Efficient Barrier Option Greeks using Automatic Differentiation

A Master Thesis with Handelsbanken Capital Markets

Gustav Hedin

Lunds Tekniska Högskola

January 2019

Abstract

Automatic Differentiation (AD) is an effective method for calculation of derivatives. It can evaluate an unlimited number of derivatives to a fixed cost relative to the computing time of the original function. The AD technique is used in many fields for large and complex calculations in order to get accurate values of derivatives fast. Banks and other financial institutions handle calculations of portfolio values which could depend on a large number of input variables. AD enables fast calculation and sensitivity analysis of these complex functions such as risks of day to day trading as well as XVA's. The accuracy of AD is also better than currently used methods for derivative calculations as it is not based on approximations. In this paper, AD in a Monte Carlo setting is studied and one way of implementing AD by operator overloading in its reverse mode is presented. Different ways to handle discontinuous payoffs are tested and applied on calculations of price and derivatives of Barrier Options. It is shown that AD combined with sigmoid smoothing of discontinuous payoffs gives accurate values for option price and Greeks with fast convergence rate.

Keywords: Automatic Differentiation, Greeks, Barrier Option, Vibrato Monte Carlo

Acknowledgements

I would like to thank Handelsbanken Capital Markets for giving me the opportunity to write my Masters Thesis together with them. A special thanks to my supervisor Martin Almqvist and the other staff at the department of quantitative analysis, for kindness, guidance and introduction to the subject.

Last but not least, I want to thank my supervisor at LTH, Professor Magnus Wiktorsson for outstanding help and feedback throughout the project.

Lund, January 17, 2019

Gustav Hedin

Nomenclature

- H - Barrier
- K - Strike price
- P - Option price
- S - Underlying asset
- σ - Volatility
- r - Interest rate
- t - Time
- θ - Input variable
- T - Maturity
- W - Wiener increment

Contents

1 Aim and Scope of Project

Pricing of portfolios and financial products is commonly done trough simulations. Besides the price, an investor needs to know how fluctuations in the market impacts the value of a his or her portfolio. To calculate these risks, or derivatives, additional simulations and calculations needs to be done. A bank, or financial institution handles large risk calculations such as CVA's and RWA which could depend on millions of variables. Evaluation of all these risks with traditional techniques soon becomes very time-consuming as the number of derivatives increases. The methods numerical accuracy is also of great importance in order to get reliable values.

Automatic Differentiation (AD) is a set of techniques for calculation of derivatives in a fast and efficient way. It is based on the chain rule and gives function values as well as derivatives with respect to all initial variables to a fix cost relative to the time it takes to evaluate the function itself. AD is a general computation technique which can be applied to a broad range of fields. It is already used in neural networks where thousands of derivatives needs to be evaluated, accurate and fast. AD can be performed in many ways and it does not seem to exist a clear and obvious best way of implementation.

Based on the problem formulated above, Handelsbanken Capital Markets wanted to investigate how AD can be used in calculation of prices and sensitivities of financial derivatives with Monte Carlo simulations. In particular, the scope of the thesis was to investigate how one can deal with barrier options and discontinuous payoffs when using AD in it's reverse mode. In this report, one way to implement reverse mode AD is presented. Different ways to deal with discontinuities are tested, both with and without AD. This is however neither a project to find the most accurate technique to calculate barrier option derivatives in general, nor a project to find the fastest and most efficient way to implement AD.

2 Background

The following section addresses theory and some definitions of financial modelling, used in this thesis.

2.1 Brownian Motion and Stochastic Differential Equations

Brownian motion describes the random individual movements of particles in a medium. The theory of these random fluctuations has been adapted throughout a broad range of fields and Norbert Wiener formulated a stochastic process for stationary independent increments. This is widely known as the Wiener Process and is used for describing fluctuations in asset prices. A Wiener process is characterized by the following properties:

1. $W_0 = 0$
2. Increments $W_{t+d} - W_t$, d ≥ 0 are independent from past values $W_s, s < t$
3. Increments $W_{t+d} - W_t$ is Gaussian and normally distributed $\mathcal{N}(0, u)$
4. W_t has continuous paths

Kiyoshi Itô developed a calculus framework for these stochastic processes which has enabled a broad range of caluculations to be made. Tomas Björk [2] gives a presentation of some common and useful definitions and formulas of SDE's and Itô calculus:

Definition: One-Dimensional Itô Formula - *Let f(t,x)* $\in C^{1,2}([0,T],\mathbb{R})$ *. Then,*

$$f(T,W_T) - f(0,0) = \int_0^T \partial_t f(t,W_t)dt + \int_0^T \partial_x f(t,W_t)dW_t + \frac{1}{2}\int_0^T \partial_{xx}^2 f(t,W_t)dt$$

where ∂_t *is the derivative w.r.t. the first argument and* ∂_x *to the second.*

When applying Itô's formula to the stochastic process $X_t = f(t, W_T)$, one gets

$$X_t = X_0 + \int_0^t \mu(s,W_s)ds + \int_0^t \sigma(s,W_s)dW_s$$

The functions μ and σ are called drift and diffusion and can be chosen freely to create any type of model dynamics. A common way to express a model like this is the *stochastic differential form*

$$dX_t = \mu(t,W_T)dt + \sigma(t,W_t)dW_t$$

A common and simple case of an SDE which underlies the Black-Scholes model is geometric brownian motion (GBM)

$$dS(t) = rSdt + \sigma SdW$$

which also can be expressed in its integrated form

$$S(T) = S_0 \exp\left((r - \frac{1}{2}\sigma^2)T + \sigma W(T)\right) \tag{1}$$

GBM is a common approximation for stock price simulations. When simulating GBM, the Euler discretization is useful

$$\hat{S}_{n+1} = \hat{S}_n + r\hat{S}_n h + \sigma\hat{S}_n \Delta W_n \tag{2}$$

where h is the simulation step size, r the interest rate, σ the volatility and ΔW_n a Wiener increment. It should be said that if one can use the exact expression for simulation, as given in equation 1, this gives a more thorough and exact result.

2.2 Black Scholes Framework for Financial Derivatives

A financial derivative is a contract based on, or *derived* from, one or many underlying assets. The contract can be specified to a particular market event and can also be customized to suit a certain buyer or investor. The simplest and most common case of a financial derivative is a call (or put) option. A call option gives the holder the right but not the obligation to buy a certain asset in the future for a specified price. The price paid for the underlying asset at exercise is the strike price K. The price P paid for the option is the maximum amount of money that can be lost. Based on the assets performance in the market during the time until maturity, an investor can gain a lot more. The payoff is the difference between the value of the asset at expiry date and the strike price. The Black and Scholes differential equation (3) is a well known tool when it comes to option pricing.

$$\frac{\partial P}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial P^2}{\partial S^2} + rS\frac{\partial P}{\partial S} - rP = 0 \tag{3}$$

Using the Black Scholes framework [7], the price of a plain vanilla call option can be expressed as

$$P_c(S,t) = S \cdot N(d_1) - Ke^{-r(T-t)} \cdot N(d_2)$$

where

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right]$$
$$d_2 = \frac{1}{\sigma\sqrt{T-t}}\left[ln\left(\frac{S}{K}\right) + \left(r - \frac{\sigma^2}{2}\right)(T-t)\right]$$

and $N(d)$ is the cumulative normal distribution

$$N(d) = \frac{1}{2\pi}\int_{-\infty}^{d} e^{\frac{-z^2}{2}} dz$$

Options can have various exercise properties and some of the most common ones are *European*, *American* and *Bermudan*. The difference between the categories lies in when the option is exercised. European options are always exercised on the last day of the contract, i.e. at maturity. American options on the other hand can be exercised at any time until maturity and Bermudan options is a mix of the two other categories such that the option can be exercised at two or more prespecified dates.

2.3 The Greeks

Hedging is an important and central concept throughout the financial sector. Whenever a bank buys or sells an asset a need may arise for another transaction to be carried out, in order to reduce the exposure to fluctuations in price of the asset. To know how changes in stock prices, volatility and interest rates impacts the banks

gains or losses, one needs to know the derivatives of prices with respect to a number of variables. Derivatives of option prices with respect to different market variables are known as *"the Greeks"*. The most commonly used Greeks are listed below.

- **Delta** measures changes in the value relative to the underlying asset; $\Delta = \frac{\partial P}{\partial S}$.
- **Vega** measures changes in the value relative to the volatility; $\nu = \frac{\partial P}{\partial \sigma}$
- **Theta** measures changes in the value relative to the time to expiry; $\Theta = \frac{\partial P}{\partial t}$
- **Rho** measures changes in the value relative to the interest rate; $\rho = \frac{\partial P}{\partial r}$
- **Gamma** is a second order derivative and measures changes in Delta with respect to the underlying asset; $\Gamma = \frac{\partial^2 P}{\partial S^2}$

2.4 Evaluating Sensitivities and Monte Carlo Simulations

Monte Carlo is a simulation method for evaluating non-trivial functions. When it comes to evaluation of functions of a random variable Z, one can use the Monte Carlo Simulation

$$\mathbb{E}[f(Z)] \sim \hat{f}(z) = \frac{1}{N}\sum_{i=1}^{N} f(z_i)$$

where N is the number of simulations and needs to be large in order to obtain an accurate value of $\mathbb{E}[Z]$ and z_i are random samples of Z. When a derivative of f is sought, the standard approach is to repeat the simulation with a slight shift in a specific variable of interest, θ_i. Then the differential can be obtained using finite (central) differences

$$\frac{\partial P}{\partial \theta} \approx \frac{P(\theta + \Delta\theta) - P(\theta - \Delta\theta)}{2\Delta\theta}$$

and second order derivatives via

$$\frac{\partial^2 P}{\partial \theta^2} \approx \frac{P(\theta + \Delta\theta) - 2P(\theta) + P(\theta - \Delta\theta)}{(\Delta\theta)^2}$$

Using multiple simulations and finite differences like this is a simple and straightforward method for derivative calculations but it is demanding with regards to computation time.

A Brownian Bridge is a stochastic process similar to Wiener process but conditioned on a start and ending value:

$$B_t \equiv (W_t \mid W_0 = a \ \ and \ \ B_T = b) \overset{\mathrm{d}}{=} a - \frac{t}{T}(b-a) + \tilde{W}_t - \frac{t}{T}\tilde{W}_t \tag{4}$$

where $\tilde{W}_t$ is a standard BM, starting at zero. The expectation of B is a straight line from a to b, $E[B_t] = a(1 - \frac{t}{T}) + b\frac{t}{T}$. Together with the randomness in the Wiener increments W_t, this gives a continuous path between a and b which can be formed in infinitely many ways.

2.5 Barrier Options

Barrier options is the term for option contracts with payoffs that are separated by a barrier which makes the payoff discontinuous. The four main categories of barrier options are

- Up and In
- Up and Out
- Down and In
- Down and Out

Using an up and out option as an example, it behaves as an usual option under the barrier. If the underlying price reaches this level, the option becomes worthless. The payoff of an option like this can be seen in figure 1. These types of barriers can be combined in a lot of versions and ways. The barrier itself can, analogous to its exercise times, be European, Bermudan or American. The simplest case is when only the value at maturity is checked. A contract with several discreet checks is also possible, i.e. a Bermudan barrier.

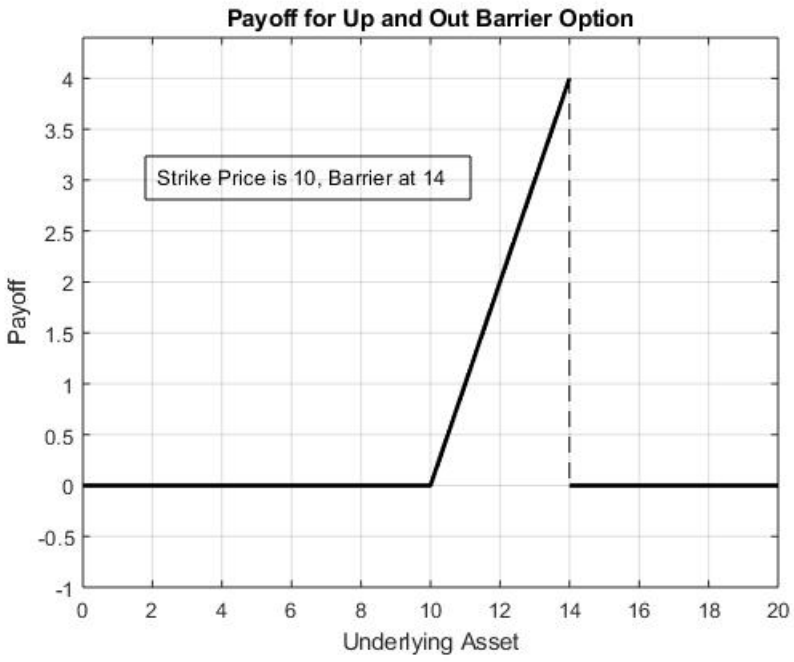


Figure 1: Up and Out Barrier Call Option - Payoff at expiry

In general, the payoff of a Barrier Option (up and out) takes the form

$$P = (S_T - K)^+ \prod_{\tau=\tau_1}^{\tau_n} I(S_\tau < H) \tag{5}$$

where I is the indicator function,

$$I(S < H) = \begin{cases} 1 & \text{if } \ S < H \\ 0 & otherwise \end{cases}$$

The derivative of the indicator function with respect to S exist in a theoretical point of view, namely as a dirac function

$$\delta(S) = \begin{cases} \infty & \text{if } \ S = 0 \\ 0 & otherwise \end{cases}$$

This gives the derivative of the Barrier option in (5) with respect to the underlying asset S to be

$$\frac{\partial P}{\partial S} = \frac{\partial}{\partial S}\left(\prod_{q=\tau_1}^{\tau_N} I(S_q < H)\right) \cdot (S_T - K)^+ + \prod_{q=\tau_1}^{\tau_N} \left(I(S_q < H)\right) \frac{\partial}{\partial S}(S_T - K)^+$$

Consider the simple case when the barrier is checked only at one point prior to expiration date i.g. $\tau_1 = \tau_N = T - 1$. The Δ for the option then takes the form

$$\begin{aligned} \Delta &= \frac{\partial}{\partial S}\left(I(S_q < H)\right) \cdot (S_T - K)^+ + I(S_q < H)\frac{\partial}{\partial S}(S_T - K)^+ \\ &= \delta(S_q - H) \cdot (S_T - K)^+ + I(S_q < H)\frac{\partial}{\partial S}(S_T - K)^+ \end{aligned}$$

All the terms in this expression can be evaluated with the above introduced theory, except for the dirac function, δ. Neither does it make sense in a theoretical way, nor is it in practise possible to multiply an expression with a factor of infinity in a numerical simulation. In other words, to obtain values for prices and Greeks of barrier options, one needs to make numerical round-offs and simplifications in simulations.

3 Automatic Differentiation

Automatic (or Algorithmic) Differentiation (AD) is a method for calculating derivatives. This is neither a numeric nor a symbolic approach, but rather a third way of calculating derivatives. It is based on the chain rule and the fact that basically every algorithm for evaluation or calculation of a function value consist of a sequence of basic operations such as plus, minus, times, exp etc. AD is not newly invented but

with the rise of more complex models in numerical science, financial computation and with neural networks, the need for a large number of derivatives to be calculated quickly, has risen dramatically.

> "Automatic Differentiation (AD) is a set of techniques based on the mechanical application of the chain rule to obtain derivatives of a function given as a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations such as additions or elementary functions such as exp(). By applying the chain rule of derivative calculus repeatedly to these operations, derivatives of arbitrary order can be computed automatically, and accurate to working precision"
>
> autodiff.org [11]

3.1 AD - Theory and Definitions

As stated above, the key to AD is the chain rule. When calculating derivatives, a long expression of functions and operations can be separated into different partial derivatives

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x}$$

Since scalar multiplication is a commutative operator, it does not matter if the partial derivatives are multiplied with each other in reverse order

$$\frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x} = \frac{\partial h}{\partial x}\frac{\partial g}{\partial h}\frac{\partial f}{\partial g}$$

This is why AD comes in two different modes; forward and reverse accumulation. The reverse mode of AD is often called adjoint mode AD or AAD for short. The difference between the two lies only in the implementation and calculation time. The same calculations are being made, just in opposite orders and hence it follows that the numerical results of the methods always match.

3.1.1 Forward Mode

In Forward Mode AD, the derivatives of a function are calculated by applying the chain rule starting from the input parameters. The calculation can be done via many intermediate variables, S, before the final function P and its derivatives are evaluated.

$$\dot{\alpha} \to \dot{S} \to \dot{P} \tag{6}$$

We want to find $\frac{\partial P}{\partial \alpha}$, the derivative of P with respect to a variable α. The expression is broken down to multiple terms via the chain rule. Using the notation common in the AD community, $\dot{\alpha}$, $\dot{S}$ and $\dot{P}$ denotes the derivative with respect to one single component,

$$\dot{S} = \frac{\partial S}{\partial \alpha}\dot{\alpha}, \quad \dot{P} = \frac{\partial P}{\partial S}\dot{S}$$

which implies

$$\dot{P} = \frac{\partial P}{\partial S}\frac{\partial S}{\partial \alpha}\dot{\alpha}$$

α can be scalar or a vector consisting of a large number of variables, a_1 ... a_n. $\frac{\partial S}{\partial \alpha}$ is then expressed as

$$\frac{\partial S}{\partial \alpha} = \frac{\partial S}{\partial \alpha_1}\dot{\alpha_1} + ... + \frac{\partial S}{\partial \alpha_n}\dot{\alpha_n}$$

When moving from step to step next step in the calculations, as in (6), one value of a partial derivative for each initial variable are calculated and pushed forward. This means that the computing time for derivatives scales linear with respect to the number of initial variables.

3.1.2 Reverse Mode

Using the reverse mode of AD, the assembly of derivatives is done in reverse order. When a final output function value is calculated, information of the order of calculations is stored. Then in a reverse sweep, the partial derivatives is accumulated all the way down to the input variables.

$$\bar{\alpha} \leftarrow \bar{S} \leftarrow \bar{P} \tag{7}$$

The notation in reverse mode is the following; $\bar{\alpha}$, $\bar{S}$ and $\bar{P}$ represents the differential of P with respect to α, S and P respectively:

$$\bar{\alpha} = \frac{\partial P}{\partial \alpha} = \frac{\partial P}{\partial S}\frac{\partial S}{\partial \alpha} = \frac{\partial S}{\partial \alpha}\bar{S}$$

and

$$\bar{S} = \frac{\partial P}{\partial S}\bar{P}$$

which gives

$$\bar{\alpha} = \frac{\partial S}{\partial \alpha}\frac{\partial P}{\partial S}\bar{P}$$

One can note that $\bar{\alpha}$ in reverse mode AD is the same as $\dot{P}$ in forward mode. $\dot{\alpha}$ in forward mode is equivalent to $\bar{P}$. Both of these are equal to 1. This is the *seeding* of the code done in order to start accumulating all derivatives.

Since reverse mode AD means that calculations are done in more complex ways than usual, calculating function values therefore takes somewhat longer time than

if evaluated with traditional techniques. On the positive side, derivatives to this function, with respect to all initial variables can be obtained simultaneously in a second sweep. In theory, this second sweep takes as long time to do as the first sweep but due to technical challenges in implementation, the second sweeps usually requires at least a triple amount of time.

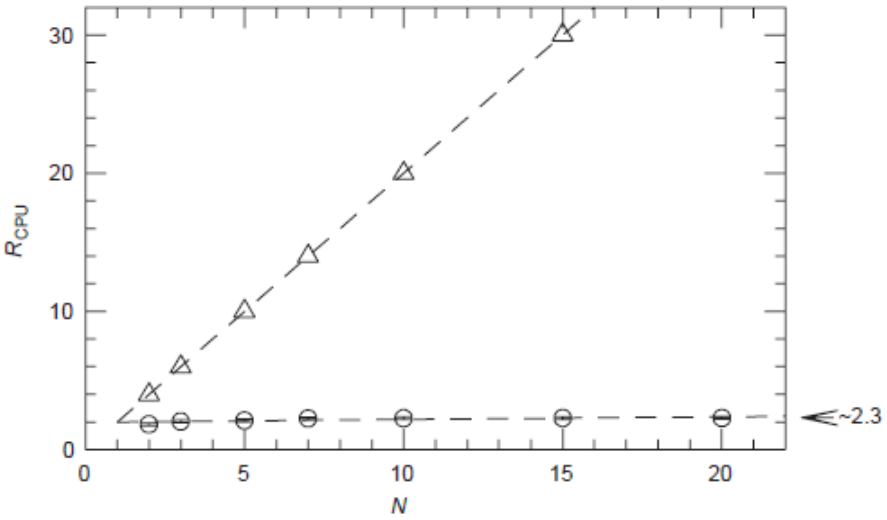


Figure 2: Processing time ratios for the calculation of Delta and Vega with reverse mode AD (circles) as a function of the number of underlying assets. The same calculations done with finite difference method is shown with triangles. Figure from Capriotti(2011), [4].

Luca Capriotti shows that with efficient implementation, it is possible to obtain a factor as low as 2.3 for calculating all derivatives in the second sweep relative to the cost of evaluating the function itself in the first sweep [4]. This is shown in figure 2. R_{CPU} symbolizes the processing time needed to calculate delta and vega of a basket call option divided by the time to calculate the option value.

$$R_{CPU} = \frac{Cost[Value + Risk]}{Cost[Value]}$$

One can see that the time needed for evaluation of all derivatives remains constant (relative to the calculation time for the option value) as the number of underlying assets increases. This astonishing property of reverse mode AD makes it very powerful. With traditional finite difference methods the computational time increases linearly and the calculations eventually becomes impossible to perform within a reasonable timely manner.

Since the computing time for derivatives in forward mode AD scales linear in respect to the number of derivatives wanted, the main part of the project has been devoted to reverse mode AD.

3.2 AD-tools

In order to make implementation of Automatic Differentiation easy, two different kinds of tools has been developed; Source Code Transformation and Operator Overloading.

Source Code Transformation is a type of compiler which takes ordinary numeric functions and generates new, differentiated functions which then gets executed instead. Havard Berland [1] lists some pros and cons with Source Code Transformation:

+ Possible in all computer languages
+ Can be applied to old legacy Fortran/C code and existing code in general
+ Allows easier compile time optimization
− Source code swell
− More difficult to code the AD tool

Operator Overloading on the other hand, is based on the fact that every evaluation or computation can be broken down to a combination of basic elementary operations. These elementary operations such as plus, minus and times are then redefined in order to handle derivative values parallel to the function evaluation. This project considers forward and reverse mode AD through operator overloading in Matlab. Pros and cons with operator overloading are among other things [1]:

+ Small changes in your original code
+ Flexible when you change your code or tool
+ Easy to code the AD tool (at least for forward mode)
− Only possible in selected languages
− Current compilers lag behind, code runs slower

AD is used by scientist in many areas. For more information on how AD is used in today's research and the different existing AD-tools, the reader is referred to *www.autodiff.org*.

4 Barrier Handling in Monte Carlo Simulation

Barriers and discontinuities needs to be smoothed out in order to evaluate derivatives. This can be done in many ways. In this project, some function smoothing techniques are tested besides the more rigorous Vibrato Monte Carlo method.

4.1 Vibrato

> "The Oxford English Dictionary describes "vibrato" as "a rapid slight variation in pitch in singing or playing some musical instruments". The analogy to Monte Carlo methods is the following: whereas a path simulation in a standard Monte Carlo calculation produces a precise value for the output values from the underlying stochastic process, in the vibrato Monte Carlo approach the output values have a narrow probability distribution."
>
> M.Giles (2009) [6]

In 2007, Michael Giles [5] presents a new method for sensitivity evaluations for discontinuous payoffs. It combines a pathwise derivative method with the Maximum Likelihood Ratio method. Instead of differentiating the trajectory itself, the method is based on a Monte Carlo simulation of the differentiated probability distribution on the other side of the barrier.

$$p_S(\hat{S}_N) = \frac{1}{\sqrt{2\pi}\sigma_W} \exp\Big(- \frac{(\hat{S}_N - \mu_W)^2}{2\sigma_W^2}\Big)$$

where

$$\begin{aligned} \mu_W &= \hat{S}_{N-1} + a(\hat{S}_{N-1}, T-h)h, \\ \sigma_W &= b(\hat{S}_{N-1}, T-h), \ \ and \\ \hat{S}_N &= \mu_W + \sigma_W Z \end{aligned}$$

Then, a differentiation of the mean is replaced with the mean of the differentiated distribution. The option value is obtained by taking the expectation first over the simulation over the barrier, then over all Monte Carlo simulations:

$$\hat{V} = \mathbb{E}\left[\mathbb{E}\left[f(\hat{S}_N)|W\right]\right] = \int \left(\int f(\hat{S}_N) p_S(\hat{S}_N)|W \right) p_W(W) dW$$

The corresponding derivatives with respect to some input variable θ can be obtained with the same concept, but now with an inner expectation of the differentiated distribution over the barrier:

$$\frac{\partial \hat{V}}{\partial \theta} = \mathbb{E}\left(\frac{\partial}{\partial \theta} \mathbb{E}\left(f(\hat{S}_N)|W \right) \right) = \mathbb{E}_W \left(\mathbb{E}_Z \Big(f(\hat{S}_N) \frac{\partial log(p_S)}{\partial \theta} \Big) \right)$$

where

$$\frac{\partial (log(p_S))}{\partial \theta} = \frac{\partial (log(p_S))}{\partial \mu_W} \frac{\partial \mu_W}{\partial \theta} + \frac{\partial (log(p_S))}{\partial \sigma_W} \frac{\partial \sigma_W}{\partial \theta} \tag{8}$$

and $\hat{S}_N$ is an instance of the underlying process, obtained via Euler discretization as in equation 2.

The Monte Carlo estimators for $\hat{V}$ and $\frac{\partial \hat{V}}{\partial \theta}$ are expressed as

$$M^{-1}\sum_{m=1}^{M}\hat{Y}^{(m)}, \quad M^{-1}\sum_{m=1}^{M}\hat{Y}_{\theta}^{(m)}$$

where $\hat{Y}^{(m)}$ is an unbiased estimator for $\mathbb{E}_Z\left[f(\hat{S}_N)|W\right]$ and $\hat{Y}_{\theta}^{(m)}$ is an unbiased estimator for $\mathbb{E}_Z\left[f(\hat{S}_N)\frac{\partial log(p_S)}{\partial \theta}|W\right]$

4.1.1 Variance Reduction

In his papers from 2007 and 2009 [5] [6], M. Giles also addresses ways to reduce the variance of the derived estimators $\hat{Y}^{(m)}$ and $\hat{Y}_{\theta}^{(m)}$. If the variance in the estimators can be reduced, fewer Monte Carlo simulations are required to obtain evaluations with a certain accuracy. One way to do this is to use antithetic variates

$$\mathbb{E}_Z = \left[f(\hat{S}_N)|W\right] = \mathbb{E}_W\left[\frac{1}{2}\left(f(\mu_W + \sigma_W Z) + f(\mu_W - \sigma_W Z)\right)\right]$$

which makes the estimator take the form

$$\hat{Y} = M^{-1}\sum_{m=1}^{M}\frac{1}{2}\left(f(\mu_W + \sigma_W Z) + f(\mu_W - \sigma_W Z)\right)$$

The variance is reduced since $(f(\mu_W + \sigma_W Z))$ and $f(\mu_W - \sigma_W Z)$ are at best, perfectly negative correlated. For a monotone function f, with random variables U and U' with same distribution, one has that

$$\big(f(U) - f(U')\big)\big(f(-U) - f(-U')\big) \leq 0$$

Taking the Expectation of the above entity gives

$$\mathbb{E}[f(U)f(-U)] - \mathbb{E}[f(U)]\,\mathbb{E}[f(-U')] - \mathbb{E}[f(U')]\,\mathbb{E}[f(-U)] + \mathbb{E}[f(U')f(-U')] \leq 0$$

Using that U and U' are identically distributed one obtains the covariance

$$Cov(f(U), f(-U)) = \mathbb{E}[f(U)f(-U)] - \mathbb{E}[f(U)]\,\mathbb{E}[f(-U)] \leq 0$$

The payoff of an up and out barrier option as in figure 1 is monotone up to the barrier H. After that the function is identically zero and it then gives no contribution to the covariance.

4.2 Vibrato and Path Dependent Discontinuous Payoffs

The idea of Vibrato can be extended to handle path dependent payoffs as well. Burgos and Giles [3] presents a way to handle payoffs depending on values at discrete intermediate times. The simulation proceeds as usual until one time step before the discontinuity. A time step twice as long is then applied over the barrier and the simulation continues as usual on the other side. A brownian bridge, as defined in expression 4 is set up over the discontinuity to obtain expressions and values for $\frac{\partial \mu}{\partial \theta}$ and $\frac{\partial \sigma}{\partial \sigma}$ as in expression 8. The length of the Brownian bridge does not have to be twice as long as the other time steps. The choice of time step over the barrier is a tradeoff between bias and variance which *Burgos* and *Giles* addresses in their paper [3].

5 Implementation

This section describes the implementation of AD and Vibrato Monte Carlo.

5.1 Forward Mode AD

For implementation of forward mode AD, the class "valder", written by Richard D. Neidinger [9] was used. The concept is simple, valder objects have a value and a derivative, hence the name valder. Every binary operation is then redefined to calculate the derivatives parallel to the function value.

$$\mathbf{a = valder(value,\ derivative)} \Rightarrow \mathbf{a = [value,\ derivative]}$$

As an example, multiplication between a and b then becomes:

$$\mathbf{f = a \cdot b = [a.value \cdot b.value,\ a.value \cdot b.derivative + a.derivative \cdot b.value]}$$

Since the computing time for derivatives in forward mode AD scales linear in respect to the number of derivatives wanted, the main part of the project has been devoted to reverse mode AD.

5.2 Reverse Mode AD

"...not easy to implement but hard to debug"

J. Utke 2013

Operator Overloading for Reverse mode AD can be done in many ways. Scientist seems to agree that operator overloading in reverse mode AD is very efficient and powerful, but not at all intuitive. To cite Michael Giles: "*Operator overloading for reverse mode computation is very much harder to understand (than forward)*".

Despite the efficiency and large computational gains to be made using this concept in calculations, there exist no general or obvious implementation scheme for reverse mode AD.

The following subsection describes how implementation of reverse mode AD through operator overloading was done in this project. The conceptual implementation idea is based on a short code example presented by Laksh Gupta. [8]

5.2.1 ADRev

For impementation of reverse mode AD in this projet, a Matlab class called *ADRev* was written. Objects of type ADRev have 4 attributes which are value, derivative, derivative function and parents (a list of other ADRev-objects). When one creates an ADRev object the only input parameter needed is the value. The other attributes are filled when the different ADRev-objects are linked together via some basic binary operation such as addition or multiplication. The ADRev objects can be seen as nodes which together forms a tree graph as in figure 3.

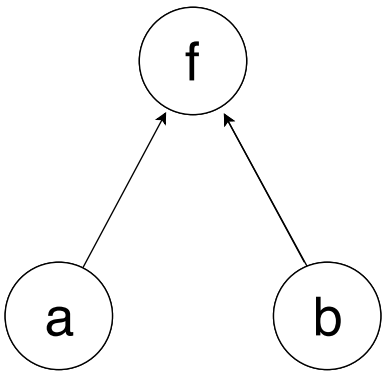


Figure 3: Two nodes are linked together to form a new value

Nodes a and b are parents to the child node, f. A link between two nodes carries value of the parent node in one way and the derivative in the other way, as seen in figure 4.

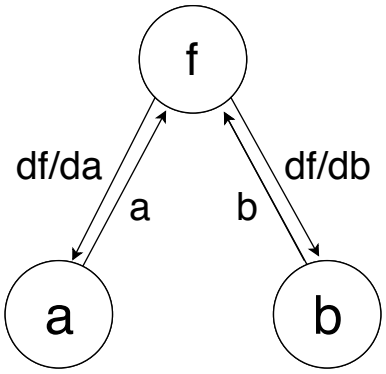


Figure 4: Values are passed on upwards in the calculation graph, derivatives are passed in the opposite direction according to the accumulation rules in expression 7

Consider multiplication between the numbers 3 and 4 as an example. First, one creates two ADRev-objects with the values 3 and 4 respectively.

$\mathbf{a = ADRev(3)}$,

$\mathbf{b = ADRev(4)}$

Then, form the product of a and b:

$\mathbf{f = a \cdot b}$.

Now, f is an ADRev object with value 12 as shown in figure 5.

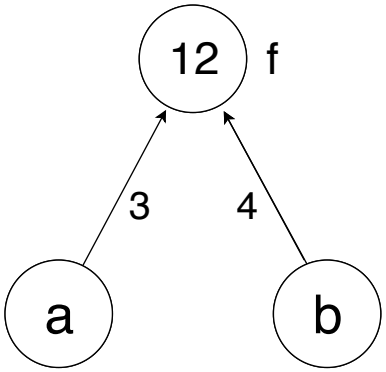


Figure 5: Calculation graph of a short example. The nodes a and b are linked together via multiplication and forms a new node, *f*, with value 12.

f also knows that it originates from a multiplication from two other ADRev objects. Therefore, f has been given a derivative-operation function of type $adrmultD$. This

function is based on the ordinary multiplication rule of differentiation, namely if

$$f(a, b) = a \cdot b$$

then it holds that

$$f'_a = b \quad and \quad f'_b = a$$

The chain rule is then applied to the node *f*. The nodes derivative-operation functions are called which pushes the corresponding derivatives recursively, down to the initial variables, as in figure 6.

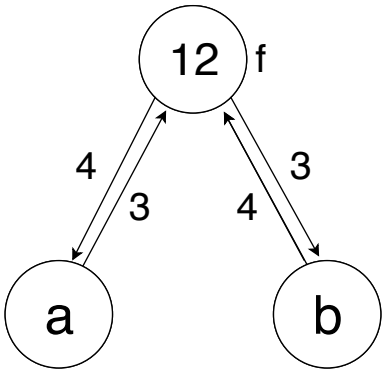


Figure 6: Calculation graph of a short multiplication example

In our case this is finished after just one iteration. Now, *a.derivative* consists of the number 4 and *b.derivative* is 3. The program works in the same principle regardless of the length and complexity of the target function. This is the nice part about operator overloading - one can write functions as usual and the structure of the program handle all derivatives for you.

In the AD-library implemented in this thesis, the following functions have been overloaded:

- addition, +
- subtraction, -
- multiplication, *
- division, /
- natural exponent, exp()
- natural logarithm, ln()
- power
- sin, sin()

- cosine, cos()
- vanilla call option payoff
- linear smoothed payoff
- sigmoid smoothed payoff

As mentioned in section 3.1.2, calculation of derivatives in reverse mode AD requires two sweeps. Most common is that all calculations, values as well as derivatives are calculated in the first sweep. Besides this, information about how the nodes are linked to each other is stored. The second sweep then only consists of accumulating all the derivatives.

In this project, implementation of reverse mode AD was done in a slightly different way. In the first sweep the calculation tree graph is constructed and function values are calculated. The derivatives on the other hand are calculated in the reverse sweep and the main reason for this is that it was easier to implement it this way. This means that if only a (final) function value is sought e.g. a payoff, then no time is wasted on calculating derivatives. A disadvantage of this method is that the whole calculation graph must be stored and used in the second sweep. This requires more memory space and makes the calculations slower.

When applying reverse AD to options and barrier options, GBM was used as the underlying process and simulated according to equation 1.

5.3 Barrier Smoothing

Calculating derivatives using only AD is possible but, as mentioned before, all derivative functions have to defined in advance. This means that the problem with discontinuities still has to be dealt with. In this project, three types of barrier smoothing have been considered. Two of them are different ways to approximate the Heaviside function

$$\theta(x) = \begin{cases} 1 & \text{if } \ x \geq 0 \\ 0 & otherwise \end{cases}$$

with a smoothed, differentiable function, the third is the Vibrato technique introduced in section 4.1.

5.3.1 Linear Smoothing

A first approach is a linear smoothing. Here the discontinuity has been replaced with a straight line on the form $y = kx + m$. To make the payoff the equation for the line are expressed as

$$y = -\left(\frac{1}{\delta}\right)x + \left(2 + \frac{H}{\delta}\right) \tag{9}$$

where δ is the smoothing parameter. The smaller δ, the smaller smoothing interval and steeper barrier. With this smoothing, a payoff for an up and out barrier option takes the form as shown to the left in figure 7.

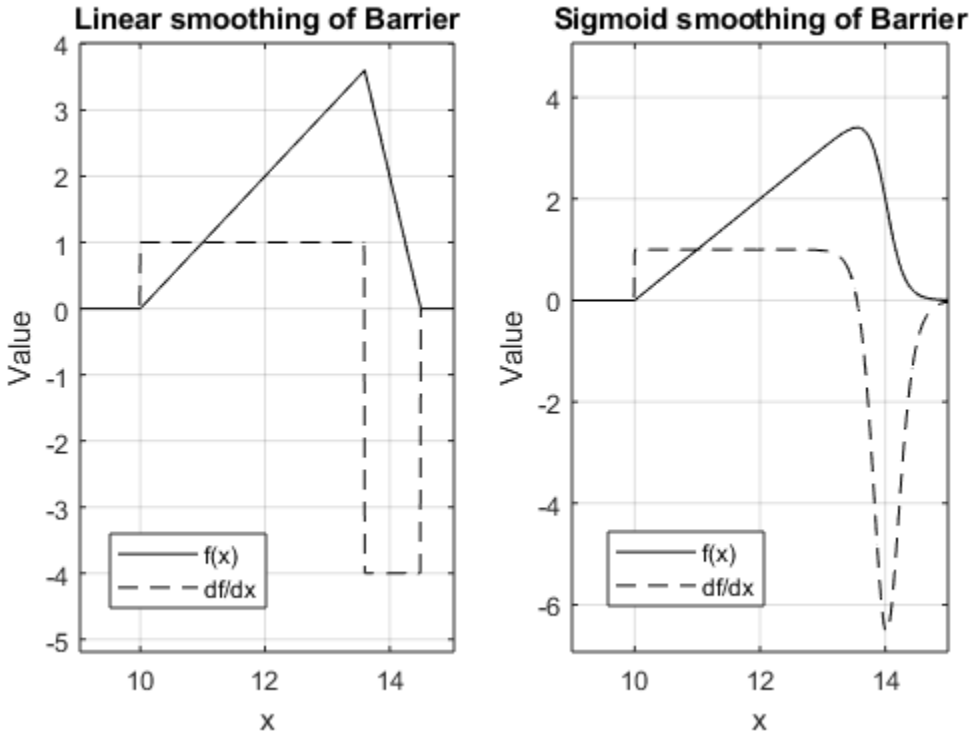


Figure 7: Linear and Sigmoid smoothing of an up and out barrier payoff as displayed in figure 1. The derivative of each smoothed function is shown with dashed lines.

5.3.2 Sigmoid Smoothing

The logistic function

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

is used in a range of fields of science. It is often involved when describing population distributions in biology or physics. The Fermi-Dirac population distribution of electrons in an atom is described with a version of the logistic function. In neural networks, it is often applied an activation function for categorizations and it is also used in logistic regression problems.

The payoff of an up and out barrier option is approximated with different sigmoid

shaped curves on the form

$$f(x) = \frac{1}{1 + e^{\frac{(x-H)}{\delta}}} \tag{10}$$

A sigmoid-smoothed barrier is shown to the right in figure 7.

5.4 Vibrato

For implementation of the Vibrato Monte Carlo method, a scheme presented by G. Pagès et al [10] was used. This is a conceptual algorithm for calculation of value and derivatives of a derivative following a Black-Scholes Model. It uses Geometric Brownian Motion as an underlying process and an Euler discretization for simulation:

1. Generate M simulation paths with time step h = $\frac{T}{h}$ of the underlying asset X and its tangent process Y = $\frac{\partial X}{\partial \theta}$ with respect to a parameter θ for k= 0,...n-2.

$$\begin{cases} \bar{X}^n_{k+1} = \bar{X}^n_k + rh\bar{X}^n_k + \bar{X}^n_k \sigma\sqrt{h} Z_{k+1} \ \ where \ \ \bar{X}^n_0 = X_0 \ \ and \ \ \bar{Y}^n_0 = \frac{\partial X_0}{\partial \theta} \\ \bar{Y}^n_{k+1} = \bar{Y}^n_k + rh\bar{Y}^n_k + \frac{\partial}{\partial \theta}(rh)\bar{X}^n_k + (\bar{Y}^n_k \sigma\sqrt{h} + \frac{\partial}{\partial \theta}(\sigma\sqrt{h})\bar{X}^n_k) Z_{k+1} \end{cases}$$

2. For each simulation path

   (a) Generate M_Z last time steps ($\bar{X}_T = \bar{X}^n_n$)

   $$\bar{X}^n_n = \bar{X}^n_{n-1}(1 + rh + \sigma\sqrt{h} Z_n)$$

   (b) Compute the first derivative with respect to θ by Vibrato using the antithetic technique

   $$\frac{\partial V_T}{\partial \theta} = \frac{\partial \mu_{n-1}}{\partial \theta} \frac{1}{2} (V_{T+} - V_{T-}) \frac{Z_n}{\bar{X}^n_{n-1}\sigma\sqrt{h}} + \frac{\partial \sigma_{n-1}}{\partial \theta} \frac{1}{2} (V_{T+} - 2V_{T\cdot} + V_{T-}) \frac{Z_n^2 - 1}{\bar{X}^n_{n-1}\sigma\sqrt{h}}$$

   with $V_{T\pm\cdot} = (\bar{X}_{T\pm\cdot} - K)^+$,

   $$\begin{cases} \bar{X}_{T_\pm} = \bar{X}^n_k + rh\bar{X}^n_k \pm \bar{X}^n_k \sigma\sqrt{h} Z_{k+1} \\ \bar{X}_{T\cdot} = \bar{X}^n_k + rh\bar{X}^n_k \end{cases}$$

   and

   $$\frac{\partial \mu_{n-1}}{\partial \theta} = \bar{Y}^n_{n-1}(1 + rh) + \bar{X}^n_{n-1} \frac{\partial}{\partial \theta}(rh)$$
   $$\frac{\partial \sigma_{n-1}}{\partial \theta} = \bar{Y}^n_{n-1}\sigma\sqrt{h} + \bar{X}^n_{n-1} \frac{\partial}{\partial \theta}(\sigma\sqrt{h})$$

   If θ = T or θ = r, one have to add $\frac{\partial}{\partial \theta}(\exp -rT)V_T$ to result above.

(c) Apply an AD method to obtain second order derivatives with respect to some θ at some $\theta*$ (Optional)

(d) Compute the mean per path, i.e. over M_Z.

3. Compute the mean of the resulting vector (over the M simulation paths) and discount it.

Other schemes than Euler discreetization can of course be used instead. The conceptual algorithm can also be adapted to other models where e.g. non constant interest rate or a time and asset dependent volatility, $\sigma = \sigma(t, X_t, \theta)$.

6 Tests and Results

In this section, different tests are carried out to evaluate the accuracy and efficiency of implemented methods.

6.1 Plain Vanilla Call Option

To begin, a simple case is examined to test the accuracy of the implemented reverse mode AD calculation method. Figure 8 displays price and Greeks for a European Call option. It can be seen that the reverse mode AD method calculates function values and derivatives (shown with various markers) with good precision compared to the analytic expressions for price and Greeks (lines).

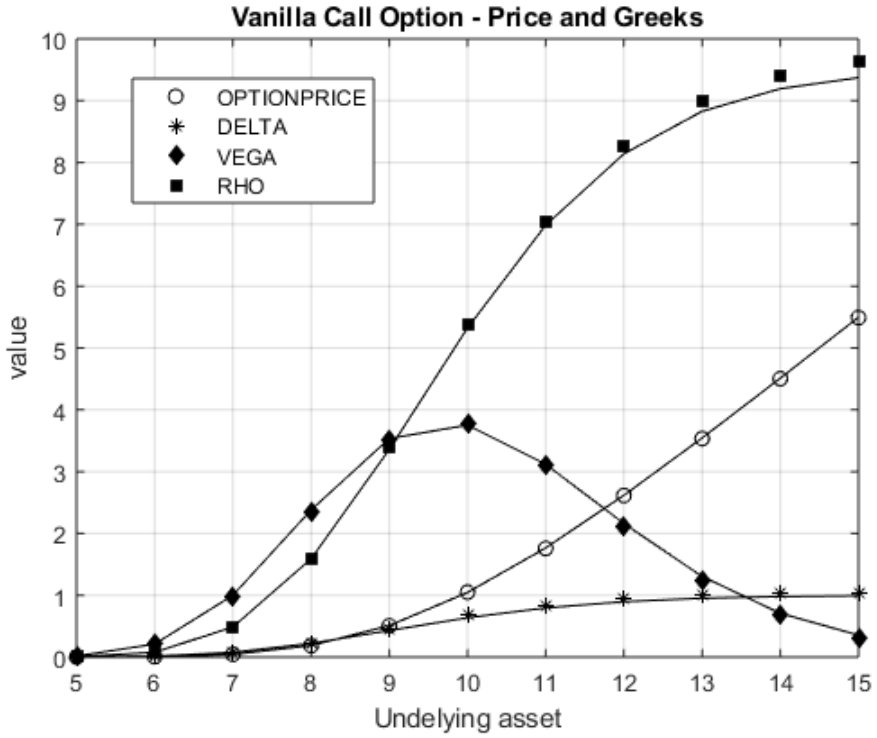


Figure 8: Price and Greeks for a European call option. The lines describes the theoretical analytic solution, the dots are the result of Monte Carlo simulation with reverse mode AD. 10000 Monte Carlo iterations are used in the simulation. The option underlies a GBM with interest rate 0.05, volatility 0.2 and strike price at 10.

6.2 Up and Out-Barrier Option

In this section, the different smoothing techniques presented above have been adapted to calculate price and Greeks for an option with an European Up and Out-barrier

at maturity. For all simulation in this section the different simulation parameters used are:

- Number of Monte Carlo simulations = 10000
- Path simulation steps = 12
- r = 0.05
- $\sigma = 0.2$
- T = 1 (one year)
- K = 10
- H = 14

6.2.1 Linear Smoothing

The following figures displays price and Greeks using linear smoothing of barrier, as in equation 9.

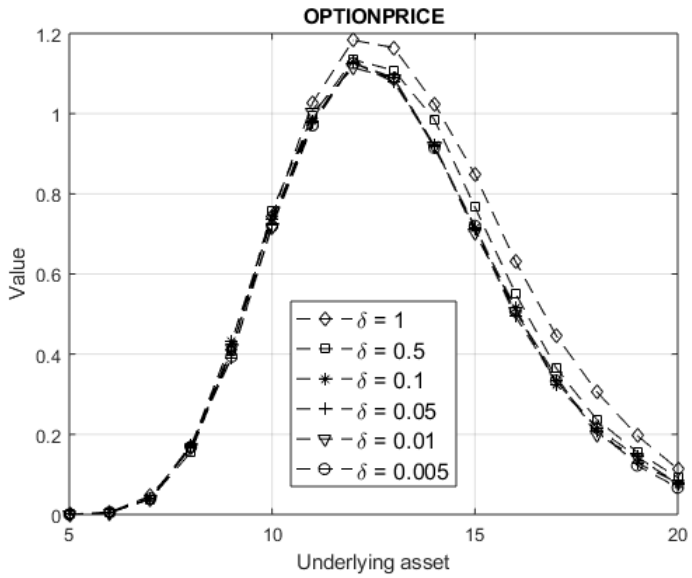
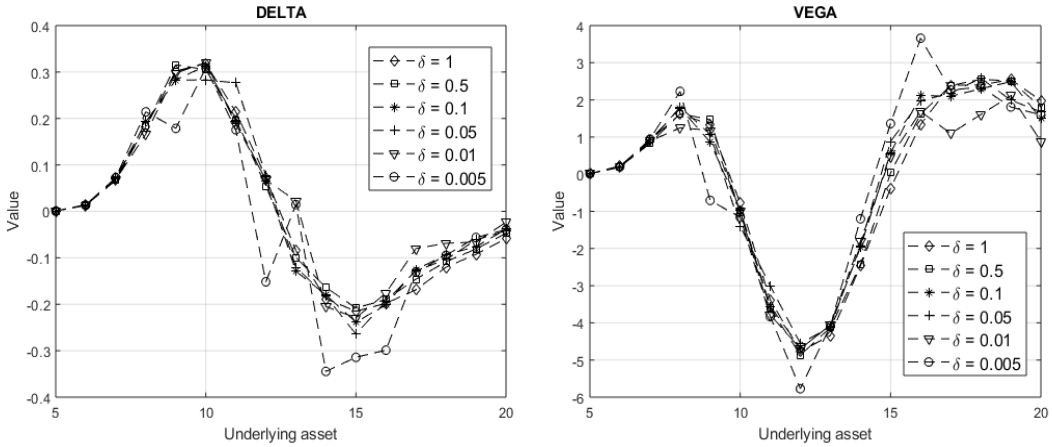
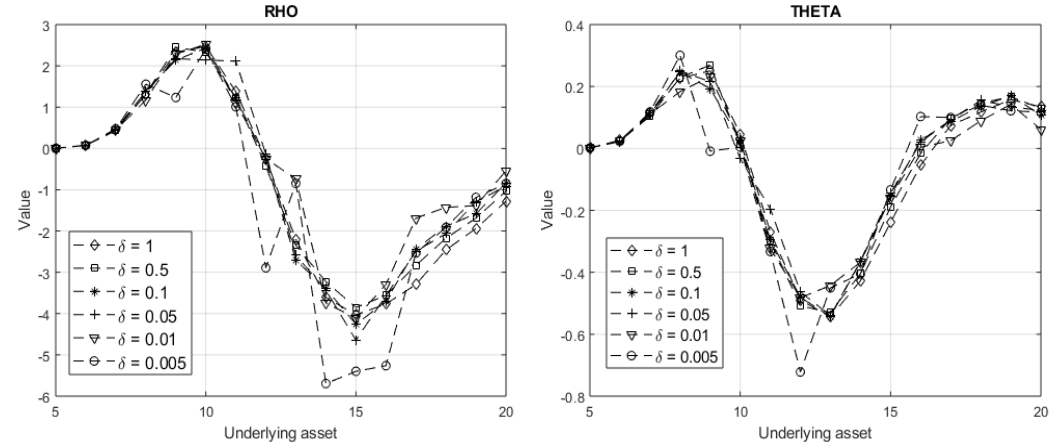


Figure 9: Price of Barrier Option for different smoothing parameters.



(a) Delta for different smoothing parameters **(b)** Vega for different smoothing parameters

Figure 10: Sensitivities with linear smoothing



(a) Rho for different smoothing parameters **(b)** Theta for different smoothing parameters

Figure 11: Sensitivities with linear smoothing

The figures 10 and 11 shows that with the right smoothing parameter one can obtain good values for the Greeks. More smoothing gives less variance in the result but large bias because of the smoothing parameter. Less smoothing means steeper and more accurate barrier but it causes lager variance makes the curves fuzzy.

6.2.2 Sigmoid Smoothing

The following figures displays price and Greeks using sigmoid barrier smoothing, as in equation 10.

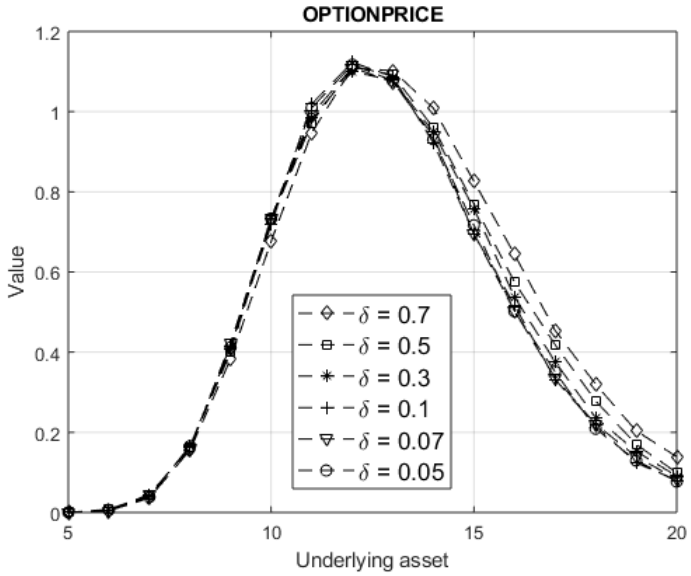
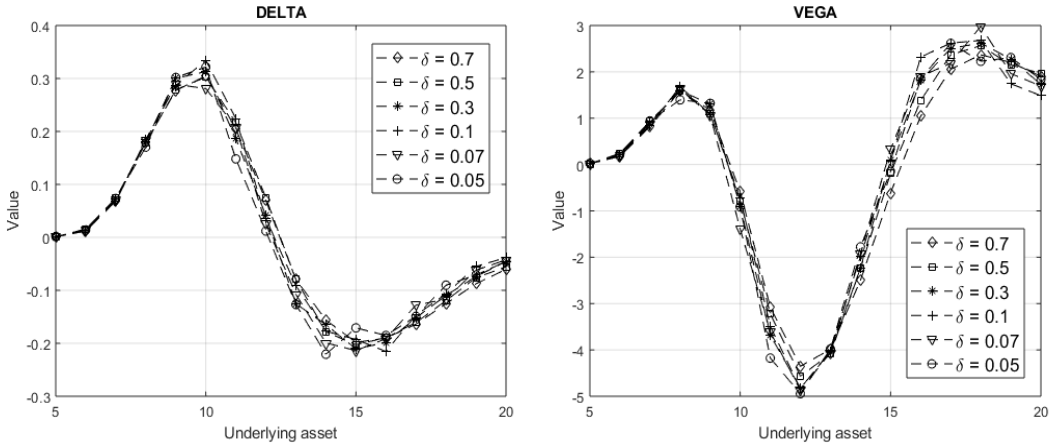
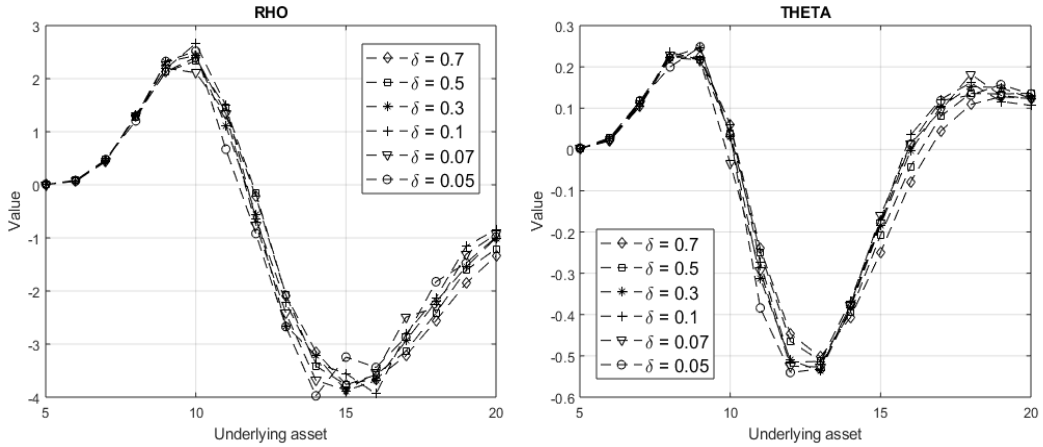


Figure 12: Price of Barrier Option for different sigmoid smoothing parameters.



(a) Delta for different smoothing parameters **(b)** Vega for different smoothing parameters

Figure 13: Sensitivities with sigmoid smoothing



(a) Rho for different smoothing parameters **(b)** Theta for different smoothing parameters

Figure 14: Sensitivities with sigmoid smoothing

In figures 12, 13 and 14 one can see that sigmoid smoothing of barriers gives results with lower variance than linear smoothing. As can be seen in the figures above, the overall choice of smoothing parameter plays a smaller role here than in the case of linear smoothing. The figures shows curves for smoothing parameters in range 0.05 to 0.7. With δ equal to 1 or larger, the smoothing was to large in order to get meaningful results and plots. It should also be said that a best choice of smoothing parameter is different for different contracts, depending on parameters such as interest rate and time until maturity. When little time is left until maturity or when close to a barrier checking time, one needs a smaller smoothing parameter to obtain reliable results.

With the right choice of smoothing parameter, both linear and sigmoid smoothing of barriers gives accurate result when calculating derivatives with reverse mode AD. Figure 15 displays delta calculated with linear and sigmoid smoothed European barrier together with result obtained from a pde-solver. Both methods are therefore reliable.

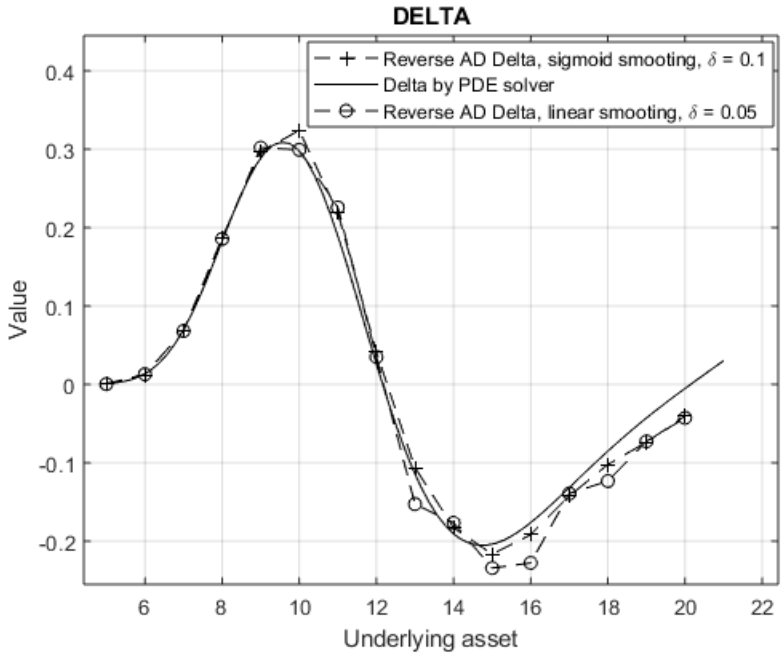


Figure 15: Linear and sigmoid smoothed European Up and Out barrier delta besides delta obatined with a pde-solver. Again, the strike price is 10 and the barrier is at 14.

6.3 Vibrato

In this subsection, the Vibrato Monte Carlo method is tested for European and Bermudan barriers.

6.3.1 Vibrato with European Barrier

The Vibrato Monte Carlo algorithm discussed in section 5.4 was run with 100000 Monte Carlo Simulations and 10 simulations over the barrier. The payoff function used in this case is the same one as displayed in figure 1.

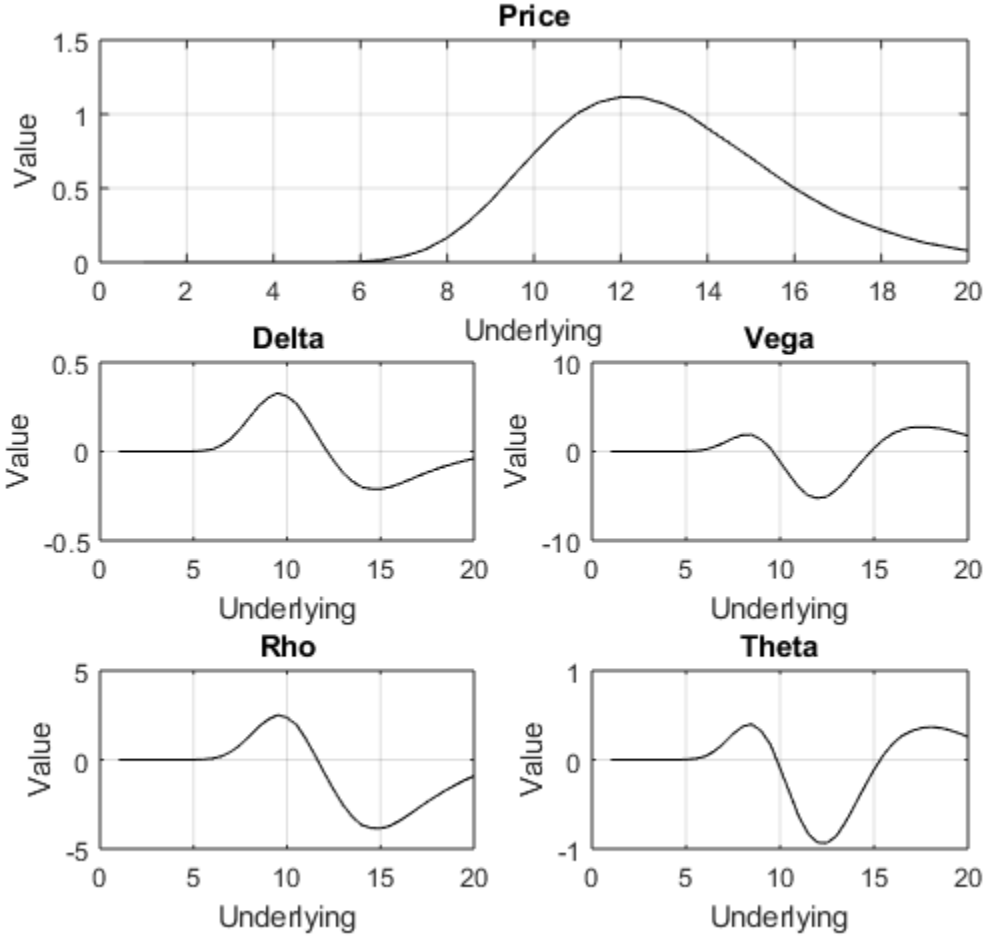


Figure 16: Price and Greeks for an option with a European Up and Out Barrier, calculated with Vibrato Monte Carlo.

Figure 16 displays price and Greeks for a European Up and Out barrier option, calculated with the Vibrato Monte Carlo method. The smooth curves validate that Vibrato is an effective and low-variance method for barrier smoothing.

6.3.2 Vibrato with Bermudan Barrier

To evaluate price and Greeks for options with barrier checked at multiple times, the vibrato scheme discussed in section 4.2 was tested with 100000 Monte Carlo simulations, and 10 simulations over each barrier checking time. These checking times was set to be at 0.7·T i.e. after 8.4 months after starting day of the contract,

and at maturity. 12 steps for simulation of the trajectory was used.

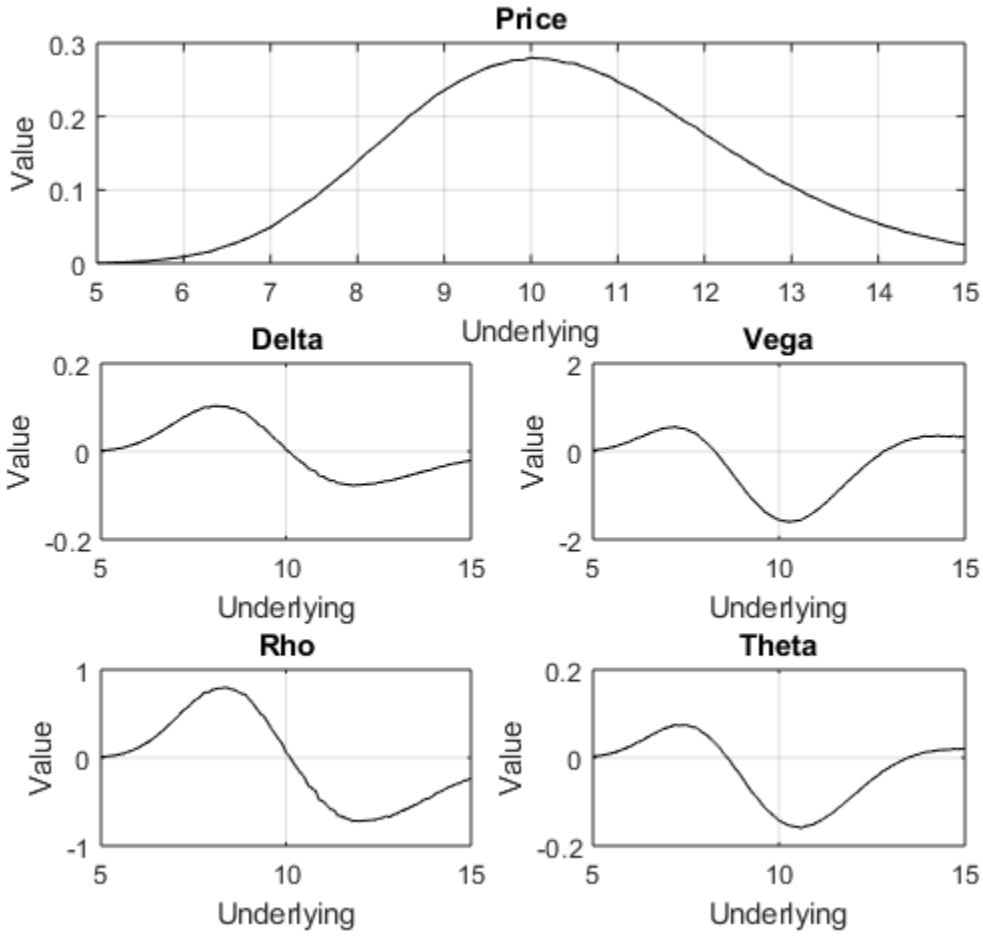


Figure 17: Price and Greeks for an option with a Bermudan Up and Out Barrier, calculated with Vibrato Monte Carlo.

Price and sensitivities for an Up and Out call option with Bermudan barrier can be seen in figure 17. The vibrato concept is applied over each barrier resulting in the smooth curves, despite two discontinuous checking times.

6.4 Vibrato AD

According to step 2c in the Vibrato Monte Carlo algorithm in section 5.4, a combination of AD and the Vibrato Monte Carlo algorithm is possible. In theory it is fully possible to run the simulations as in section 6.3.2 and obtain first order

Greeks with Vibrato precision and second order Greeks e.g. Γ with AD. Due to the practical limitations of this project, results of calculations with the Vibrato AD techniques are not presented here. As of today, the ADRev class cannot handle vectors of ADRev-objects. Nevertheless there exists no theoretical restraints for a future implementation of Vibrato AD based on the work presented in this paper.

To obtain a second order derivative with the Vibrato AD setting, simply apply the chainRule-algorithm on a first order derivative. For the case of Γ, apply the chain rule on the vibrato-calculated Delta and fetch the value of Γ in the derivative-attribute of the underlying variable.

1. Define a variable of interest as an ADRev object: $S = \mathit{Underlying}$
2. Apply the Vibrato algorithm to compute *Delta* (of type ADRev).
3. Apply chainRule function on *Delta*.
4. Now, Γ is stored in *Underlying.derivative*.

6.5 Convergence

An important property of numerical methods and simulations is the rate of convergence. With a fast convergence, fewer simulations is needed which means faster computing time. Figure 18 displays the convergence of Delta, calculated with finite difference and reverse AD, respectively. The displayed values is for an Up and Out option, with European barrier. Starting value of the asset was set to 9. Apart from that, the same simulation parameters as listed in section 6.2 are being used. It can be seen that the finite difference method converges very slowly while reverse AD with sigmoid smoothing more or less stabilizes at 0.3 after around 10000 simulations.

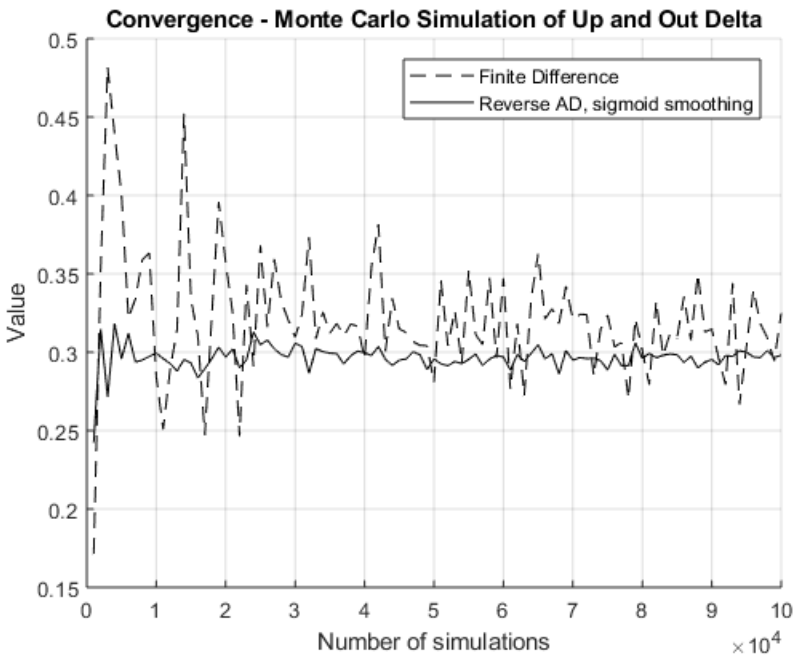


Figure 18: Convergence of Delta calculated with finite difference and reverse AD methods. Simulations run for an European Up and Out Barrier at 14. Strike price at 10 and a starting value for the underlying GBM at 9.

In figure 19, it can be seen that the Vibrato method also has a high degree of convergence compared to a finite difference method. This means that even in the cases where AD not is used, Vibrato should be used in Monte Carlo simulations in order to get accurate results fast. Remember, the only requirement needed for using Vibrato is that the distribution for the underlying movement is known. The choice of smoothing parameter in reverse mode AD affects the numerical result slightly, see figure 13a. Hence, it is not strange that there exist a stationary difference between the two methods in figure 19. This difference is however just of size of around 0.01 which gives the small relative difference of 3% between the two methods.

Both Vibrato and AD evaluates derivatives with just one round of Monte Carlo simulations (Compared to finite difference which requires at least two). If just a few number of derivatives is sought, then both Vibrato and AD with satisfying smoothing are giving fast and accurate results. Depending on how AD is implemented, it is possible that Vibrato is a faster method in this case. It is when then number of derivatives sought that reverse mode AD becomes the clear and obvious choice.

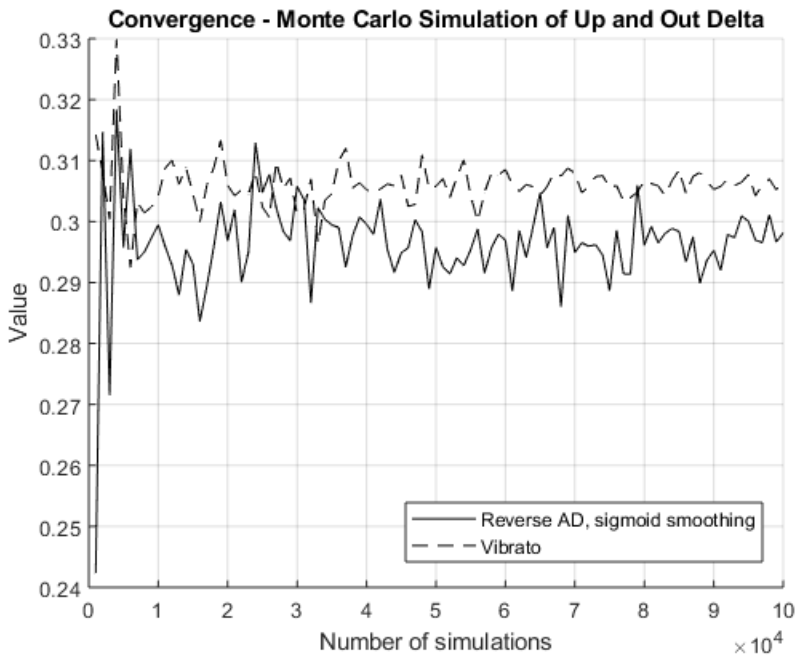


Figure 19: Convergence of Delta calculated with reverse AD and Vibrato methods. Simulations run for an European Up and Out Barrier at 14. Strike price at 10 and a starting value for the underlying GBM at 9.

7 Conclusion

In this project, one way to implement reverse mode AD via operator overloading is presented. The concept is tested on simple financial contracts with and without discontinuous payoffs. When calculating prices and derivatives of complex contracts, one traditionally needs to do a number of simulations to eventually circle down to accurate values. Since no numerical approximations needs to be done in order to calculate derivatives with AD (except for smoothing of barriers), Greeks calculated with AD therefore have a high degree of accuracy. AD methods also converge faster compared to existing finite difference methods used today. The benefits of AD techniques will be even more notable when applied to contracts which depends on a large number of variables. When using AD, functions and differentials need to be defined in advance. Discontinuities can be smoothed out with linear or sigmod shaped curves. This enables calculations on barrier options with high precision. Which smoothing parameters to use is a trade-off between bias and variance. The type of smoothing function, contract specifications and simulation parameters also affect the best choice of parameter.

The Vibrato Mote Carlo method also enables precise calculation of prices and derivatives of options with discontinuous payoffs. For contracts containing Bermudan barriers, Brownian bridges can be used to obtain fist order Greeks. With antithetic techniques variances can be reduced even further.

Apart from the numerical results and conclusions, the main result of this thesis is that it is fully possible to adapt the concept of reverse mode AD into Monte Carlo simulations for pricing of options with discontinuous payoffs.

Further work is to modify the implemented code to handle vectors of ADRev-objects. This enables a broad range of algorithms to be evaluated in a reverse AD setting. With vector handling, the existing vibrato schemes can be evaluated with reverse AD variables which then can give values of Γ with the accuracy of vibrato and speed of AD. More research is required to obtain a reverse AD framework to handle options with Bermudan or American barriers and contracts depending on the average of passed values.

References

[1] Håvard Berland. Automatic Differentiation - Presentation slides, page 15, 2006, http://www.robots.ox.ac.uk/ tvg/publications/talks/autodiff.pdf (2018-11-21).

[2] Tomas Björk. *Arbitrage Theory in Continuous Time*. Oxford University Press, 2004.

[3] Sylvestre Burgos and M Giles. The computation of greeks with multilevel monte carlo. *arXiv preprint arXiv:1102.1348*, 2011.

[4] Luca Capriotti. Fast greeks by algorithmic differentiation. *The Journal of Computational Finance*, 14(3), 2011.

[5] Michael B Giles. Monte carlo evaluation of sensitivities in computational finance. Technical report, Oxford University Computing Laboratory, 2007.

[6] Michael B Giles. Vibrato monte carlo sensitivities. In *Monte Carlo and Quasi-Monte Carlo Methods 2008*, pages 369–382. Springer, 2009.

[7] Thomas Guhr. *Econophysics*. Matematisk Fysik, Lund University, 2007.

[8] Laksh Gupta. Automatic differentiation using operator overloading, 2016, http://lakshgupta.github.io/2016/11/06/autodiffbasic/, (2018-10-03).

[9] Richard D Neidinger. Introduction to automatic differentiation and matlab object-oriented programming. *SIAM review*, 52(3):545–563, 2010.

[10] Gilles Pagès, Olivier Pironneau, et al. Vibrato and automatic differentiation for high order derivatives and sensitivities of financial options. *Journal of Computational Finance 22(2, 1-34)*, 2018.

[11] Introduction to AD. http://www.autodiff.org/?module=introduction, (2018-11-07).

Appendix A - Code Example

This is a short, detailed example of how partial derivatives can be calculated with reverse mode AD.

Consider the case

$$f(a,b) = a \cdot b$$

with corresponding partial derivatives with respect to a and b:

$$f'_a = b$$
$$f'_b = a$$

If $a = 3$ and $b = 2$, the partial derivatives becomes

$$f'_a = 2$$
$$f'_b = 3$$

To calculate these derivatives using reverse mode AD, start with defining the initial variables a and b as variables of ADRev-type. When defining ADRev variables the only input argument needed is the value of the variable. This value can be either a scalar or a vector.

```
% Constructor in the ADRev-class:
function obj = ADRev(val)
    obj.value = val;
    obj.derivative = 0;
    obj.derivativeOp = @ adr_constD;
    obj.parents = [];
end
```

Now, define the desired function. In this case the only thing which needs to be typed is $f = a \cdot b$. Since the operator for multiplication is overloaded, whats get executed instead of the usual multiplication, is:

```
% Overloaded Multiplication in ADRev:
function result = mtimes(x,y)
    if ~isa(x,'ADRev')
        x = ADRev(x);
    elseif ~isa(y,'ADRev')
        y = ADRev(y);
    end
    result = ADRev(x.value .* y.value);
    result.derivativeOp = @ adr_mulD;
    result.parents = [x y];
end
```

f is now an object of ADRev-type. f, a and b can be seen as nodes in a calulation graph. The links between the nodes carries information of values and derivatives up and down, respectively.

To start, or *seed* the chain rule algorithm to push derivatives down to the initial variables, the derivative of the to node is set to 1, since $df/df = 1$.

```
>> f =
                value:  6
           derivative:  1
       derivativeOp:  @adr_mulD
              parents:  [1x2 ADRev]
```

Then, call the current node's derivative operation function:

>> node.derivativeOp(node.derivative, node.parents);

In our case the node f's derivative-operation function is derived from the product rule of differentiation, whats gets executed is:

```
% Multiplication derivative-operation function in ADRev:
function modified_parents = adr_mulD(prevDerivative, adNodes)
    adNodes(1).derivative = adNodes(1).derivative ...
        + prevDerivative.*adNodes(2).value;
    adNodes(2).derivative = adNodes(2).derivative ...
        + prevDerivative.*adNodes(1).value;
    modified_parents = adNodes;
end
```

This fills the derivative slots of a and b. One can now obtain the partial derivatives of f with respect to a by typing a.derivative and b.derivative respectively.

```
>> a =
                value:  2
           derivative:  3
       derivativeOp:  @adr_constD
              parents:  []
```

and

```
>> b =
                value:  3
           derivative:  2
       derivativeOp:  @adr_constD
              parents:  []
```

In the general case, the pushing of derivatives from one node to it's parents, is done recursively. This enables calculations of partial derivatives for long and complex functions.