDiscount;
  LocalDate date;
  double value;
  Currency ccySensitivity;
  ...
}
```

Note that there are two currencies, one for the currency in which the discounting is performed, i.e. the X in the formula and one for the unit in which the sensitivity is represented. There is good argument to claim that one currency would be enough for discount factor sensitivity and I'm ready to accept those arguments, even if in some cases you may want to express all the results in a common currency or convert from one currency to another.

But in the case of the sensitivity to forward Ibor rate, which is the object represented in Listing 3.8, one definitively need two information: the index of the forward and the currency in which the amount is paid. Even if a currency is implicit in the index, we need a second currency to know in which unit the payment is done. This is important for quanto-like payments in particular.

Listing 3.8 Object for sensitivities to forward rates

```
public class IborRateSensitivity implements PointSensitivity {
  IborIndex index;
  LocalDate fixingDate;
  Currency currency;
  double sensitivity;
  ...
}
```

With each of those objects we can store the sensitivity to one discount factor or one forward rate. With a list of them, we can store as many sensitivities as necessary. An object collecting the sensitivities is summarized in Listing 3.9.

Note that at this stage the data is stored in a `List`, not an array. This is important to note in relation to our remarks above. There is no fixed size array for all the potential dates of a point sensitivity. The length of the `PointSensitivities`

[4] The implementation used in practice is slightly more complex than the one described in this text. In particular the actual implementation has features for serialization, builders and combination of objects. We emphasize here only the AD related features.

Listing 3.9 Object for point sensitivities in the multi-curve framework

```
public class PointSensitivities {
  List<PointSensitivity> sensitivities;
  ...
}
```

embedded list depends on the instrument priced and potentially the model and calibration procedure used.

With the information in that object, we have enough data to answer the questions like, to which date do I have discounting factor sensitivity or to which date do I have fixing risk. In terms of market risk, to my opinion, the most important is the fixing date sensitivity. As a fixing is a daily one-off there is limited diversification one can do, it is not possible to take the fixing from another source, like one can borrow from another counterpart, one cannot really postpone the fixing to the next day, while is it often possible to borrow overnight. On the other side, knowing the exact profit impact of each daily fixing would allow to foresee the impact of fixing manipulation. But I, naively, cannot believe that one would ever use AD for a such evil purpose as market manipulation.

We call the first level of sensitivity, as described above, the *point sensitivity*. Each potential point on the curves can produce a sensitivity. But this type of sensitivity would not be produced by an Algorithmic Differentiation sensu stricto. The sensitivities or derivatives computed are not with respect to an input but to an intermediary value, they can nevertheless be very useful and should not be excluded from a general financial library. Often in libraries where the sensitivities are produced by finite difference or automatic AD, those point sensitivities are unfortunately not available. In the case of finite difference they are not even computed, and in the case of automatic AD they may have been computed and recorded in the tape, but finding them may be more complex as they are not earmarked. We will come back to similar issues in Chapter 4 on automatic AD and in Chapter 5 on adaptation of AD to compute numbers closely related to derivatives.

The second level of sensitivities we produce is called *parameter sensitivity* or *internal representation sensitivity*. This is the sensitivity to the internal parameters used to represent the calibrated curves. The most commonly used curve descriptions is through interpolated zero-coupon curves, but many other representation are possible. Some useful representations are described in (Henrard 2014, Chapter 5). This sensitivity type is probably, or should be, the least useful of the three we describe here. It depends on the internal representation of the curve in the library. This internal representation is not the most useful to the business users. A curve can be represented by zero-coupon rates or discount factor; by changing the interpolation mechanism, both representations can produce exactly the same result. The internal representation would be invisible to an outside user. Providing the parameters sensitivity to the end user is providing number potentially widely different for the same inputs and same outputs; this can be surprising for a user that does not know about the internal working of the library.

In particular, the sensitivity to zero-coupons rates are often used as a risk measure. Zero-coupon rate are not defined for all curves. In particular forward Ibor curves described directly through forward rate and not through pseudo-discount factors do not have zero-rate representation. The theoretical description of those curves in the multi-curve framework is provided in (Henrard 2014, Chapter 3). It means that the parameters that are used to describe curves are in general not homogeneous and often don't have a direct intuitive meaning. Relying on those numbers for risk management may be unwise. The zero-coupon PV01 enter, to my opinion, in this category of ill-defined risk measures. Or more exactly is it defined only in a restricted set of curve description. Relying on them would restrict the risk manager view of the risk world.

The algorithmic differentiation meaning of the parameter sensitivities is clear, it is the derivatives of the final present value with respect to the parameters used to internally store the curve. It is the derivative to an intermediary variable. Like for the point sensitivities, how to extract the information computed as some stage of the process will depend on the implementation. But what the external business user can do with those numbers dependent of the internal representation and not of financial information is unclear.

In implementation terms, the parameter sensitivity can be computed as a *projection* of the point sensitivity to the parameters of the curves. By projection, we mean here multiplying the partial derivative of one of the functions composing the algorithm by the derivative of the remaining composed functions with respect to the intermediary variables used. A version of that approach can be found in the class `AbstractRatesProvider` of the OpenGamma's Strata library. This is a generic algorithm that applies to any point sensitivity. This part of the code is not instrument specific but curve description specific. Only a unique implementation in a library is required.

One of the important practical aspect of the parameter sensitivity is the object containing the sensitivity. From a pure data point of view, the result is simply a long vector; from a business point of view it is important to know to which curve and point each number refers and in which currency it is expressed. Moreover, the number of curves and currencies to which an instrument present value depends is a priori unknown. If the data is stored as a unique vector, it would need to be a vector with all the curves used in the system, not only the one related to the instrument analyzed. This would mean again a huge quantity of 0 stored. For that reason we suggest to store the parameter sensitivity in an object composed of a list of sensitivities to a single curve. Each single curve sensitivity contains the actual sensitivity vector, the currency in which the sensitivity is expressed and the curve metadata, i.e. its name and the description of the parameters. An example of such an object is provided in Listing 3.10.

The third level of sensitivity is the *market quote sensitivity*. Another term often used for those sensitivities is *par rate sensitivity*. I prefer to use the market quote terminology, as some instruments are quoted through par rate but others are quoted using other mechanism as spread, futures price or bond price. The par rate sensitivity term appears more restrictive and we would like to have an approach which is as generic as possible.

Listing 3.10 Object for parameter sensitivities in the multi-curve framework

```
public class CurveCurrencyParameterSensitivities {
  List<CurveCurrencyParameterSensitivity> sensitivities;
  ...
}

public class CurveCurrencyParameterSensitivity {
  CurveMetadata metadata;
  Currency currency;
  double[] sensitivity;
  ...
}
```

Computing the market quotes sensitivities means that the curve have been calibrated from market quotes. The curve calibration process is discussed in more details in Section 6.1. One of the output of that procedure is obviously the calibrated curves, i.e. the set of internal parameters describing the curves. Another very important output is the *transition* or *Jacobian* matrices. Those matrices are the partial derivatives of the internal parameters of the curves with respect to the market quotes of the instruments used to calibrate the curves.

With the description above, the numerical procedure to obtain the market quote sensitivities is direct. The derivative of the present value (or any other measure) with respect to the market quotes is the multiplication of the transition matrix by the parameter sensitivity computed at the previous step:

$$\frac{\partial \mathrm{PV}}{\partial q_i} = \sum_j \frac{\partial \mathrm{PV}}{\partial p_j} \frac{\partial p_j}{\partial q_i}.$$

Like for the parameter sensitivity, one of the challenges to the market quotes sensitivities is the object representing it. We suggest to use the same object than the one described in Listing 3.10. Even if the business description of the two results are not the same, the structure of the data is the same. The data has the same format, but the metadata will distinguish between the two types of sensitivities. For several curves, there is a sensitivity, represented by an array, to data in a given currency.

The object containing the *transition matrices* is not trivial. Each calibrated curve potentially depends on several market quote curves. The different curves can have different sizes. The order in which the curves are calibrated is itself important. An example of an object containing a representation of the transition matrices for one curve is proposed in Section 6.1 on curve calibration.

The performance obtained by this approach is presented in Table 3.3. In all cases, the sensitivities are computed in a time which is less than five times the time of the present value. The time presented in the table does not include the curve calibration.

Table 3.3 Computation time for the present value and different type of sensitivities. The multi-curve set has two curves and a total of 30 nodes

Repetitions	Computation	Time	Ratio
1,000	Swaps present value	195	1.00
1,000	Swaps point sensitivity	280	1.44
1,000	Swaps (point + parameter) sensitivity	715	3.67
1,000	Swaps (point + parameter + quotes) sensitivity	900	4.62

Figures in milliseconds. Each repetition is with three swaps with maturities 5, 10 and 30 years.

3.4 Monte Carlo

Monte Carlo pricing of financial derivatives is certainly a numerically challenging problem in quantitative finance. But the Algorithmic Differentiation version of it does not present any specific challenges on top of the pure pricing challenges. Obviously the different steps have to be treated carefully but they can be seen as a quite direct application of the AD methodologies.

A Monte-Carlo pricing for a European option can be described in several steps:

Random number Generate a set of random numbers $r_{f,s}$ where f $(1 \leq f \leq F)$ represents a factor of the model and s $(1 \leq s \leq S)$ a scenario.

Scenario From each scenario, generate the market at the expiry date θ. The scenario market will depend on the model parameters V_p $(1 \leq p \leq P)$ and the underlying $M_{0,u}$ $(1 \leq u \leq U)$ - yield curves for example - in a given numeraire

$$M_{\theta,s} = g_1(r_{.,s}, V, M_0).$$

where $M_{\theta,s} = (M_{\theta,s,u})_{u=1,\ldots,U}$ is a market vector for each scenario and $g_1 : \mathbb{R}^F \times \mathbb{R}^P \times \mathbb{R}^U \to \mathbb{R}^M$. Through the stochastic model, the market description is obtained from the initial market and the model parameters. The market is for example the full yield curves in an interest rate term structure model.

Numeraire and pay-off From the different markets, compute the pay-off and the numeraire

$$P_{\theta,s} = g_2(M_{\theta,s}) \quad \text{and} \quad N_{\theta,s} = g_3(M_{\theta,s}).$$

Average The present value is the average or expectation of the pay-off discounted by the numeraire:

$$\mathrm{PV}_0 = N_0 \frac{1}{S} \sum_{s=1,\ldots,S} \left(N_{\theta,s}^{-1} P_{\theta,s} \right).$$

where N_0 is the value in 0 of the numeraire.

The Adjoint Algorithmic differentiation start from the last step. The derivatives are computed with respect to the underlying parameters M_0 and the model parameters V

Average There is one sensitivity for each scenario s:

$$\bar{N}_{\theta,s} = -N_0 N_{\theta,s}^{-2} P_{\theta,s} \quad \text{and} \quad \bar{P}_{\theta,s} = N_0 N_{\theta,s}^{-1}.$$

Numeraire and pay-off For each of the dimension m representing the market and each scenario s we have one sensitivity:

$$\bar{M}_{\theta,s,m} = D_m g_2(M_{\theta,s}) \bar{P}_{\theta,s} + D_m g_3(M_{\theta,s}) \bar{N}_{\theta,s}.$$

Random number The starting parameters and curves are used in each scenario s and for each dimension of the underlying market m:

$$\bar{V}_p = \sum_{s=1,\dots,S} \sum_{m=1,\dots,M} D_{V_p} g_{1,m}(r_{.,s}, V, C) \bar{M}_{s,m}$$
$$\bar{C}_u = \sum_{s=1,\dots,S} \sum_{m=1,\dots,M} D_{C_u} g_{1,m}(r_{.,s}, V, C) \bar{M}_{s,m}$$

Even if the final formulas contains several layers of sums, they are very easy to compute provided than the derivatives of g_1, g_2 and g_3 can be computed. Obviously those functions will be instrument and model dependent. But in general they are relatively simple. For standard options, g_2 is often a piecewise linear function and g_3 is often a linear combination of market factors. The function g_1 is usually the most complex part, it describe how to generate future market data from current data and model parameters. But even in a Libor Market Model with the forward market generated by Euler or Predictor-Corrector approach, it can be decomposed in exponentials, divisions, multiplication and additions. The AD development for those functions is usually straightforward, even if tedious.

Bibliography

Bachelier, L. (1900). *Théorie de la Spéculation*. PhD thesis, Ecole Normale Supérieure.

Black, F. (1976). The pricing of commodity contracts. *Journal of Financial Economics*, 3(1–2):167—179.

Hagan, P., Kumar, D., Lesniewski, A., and Woodward, D. (2002). Managing smile risk. *Wilmott Magazine*, Sep, 2002(5):84–108.

Henrard, M. (2003). Explicit bond option and swaption formula in Heath-Jarrow-Morton one-factor model. *International Journal of Theoretical and Applied Finance*, 6(1):57–72.

Henrard, M. (2014). *Interest Rate Modelling in the Multi-curve Framework: Foundations, Evolution and Implementation*. Applied Quantitative Finance. London:Palgrave Macmillan. ISBN: 978-1-137-37465-3.

Chapter 4
Automatic Algorithmic Differentiation

Big brother is watching you: AD tapes – Automatic AD saves programing time, not execution time – Human are still useful to computers – On n'oublie rien de rien. On s'habitue c'est tout

On top of theoretical Algorithmic Differentiation as described in most of this book, there is also the computer science art of *Automatic Algorithmic Differentiation*. This is the art of writing code that performs the Algorithmic Differentiation automatically. The developments required are relatively heavy at the start, but from there on, there should be only a minimal development cost.

Automatic Algorithmic Differentiation is not the main subject this book. In most of the chapters, we discuss AD's principles and how it can be applied to finance. Nevertheless to understand the subject globally a minimum discussion on the automation is required.

Generally, automatic differentiation can be achieved in two ways. One approach is to generate the AD code automatically from the original function code. A second approach is by "operator overloading." The code is almost unchanged but the objects on which it operates are not primitive `double` anymore but *augmented doubles*. The augmented doubles contains the double information on its value and also some extra information required to perform the AD calculations automatically. Those augmented doubles can be created for Standard AD and Adjoint AD. In our implementation, for the standard version, the extra information is the array of already computed derivatives. For the adjoint version the extra information is the index of the operation in an AD *tape*. The notion of AD tape is described later in this chapter. In this book we chose to present the operator overloading approach. To be precise, as we use Java as our example language, we don't actually use operator overloading but operator replacement by methods. This is maybe not as neat when writing/reading the code. But on the other side it is very explicit on which parts are the simple direct code on primitive doubles and which parts are the overloaded versions on augmented doubles. From a didactical point of view it is important to be able to distinguish clearly between them. Operator overloading may appear in Java 10, in which case I would rewrite the code to use the new feature.

Many references regarding the computer science art of automatic Algorithmic Differentiation can be found in the recent book Naumann (2012). The website `http://www.autodiff.org/` can also be used as a starting point for many tools and references.

M. Henrard, *Algorithmic Differentiation in Finance Explained*,
Financial Engineering Explained, DOI 10.1007/978-3-319-53979-9_4

In the advanced chapters of this book we argue that Algorithmic Differentiation done by subject matter experts can improve the already amazing performance of Algorithmic Differentiation. The automatic differentiation is certainly a very useful approach and once fully implemented can save a lot of development time. But there is more to algorithmic differentitation than simple and blind application of rules without an expert eye.

Regarding the efficiency of AD with an expert eye on the implementation, we will show in the "at best" calibration cases described in a later chapter, that the derivatives of complex process, like pricing complex derivative with ad hoc model calibration, can be obtained for almost free. The ratio between the price alone and the (numerous) derivative will be barely above 1; in the region of 1.1. To compute the price plus 100 of its derivative, the computation time is the one of the price plus 10%. You get 100 extra numbers in 10% of the time required for the first number! This was not achieve by a generic AD implementation that reproduces in the derivative computation the algorithm of the pricing but a specific AD computation that take into account the structure of the problem.

In Naumann and du Toit (2014), the author states "*Doing AD by hand on any production size code is simply not feasible and is incompatible with modern software engineering strategies.*" I disagree with this statement for the following reason. The *philosophy* of AD is to use the composition (see Section 2.2). Each part of the composition need to have a derivative version. It should be mainly irrelevant if each block of the composition is written automatically or manually. Disregarding form the start any implementation approach, it being manual or automatic (or finite difference), as part of a general AD is incoherent with my view of the general AD philosophy. The exclusion of an approach under the authority argument that is not "modern" is not a strong argument. A general AD implementation should accept any implementation for each part of the composition. An implementation that accepts only one specific type of implementation for each part would be a weak one. To have the best team you need to be allowed to take the best element for each position and make sure they work together smoothly, this is the case for AD as for team sport. We will present later examples of mixed implementations involving both manual and automatic AD in the same computation.

One of the drawbacks of fully automatic AD, is that it requires full access to source code, even for third party libraries. It is not enough to have a derivative version of each method; it has to be in the exact same code approach. The manual implementation does not have this problem, different approaches can be combined for each line of code. By extension an approach that combines automatic differentiation with manual glue will allow diversity and efficiency.

Another of the drawbacks of automatic AD is the memory requirements. For example in Naumann and du Toit (2014), the authors states "*Moreover, AD implementations based on operator overloading techniques (such as dco/c++) amplify this problem since they need to store and recover at runtime an image of the entire computation (commonly referred to as the tape) in order to ensure the correctness of the calculation.*" Roughly speaking standard automatic AD with operator overloading record each code operation with entries logging the type of operation, the

number involved, the value computed and, in a second phase, the derivatives. The record of all the those operations is called the AD tape. At the very end of the computation, that very long list of values and derivatives can be replayed backward to get the derivatives results. In any case, all the steps required for the final computation, even if there are many of them, are kept in memory up to the very end of the process to achieve the derivative computation. The more manual implementation can release memory at each stage of the computation if the intermediary numbers are not used any more. The intermediary garbage can be collected when not used anymore.

The memory drawback is also accompanied by a lost performance. Not only the results and derivatives are recorded but also all the description on how the results are obtained. Manipulating all this data is by itself time consuming. In some of the examples we will compare the time required to do the computation of a value and no derivative with primitive doubles to the time with augmented doubles. This shows the cost of the operator overloading which is not linked to the computation of derivatives itself but to the gathering of data required to do it automatically.

Obviously different implementation will have different costs. As an indicative level, for our implementation of automatic Adjoint AD, performing the computation with the augmented doubles instead of the simple doubles multiply the computation time by a factor of two to five. This is only for the recording of the operation and not the computation of the derivatives. The total cost of the recording and the computation of the derivatives is between two and ten. The theoretical results indicate an upper bound for the ratio around three to four. The implementations by subject matter experts as discussed in different places in this book provide ratios between 1.1 and 4.

A general Automatic Algorithmic Differentiation tool is proposed in Naumann and du Toit (2014). Some part of it are also described in Naumann (2012). The C++ implementation is called `dco/c++` (derivative computation by overloading) and is developed at RWTH Aachen University in collaboration with NAG. In the financial library sector, the vendor Fincad also propose a variation of AD that they call Universal Algorithmic Differentiation™. This last implementation being closed source, it is difficult to assess to which extend it differs or overlap with the description done in this book. Another automatic AD implementation is TapeScript[1]. TapeScript is an open source library for adjoint algorithmic differentiation (AAD) developed and maintained by CompatibL

4.1 Standard Algorithmic Differentiation by Operator Overloading

The automatic algorithmic differentiation approach by operator overloading applies the standard code to *augmented objects*, while the manual AD approach applies

[1] https://github.com/compatibl/tapescript

augmented code to standard objects. Those augmented objects need to be created carefully.

A potential implementation for an object storing a value and its derivatives, suitable for automatic standard AD, is proposed in Listing 4.1. The code of the object can be found in the GitHub repository described in the introduction in the package `type`. The object is called `DoubleSad`. This is an augmented `double` to be used in Standard Algorithmic Differentiation. In standard AD, at each step the derivatives with respect to each input is computed. The number of inputs is known and an array of the correct size can be carried around. The computation of the derivatives is done at the same time as the computation of the value itself.

Listing 4.1 Object storing information for Standard Algorithmic Differentiation by operator overloading

```
public class DoubleSad {
  private final double value;
  private final int nbDerivatives;
  private final double[] derivatives;

  public DoubleSad(double value, double[] derivatives) {
    this.value = value;
    this.derivatives = derivatives;
    nbDerivatives = derivatives.length;
  }
  ...
  static public DoubleSad[] init(double[] inputs) {
    int nbInputs = inputs.length;
    DoubleSad[] init = new DoubleSad[nbInputs];
    for(int loopi = 0; loopi < nbInputs; loopi++) {
      double[] initDot = new double[nbInputs];
      initDot[loopi] = 1.0d;
      init[loopi] =  new DoubleSad(inputs[loopi], initDot);
    }
    return init;
  }
}
```

The first step in each computation is to initialize the different `DoubleSad` with the input values and the relevant derivatives. The derivatives are trivial as the derivative of an input with respect to the input itself is 1.0 and with respect to the other inputs is 0.0. This initialization is done by the `init` method.

For each elementary operation, like addition, subtraction, multiplication, sinus, exponential, power, normal cumulative density, etc. we have to explain how the operation acts on the value and on its derivatives. This is done through the different static methods with the name of the operation stored in the `MathAad` class. Some of the method are provided in Listing 4.2. For example the addition is described by the `plus` method. The addition of two `DoubleSad` is done by adding the values, which are simple doubles and by adding the derivatives arrays element by element.

Similar descriptions are required for each operation (subtraction, multiplication, sine, cosine, exponential, logarithm, square root, etc.). The code is not presented for the other operation, as it would be a very long listing without real didactic value. The full code can be found in the open source repository.

Listing 4.2 Mathematical operations for Standard Algorithmic Differentiation by operator overloading

```
public class MathSad {

  public static DoubleSad plus(DoubleSad d1, DoubleSad d2) {
    int nbDerivatives = d1.getNbDerivatives();
    double valueOutput = d1.value() + d2.value();
    double[] derivativesOutput = new double[nbDerivatives];
    for (int loopd = 0; loopd < nbDerivatives; loopd++) {
      derivativesOutput[loopd] =
        d1.derivatives()[loopd] + d2.derivatives()[loopd];
    }
    return new DoubleSad(valueOutput, derivativesOutput);
  }
  ...
}
```

Using that object, the code for the initial test function described by Listing 2.1 in the previous chapter can be adapted easily. The code for the automatic Standard Algorithmic Differentiation version is given in Listing 4.3. With respect to the original code, the operator – like + – have been replaced by the method of the similar name and the mathematical functions – like exponential – have also been replaced by a method of similar name but in the `MathSad` class instead of the `Math` class. For the rest the code looks very similar.

Listing 4.3 Simple function code by Automatic Standard Algorithmic Differentiation

```
static public DoubleSad f_Sad_Automatic(DoubleSad[] a) {
  DoubleSad b1 = MathSad.plus(a[0], MathSad.exp(a[1]));
  DoubleSad b2 = MathSad.plus(MathSad.sin(a[2]), MathSad.cos(a[3]));
  DoubleSad b3 = MathSad.plus(MathSad.pow(a[1], 1.5d), a[3]);
  return MathSad.plus(MathSad.multipliedBy(MathSad.cos(b1), b2), b3);
}
```

As the algorithm is progressing through the code, the value is computed and stored in the `value` part of the `DoubleSad`. The derivatives are computed at the same time and stored in the `derivatives` array part of the augmented double.

Performance results of this implementation will be provided later in the chapter, in particular in Table 4.2, when we compare them with different versions of manual and automatic Adjoint AD.

4.2 Adjoint Algorithmic Differentiation by Operator Overloading

The implementation of the adjoint mode is usually technically more complex. The reason is that each line of code is analyzed twice, once in the forward sweep where the value is computed and once in the backward sweep where the derivatives are computed. It means that all the operations done in the original code through the forward sweep have to be recorded and to be replayed later. This recording of operations is not only at the level of one method but through the full code, between the different methods. If two methods are composed, the recording will contain the full list of all operations in both methods. The accumulation of basic operations span the full code run.

The list of recorded operations is usually called the *tape*. The tape contains the list of all elementary operations performed and enough information to know to which variables the operations applied. The unit for the recording is not the *line of code* as in the high level language but at the level of more fundamental operations within one line of code. Each operation acts on 0, 1 or 2 variables. In our example implementation, we have implemented the following fundamental operations: `INPUT`, `MANUAL`, `ADDITION`, `ADDITION1`, `SUBTRACTION`, `MULTIPLICATION`, `MULTIPLICATION1`, `DIVISION`, `SIN`, `COS`, `EXP`, `LOG`, `SQRT`, `POW`, `POW1`, `NORMALCDF`. For most of them, the name is explicit enough and they do not require more explanation. The ones requiring more explanations are explained below.

`INPUT`: The fundamental variables with respect to which the derivatives need to be computed, this is the starting point of the recording and at the same time the end point of the backward sweep. Once the backward algorithm reaches one of those record, nothing else need to be done with that information, the required derivative is obtained. The tape will keep the records available for future extraction.

`MANUAL`: This is the marker in the automatic AD to indicate non-automatic computations, i.e. that some computation of derivatives have been done in a different way to the current implementation of automatic AD and the result obtained with a different format is fed back into the tape. To my opinion, the main principles of AD is composition of functions and its derivatives. A composition can be achieved by combining adjoint, standard, automatic or manual implementation. Those different approaches should not be exclusive from each other. We will come back to this special operation when analyzing mixed implementations later in the chapter.

`ADDITION1`, `MULTIPLICATION1` and `POW1`: When adding, multiplying or taking the power of numbers, two possibilities arise. If the two numbers are *augmented doubles*, the derivatives with respect to both is required. On the other side, if one of the number is a constant, not dependent of the inputs, the derivative with respect to only one of them is required. The "1" versions of the three operations above are to be used if only the first

element is a augmented double and the other one is a primitive double – a constant for derivatives computation purposes.

The operations for the adjoint AD are performed on augmented doubles that we call `DoubleAad` in the code associated to the book. A shorten version of the code, where only a couple of the ingredients are provided, can be found in Listing 4.4. The full code can be found on the GitHub repository. The information recorded in that object is the value itself and the *index*, *virtual address* or *recording number* in the tape where the operations that led to that result are located. The number called *tape index* in this book is also called *virtual address* in the literature.

Listing 4.4 Object storing information for Adjoint Algorithmic Differentiation by operator overloading

```
public class DoubleAad {
  private final double value;
  private final int tapeIndex;

  public DoubleAad(double value, int tapeIndex) {
    this.value = value;
    this.tapeIndex = tapeIndex;
  }
}
```

For each operation done on those numbers, an entry is recorded in the tape. The description of the tape entry is provided in Listing 4.5. The first information recorded is the type of basic operation performed. The list of those operations was provided above and in the code is described as an `enum`. The operations are elementary operations and operate on zero, one or two arguments. The tape index of the arguments are recorded in the `indexArg1` and `indexArg2` fields. If one or both arguments are irrelevant, the tape index is filled with -1. The next information is the value itself, this is the result of the operation. The *extra value* is some supplementary information required in some cases. This can be the multiplicative factor for the `MULTIPLICATION1`. It is also used for `MANUAL` entries as explained in a subsequent section.

Listing 4.5 Tape entry for Adjoint Algorithmic Differentiation by operator overloading

```
public class TapeEntryAad {
  private final OperationTypeAad operationType;
  private final int indexArg1;
  private final int indexArg2;
  private final double value;
  private final double extraValue;
  private double valueBar;
  ...
}
```

The last information is the value of the derivative of the final output with respect to the variable represented by the entry. It is called `valueBar` in the code to remind the notation used in the theoretical developments. This information is filled only in the backward sweep phase, which is called in this context the *interpretation* of the tape. That phase is described later.

The tape itself is described in Listing 4.6. It consists in a list of `TapeEntryAad`. One important information for the different values stored is the index of the corresponding operation. For that reason, when one entry is added to the tape, through the `addEntry` method, the index of the operation is returned to the main code.

Listing 4.6 Tape code

```
public class TapeAad {
  private final List<TapeEntryAad> tapeList;
  private int size;

  public TapeAad() {
    this.tapeList = new ArrayList<TapeEntryAad>();
    size = 0;
  }
  public int size() {
    return size;
  }
  public TapeEntryAad getEntry(int index) {
    return tapeList.get(index);
  }
  public int addEntry(TapeEntryAad entry) {
    tapeList.add(entry);
    size++;
    return size - 1;
  }
}
```

Using those objects, the code for the example function described by Listing 2.1 in the previous chapter can be adapted easily. The code for the automatic Adjoint Algorithmic Differentiation version is given in Listing 4.7. With respect to the original code, the operators – like + – have been replaced by the method of the similar name and the mathematical functions – like exponential – have also been replaced by a method with similar name but using the `MathAad` method. For the rest the code looks very similar. Note that the tape need to be passed in each operation as each operation add an entry to the same tape and each operation create a new index.

Like for the standard AD case, the operations on the augmented doubles require a new math library. It is implemented in the `MathAad` class. An extract of that class is provided in Listing 4.8. The class consists of a list of static methods for the standard operations, like addition and multiplication and the basic functions, like exponential, sine, etc.

Listing 4.7 Simple function code by Automatic Adjoint Algorithmic Differentiation

```
static DoubleAad f_Aad_Automatic(DoubleAad[] a, TapeAad tape) {
  DoubleAad b1 = MathAad.plus(a[0],
    MathAad.exp(a[1], tape), tape);
  DoubleAad b2 = MathAad.plus(MathAad.sin(a[2], tape),
    MathAad.cos(a[3], tape), tape);
  DoubleAad b3 = MathAad.plus(MathAad.pow(a[1], 1.5d, tape),
    a[3], tape);
  DoubleAad b4 = MathAad.plus(MathAad.multipliedBy(
    MathAad.cos(b1, tape), b2, tape), b3, tape);
  return b4;
}
```

Listing 4.8 The mathematical library to be used with adjoint AD augmented doubles

```
public class MathAad {

  public static DoubleAad plus(DoubleAad d1, DoubleAad d2,
      TapeAad tape) {
    double valueOutput = d1.value() + d2.value();
    int index = tape.addEntry(new TapeEntryAad(
      OperationTypeAad.ADDITION, d1.tapeIndex(),
      d2.tapeIndex(), valueOutput));
    return new DoubleAad(valueOutput, index);
  }
  ...
}
```

The code presented in Listing 4.7 can be run on an example data. The resulting tape is presented in Table 4.1. The first column corresponds to the operation tape index or virtual address and the other columns to the information in the tape entries. The inputs used for the example are visible in the `INPUT` lines and are 0.0, 1.0, 2.0 and 3.0. The content presented is the tape after the interpretation which is described in later. Before the interpretation, the last column, with the derivatives values, is 0.0 for all entries.

The inputs are not computed from previous elements in the list and the argument indices are –1 for all of them. The first operation is the exponential which is applied to the value at index 1. The result, 2.7183, and the index are visible at line with index 4. The next line add two augmented doubles, the one at index 0 and the one at index 4, i.e. the second one is the exponential computed previously. The composition mechanism is starting here. The same process is performed for each line, up to the final addition at index 13. The output value is 4.0738.

The implementation proposed so far refers to the computation of the values themselves and the recording of the operations in the tape. All that make sense only if it is used later to compute the derivatives. The computation of the derivatives from the tape records is called the *tape interpretation*.

Table 4.1 Tape for the simple function described in Listing 2.1

Index	Operation	:	Arg 1	Arg 2	Value	Derivative
0:	INPUT	:	–1	–1	0.0	0.0331
1:	INPUT	:	–1	–1	1.0	1.5901
2:	INPUT	:	–1	–1	2.0	0.3794
3:	INPUT	:	–1	–1	3.0	1.1287
4:	EXP	:	1	–1	2.7183	0.0331
5:	ADDITION	:	0	4	2.7183	0.0331
6:	SIN	:	2	–1	0.9093	–0.9117
7:	COS	:	3	–1	–0.9900	–0.9117
8:	ADDITION	:	6	7	–0.0807	–0.9117
9:	POW1	:	1	–1	1.0	1.0
10:	ADDITION	:	9	3	4.0	1.0
11:	COS	:	5	–1	–0.9117	–0.0807
12:	MULTIPLICATION	:	11	8	0.0736	1.0
13:	ADDITION	:	12	10	4.0738	1.0

The inputs are in the first four lines and are 0.0, 1.0, 2.0 and 3.0.

In our demonstration code, the interpretation is done in a method called `interpret`. The method is located in the class `TapeUtils`. The goal of that method is to implement the equivalent of the theoretical formula given by Eq. 2.4 to the practical case under consideration.

With respect to the theoretical formula, our task is facilitated as we work only on elementary operations that take at most two arguments. The operations are called g_k in the theoretical formula. One of the $\bar{b}[k]$ of the formula will play a role in at most two $\bar{b}[j]$ with $j > k$. Instead of working on the formula as written in the theoretical part and for one given j summing all the relevant k, we take one k at a time and see for which j it plays a role. Those j's are easy to obtain through the tape entries, those are given by the tape index of the operation inputs. The quantities $\frac{\partial}{\partial b_j} g_k$ still need to be computed. The formulas for those quantities, for elementary operation, can be found in any calculus book. In Listing 4.9 we have reproduced three of those operations. The addition of two augmented doubles, the addition of an augmented double with a primitive double and the sine of an augmented double. Once more, all the detailed formulas and cases can be found in the associated code.

The interpretation starts, as described in Table 2.3, by filling the derivative entry of the output with the value of the derivative of the output with respect to the output, this is 1.0. This step is done in line 3 of Listing 4.9. Then each entry is run through in reverse order. This is done using the loop defined in line 4.

The interpretation of the `SIN` operation is done in the following way. The sine takes only one argument, so only one derivative is impacted. The derivative variable for the entry referred by the index of the first argument will be impacted, and only that number. The index is retrieve in line 22. The derivative of the sine function applied to a double is the cosine applied to the same number – line 23. The value is recorded in the tape entry with index given by the same first argument. That cosine value is multiplied by the $\bar{b}[k]$, this is the derivative value for the entry

Listing 4.9 Tape interpreter for Adjoint Algorithmic Differentiation by operator overloading

```
1  public static void interpret(TapeAad tape) {
2    int nbEntries = tape.size();
3    tape.getEntry(nbEntries-1).addValueBar(1.0d);
4    for(int loope = nbEntries-1; loope>=0; loope--  ) {
5      TapeEntryAad entry = tape.getEntry(loope);
6      switch (entry.getOperationType()) {
7        case INPUT:
8          break;
9          ...
10       case ADDITION: // Addition of two DoubleAads.
11         tape.getEntry(entry.getIndexArg1())
12           .addValueBar(entry.getValueBar());
13         tape.getEntry(entry.getIndexArg2())
14           .addValueBar(entry.getValueBar());
15         break;
16       case ADDITION1: // Addition of a DoubleAad with a primitive double.
17         tape.getEntry(entry.getIndexArg1())
18           .addValueBar(entry.getValueBar());
19         break;
20       ...
21       case SIN: // Sine of a DoubleAad
22         tape.getEntry(entry.getIndexArg1()).addValueBar(
23           Math.cos(tape.getEntry(entry.getIndexArg1()).getValue())
24           * entry.getValueBar());
25         break;
26       ...
27       default:
28         break;
29     }
30   }
31 }
```

we are working with – line 24. By running through the tape in reverse order we cover all the recorded operations down to the input. When the loop is finished, all the required derivatives have been computed. The results can be collected easily. It suffice to go through the interpreted tape a collect all lines with an `INPUT` operation. In the code this is done by the `extractDerivatives` method of the class `TapeUtils`.

Up to now we have provided one implementation for automatic Standard AD and one for automatic adjoint AD and explained how to use the code for the very simple starter function described in Listing 2.1. We thus have now numerous versions of the derivative code for that simple function: four finite difference implementation, three standard AD – two manual and one automatic – and three adjoint AD – two manual and one automatic. From the versions discussed in previous chapter, we keep only three versions for further comparisons: the forward finite difference and the optimized manual standard AD and adjoint AD. To those versions we now add the automatic versions. The performance results are provided in Table 4.2.

Table 4.2 Computation time for the example function and four derivatives

Repetitions	Computation	Time	Ratio
100,000	Function	110	1.00
100,000	Function + FD forward	575	5.23
100,000	Function + SAD Optimized	195	1.77
100,000	Function + SAD Automatic	310	2.82
100,000	Function + AAD Optimized	165	1.50
100,000	Function + AAD Automatic	350	3.18
100,000	Function + AAD Automatic (no `interpret`)	260	2.36

Figures in milliseconds. Each repetition is with 5 data sets so that there is actually 500,000 repetitions.

For this very simple function, the manually crafted implementation perform significantly better than the automatic ones, both for forward and adjoint AD. For the standard mode, the derivatives add 0.77 to the function cost for the manual version and 1.82 for the automatic one. For the adjoint version, the figures are 0.50 and 2.18 respectively. In the case of the automatic adjoint version, we have also assessed the time to run the code using the augmented doubles and storing the tape but without running the interpretation code. This can be viewed as the extra cost to manipulate the data in an automatic way, but not doing the actual derivatives computation. Doing the actual derivative computation add only a cost of 0.82, which is not too far from the manual cost. From this very simple example one can see that the data manipulation with the augmented doubles is the part of the process that is adding the most to the computation time. The actual derivatives computations are still very efficient.

4.3 Automatic Algorithmic Differentiation Applied to Finance

In this section, we repeat the analysis done at the end of the previous section but for the functions related to finance analyzed in Chapter 3.

The first of those functions is the Black price function for options. The performance results are displayed in Table 4.3. For this simple function with only five inputs, the automatic AD does not perform as well as one would like. The recording of the tape takes a lot of time with respect to the simple operations involved. This can be seen in the last line where we performed the computation of the value using the augmented doubles, but without the interpretation of the tape, i.e. without actually computing the derivative. That recording on its own reduce the speed by a factor of six with respect to s straight implementation with primitive doubles.

The automatic AD version has still the advantage of the precision, but the speed advantage with respect to the single side (forward) finite difference implementation is lost. This is specific to the low number of inputs in our example. Nevertheless it appears to be a relatively standard result in implementations that the augmented doubles slow down the value computation by a factor between two to six and that the actual derivatives computation can be three to ten times the value computation time, against a maximum of three to four in theory and a factor of one to five in different manual implementation we analyzed.

Table 4.3 Computation time for the Black function and five derivatives

Repetitions	Computation	Time	Ratio
100,000	Function	75	1.00
100,000	Function + FD forward	525	7.50
100,000	Function + SAD	235	3.36
100,000	Function + SAD Automatic	575	8.21
100,000	Function + AAD Optimized	125	1.79
100,000	Function + AAD Automatic	760	10.13
100,000	Function + AAD Automatic (no `interpret`)	450	6.00

Figures in milliseconds. Each repetition is with 5 data sets and call/put so that there is actually 1,000,000 repetitions.

Table 4.4 Computation time for the SABR volatility function and seven derivatives

Repetitions	Computation	Time	Ratio
100,000	Function	75	1.00
100,000	Function + FD forward	630	8.40
100,000	Function + AAD Optimized	130	1.73
100,000	Function + AAD Automatic	860	11.47
100,000	Function + AAD Automatic (no `interpret`)	520	6.93

Figures in milliseconds. Each repetition is with 5 data sets so that there is actually 500,000 repetitions.

The next example is the SABR volatility function that was described in Listing 3.5. We implemented only the AAD version of the function and not the SAD one. The performance results are presented in Table 4.4. In this case also, the automatic version does not compete very well with the manual implementation and barely with the finite difference version in term of speed.

The last example in this section is the SABR with implied volatility option price. The code is the composition of the two previous methods. The performance results are presented in Table 4.5. We see the real use of the AD philosophy of applying derivatives to composition. Now there are also more inputs, in total the function has eight inputs. The automatic AAD provides results in a time roughly similar to the finite difference but with the advantage for precision. The function value and eight derivatives take roughly 12 value computation time. Note also that even for this relatively simple method, the AAD tape recording is of non-negligible length; in total the tape for SABR price has 81 entries.

4.4 Mixed Algorithmic Differentiation Implementations

By mixed Algorithmic Differentiation, we mean a mixture of manually written code and automatic operator overloading code. Some formulas that are used on a regular basis or are time consuming can be manually optimized while other that are less important performance-wise can be implemented as straight forward automatic AD.

Table 4.5 Computation time for the SABR option price function and eight derivatives

Repetitions	Computation	Time	Ratio
100,000	Function	240	1.00
100,000	Function + AAD	440	1.83
100,000	Function + AAD Automatic	2935	12.23
100,000	Function + AAD Automatic (no `interpret`)	1140	4.65

Figures in milliseconds. Each repetition is with 5 data sets and call/put so that there is actually 1,000,000 repetitions.

For this reason we have created in our automatic AD operation types, referenced in the enumeration `OperationTypeAad`, a semi-mysterious type called `MANUAL`. This is the mechanism by which the developer can introduce the result of derivatives computation in the automatic code. The tape entries with that type have the `extraValue` field populated. The extra-value is the derivative of the variable referenced by the entry with respect to one of the previous variables that were used to compute it. That previous variable does to need to be the immediately previous variable in the composition chain, it can be any previous variable. This allows the derivative code to jump several SAC lines. In practice for the examples we provide, we will directly jump from the inputs to the output of a method, without recording any of the method intermediary steps in the tape. The previous variable to which the derivative in the extra-value refers is indicated by the index in `indexArg1`. This way of bypassing the automatic AD while still keeping the automatic structure has two advantages with respect to a fully automated AD. The optimization done in the code by manual AD can be transferred to the automatic AD and there are less SAC lines recorded in the tape, so less memory requirements.

When it comes to the interpretation of the tape, the derivative of the argument variable is incremented by the extra-value multiplied by the derivative of the output with respect to the analyzed variable. In the notation of Eq. 2.4, this is $\bar{b}[\text{indexArg1}] += \bar{b}[k] \cdot \frac{\partial}{\partial b[\text{indexArg1}]} g_k$, where k is the index of the variable analyzed. This is the standard multiplication representing the composition. As we are interested by the first order derivative, we don't need to know really how the computation to obtain the variable, described by the function g_k, was performed. Only its first order derivative is important.

But with what we have described above, only one of the arguments of the skipped code is automatically interpreted for AD. A second variable could be added in the `indexArg2`. But the manual part result can a priori depends on more than two arguments, using the second index in that way would not solve the general case. Remember that the tape's entries have only two arguments because the elementary operation which are recorded by the automatic code have at most two arguments. The method we propose here to solve this problem without adding more arguments, is to have the second argument pointing to another version of the same variable where the second input variable derivative is represented in the extra-value. But that second version does not have the derivative of the output with respect to the analyzed

variable, as recursively we have reached only the last version of that variable in the tape. It is then enough to copy that derivative of the analyzed variable to the `valueBar` field of that other copy to be able to continue the process. We create as many copies of the variable as number of input in the manual code. Each copy, but the first, pointing to the previous copy to continue the recursive process. The part of the `interpret` method which deal with automatic adjoint AD computation for the `MANUAL` type is represented in Listing 4.10.

Listing 4.10 The part of the tape `interpret` method related to the `MANUAL` operation

```
...
  switch (entry.getOperationType()) {
    case INPUT:
      ...
    case MANUAL:
       tape.getEntry(entry.getIndexArg1()).addValueBar(
         entry.getExtraValue() * entry.getValueBar());
       if(entry.getIndexArg2() != -1) {
         tape.getEntry(entry.getIndexArg2())
           .addValueBar(entry.getValueBar());
       }
       break;
   ...
```

With the previous explanation on the way to incorporate manual code in an automatic AD setting, we are ready to review some of our previous examples. In the first financial code we have analyzed, we looked at the Black formula. The manual code with automatic AD wrapping is available in the method `price_Aad_Automatic2` of the class `BlackFormula`. The code in that method is the same that in the `price_Aad_Optimized` for the main part. Only at the end, the storage of the manual computation is done in the relevant tape entries. From a signature point of view, the method looks like the automatic version. The last part of the relevant code is proposed in Listing 4.11.

The performance of this new implementation is compared to the one obtained earlier is displayed in Table 4.6. The performance is improved with respect to the full automatic version. The optimization done in the manual code still produces its effects. Nevertheless, the recording and manipulation of tapes is still significantly slower that dealing with primitive double.

The main philosophy of algorithmic differentiation is to use the composition efficiently. How does the manual/automatic duality transfers to composition. To show some possibilities we reimplemented the SABR with implied volatility option price with several mixed manual/automatic combination. The first one is an automatic implementation where both the SABR volatility and the Black price are computed by the manual version wrapped in the automatic language. This is the composition of the equivalent of two `Automatic2` codes as described above. The second implementation, called *Mixed M 1* uses the appearance of manual

Listing 4.11 The final part of the manual code with automatic wrapping for the Black formula

```
...
  int indexPrice0 = tape.addEntry(new TapeEntryAad(
    OperationTypeAad.MANUAL, forwardAad.tapeIndex(),
    price, inputBar[0]));
  int indexPrice1 = tape.addEntry(new TapeEntryAad(
    OperationTypeAad.MANUAL, volatilityAad.tapeIndex(),
    indexPrice0, price, inputBar[1]));
  int indexPrice2 = tape.addEntry(new TapeEntryAad(
    OperationTypeAad.MANUAL, numeraireAad.tapeIndex(),
    indexPrice1, price, inputBar[2]));
  int indexPrice3 = tape.addEntry(new TapeEntryAad(
    OperationTypeAad.MANUAL, strikeAad.tapeIndex(),
    indexPrice2, price, inputBar[3]));
  int indexPrice4 = tape.addEntry(new TapeEntryAad(
    OperationTypeAad.MANUAL, expiryAad.tapeIndex(),
    indexPrice3, price, inputBar[4]));
  return new DoubleAad(price, indexPrice4);
```

Table 4.6 Computation time for the Black function and five derivatives

Repetitions	Computation	Time	Ratio
100,000	Function	75	1.00
100,000	Function + AAD Optimized	125	1.79
100,000	Function + AAD Automatic	760	10.13
100,000	Function + AAD Automatic / Manual	530	7.07

Figures in milliseconds. Each repetition is with 5 data sets and call/put so that there is actually 1,000,000 repetitions.

AD but inside, the implied volatility is obtained by manual code and the Black formula by automatic code. The third implementation, called *Mixed A 1*, has the automatic appearance and inside, the volatility is automatic code and the Black formula is manual code. The fourth implementation, called *Mixed A 2*, has the automatic appearance and inside, the volatility is manual code and the Black formula is automatic code. There would be more combination possible, but it seems enough to have a view of the impact. The performance results are displayed in Table 4.7.

The main conclusion from this section is that the manual and automatic algorithmic differentiation can be used in combination. This is the real philosophy of AD as I see it: use the *efficiency of composition* to achieve the *best result*. There is no a priori superiority of one approach with respect to another. The secondary conclusion is that automatic AD is less efficient that expertly crafted AD and that the ratio between the two can be larger than five. We will come back to the advantages of expertly crafted AD code in the next chapter.

Table 4.7 Computation time for the SABR price function and eight derivatives

Repetitions	Computation	Time	Ratio
100,000	Function	240	1.00
100,000	Function + AAD Optimized	440	1.83
100,000	Function + AAD Automatic	2935	12.23
100,000	Function + AAD Automatic 2	980	4.08
100,000	Function + AAD Mixed M 1	1230	5.13
100,000	Function + AAD Mixed A 1	2480	10.33
100,000	Function + AAD Mixed A 2	1340	5.58

Figures in milliseconds. Each repetition is with 5 data sets and call/put so that there is actually 1,000,000 repetitions.

Bibliography

Naumann, U. (2012). *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.* Philadelphis:SIAM.

Naumann, U. and du Toit, J. (2014). Adjoint algorithmic differentiation tool support for typical numerical patterns in computational finance. Technical report, RWTH Aachen University.

Chapter 5
Derivatives to Non-inputs and Non-derivatives to Inputs

You didn't know you wanted it – You get what you have not asked for – Volatility can stick.

The title of this chapter may appear a little bit cryptic. The main advertised goal of Algorithmic Differentiation (AD) is to compute in an efficient way the derivatives of functions with respect to their inputs. Each part of this chapter's title may appear in contraction with that general goal.

Nevertheless, to my opinion, the content of this chapter is highly relevant for this book in the context of application of AD to finance. Instead of reading it as a negation of the book's main goal, the content of this chapter should be viewed as an extension of it. A less cryptic, but significantly longer and less fun title could have been: "computing derivatives with respect to financially meaningful numbers that are not direct inputs to the global computation and computing numbers closely related to derivatives but not matching exactly the theoretical definition of derivative."

The techniques described below are in general not available to automatic AD. They are producing results relevant for a subject matter expert but that do not appear directly in the data structure. Automating the interpretation of the financial meaning of code is probably beyond the current reach of programming and will need to wait for more developments in artificial intelligence. We still need to trust human experts to extract as much insight as possible from the existing developments.

5.1 Derivatives with Respect to Non-inputs

As a first example, we look at the question of the vega hedging of caps and floors. Suppose that we use a description of the cap price through normal[1] implied volatilities. In our example, the so-called "smile" of normal volatilities is described for this purpose by a set of fixed strikes volatilities and an interpolation mechanism is

[1] We use normal/Bachelier (1900) volatilities and not log-normal/Black (1976) volatilities to avoid the problem with negative strikes and rates.

M. Henrard, *Algorithmic Differentiation in Finance Explained*,
Financial Engineering Explained, DOI 10.1007/978-3-319-53979-9_5

used to obtain the volatility between them. For the moment we ignore the expiry dimension.

A direct implementation of algorithmic differentiation will provide the sensitivity of the price to the different strike nodes describing the smile, this is to a fixed set of strikes. This may not be what the financial user is interested in. Even if the smile is described using fixed strikes node strikes, those strikes may not be fundamentally different from others strikes. A good complete set of financial information provided to the risk manager will contains on one side the sensitivity to the interpolated volatility used in the price computation and on the other side the sensitivity to each individual node used to describe the smile. The volatility interpolated between the nodes is not one of the original input data provided but an intermediary number which is part of the computation. The business user may be interested by information on derivatives with respect to intermediary values. The architecture of the system should allow him to request and obtain those numbers. Moreover the computation time for providing one or both of the information described above should not be fundamentally different. The sensitivity to an intermediary value should be available without restarting the full computation.

In the swap market, swaps with yearly tenors are liquid and the building blocks of the market. The interest rate curves are calibrated with those data points and the macro-hedging of a portfolio is usually done with the same instruments. The inputs use in the computer implementation are matching the information the trader uses. For example trading a 5-year, 7-month and 10-day swap to hedge some interest rate level is cumbersome and attract a larger bid/offer due to illiquidity. In this context, computing the sensitivity with respect to the yearly swap rates for an interest rate portfolio makes sense, both from a financial and a technological point of view.

We have already described a similar requirement for interest rate curve in Section 3.3. The information provided to the end-user for interest rate curves should include the point sensitivity, the parameter sensitivity and the market quote sensitivity or in a more generic language, the sensitivity to each individual point used in the computation, to the model parameters and to the market data used to calibrate the model.

A natural number computed for interest rate books through AD is the sensitivity of the book to the different market quotes used to calibrate the curves. This is a Risk Measurement number. To achieve Risk Management, the risk manager has to convert this number into an action. The question is: *Which amount of the financial product represented by the market quote should I buy/sell to eliminate the risk described by the sensitivity?*

From the market quote sensitivities $\partial \mathrm{PV}/\partial q_k$ computed in the risk measurement step, one would like to establish the notional of each instrument that should be traded. First note that the present value PV_k of each instrument with unit notional k used in the curve calibration, is such that

$$\frac{\partial \mathrm{PV}_k}{\partial q_l} = 0 \quad \text{for} \quad k \neq l.$$

Each instrument has a present value of zero if its quote is the one used in the calibration step. Changing the other quotes does not affect that value. The value

of a given instrument used in the curve calibration is not sensitive to the change of quotes associated to the other instruments. We have only one non-zero number in those sensitivities for each node k; we denote it

$$T_k = \frac{\partial \mathrm{PV}_k}{\partial q_k}$$

for a calibrating instrument with unit notional.

This number is easy to compute, it suffices to apply the principles described in the previous section for the present value sensitivity to each of the n instruments in turn, obviously using AD. That step can be done once at curve calibration time and stored to be used as described below as many time as necessary.

Suppose that the sensitivity of a portfolio to the different market quotes is given by

$$s_k = \frac{\partial \mathrm{PV}}{\partial q_k}.$$

To obtain a portfolio with first order sensitivity equal to zero for all buckets, it is enough to trade a notional

$$N_k = -\frac{s_k}{T_k}$$

in each of the calibration instruments.

Adding those trades to the portfolio we obtain a total sensitivity to the market quote q_k of

$$s_k + N_k \cdot T_k = s_k - \frac{s_k}{T_k} \cdot T_k = 0.$$

As promised we obtain the perfect (local) hedging of the portfolio with the instruments used to calibrate the curve at the cost of one multiplication for each instrument in the calibration set. The multiplication factor $-1/T_k$ is computed as a by-product of the AD implementation. It is related to the derivative of the PV of an intermediary instrument computed during the curve calibration process. It is useful for the risk manager to be able to request that quantity to be available for further processing. The quantity $-1/T_k$ is not strictly speaking the derivative with respect to one of the input, but the derivative of the theoretical quote when the present value of the underlying instrument changes. It not a quantity automatically computed, but a portfolio manager or trader will certainly find it a very useful quantity to have at his disposal.

5.2 Non-derivatives with Respect to Inputs

In most cases, the inputs to valuation are market data, this is market quotes of financial instruments. For some instruments, like bonds and futures, the quote may be

a price, for others, like FRA, IRS and inflation swaps, the quote may be a rate and for other, like basis swaps or Forex swaps, it may be a spread. For some, the quoting mechanism can be indirect, like implied volatility for options or yield for bonds; a standard or conventional formula needs to be used to obtain the actual term sheet of the trade.

Using the conventional formula is a mechanism to facilitate the communication between the market participant, it is not a religion nor an indication that the user believes that the intuition behind the formula is correct. It is not because someone quotes the Black or Bachelier formula implied volatility for an option, that he necessarily uses the log-normal or normal dynamic implicit from the formula to risk manage his option book. Even if we restrict ourselves to a Black formula with an implied volatility smile or surface, the description of the current smile, which is what can be observed in the market, should not be read as a estimation on how it will evolve in the future. Someone may describe the smile of interest rate cap/floors in term of implied volatility by strike, this does not implies that he believes that the best prediction for the future is that the implied volatility for a given strike will be constant. Predicting that the implied volatility for a given strike is stable when the underlying market evolves is called a *sticky strike* dynamic for obvious reasons. The fact that cap/floor market conventionally quote volatility by strike does not mean that all cap/floor market participants use the sticky strike approach for risk management purposes.

One of the direct applications of Algorithmic Differentiation in finance is the computation of greeks. The algorithmic differentiation technique itself is the art of computing derivatives of the output with respect to the input. There is a semantic slide between the two previous sentences. Greeks and derivatives are almost synonyms in finance, except in this section where they are almost antonyms. The greeks we are interested in may be different from the direct derivatives of the formula, they may be a combination of different derivatives to match a specific intuition or derivatives with respect to a number which is not explicitly a market quote but that can be implied from them.

In the martingale approach to pricing, the choice of model is almost synonymous of the choice of delta with respect to the underlying. The risk manager should have the choice of selecting the data that to his opinion provides the best representation of the current market and, separately, he should have the choice to decide of the modeling approach that represents his best view of the market future evolution. It is the job of the developer to allow those choices to be made by the end user. Not only the straight "derivative with respect to the input" should be available but also any sensible modification thereof as imagined by the risk manager. The terms "sensible" and "imagined" can be somewhat contradictory, this is why the dialogue between the two characters above is important.

Note that in the *Fundamental Review of Trading Book*, the Basel Committee on Banking Supervision (2014)[2] (BCBS) prescribes that the computation of

[2] http://www.bis.org/bcbs/

sensitivities for regulatory computation of capital related to the trading book should be done in a *sticky delta* approach – that we describe below. Even if I personally disagree with that recommendation, I agree with the fact that a risk manager should be able to choose his preferred way, whatever the market conventions are.

5.2.1 Sticky to Something – The Problem

In this section, we take a concrete example of the above abstract discussion and describe how to implement it in practice. In this case, we suppose we have a financial option priced by a Black formula. This can be an equity, forex or interest rate option. Obviously for interest rate options, the Black/log-normal model may not be the best choice as rates and strikes can be negative. If you are working in interest rate, you can replace Black by Bachelier everywhere in this section; all the results remain valid with very minor adjustments. It is also valid for any base formula depending of similar parameters: forward, strike and a unique model parameter.

We consider an option with strike K and time to expiry θ. The Black formula, already described in Eq. 3.1 is given, ignoring the numeraire which is not important for our discussion, by

$$\text{Black}(K, \theta, F_0, \sigma_0) = \omega \left(F_0 N(\omega d_+) - K N(\omega d_-)\right)$$

where $\omega = 1$ for a call, $\omega = -1$ for a put and

$$d_\pm(K, \theta, F_0, \sigma_0) = \frac{\ln\left(\frac{F_0}{K}\right) \pm \frac{1}{2}\sigma_0^2 \theta}{\sigma_0 \sqrt{\theta}}. \tag{5.1}$$

The base present value formula is

$$\text{PV}_0 = \text{PV}(K, \theta, F_0) = \text{Black}(K, \theta, F_0, \sigma_0) \tag{5.2}$$

with F_0 the underlying current forward price and σ_0 the implied volatility for that option. By implied volatility we mean the usual "wrong parameter in the wrong formula to obtain the correct price." By saying that, we implicitly mean that we do not trust the model that leads to the formula but we use the formula as a convenient mean to store information about the market through the implied volatility. We can observe the same market, not only for the strike price we are interested in, but for other strikes as well. We can obtain the current implied volatility for all strikes, called the *market smile*: $\sigma = \sigma^{\text{Mkt}}(K)$. For different strikes, the market prices are given by

$$\text{PV}(K, \theta, F_0) = \text{Black}(K, \theta, F_0, \sigma^{\text{Mkt}}(K)) \tag{5.3}$$

We are interest in the derivative of the price with respect to the forward price or rate. We cannot differentiate directly the PV function from Eq. 5.2. The formula gives the correct price for the current forward rate F_0 but there is no indication that the same formula would be correct if the underlying value was to change. The notation in the above formula may let us think that the volatility is simply strike dependent and we can differentiate Eq. 5.3. This is not the case either. The volatility shows only a strike dependency because of our ignorance, we don't know what happens for a different market F_0. The present value formula should actually be written as

$$\text{Black}(K, \theta, F, \sigma(K, \theta, F, \ldots)). \tag{5.4}$$

But we know only a small part of the volatility function, we know it only for one value of F – the current value F_0 – and we want to differentiate with respect to that same F. In other words, we are looking for the derivative in a direction where we know almost nothing. We want a short movie about the main character – the present value – but we only have a still picture of the opening sequence of the movie.

It means that we have to add some hypothesis to the framework to obtain what we are looking for. One potential way to select those hypothesis is to suppose a *sticky something* behavior for the smile. The standard "something" that can be sticky are *strike*, *moneyness* or *delta*.

Before going to the details of what we mean by *sticky*, we describe the other terms. The strike term does not need further explanation. The moneyness can be express as a *simple moneyness*, which is the difference between the strike and the forward ($K - F$) or as a *log-moneyness*, which is the logarithm of the ratio of strike and forward ($\log(K/F)$). The delta is the Black formula delta given by $N(d_+)$ with d_+ described in Equation (5.1).

What does it mean that the volatility description *sticks* to one of those quantities? It means that we associate the volatility for a new market value $F \neq F_0$ by using the current market smile σ^{Mkt} with that quantity constant. Let $f(K, F)$ be the quantity that is preserved.

For the cases mentioned above, those quantities are

Strike: $f(K, F) = K$

Simple moneyness: $f(K, F) = K - F$

Log-moneyness: $f(K, F) = \log(K/F)$

Delta: $f(K, F, \sigma) = N(d_+(K, F, \sigma))$

The delta case is more involved as it refers to the volatility and will be treated separately later.

For a change in the underlying market to a different forward $F \neq F_0$, the new volatility to be used to price the option with strike K is

$$\sigma = \sigma^{\text{Mkt}}(L) \quad \text{where } L \text{ is such that} \quad f(K, F) = f(L, F_0). \tag{5.5}$$

The volatility to use is given by the market volatility at an implied strike L. This implied strike is obtained by using a conservation law.

For the sticky strike case, the solution is easy, $F(K, F) = K$, $L = K$ and

$$\sigma = \sigma^{\text{Mkt}}(K).$$

This is the simplest case; the implied volatility for a given option with a given strike does not change with the change of the underlying market.

For the sticky moneyness case, the solution of $K - F = L - F_0$ is given by $L = K - F + F_0$ and the volatility to be used is

$$\sigma = \sigma^{\text{Mkt}}(K - F + F_0).$$

The new volatility is obtained by a simple translation.

By the sticky extension, we have an extension of the implied volatility to all forward F and we can write

$$\text{PV}(K, F) = \text{Black}(K, F, \sigma)$$

with σ satisfying Eq. 5.5.

5.2.2 *Sticky Something, Volatility Independent*

In this section, we suppose that the function f defining the stickiness depends on F and K only and not on σ.

Remember, what we are trying to compute is the sensitivity of the price to the change of the underlying forward price or rate. We are looking for the derivative with respect to F, the third variable in the present value function,

$$D_3\text{PV}(K, \theta, F_0) = D_3\text{Black}(K, \theta, F_0, \sigma_0) + D_4\text{Black}(K, \theta, F_0, \sigma_0)D_3\sigma(K, \theta, F, \ldots).$$

In the above formula, most of the pieces are known. The derivative of the Black formula with respect to the forward is called the (forward) *delta* and the derivative with respect to the volatility is called the *vega*. The formula is thus

$$D_3\text{PV}(K, \theta, F_0) = \Delta(K, \theta, F_0, \sigma_0) + \text{Vega}(K, \theta, F_0, \sigma_0)D_3\sigma(K, \theta, F, \ldots).$$

The only piece we still have to provide is the $D_3\sigma$. For the two special cases of the sticky strike and sticky simple moneyness, the formula of the volatility in term of forward is explicit and we could compute the derivative directly.

In a more general setting, we can still obtain the result by using the implicit function theorem which is detailed in Appendix A.3. The implicit result is given by Equation (5.5). We have a point with a root of the equation: $f(K, F_0) - f(K, F_0) = 0$ and we would like to find an implicit function g which solved the equation

$f(K,F) - f(g(F),F_0) = 0$ for points around F_0. With that notation, the volatility is $\sigma(K,\theta,F,\ldots) = \sigma^{\text{Mkt}}(g(F))$. We want to compute $D_3\sigma$, which is possible by the implicit function theorem. Note that, by definition, $g(F_0) = K$.

By the implicit function theorem, the derivative of g is given by

$$D_1 g(F_0) = \frac{D_2 f(K,F_0)}{D_1 f(K,F_0)}.$$

We have not mentioned yet the proof of the existence of g; by the same implicit function theorem, the function g exists around F_0 if $D_1 f(K,F_0) \neq 0$.

If this is the case, we have as final solution

$$D_3\text{PV}(K,\theta,F_0) = \Delta(K,\theta,F_0,\sigma_0) + \text{Vega}(K,\theta,F_0,\sigma_0) D\sigma^{\text{Mkt}}(K) \frac{D_2 f(K,F_0)}{D_1 f(K,F_0)}.$$

Obviously this requires that σ^{Mkt} is differentiable. In particular a simple linear interpolation of market volatilities would not provide very good results.

In the *sticky strike* case, we have $F(K,F) = K$ and $D_2 f(K,F_0) = 0$. So there is no vega part in the formula and

$$\Delta_{\text{StickyStrike}} = D_3\text{PV}(K,\theta,F_0) = \Delta(K,\theta,F_0,\sigma_0).$$

In the *sticky simple moneyness* case, we have $F(K,F) = K - F$, $D_2 f(K,F_0) = -1$ and $D_K f(K,F_0) = 1$. So the final solution is

$$\begin{aligned}\Delta_{\text{StickyMoneyness}} =& D_3\text{PV}(K,\theta,F_0)\\ =& \Delta(K,\theta,F_0,\sigma_0) - \text{Vega}(K,\theta,F_0,\sigma_0) D_1\sigma^{\text{Mkt}}(K). \end{aligned} \tag{5.6}$$

In the *sticky log-moneyness* case, we have $F(K,F) = \log(K/F)$ and $D_2 f(K,F_0) = -1/F_0$ and $D_1 f(K,F_0) = 1/K$. So the final solution is

$$\begin{aligned}\Delta_{\text{StickyLogMoneyness}} =& D_3\text{PV}(K,\theta,F_0)\\ =& \Delta(K,\theta,F_0,\sigma_0) - \text{Vega}(K,\theta,F_0,\sigma_0) D_1\sigma^{\text{Mkt}}(K) \frac{K}{F_0}. \end{aligned} \tag{5.7}$$

5.2.3 *Sticky Something, Volatility Dependent*

The sticky delta case is a slightly more complex problem. The reason is that the constraint is now of the form $f(K,F,\sigma) = f(L,F_0,\sigma^{\text{Mkt}}(L))$. The constraint f is on the delta and the delta itself depends on the σ. We cannot solve the two problems consecutively, we have to solve them simultaneously.

The problem is now

$$\begin{aligned}\sigma &= \sigma^{\text{Mkt}}(L)\\ f(K,F,\sigma) &= f(L,F_0,\sigma^{\text{Mkt}}(L))\end{aligned}$$

where f is given by d_+ from Eq. 5.1. It can be written as a systems of equations, considering K and F_0 as constants, as

$$h(L,F,\sigma) = 0$$

We now have a system of two equations with three unknowns L, F and σ. The goal is to obtain (implicitly) $\sigma(F)$ and (explicitly) $D\sigma(F_0)$. The partial derivatives we need are given by

$$D_{(1,3)}h(K,F_0,\sigma_0) = \begin{pmatrix} -D_1\sigma^{\text{Mkt}}(K) & 1 \\ -D_1 f(K,F_0,\sigma_0) - D_3 f(K,F_0,\sigma_0) D_1\sigma^{\text{Mkt}}(L) & D_3 f(K,F_0,\sigma_0)\end{pmatrix}$$

$$D_2 h(K,F_0,\sigma_0) = \begin{pmatrix} 0 \\ D_2 f(K,F_0,\sigma_0)\end{pmatrix}$$

And the inverse to be used is

$$\left(D_{(1,3)}h(K,F_0,\sigma_0)\right)^{-1} = \frac{1}{D_1 f(K,F_0,\sigma_0)} \begin{pmatrix} D_3 f(K,F_0,\sigma_0) & -1 \\ D_1 f(K,F_0,\sigma_0) + D_3 f(K,F_0,\sigma_0) D_1\sigma^{\text{Mkt}} & -D_1\sigma^{\text{Mkt}}(K)\end{pmatrix}$$

After easy algebraic manipulations, we have

$$D_F \begin{pmatrix} L \\ \sigma\end{pmatrix} = -\frac{1}{D_1 f(K,F_0,\sigma_0)} \begin{pmatrix} -D_2 f(K,F_0,\sigma_0) \\ -D_2 f(K,F_0,\sigma_0) D_1\sigma^{\text{Mkt}}(K)\end{pmatrix}$$

The final result for the case of the sticky delta is

$$D_F \text{PV}(F_0,K) = \Delta(F_0,\sigma_0) + \text{Vega}(F_0,\sigma_0) D_1\sigma^{\text{Mkt}}(K) \frac{D_2 d_+(K,F_0,\sigma_0)}{D_1 d_+(K,F_0,\sigma_0)}$$

with

$$\begin{aligned} D_1 d_+(K,F_0,\sigma_0) &= -\frac{1}{\sigma_0\sqrt{\theta}}\frac{1}{K}\\ D_2 d_+(K,F_0,\sigma_0) &= \frac{1}{\sigma_0\sqrt{\theta}}\frac{1}{F_0}.\end{aligned}$$

This leads to a final result

$$\Delta_{\text{StickyDelta}} = D_3 \text{PV}(K,\theta,F_0) = \Delta(K,\theta,F_0,\sigma_0) - \text{Vega}(K,\theta,F_0,\sigma_0) D_1\sigma^{\text{Mkt}}(K) \frac{K}{F_0}.$$

Note that the *sticky delta* risk is locally the same as the *sticky log-moneyness* case.

In this section we have constructed different deltas, the original formula Delta, which is equivalent to the sticky strike Delta but also the sticky moneyness Delta, the sticky log-moneyness Delta and the sticky delta Delta. All those numbers have been obtained from the same set of market data, i.e. the non-dynamic market smile $\sigma^{\text{Mkt}}(K)$. There is no need to manipulate the original data to obtain the different results. An implementation of pricing through Black formula with smile should provide the three different deltas as standard output. Then it is up to the risk manager to use one or all of them, or even to request a different one. The risk manager should not be required to transform the input data to be able to obtain the risk measure he would like to see; the system should provide this feature implicitly. The non-derivatives to the input can be computed directly in the library by using simple manipulations of the results provided by the AD framework. Remember, Algorithmic Differentiation is not a mere technique, it is really a *philosophy of implementation*.

Bibliography

Bachelier, L. (1900). *Théorie de la Spéculation*. PhD thesis, Ecole Normale Supérieure.

Basel Committee on Banking Supervision. (2014). Fundamental review of the trading book: outstanding issues. Consultative document, Basel Committee on Banking Supervision.

Black, F. (1976). The pricing of commodity contracts. *Journal of Financial Economics*, 3(1–2):167–179.

Chapter 6
Calibration

What we need is implicit – The fast lane to calibration.

The term *calibration* used in the chapter's title is the term used in finance for root finding or finding an optimal in the least-square sense. The type of problem encountered can be summarized in the following way. A certain number of financial market information, called *market quotes*, are available. This is for example the par rates of swaps, the prices of futures, the yields of bonds, or the prices of options. We want to use a specific model to compute the present value of other instruments, similar to the original ones, or to compute the risks associated to them. The model is based on a certain number of parameters. Those parameters can be the zero-coupon rates of a curve, the Black implied volatilities, or the volatility parameters of a Libor Market Model.

From the model, one can compute the *theoretical quotes* that would be prevalent if the model was correct. Those theoretical quotes depend on the model parameters. The calibration consists in searching for the parameters of the model which are such that the theoretical quotes are equal to or as close as possible to the market quotes. When the theoretical quotes are equal to the market quotes, we say that we have an *exact calibration*; when the theoretical quotes are not equal to the market quote and found through a least-square-like approach, we say that we have an *at-best calibration*. Usually in the exact calibration we have the same number of model parameters as the number of market quotes targeted. In the at-best calibration, there are more market quotes than parameters in the model.

6.1 Interest Rate Curves Calibration

In the approach we present in this section, we deal only with exact calibration for interest rate curves in the multi-curve framework. We suppose that we have a certain number of instruments in the calibration basket and the same number of parameters in the curves. The parameters are split among different curves. We suppose also that the multi-dimensional root finding problem defined by the constraints on each instrument and the curve parameter has one non-singular solution.

M. Henrard, *Algorithmic Differentiation in Finance Explained*,
Financial Engineering Explained, DOI 10.1007/978-3-319-53979-9_6

It is quite difficult to describe financial conditions that lead to those mathematical hypothesis without prescribing specific curve descriptions and instrument restrictions. We would like to leave the approach as general as possible, so we refrain going in that direction.

We have n instruments that are used in the calibration and we index them by k ($1 \leq k \leq n$). Let $p = (p_l)_{1 \leq l \leq n}$ denote the parameters of the curves and $S = (S_k)_{1 \leq k \leq n}$ the functions for which we have to find a root for the k-th instrument. This is in general the market quote par spread or a similar function. The market quotes themselves are denotes $q = (q_k)_{1 \leq l \leq n}$. See Henrard (2014b) for more details on the curve descriptions and the functions to solve. The problem to solve is

$$S_k(p) = 0 \quad (1 \leq k \leq n). \tag{6.1}$$

In general, the problem is non-linear and strongly coupled. The solution has to be obtained globally for all parameters p as one computation block, it is not possible to bootstrap the different equations.

In some special cases, the problem can be divided in sub-problems, easier to solve numerically. The most extreme simplification is when the problem can be divided into one dimensional sub-problems. This is the case of the bootstrapping approach, where the interpolation scheme is local and the curves are not entangled. This is an extreme case, quite non-realistic in practice and we do not analyze it here.

A more frequent numerical simplification is when some curves are not fully entangled and the problem can be solved inductively. First calibrating some curves and then using the resulting curves as an input to calibrating the next curves. In the sequel we call *curve units* the set of curves calibrated simultaneously in one unique root-finding process. The complete set of all the related curves is called a *group*.

The standard application of this is when the curves are build one by one. More involved examples are when units of curves are first calibrated and then another unit of several curves is calibrated based on the previous units. Let u be the number of units and n_i the total number of parameters in the first i units. The steps in the induction will look like:

$$\begin{aligned} S_k((p_l)_{0<l\leq n_1}) &= 0 \quad (0 < k \leq n_1) \\ S_k((p_l)_{0<l\leq n_1}, (p_l)_{n_1<l\leq n_2}) &= 0 \quad (n_1 < k \leq n_2) \\ &\vdots \\ S_k((p_l)_{0<l\leq n_{u-1}}, (p_l)_{n_{u-1}<l\leq n}) &= 0 \quad (n_{u-1} < k \leq n) \end{aligned} \tag{6.2}$$

Each vector equation represents one unit. In the function S_k above we represent only the parameters that impact the equalities.

Solving the above system of equations, this is finding the parameters $(p_l)_{0<l\leq n}$, should not be the only output of the calibration exercise. The output of the procedure should facilitate the computation of derivatives related to other functions. The calibration function is only one step in the general AD implementation for interest rate.

For example the calibrated curves are used to compute the present value of instruments – denoted PV. What we are also interested in are the sensitivities or first order derivatives of the present value to the market quotes q_k used in the curve building procedure, i.e. we are also interested in

$$\frac{\partial \mathrm{PV}}{\partial q_k}.$$

Those numbers are often called the *bucketed deltas*, *partial PV01* or *key rate duration* of the instruments.

Using the calibrated curves, it is possible to compute the derivatives of the present value to the curves parameters p. Algorithmic differentiation generally helps for that also. The derivative with respect to the market quotes can be obtained through the usual derivative of composition by

$$\frac{\partial \mathrm{PV}}{\partial q_k} = \sum_{l=1}^{n} \frac{\partial \mathrm{PV}}{\partial p_l} \frac{\partial p_l}{\partial q_k}.$$

This is the usual composition used in AD, first calibrate the curves, that is compute the function $p(q)$, and then compute the present value of the other instruments. What we have not described above is how to compute in practice the matrix

$$D_q p = \left(\frac{\partial p_l}{\partial q_k} \right)_{1 \le l \le n, 1 \le k \le n}.$$

That matrix is often called *inverse Jacobian matrix* or *inverse transition matrix*. Obviously you could implement AD for the full root-finding procedure and par spread computation and obtain the result that way. What we show below is that this is not required, it is sufficient to have the AD implemented for the par spread part.

The functions we suggest to use for the root-finding procedure in the calibration are the par market quotes spreads. They are the additive spreads to the market quotes for which the present value of the instrument is equal to zero, this is for q^{Mkt} the market quotes and $\tilde{q}$ the computed quotes from the curves,

$$S(p, q^{\mathrm{Mkt}}) = \tilde{q}(p) - q^{\mathrm{Mkt}}. \tag{6.3}$$

We rewrite Eq. 6.1 to indicate explicitly the market quote dependency. The calibrated parameters p_0 are such that

$$S(p_0, q^{\mathrm{Mkt}}) = 0.$$

The market quotes q^{Mkt} are fixed numbers and are not part of the parameters to change to find the root. Provided that some regularity conditions are satisfied around the solution (p_0, q^{Mkt}), one can apply the *implicit function theorem* described in Appendix A.3. There is a function $p : \mathbb{R}^n \to \mathbb{R}^n; q \mapsto p(q)$ defined in a neighborhood of q^{Mkt} such that

$$S(p(q), q) = 0$$

for all q in the neighborhood. Using the derivative part of the implicit function theorem, the derivatives of the calibrated parameters with respect to the quotes q at the market quote q^{Mkt} can be obtained by

$$D_q p(q^{\text{Mkt}}) = -(D_p S)^{-1}(p_0, q^{\text{Mkt}}) D_q S(p_0, q^{\text{Mkt}}) = (D_p S)^{-1}(p_0, q^{\text{Mkt}}).$$

The last equality is obtained using the fact that the derivative of S given in Eq. 6.3 to the quote q is –1.

When the root-finding problem is solved with a Newton-like algorithm, the same type of matrix is computed internally in the solver. The Jacobian matrix is used to find the direction of the next best guess. Computing the matrix for later use does not require developments beyond those already done for solving efficiently the root-finding algorithm. In what follows, we will call the *Jacobian* or *transition* matrix the matrix $D_p S$ and the *inverse Jacobian* the matrix $D_q p = \left(D_p S\right)^{-1}$.

We now describe a way to improve further the efficiency of the computation of the above matrix. To obtain the transition matrices between the market quotes and the curve parameters, one can compute directly the huge matrix with all the market quotes as input and all the curve parameters as the output of a very large root-solving problem.

This is possible, but if you work in a multi-currency and multi-curve framework with collateral – see (Henrard 2014b, Section 8.3) – with many currencies, many collateral and many ways to combine them, one has to deal easily with 10 curves with 20 parameters each, this is a total of 200 parameters and market quotes. Solving a 200×200 non-linear system of equations and computing the associated Jacobian matrix is probably not the most efficient way to achieve the result. As described above in the system of equations (6.2), the calibration can be done in an inductive way for most of the curve settings. The last curves calibrated depend on all the previous curves, but at each step only a reduced system has to be solved. Usually the sub-system, that we call unit, contains only one to three curves and not ten or more like the full system.

Suppose that the curves are obtained as a multidimensional root-finding process in an inductive way. The goal is to obtain the calibrated curves $(p_l)_{l\in(0,n]}$ from the market rates $(q_k)_{k\in(0,n]}$ but also the generalized transition matrix

$$\left(D_{q_k} p_l\right)_{l\in(0,n],k\in(0,n]}.$$

Due to the way the curves are built, we know that a good part of the matrix is full of zeros; the non-zero part, which we are interested in, is made of the sub-matrices

$$\left(D_{q_k} p_l\right)_{l\in(n_{i-1},n_i],k\in(0,n_i]}.$$

The equations solved to obtain the parameters are at the i-th step:

$$S(n_{i-1}, n_i](p(0, n_{i-1}], p(n_{i-1}, n_i]) = 0 \tag{6.4}$$

where the notation $x(i, j]$ represents the vector with values $(x_k)_{i<k\leq j}$. We have split the parameter vector in two parts: the parameters that have already been calibrated ($(0, n_{i-1}]$) and the parameters we calibrate in this step $(n_{i-1}, n_i]$.

Suppose that at each step, the previous step transition matrices $D_{q_k p_l}$ are available for $l \in (n_{i-2}, n_{i-1}]$ and $k \in (0, n_{i-1}]$. We can compute $D_{p_l q_k} = D_{p_l S_k}$ for $l, k \in (n_{i-1}, n_i]$ directly as described previously. Moreover

$$\left(D_{q_k p_l}\right)_{l,k\in(n_{i-1},n_i]} = \left(\left(D_{p_l q_k}\right)_{l,k\in(n_{i-1},n_i]}\right)^{-1}.$$

So we have the derivative of the new set of parameters $(p_l)_{(n_{i-1},n_i]}$ with respect to the new market rates $(q_l)_{(n_{i-1},n_i]}$. For the previous market rates, we use composition. Suppose that we have $D_{p_k p_l}$ for $l \in [n_{i-1}, n_i)$ and $k \in [0, n_{i-1})$. Then

$$\left(D_{q_k p_l}\right)_{l\in(n_{i-1},n_i],k\in(0,n_{i-1}]} = \left(D_{p_m p_l}\right)_{l\in(n_{i-1},n_i],m\in(0,n_{i-1}]} \cdot \left(D_{q_k p_m}\right)_{m\in(0,n_{i-1}],k\in(0,n_{i-1}]}$$

The second factor is provided by the previous steps, we still have to obtain the first factor. Using the implicit function theorem for the Equation (6.4), we have

$$\left(D_{p_m p_l}\right)_{l\in(n_{i-1},n_i],m\in(0,n_{i-1}]} = -\left(\left(D_{p_l S_j}\right)_{j\in(n_{i-1},n_i],l\in(n_{i-1},n_i]}\right)^{-1} \left(D_{p_m S_j}\right)_{j\in(n_{i-1},n_i],m\in(0,n_{i-1}]}.$$

We now have all the required results to keep track of the full transition matrix.

In the following example we use an implementation[1] of the above construction. We build four curves, all of them for instruments collateralized at USD Fed Fund: USD Fed Fund and discounting, USD Libor 3M, EUR discounting and EUR Euribor 3M. The curves are built in three units. The first one with the USD discounting curve calibrated to OIS, the second one with USD forward calibrated to IRS and the third one with both the EUR discounting and the EUR Euribor forward 3M calibrated to cross-currency swaps and EUR IRS. We use the hypothesis of independence of the market quotes of IRS Euribor 3M from the collateral currency to avoid convexity adjustment. This simplify the computations but not the dependency between the curves, which is the feature we want to emphasize in this section.

We look at the transition matrix for the EUR discounting. The transition matrix will depend potentially on all the other curves. Using the notation of our implementation, we have the summary dependency in Table 6.1. It has to be interpreted as follows: the generalized transition matrix depends on four

[1] The implementation we used is the OpenGamma Strata 1.0 library. It is open source and available on GitHub at `https://github.com/OpenGamma/Strata`

Table 6.1 Summarized representation of the dependency of the EUR with collateral in USD Fed Fund discounting curve to the other curves of the example

```
name=USD-DSCON-OIS, parameterCount=11,
name=EUR-DSC-XCCY, parameterCount=11,
name=EUR-EURIBOR3M-IRS, parameterCount=10,
name=USD-LIBOR3M-IRS, parameterCount=10
```

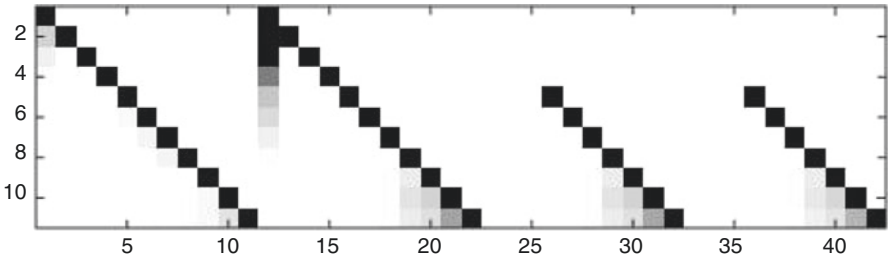

Fig. 6.1 Visual representation of the transition matrix of the EUR discounting for collateral in USD with Fed Fund rates

curves (with the listed names), for each curve the associated integer is the number of parameters of the curve. In this simplified example we have 42 parameters.

The total transition matrix for the last curve has 11 rows and 42 columns and is too large to be represented directly in this document. Instead we give a graphical representation of it in Fig. 6.1. The dark squares indicate a strong dependency (absolute value above 0.10) and the grey squares a low dependency down to the white squares where the dependency is 0.

The discounting curve in EUR for collateral in USD obviously depends on the cross-currency instruments (FX swaps and cross-currency swaps), but also on the USD discounting (OIS), USD single currency IRS and EUR single currency IRS.

Using this huge quantity of information and managing the dependencies is probably the main challenge in collateralized cross-currency trading and risk management. Algorithmic differentiation is helping not only to quicken the computation but also to clarify the dependencies.

Once the curve calibration process is finished, we suggest to store the inverse Jacobian matrix, at least the non-zero blocks of it, for later use. The information required for one curve, is the list of curves of which its parameters depends and the number of those parameters. In Strata, the object containing the curve name and its parameter count is called `CurveParameterSize`. The matrix itself is stored in a `JacobianCalibrationMatrix`. The simplified versions of the code are provided in Listing 6.1. An example of content for such an object is proposed in Table 6.1.

Listing 6.1 Curve Jacobian calibration matrix with reference to other curves and their market quote sizes

```
public class CurveParameterSize {
  /** The curve name. */
  private CurveName name;
  /** The number of parameters. */
  private int parameterCount;
  ...
}

public class JacobianCalibrationMatrix{
  /** The curve order. */
  private List<CurveParameterSize> order;
  /** The inverse Jacobian matrix. */
  private DoubleMatrix jacobianMatrix;
  ...
}
```

6.2 Model Calibration: Exact Calibration

We now discuss model calibration beyond the interest rate curves calibration. The content of this section is inspired from Henrard (2014a). Similar developments in the framework of PDE solutions in finance are proposed in Capriotti et al. (2015).

The method is first presented using a simple example, allowing simplified notation. The notation will be generalized later.

Suppose we have implemented the code for the function $f : \mathbb{R}^{p_a} \to \mathbb{R}^{p_z}$

$$z = f(a).$$

We suppose also that within the algorithm to compute f, there is an equation to solve. The algorithm is decomposed into

$$\begin{aligned} b &= g_1(a) \\ c \text{ s. t. } & g_2(b,c) = 0 \\ z &= g_3(c) \end{aligned}$$

with $g_1 : \mathbb{R}^{p_a} \to \mathbb{R}^{p_b}$, $g_2 : \mathbb{R}^{p_b} \times \mathbb{R}^{p_c} \to \mathbb{R}^{p_c}$ and $g_3 : \mathbb{R}^{p_c} \to \mathbb{R}^{p_z}$. The second part of the algorithm is a multi-dimensional equation that must be solved. It is assumed that all intermediary functions g_i are differentiable. How to write the Algorithmic Differentiation version of this function? The middle part of the algorithm does not match the Single Assignment Code property described in Section 2.3.

Suppose that the adjoint versions of the functions g_i ($1 \leq i \leq 3$) are known but the adjoint version for the solver is unknown, this is the derivatives of the function that computes c from b are unknown. The implicit function theorem ensures – under certain mild conditions which are described in Appendix A.3 – that the process that

produces c as function of b is actually differentiable and links its derivative to the derivatives of g_2. We denote by g_4 the implicit (and unknown) function associating c to b, this is the second part of the algorithm can be replaced by $g_4(b) = c$. We don't have code for the function g_4, we only know that the abstract version of the function exists around the point of interest. The implicit function theorem guarantees that g_4 exists and provides a way to compute its derivative based on the derivatives of the known function g_2 as

$$Dg_4(b) = -\left(D_2 g_2(b,c)\right)^{-1} D_1 g_2(b,c).$$

Solving the equation $g_2(b,c) = 0$ is usually much more time-consuming than simply computing one value of g_2. The standard approach is to use a Newton-like algorithm and to repeat the valuation of g_2 with guessed values. In the implicit function theorem approach, using the adjoint version, there is no need to solve the equation again for the derivatives and there is no requirement to have adjoint version of the solver and its repeated guesses, only the derivatives of the function g_2 are used. Those derivatives are usually computed themselves by algorithmic differentiation. The adjoint method used in this way will give better results than the normal approach as there is no need to solve the equation for g_2 again. If the root-finding is a significant part of the computation time for f, the time required to compute the function f and all of its derivatives will be usually less than twice the time taken to calculate one value. The slower the root-finding problem is, the better on a relative basis for the enhanced AD approach.

The standard notation in AAD is to denote the derivative of the final value z with respect to an intermediate value x by $\bar{x}$. The literature uses different notations with regards to the transposition of $\bar{x}$; here we use

$$\bar{x} = \left(D_x z(x)\right)^T,$$

this is the *bar* variables are column vectors if z is of dimension one, or matrices with p_x rows and p_z columns if z is of dimension greater than one.

The adjoint version of the algorithm is

$$\begin{aligned}
&\bar{z} = I \quad \text{(with } I \text{ the } p_z \times p_z \text{ identity)}\\
&\bar{c} = \left(D_c g_3(c)\right)^T \bar{z}\\
&\bar{b} = \left(D_b g_4(b)\right)^T \bar{c} = -\left(\left(D_c g_2(b,c)\right)^{-1} D_b g_2(b,c)\right)^T \bar{c}\\
&\bar{a} = \left(D_a g_1(a)\right)^T \bar{b}.
\end{aligned}$$

All the parts of the algorithm are now explicit. The implicit root-finding part has been replaced by an explicit matrix manipulation involving one matrix inversion.

6.2.1 Calibration Technique Description

One frequent part of exotic instruments pricing in finance is a *complex model calibration*. The process involves two model, one base model to price vanilla instruments and one complex model to price the exotic instrument. The process can be synthetizes as follows:

- The price of an *exotic instrument* is related to a specific basket of *vanilla instruments*;
- The price of these vanilla instruments is computed in a given *base model*;
- The *complex model parameters* are calibrated to fit the vanilla option prices from the base model. This step is usually done through a generic numerical equation solver; and
- The exotic instrument is then priced with the calibrated complex model.

We want to differentiate the exotic price with respect to the parameters of the base model. In the bump and recompute approach, this corresponds to computing the risks with model recalibration. As the calibration can represent a major fraction of the total computation time of the procedure listed above, an alternative method is highly desirable.

In this section, the generic AD and implicit function theorem method described in the first part of the section is applied to an interest rate model calibration case. Let $\mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Base}}$ be the present values of a set of vanilla financial instruments used for calibration of the base model. The data required for the pricing are the yield curves in a multi-curve framework, which are denoted C, and market volatility parameters – for example SABR parameters or Black volatilities – for the base model. The parameters for the base model are denoted Θ. The exotic model provides a present value for the same vanilla options with the same curves but using a different set of parameters with different meaning. The set of parameters for the exotic model are denoted Φ. The pricing function for the vanilla options in the calibrated model is denoted $\mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}$. The calibration procedure consists in finding the parameters Φ which solve

$$f(C, \Theta, \Phi) = 0. \tag{6.5}$$

For perfect calibration, the function is simply

$$f(C, \Theta, \Phi) = \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Base}}(C, \Theta) - \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(C, \Phi).$$

Equation 6.5 will be multi-dimensional when there are several calibrating instruments. We suppose that there are as many calibration instruments as parameters to be calibrated in Φ.

In practice, some models may have more free parameters than calibrating instruments. In this case the model parameters are constrained in such a way that there are

the same number of degrees of freedom as the number of calibrating instruments. The second example below calibrates a two-factor LMM with many parameters. The parameters Φ used here are those degrees of freedom, not the original model parameters.

With the calibration procedure, we obtain implicit calibrated model parameters from the original model parameters and the curves:

$$\Phi = \Phi(C, \Theta).$$

The function is obtained through the equation solving procedure; there is no explicit solution or even explicit code that produces the function Φ directly from the inputs C and Θ.

The exotic option is priced from the calibrated model through the pricing $\mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}(C, \Phi)$. With the implicit function above we can define

$$\mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Base}}(C, \Theta) = \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}(C, \Phi(C, \Theta))$$

We are interested in the derivative of the exotic option with respect to the curves and the base model parameters Θ. The goal is to see all the risk, for the vanilla instruments and the exotic instruments written in term of the market quotes used in the liquid vanilla market. This gives a coherent view of the risk for a portfolio using heterogeneous models.

With the derivative versions of $\mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}$, we can compute the derivatives

$$D_C \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}} \text{ and } D_\Phi \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}.$$

The quantities of interest are

$$D_C \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Base}} \text{ and } D_\Theta \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Base}}.$$

Through composition we have

$$\begin{aligned} D_C \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Base}}(C, \Theta) = {} & D_C \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}(C, \Phi(C, \Theta)) \\ & + D_\Phi \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}(C, \Phi(C, \Theta)) D_C \Phi(C, \Theta), \end{aligned}$$

and

$$D_\Theta \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Base}}(C, \Theta) = D_\Phi \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}(C, \Phi(C, \Theta)) D_\Theta \Phi(C, \Theta).$$

The quantities $D_C \Phi$ and $D_\Theta \Phi$ are still unknown at this stage. Using the implicit function theorem, the function Φ is differentiable and its derivatives can be computed from the derivative of f:

$$D_\Theta \Phi(C, \Theta) = -\left(D_\Phi f(C, \Theta, \Phi(C, \Theta))\right)^{-1} D_\Theta f(C, \Theta, \Phi(C, \Theta))$$

and

$$D_C\Phi(C,\Theta) = -\left(D_\Phi f(C,\Theta,\Phi(C,\Theta))\right)^{-1} D_C f(C,\Theta,\Phi(C,\Theta)).$$

In the perfect calibration case, those equations reduce to

$$D_\Theta\Phi(C,\Theta) = \left(D_\Phi \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(C,\Phi(C,\Theta))\right)^{-1} D_\Theta \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Base}}(C,\Theta)$$

and

$$D_C\Phi(C,\Theta) = \left(D_\Phi \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(C,\Phi(C,\Theta))\right)^{-1} \left(D_C \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Base}}(C,\Theta) - D_C \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(C,\Phi(C,\Theta))\right).$$

We have described the theoretical results leading to an efficient implementation of sensitivity computation through the calibration procedure. The procedure only requires the implementation of the derivatives of the PV functions and a little bit of linear algebra.

6.2.2 Examples

In line with the above technique, we would like to price and compute the sensitivities of exotic swaptions in a physical delivery SABR framework. For all the required pricing algorithms the adjoint versions have been implemented[2].

6.2.2.1 Cash Swaptions in the Hull-White Model

In the first example our *exotic* instrument is a cash-settled swaption and our vanilla basket is composed of a unique physical delivery swaption. The base model is a SABR model on the swap rate. The curve framework is multi-curve (one discounting curve and one forward curve). The model parameters Θ are the SABR parameters α, ρ and ν (β is set to 0.50). The calibrated model is a Hull-White one factor (extended Vasicek) model with constant volatility. The parameter of the Hull-White model to calibrate is the constant volatility. The pricing algorithm in the Hull-White model for the physical delivery swaption is described in Henrard (2003), and the pricing algorithm used for the cash-settled swaption is the efficient approximation described in Henrard (2010a).

For this example, we use a 1Y $\times$ 9Y swaption on an annual vs 6m Euribor swap. There are three SABR sensitivities (α, ρ, and ν) and 38 rate sensitivities (19 on each curve). The performance results are provided in Table 6.2. The computation of the

[2] The implementations used for the performance figures are those in the OpenGamma analytics library. The computations are done on a Mac Pro 3.2 GHz Quad-core.

Table 6.2 Performance for different approaches to derivatives computations: cash settled swaption in Hull-White one factor model

Risk type	Approach	Price time	Risks time	Total
SABR	Finite difference	1.00	3×1.00	4.00
SABR	AAD and implicit function	1.00	0.28	1.28
Curve	Finite difference	1.00	38×1.00	39.00
Curve	AAD and implicit function	1.00	0.56	1.56
Curve and SABR	Finite difference	1.00	41×1.00	42.00
Curve and SABR	AAD and implicit function	1.00	0.83	1.83

Times relative to the pricing time. The pricing time is 0.45 second for 1000 swaptions.

three SABR derivatives add less than 30% to the pricing computation time in this approach; a one-sided finite difference computation would have add at least 300%.

The same comparison was performed for the interest rate sensitivities. The finite difference would require at least 39 price time (3900%). The proposed approach adds only 0.55 price time (55%). In total, the proposed algorithm is around 23 times faster than a finite difference approach and is numerically more stable.

To partly compare the above numbers with the results of Schlenkirch (2012), we also report figures for a Hull-White one factor model with piecewise constant volatility. The set-up in the above paper is different but the underlying model is similar. The computation time for the finite difference and adjoint evaluations of the Jacobian with respect to the piecewise constant volatility for a 30Y and 100Y swap (annual volatility dates) is provided. The Jacobian is the derivative of all European swaption prices with respect to all volatilities in the model. The 30Y Jacobian computation requires 0.110 second by finite difference and less than 0.015 second by algorithmic differentiation. This is approximately 14% of the runtime and is in line with Schlenkirch (2012) figures (20%). The corresponding figures for the 100Y case are 20.2 s and 0.42 s (2%).

6.2.2.2 Amortized Swaptions in LMM

In this example, the exotic instrument is an amortized European swaption (i.e. a swaption with decreasing notional), and the vanilla basket is composed of vanilla European swaptions with same expiry and increasing maturities. The amortized swaption has a 10Y maturity and yearly amortization. The calibrating instruments are ten vanilla swaptions with yearly maturities between 1Y and 10Y and same strike as the amortized swaption.

The base model is a SABR model on each vanilla swaption. The complex model is a two-factor LMM with displaced diffusion and Libor period of six months. The pricing method for the vanilla and the amortized swaption is the efficient approximation described in Henrard (2010b).

The calibration is performed as follows: for each yearly period the weights of the different parameters (four in each year) are fixed. The calibration is done by

Table 6.3 Performance for different approaches to derivatives computations: amortized swaption in the LMM

Risk type	Approach	Price time	Risks time	Total
SABR	Finite difference	1.00	30 × 1.00	31.00
SABR	AAD and implicit function	1.00	0.18	1.18
Curve	Finite difference	1.00	42 × 1.00	43.00
Curve	AAD and implicit function	1.00	0.74	1.74
Curve and SABR	Finite difference	1.00	72 × 1.00	73.00
Curve and SABR	AAD and implicit function	1.00	0.75	1.75

Times relative to the pricing time. The valuation time is 0.425 second for 250 swaptions.

multiplying those weights by a common factor. The parameter Φ in the previous section are the multiplicative factors (10 in total), even if in practice the derivatives with all the model parameters (40 in total) are computed as an intermediary step.

The computation time results for the SABR and curve sensitivities are reported in Table 6.3. There are 30 SABR sensitivities (α, ρ, ν for 10 vanilla swaptions). In the described approach, the 30 sensitivities add only 20% to the computation time with respect to the calibration and price computation .

There are 42 curve sensitivities (two curves, semi-annual payments over 10 years). The computation of the 42 sensitivities takes only 74% of the price time. In total, the AAD approach is approximately 2.5% of the time required by finite difference. Note that computing the curve and SABR sensitivities take approximately the same amount of time as computing the curve sensitivities alone as most of the computations are common.

Similar results for amortized swaptions of different maturities and with different numbers of calibrating instruments are reported in Fig. 6.2. The ratios between the present value and sensitivities time and the present value time are reported for the finite difference approach and Adjoint Algorithmic Differentiation approach using the implicit function method described in the previous section. The implicit function AAD method ratio is almost independent of the number of sensitivities. In all cases but the 30Y swaptions (where 212 sensitivities are calculated), the ratio is below two. For the finite differences, the ratio is above 200, a gain of more than a factor 100. This mean the computation time is reduced from one hour to less than one minute.

6.3 Model Calibration: Least-Square

In this section we develop techniques similar to the one developed in Section 6.2 but for the case where the calibration is not a root-finding calibration but a least square calibration.

Here we consider the case were the parameters Φ are obtained through a (weighted) least square process. Suppose that there are n calibrated parameters in Φ and $m \geq n$ instruments for the calibration process. The weights associated to

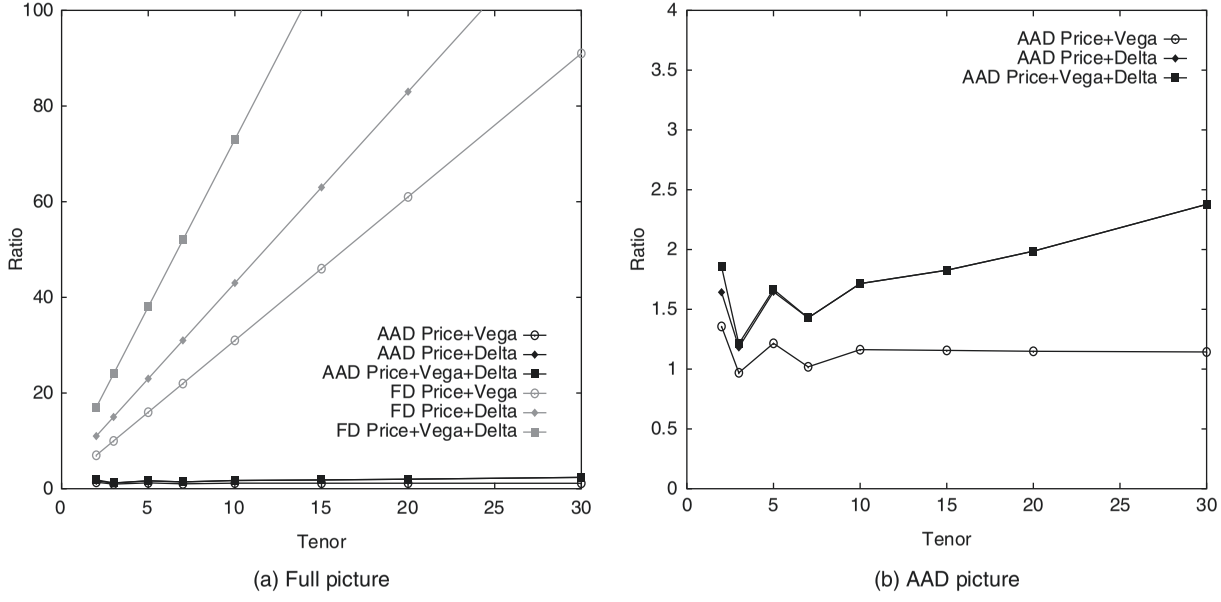

Fig. 6.2 Computation time ratios (present value and sensitivities time to present value time) for the finite difference and AAD methods. The *vega* represents the derivatives with respect to the SABR parameters; the *delta* represents the derivatives with respect to the curves. The AAD method uses the implicit function approach. Figures for annually amortized swaptions in a LMM calibrated to vanilla swaptions in valued with the SABR model

each instrument are fixed and denoted $(w_i)_{i=1,\dots,m}$. The calibration parameters are defined as

$$\begin{aligned}\Phi_0 &= \arg\min_{\Phi\in\mathbb{R}^n} h(C_0,\Theta_0,\Phi)\\ &= \arg\min_{\Phi} \sum_{i=1,\dots,m} w_i \left(\mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Base}}(i,C_0,\Theta_0) - \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C_0,\Phi)\right)^2. \quad (6.6)\end{aligned}$$

At the minimum Φ_0, the derivatives of h with respect to Φ are 0:

$$f(C_0,\Theta_0,\Phi_0) = D_\Phi h(C_0,\Theta_0,\Phi_0) = 0.$$

The minimum satisfies Eq. 6.5 with the function f defined above. This last equation is a n unknown and n equation system. The above derivatives can be computed explicitly as

$$\begin{aligned}&D_\Phi h(C,\Theta,\Phi)\\ &= -2 \sum_{i=1,\dots,m} w_i \left(\mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Base}}(i,C,\Theta) - \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi)\right) D_\Phi \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi).\end{aligned}$$

With the calibration procedure, we obtain calibrated model parameters from the original model parameters and the curves:

$$\Phi_0 = \Phi(C_0,\Theta_0).$$

The parameters are obtained through the optimization procedure; there is no explicit solution or even explicit code that produces those parameters directly. We suppose that f is regular enough and that $D_\Phi f(C_0,\Theta_0,\Phi_0)$ is invertible, so we can apply the implicit function theorem.

The implicit function theorem states that there exists a function $\Phi(C,\Theta)$ such that

$$f(C,\Theta,\Phi(C,\Theta)) = 0$$

for (C,Θ) close to (C_0,Θ_0) and there are no other solution in a neighborhood. We still need to prove that the function $\Phi(C,\Theta)$ gives a minimum of the original problem (6.6) and not only a point with 0 derivatives, like a saddle point.

Let m_0 denote the minimum value of (6.6) at Φ_0, this is $m_0 = h(C_0,\Theta_0,\Phi_0)$. As Φ_0 is a minimum, $D_\Phi f(C_0,\Theta_0,\Phi_0)$ is positively defined. As it is invertible, it is strictly positively defined. Using those properties, one can show that there exists a $\epsilon > 0$ and a sphere around Φ_0 such that $h(C_0,\Theta_0,\Phi) > m_0 + 3\epsilon$ for Φ on the sphere. As h is continuous, for (C,Θ) close enough to (C_0,Θ_0) and Φ on the sphere, $h(C,\Theta,\Phi) > m_0 + 2\epsilon$. On the other side, $h(C,\Theta,\Phi(C,\Theta)) < m_0 + \epsilon$ for (C,Θ) close enough to (C_0,Θ_0). This proves that h has a minimum in the interior of the disk in the Φ dimension. Being in the interior, the minimum has zero derivatives. From the result of the implicit function theorem, $\Phi(C,\Theta)$ is the only zero. This proves that

the implicit function $\Phi(C, \Theta)$ is not only a zero of the derivative function f but also a minimum of the least square problem.

The exotic option is priced from the calibrated model through the pricing $\mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}(C, \Phi)$. With the implicit function above we can define

$$\mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Base}}(C, \Theta) = \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}(C, \Phi(C, \Theta))$$

We are interested in the derivative of the exotic option with respect to the curves and the base model parameters Θ.

The quantities of interest are

$$D_C \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Base}} \text{ and } D_\Theta \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Base}}.$$

With the AD versions of $\mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}$, we can compute the derivatives

$$D_C \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}} \text{ and } D_\Phi \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}.$$

Through composition we have

$$\begin{aligned} &D_C \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Base}}(C_0, \Theta_0) \\ &= D_C \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}(C_0, \Phi(C_0, \Theta_0)) + D_\Phi \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}(C_0, \Phi(C_0, \Theta_0)) D_C \Phi(C_0, \Theta_0), \end{aligned}$$

and

$$D_\Theta \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Base}}(C_0, \Theta_0) = D_\Phi \mathrm{PV}^{\mathrm{Exotic}}_{\mathrm{Calibrated}}(C_0, \Phi(C_0, \Theta_0)) D_\Theta \Phi(C_0, \Theta_0).$$

Where $D_C \Phi$ and $D_\Theta \Phi$ are yet unknown. Using the implicit function theorem, the function Φ is differentiable and its derivatives can be computed from the derivative of f:

$$D_\Theta \Phi(C_0, \Theta_0) = -\left(D_\Phi f(C_0, \Theta_0, \Phi(C_0, \Theta_0))\right)^{-1} D_\Theta f(C_0, \Theta_0, \Phi(C_0, \Theta_0))$$

and

$$D_C \Phi(C_0, \Theta_0) = -\left(D_\Phi f(C_0, \Theta_0, \Phi(C_0, \Theta_0))\right)^{-1} D_C f(C_0, \Theta_0, \Phi(C_0, \Theta_0)).$$

We need to describe $D_\chi f$.

$$\begin{aligned} D_\Theta f(C, \Theta, \Phi) =& D_\Theta D_\Phi h(C, \Theta, \Phi) \\ =& -2 \sum_{i=1,\ldots,n} w_i D_\Phi^T \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i, C, \Phi) D_\Theta \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Base}}(i, C, \Theta). \end{aligned}$$

$$
\begin{aligned}
D_C f(C,\Theta,\Phi) = \; & -2 \sum_{i=1,\ldots,n} w_i D_\Phi^T \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi) \left(D_C \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Base}}(i,C,\Theta) \right. \\
& \left. - D_C \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi) \right) - 2 \sum_{i=1,\ldots,n} w_i \left(\mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Base}}(i,C,\Theta) \right. \\
& \left. - \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi) \right) D_C D_\Phi \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi)
\end{aligned}
$$

$$
\begin{aligned}
& D_\Phi f(C,\Theta,\Phi) \\
& = D_\Phi D_\Phi h(C,\Theta,\Phi) = 2 \sum_{i=1,\ldots,n} w_i D_\Phi \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi) D_\Phi^T \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi) \\
& -2 \sum_{i=1,\ldots,n} w_i \left(\mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Base}}(i,C,\Theta) - \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi) \right) D_\Phi^2 \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi).
\end{aligned}
$$

The *annoying* parts are the second order derivatives parts. Usually the first order derivatives are implemented in AD frameworks but not the second order one. Fortunately in the above formula, all the second order derivatives are multiplied by $\mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Base}}(i,C,\Theta) - \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi)$ which is small when the calibrated model can match the base prices well enough. Based on that, we can use the following approximations:

$$
\begin{aligned}
& D_C f(C,\Theta,\Phi) \\
& \simeq -2 \sum_{i=1,\ldots,n} w_i D_\Phi^T \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi) \left(D_C \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Base}}(i,C,\Theta) \right. \\
& \qquad \left. - D_C \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi) \right)
\end{aligned}
$$

and

$$
D_\Phi f(C,\Theta,\Phi) \simeq 2 \sum_{i=1,\ldots,n} w_i D_\Phi^T \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi) D_\Phi \mathrm{PV}^{\mathrm{Vanilla}}_{\mathrm{Calibrated}}(i,C,\Phi).
$$

This is very similar to the approximation done in computing Hessian as described in (Press et al., 1988, Section 14.4).

6.3.1 Least Square Examples

In this section we analyze an example similar to the second one of the previous example section.

The calibrated model is a Libor Market Model with displaced diffusion. We calibrate two parameters for each maturity: the volatility and the displacement parameters. The volatility parameter guides the general level of the smile while the

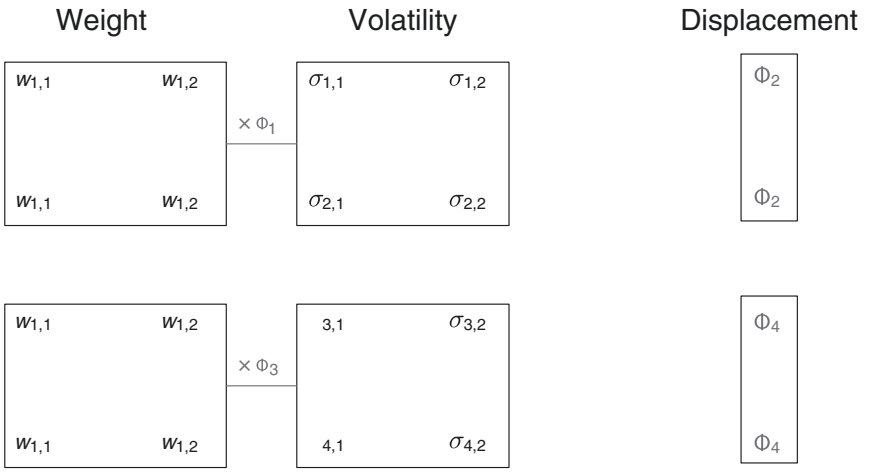

Fig. 6.3 Representation of LMM calibration for a 2 years amortised swaption. Each yearly volatility block is multiplied by a common multiplicative factor and each yearly displacement block contains the same number

displacement parameter commands the skew of the smile. We calibrate the two parameters by a least square approach on the prices of swaptions with several strikes. In the tests we use between 2 and 6 strikes.

The calibration is done for each yearly block on a multiplicative factor to given weights to obtain volatilities and on a shared displacement. The graphical representation of the calibration for a 2 year problem is provided in Fig. 6.3.

The more difficult the calibration is, the better the results of AD with implicit function will be on a relative basis. The algorithmic differentiation with implicit theorem method is using the already computed calibration in the sensitivity.

6.3.1.1 Reference

The reference example has a tenor of 5 years and there are five annual calibrations. The calibration on each tenor is done on three strikes (–100, 0, +100) bps from ATM. In the finite difference approach, the ratio is roughly 3 (SABR) + 4 (semi-annual payments with 2 curve) for each years. For a 5 years tenor, the finite difference ratio is around 36. The ratio obtained in practice in this example through AD with implicit function is 1.40.

6.3.1.2 Tenors

We run the same test with several tenors, between 2 years and 30 years. The calibration is similarly annual on three swaptions for each calibration date. In a finite difference the ratios would increase roughly linearly with the tenor. Figure 6.4 reports the results.

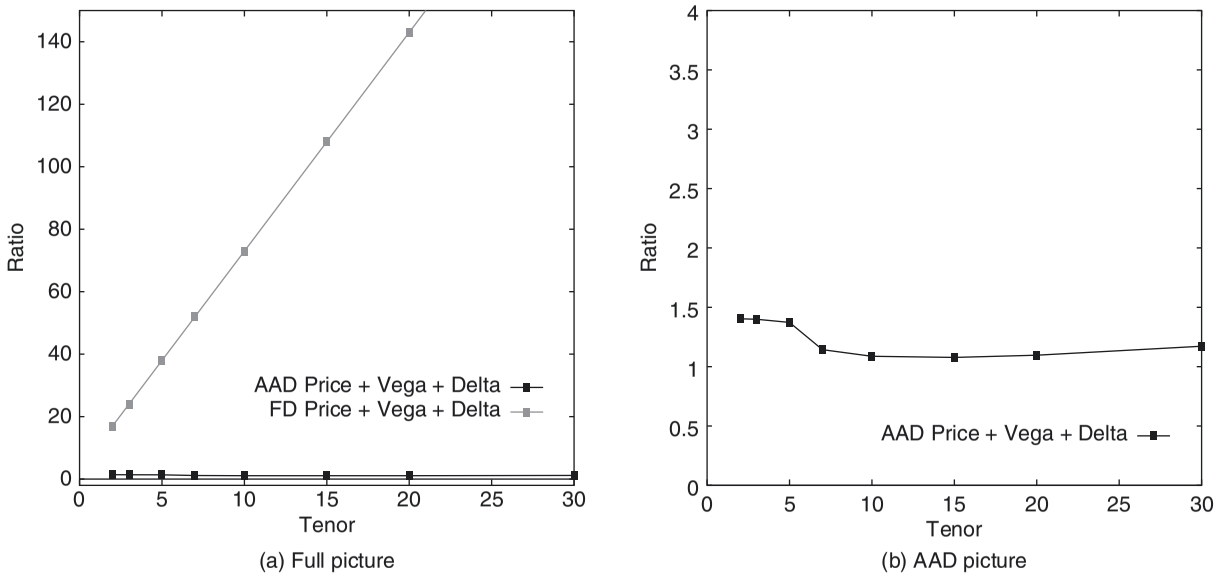

Fig. 6.4 Computation time ratios (price and sensitivities time to price time) for the finite difference and AAD methods. The *vega* represents the derivatives with respect to the SABR parameters; the *delta* represents the derivatives with respect to the curves. The AAD method uses the implicit function approach. Figures for annually amortised swaptions in a LMM calibrated yearly to three vanilla swaptions in SABR

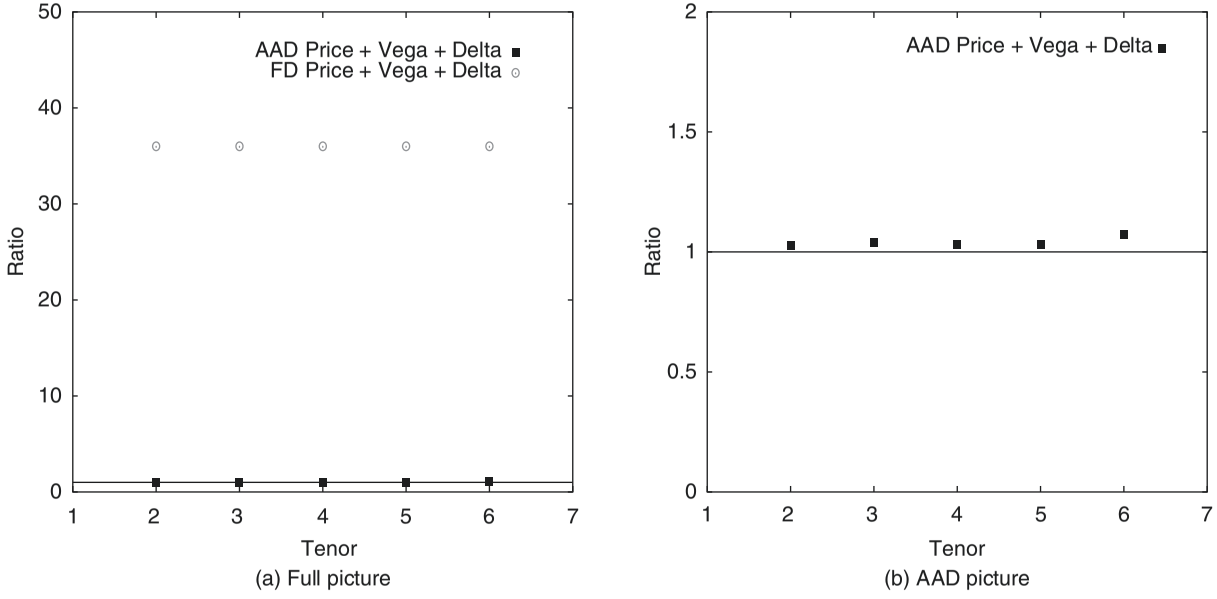

Fig. 6.5 Computation time ratios (price and sensitivities time to price time) for the finite difference and AAD methods. The *vega* represents the derivatives with respect to the SABR parameters; the *delta* represents the derivatives with respect to the curves. The AAD method uses the implicit function approach. Figures for annually amortised swaptions in a LMM calibrated yearly to the given number of vanilla swaptions with different strikes in SABR

As in the previous examples, the metric to analyze the efficiency is the ratio between price and derivatives time and price time. The derivatives are composed of the SABR and curve sensitivities.

The linear increase of the ratios with the tenors is obvious for the finite difference method. The AD with implicit function method achieves a relatively constant ratio which is barely above 1 and well below 1.5. This is well below the theoretical upper bound of $\omega_A \in [3, 4]$. In the case of the 20-year swaption, the gain between the finite difference and optimized AD with implicit function is around 100. In practice this is reducing the computation time from 1 hour 40 minutes to 1 minute.

6.3.1.3 Strikes

In this part of the analysis, we go back to a 5 year amortised swaption with annual calibration. We run similar tests with calibrations sets for each periods with 2–6 strikes. Figure 6.5 reports the results.

As can be seen, the ratios are independent of the number of calibrating strikes for the finite difference and the AD versions. The ratio obtained in the AD with implicit function case is around 1.1, which is well below the theoretical upper bound of $\omega_A \in [3, 4]$. The ratio for the finite difference approach is around 36 and independent of the number of strikes.

Bibliography

Capriotti, L., Jiang, Y., and Macrina, A. (2015). Real-time risk management: An AAD-PDE approach. *Journal of Financial Engineering*, 2(4):1–31.

Henrard, M. (2003). Explicit bond option and swaption formula in Heath-Jarrow-Morton one-factor model. *International Journal of Theoretical and Applied Finance*, 6(1):57–72.

Henrard, M. (2010a). Cash-Settled Swaptions: How Wrong are We? Working paper series 1703846, SSRN. Available at SSRN: http://ssrn.com/abstract=1703846.

Henrard, M. (2010b). Swaptions in Libor Market Model with local volatility. *Wilmott Journal*, 2(3):135–154.

Henrard, M. (2014a). Adjoint algorithmic differentiation: Calibration and implicit function theorem. *Journal of Computational Finance*, 17(4):37–47.

Henrard, M. (2014b). *Interest rate modelling in the multi-curve framework: Foundations, evolution and implementation*. Applied Quantitative Finance. London:Palgrave Macmillan. ISBN: 978-1-137-37465-3.

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. (1988). *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge:Cambridge University Press.

Schlenkirch, S. (2012). Efficient calibration of the Hull-White model. *Optimal Control Applications and Methods*, 33(3):352–362.

Appendix A
Mathematical Results

The most powerful force in the universe is composition. – What we need is implicit.

A.1 Derivative Notations

Definition A.1 (Derivative) A function $f : \mathbb{R}^m \to \mathbb{R}^n; x \mapsto f(x)$ is said to be *differentiable* at a point $x_0 \in \mathbb{R}^m$ if f is defined on that point and there exist a linear function $Df(x_0) : \mathbb{R}^m \to \mathbb{R}^n$ such that

$$\lim_{\epsilon \to 0; \epsilon \in \mathbb{R}^m} \frac{f(x_0 + \epsilon) - (f(x_0) + Df(x_0)(\epsilon))}{|\epsilon|} = 0.$$

The linear function $Df(x_0)$ is called the *derivative* of f in x_0.

If the input vector is split in several sub-vectors, like in $(a_1, a_2) \mapsto z = f(a_1, a_2)$, the partial derivative with respect of one of the partial vector is denoted $D_i f(a_1, a_2)$ or $D_{a_i} f(a_1, a_2)$ with $i = 1$ or 2.

The derivative is a linear function with $Df(x) \in \mathcal{L}(\mathbb{R}^m, \mathbb{R}^n)$. The derivative is represented by a $n \times m$ matrix (n rows, m columns). For column vectors a and ϵ, we will often used the approximation

$$f(a + \epsilon) \sim f(a) + Df(a) \cdot \epsilon.$$

A.2 Derivative of Composition

Theorem A.1 Let $g_1 : \mathbb{R}^{p_a} \to \mathbb{R}^{p_b}$ be differentiable in a and $g_2 : \mathbb{R}^{p_b} \to \mathbb{R}^{p_z}$ be differentiable in $g_1(a)$. Then the function

$$(g_2 \circ g_1) : \mathbb{R}^{p_a} \to \mathbb{R}^{p_z}; a \mapsto g_2(g_1(a))$$

is differentiable in a and

$$D(g_2 \circ g_1)(a) = D(g_2(g_1(a)) \cdot Dg_1(a).$$

M. Henrard, *Algorithmic Differentiation in Finance Explained*,
Financial Engineering Explained, DOI 10.1007/978-3-319-53979-9

The derivative operator transform the composition of function into a composition of linear function, represented by a matrix multiplication.

This result has a positive impact on the efficiency of Algorithmic Differentiation. The complex process of composition is replace at the derivative level by a matrix multiplication which can be performed very efficiently by computers.

A.3 Implicit Function Theorem

Theorem A.2 Le $f : \mathbb{R}^{n+m} \to \mathbb{R}^m$ be continuously differentiable. If (x_0, y_0) is such that

$$f(x_0, y_0) = 0$$

and if $D_y f(x_0, y_0)$ is invertible, then, in a neighborhood $X \times Y$ of (x_0, y_0), there is a (implicit) function g such that $f(x, g(x)) = 0$ for $x \in X$, $\{(x, g(x)) : x \in X\} = \{(x, y) \in X \times Y : f(x, y) = 0\}$, g is differentiable in x_0 and

$$Dg(x_0) = -(D_2 f(x_0, y_0))^{-1} D_1 f(x_0, y_0).$$

See for example Mawhin (1997) for more details and a proof of the result.

This result has also a positive impact on the efficiency of Algorithmic Differentiation. The complex process of root-finding is replaced at the derivative level by a matrix inversion which can also be performed very efficiently by computers.

Bibliography

Mawhin, J. (1997). *Analyse: Fondements, Techniques, Evolution*. Louvain-la-Neuve:De Boeck.

Index

M. Henrard, *Algorithmic Differentiation in Finance Explained*,
Financial Engineering Explained, DOI 10.1007/978-3-319-53979-9