RWTH Aachen University

Computational Differentiation in Finance

Kumari Shikha

July 29, 2018

1 Acknowledgment

This report is part of the Simulation Sciences Seminar . I would like to thank my supervisor Dr. Johannes Lotz for his consistent help throughout the seminar period , from providing papers to explaining concepts, and for his valuable inputs on improving the presentation .

Also thanks to Professor Uwe Neumann for making the course "Algorithmic Differentiation"[5] available online, the course materials of which served as an important source of explanation .

2 Introduction

The aim of the seminar presentation was to pick up a topic and break it down into simple explanations so that someone with no prior knowledge can understand . My main aim was to make sure that all students understand the financial use case , that of pricing a european call option , understand the basics of algorithmic differentiation , advantages and disadvantages of tangent and adjoint modes , get to know about `dco/c++` and get some basic idea of topics like pathwise adjoints and checkpointing utilised in `dco/c++` to decrease memory requirements and appreciate the decrease in time while using adjoint approach rather than central finite differences .

3 Use Case

For this seminar , a simple European call option on an underlying driven by a local volatility process was considered . Basic definitions of associated terms were provided :

- Underlying : An underlying security is a stock, index, bond, interest rate, currency or commodity on which derivative instruments, such as futures and options, are based.

  Examples : In a stock option to buy 100 shares of Nokia at EUR 50 in April 2011, the underlying is a Nokia share. In a futures contract to buy EUR 10 million 10-year German Government bonds, the underlying are the German Government bonds. Other examples are stock market indexes such as the Dow Jones Industrial Average and Nikkei 225, for which the underlying are the common stocks of 30 large U.S. companies and 225 Japanese companies, respectively.

- Call option : An options contract is an agreement between two parties to facilitate a potential transaction on the underlying security at a preset price, referred to as the strike price, prior to the expiration date (maturity) . The two types of contracts are put and call options. The buyer of a call option has the right but not the obligation to buy the number of shares covered in the contract at the strike price. Put buyers have the right but not the obligation to sell shares at the strike price in the contract. Option sellers, on the other hand, are obligated to transact their side of the trade if a buyer decides to execute a call option to buy the underlying security or execute a put option to sell.

  Types of Option depending of right to exercise (names have nothing to do with geographic location) :

  An **American option** is an option that can be exercised anytime during its life. American options allow option holders to exercise the option at any time prior to and including its maturity date, thus increasing the value of the option to the holder relative to European options, which can only be exercised at maturity. The majority of exchange-traded options are American.

  A **European option** is an option that can only be exercised at the end of its life, at its maturity. European options tend to sometimes trade at a discount to their comparable American option because American options allow investors more opportunities to exercise the contract.

  Example: If an investor purchased a call option on Apple Inc. in March, expiring in December of the current year, the investor would have the right to exercise the call option at any time up until its expiration date. If the call option on Apple Inc. had been a European option, the investor would only be able exercise the option at the expiry date in December. If, hypothetically, that share price became most optimal for exercise in August, the investor would still have to wait until December to exercise the call option, when it could be out-of-the-money and virtually worthless.

- Volatility : Volatility refers to the amount of uncertainty or risk about the size of changes in a security's value. It shows the extent to which the return of the underlying asset will fluctuate between now and the option's expiration. It can either be measured by using the standard deviation or variance between returns from that same security or market index. A higher volatility means that a security's value can potentially be spread out over a larger range of values. This means that the price of the security can change dramatically over a short time period in either direction. A lower volatility means that a security's value does not fluctuate dramatically, but changes in value at a steady pace over a period of time.

4 Mathematical formulation of the problem

4.1 Monte Carlo Formulation

Let $S = (S_t)_{t \geq 0}$ be the solution to the SDE :

$dS_t = rS_t dt + \sigma(log(S_t), t)S_t dW_t$

where $W = (W_t)_{t \geq 0}$ is a standard Brownian motion

r>0 is the risk free interest rate

σ is the local volatility function

The **price of the call option** is then given by : $V = e^{-rT}E(S_T - K)^+$
T : Final time / maturity / expiration time

In practice σ will typically be computed from the market observed implied volatility surface and is often represented either as a bicubic spline or as a series of one-dimensional splines.

To keep things simple , we choose to represent σ as

$\sigma(x, t) = g(x).t$

$g : R- > R_+$ is given by :

$g(x) = \frac{p_m(x)}{q_n(x)} = \frac{a_0 + a_1 x + ... + a_m x^m}{b_0 + b_1 x + ... + b_n x^n}$

p_n and q_m are polynomials of order n and m respectively .

To compute V , we consider the log process $X_t = log(S_t)$ which satisfies the SDE
$dX_t = (r - \frac{\sigma^2(X_t,t)}{2})dt + \sigma(X_t, t)dW_t$

Setting $\Delta = \frac{T}{M}$ for some integer M , and defining MC time steps $t_i = i\Delta$ for i=1,2,....,M :

$X_{t_{i+1}} = X_{t_i} + (r - \frac{\sigma^2(X_{t_i},t_i)}{2})\Delta + \sigma(X_{t_i},t_i)\sqrt{\Delta}Z_i$
where : each Z_i : standard normal random number and $X_{t_0} = log(S_0)$

4.2 Sensitivities

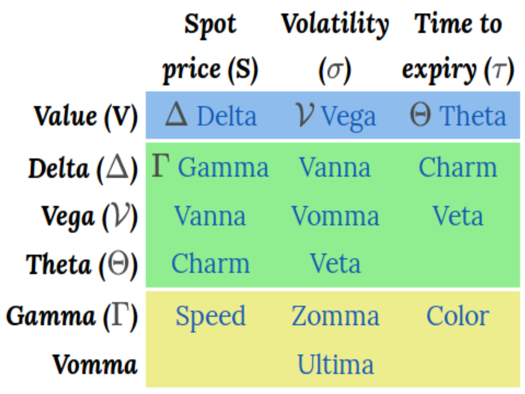
We can use AD to compute sensitivities of V with respect to the input parameters $K, T, r, S_0, a_0, ..., a_m$ and $b_0, ..., b_n$.

Active outputs : We are interested in their rate of change
Active inputs : We are interested in rate of change with respect to them

Passive outputs : We are NOT interested in their rate of change
Passive inputs : We are NOT interested in rate of change with respect to them

eg : Active input : S0
Passive input : a_i in calculation of sigma

Some relevant derivatives among active inputs and active outputs have special names (called Greeks because they are named after greek symbols)



	Spot price (S)	**Volatility (σ)**	**Time to expiry (τ)**
Value (V)	Δ Delta	$\mathcal{V}$ Vega	Θ Theta
Delta (Δ)	Γ Gamma	Vanna	Charm
Vega ($\mathcal{V}$)	Vanna	Vomma	Veta
Theta (Θ)	Charm	Veta	
Gamma (Γ)	Speed	Zomma	Color
Vomma		Ultima	

5 Modes of Differentiation

Let $z = f(a, b, c)$
Tangent mode / Forward mode : Derivative of output of every "layer" wrt input of that layer is calculated / stored . $\frac{\partial f}{\partial a}$, $\frac{\partial f}{\partial b}$, $\frac{\partial f}{\partial c}$

Reverse / Adjoint mode : Derivative of input of every "layer" wrt output of that layer is calculated / stored . $\frac{\partial a}{\partial f}$, $\frac{\partial b}{\partial f}$, $\frac{\partial c}{\partial f}$

Example Expression[4] : **z = x * y + sin(x)**
We are interested in the derivatives of output wrt input , i.e. $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$
Steps to calculate z :
x = input x
y = input y
a = x * y
b = sin(x)
z = a + b

5.1 Tangent Mode

Using chain rule : $\frac{\partial z}{\partial t} = \Sigma\left(\frac{\partial z}{\partial u_i}.\frac{\partial u_i}{\partial t}\right)$ where t is some input variable like x .
Our example[4] : **z = x * y + sin(x)**

$$\frac{\partial x}{\partial t} = ?$$
$$\frac{\partial y}{\partial t} = ?$$
$$\frac{\partial a}{\partial t} = y * \frac{\partial x}{\partial t} + x * \frac{\partial y}{\partial t}$$
$$\frac{\partial b}{\partial t} = cos(x) * \frac{\partial x}{\partial t}$$
$$\frac{\partial z}{\partial t} = \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t}$$

Now , if we want $\frac{\partial z}{\partial x}$, t = x and therefore we seed $\frac{\partial x}{\partial x} = 1$ and $\frac{\partial y}{\partial x} = 0$.

$$\begin{aligned}
\frac{\partial x}{\partial t} &= 1\\
\frac{\partial y}{\partial t} &= 0\\
\frac{\partial a}{\partial t} &= y * 1 + x * 0\\
\frac{\partial b}{\partial t} &= cos(x) * 1\\
\frac{\partial z}{\partial t} &= \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t}\\
\frac{\partial z}{\partial t} &= y + cos(x)
\end{aligned}$$

$$\frac{\partial z}{\partial x} = y + cos(x)$$

Advantage : The differential variables depend on the intermediate variables, so they can be calculated together . There is no need to hold on to the the intermediate variables until later, **saving memory**.

Interleaved calculation :

$$\begin{aligned}
x &= input\\
\frac{\partial x}{\partial t} &= ?\\
y &= input\\
\frac{\partial y}{\partial t} &= ?\\
a &= x * y\\
\frac{\partial a}{\partial t} &= y * \frac{\partial x}{\partial t} + x * \frac{\partial y}{\partial t}\\
b &= sin(x)\\
\frac{\partial b}{\partial t} &= cos(x) * \frac{\partial x}{\partial t}\\
z &= a + b\\
\frac{\partial z}{\partial t} &= \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t}
\end{aligned}$$

Disadvantages : Since we set t = x to calculate $\frac{\partial z}{\partial x}$,
we will need to set t = y for $\frac{\partial z}{\partial y}$. This means number of passes = number of input

variables .

If the number of input variables is very large , this leads to as many number of passes .

5.2 Adjoint Mode

Considering turning the chain rule upside down : $\frac{\partial s}{\partial u} = \Sigma\left(\frac{\partial z_i}{\partial u} . \frac{\partial s}{\partial z_i}\right)$

z_i : output variables of interest
u : input variables (x and y)
s : some output variable (either of the z_i)

For example problem[4] : **z = x * y + sin(x)**

$$\begin{aligned}
&\frac{\partial s}{\partial z} = ? \\
&\frac{\partial s}{\partial b} = \frac{\partial s}{\partial z} \\
&\frac{\partial s}{\partial a} = \frac{\partial s}{\partial z} \\
&\frac{\partial s}{\partial y} = x * \frac{\partial s}{\partial a} \\
&\frac{\partial s}{\partial x} = y * \frac{\partial s}{\partial a} + cos(x) * \frac{\partial s}{\partial b}
\end{aligned}$$

We put s = z , seed $\frac{\partial s}{\partial z} = 1$ and in just 1 pass get $\frac{\partial s}{\partial x}$ and $\frac{\partial s}{\partial y}$.

$$\frac{\partial s}{\partial z} = 1$$
$$\frac{\partial s}{\partial b} = 1$$
$$\frac{\partial s}{\partial a} = 1$$
$$\frac{\partial s}{\partial y} = x * 1$$
$$\frac{\partial s}{\partial x} = y * 1 + cos(x) * 1$$

$$\frac{\partial z}{\partial y} = x$$
$$\frac{\partial z}{\partial x} = y + cos(x)$$

Disadvantage : Now calculations and differential calculation cannot be interleaved. We need to save the intermediate variables also , and then calculate the differentials, thus leading to more memory use .

Disadvantage : If number of active output variables is large and number of active input variables is small , say 1 , tangent mode would be better since it requires only 1 pass .

6 Tools available for Computational Differentiation

There are various open source tools available for Automatic Differentiation which can be found at [2] :

`www.autodiff.org`

We can find tools categorized by language .

The tool I studied for the seminar is `dco/c++` by NAG .

7 dco/c++

`dco/c++` is an operator overloading AD tool developed by Numerical Algorithms Group [3]. Following pseudocode code illustrates basic use of `dco/c++` with reference to the pricing problem in section 4.1 . We suppose there are generic (templated) data types for active inputs and outputs, and regular (non-templated) data types for passive inputs and outputs:

```
template<typename ATYPE >
struct ACTIVE INPUTS {
        ATYPE S0,r,K,T;
        LocalVolSurface<ATYPE> sigmaSq ;
};
template<typename ATYPE>
struct ACTIVE OUTPUTS {
ATYPE V;
};
struct PASSIVE INPUTS {
        int N,M;
        double rngseed [6] ;
};
struct PASSIVE OUTPUTS {
        double ci ;
};
```

[1]

LocalVolSurface is a generic type representing the local volatility surface and contains the parameters of the PadÃľ approximation , while V is the Monte Carlo option price and ci is half the width of the confidence interval. Suppose there is a generic (templated) primal function

```

template<typename ATYPE>
void price(
const ACTIVE INPUTS<ATYPE> &X,
const PASSIVE INPUTS &XP,
ACTIVE OUTPUTS <ATYPE> &Y,
PASSIVE OUTPUTS &YP
);
```

[1]

mapping inputs XP and X to outputs YP and Y .

7.1 dco/c++: Tangent Mode

```
#include "dco.hpp"
typedef dco::gt1s<double> DCO_MODE;
```

```
3  typedef DCO_MODE::type DCO_TYPE;
4
5  ACTIVE_INPUTS<DCO_TYPE> X;
6  PASSIVE_INPUTS XP;
7  ACTIVE_OUTPUTS<DCO_TYPE> Y;
8  PASSIVE_OUTPUTS YP;
9
10 dco::derivative(X.S0)=1;
11 price(X,XP,Y,YP);
12 cout<<"Y=" << dco::derivative(Y.V) << endl;
13 cout<<"dY/dX.S0="<<dco::derivative(Y.V)<<endl;
14
15 dco::derivative(X.S0)=0;
16 dco::derivative(X.r)=1;
17 price(X,XP,Y,YP);
18 cout<<"dY/dX.r=" << dco::derivative(Y.V)<<endl;
```

[1]

In the above pseudocode active input and output types are instantiated with the dco/c++ first-order tangent data type dco:: gt1s<double>::type . Calling dco:: derivative in line 10 is equivalent to setting $x^{(1)}$ to the Cartesian basis vector corresponding to $\frac{\partial V}{\partial S_0}$. The option value is obtained in line 12 and the derivative value in line 13. Lines 15-16 seed $x^{(1)}$ with the Cartesian basis vector for $\frac{\partial V}{\partial r}$. The primal code is run again on line 17 and the derivative is printed on line 18.

We notice that the tangent code is run as many number of time as are the number of active input variables we are interested in .

7.2 dco/c++: Adjoint Mode

```
#include "dco.hpp"
typedef dco::ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
DCO_TAPE_TYPE* & DCO_TAPE_POINTER=DCO_MODE::global_tape;

ACTIVE_INPUTS<DCO_TYPE> X;
PASSIVE_INPUTS XP;
ACTIVE_OUTPUTS<DCO_TYPE> Y;
PASSIVE_OUTPUTS YP;

DCO_TAPE_POINTER = DCO_TAPE_TYPE::create();
DCO_TAPE_POINTER->register_variable(X.S0);
```

```
14 DCO_TAPE_POINTER->register_variable(X.r);
15 ...
16
17 price(X,XP,Y,YP);
18
19 DCO_TAPE_POINTER->register_output_variable(Y.V);
20 dco::derivative(Y.V)=1;
21 DCO_TAPE_POINTER->interpret_adjoint();
22
23 cout<<"Y=" << dco::value(Y.V) << endl;
24 cout<<"dY/dX.S0="<<dco::derivative(X.S0)<<endl;
25 cout<<"dY/dX.r="<<dco::derivative(X.r)<<endl;
26 ...
27
28 DCO_TAPE_TYPE::remove(DCO_TAPE_POINTER);
```

[1]

The tape as well as the active data types are instantiated with the `dco/c++` first-order adjoint type dco :: ga1s<double>::type . All the active variables are then registered with the tape (lines 13-15) and the primal code is run to populate the tape (line 17). Y.V is marked as the active output in line 19. In line 20 the input adjoint $y_{(1)}$ from (5) is set equal to 1 before playing the tape back in line 21. The value is printed in line 23 before all the derivatives are printed in lines 24-26. Lastly memory allocated by the tape is released in line 28.

We notice that the full gradient is obtained from one run of the adjoint code.

7.3 Checkpointing

$$F \; : \; (\mathbf{x},\tilde{\mathbf{x}}) \xrightarrow{f_1} (\mathbf{u},\tilde{\mathbf{u}}) \; , \begin{pmatrix} (\mathbf{u},\tilde{\mathbf{u}}_1) \xrightarrow{g} (\mathbf{v}_1,\tilde{\mathbf{v}}_1) \\ (\mathbf{u},\tilde{\mathbf{u}}_2) \xrightarrow{g} (\mathbf{v}_2,\tilde{\mathbf{v}}_2) \\ \vdots \\ (\mathbf{u},\tilde{\mathbf{u}}_N) \xrightarrow{g} (\mathbf{v}_N,\tilde{\mathbf{v}}_N) \end{pmatrix} , \; (\mathbf{v}_1,\dots,\mathbf{v}_N,\tilde{\mathbf{v}}_1,\dots,\tilde{\mathbf{v}}_N) \xrightarrow{f_2} (y,\tilde{\mathbf{y}})$$

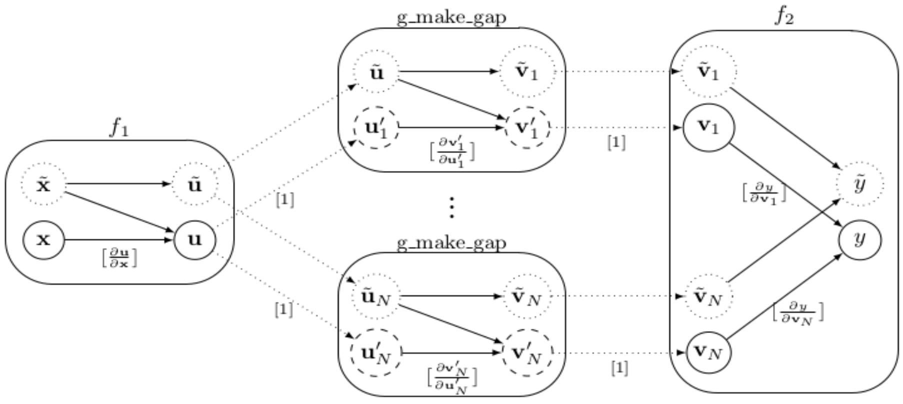
[1]

In above picture , g is the Euler-Maruyama integrator and $\widetilde{\boldsymbol{u}}_i$ represents sample-path-specific passive inputs such as the random numbers (it includes $\widetilde{\boldsymbol{u}}$ from f_1). The outputs $\boldsymbol{v}_i$ represent the N sample paths, and f_2 is the payoff function.

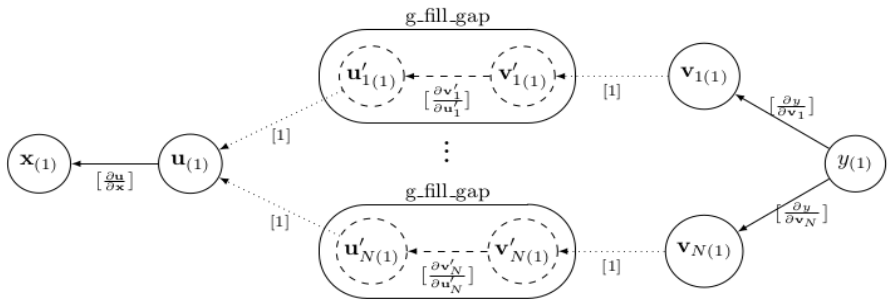
For the first-order adjoint version of the Monte Carlo application (Section 4.1) , the memory requirements of the tape scale more or less linearly with the number of sample paths . This may lead to infeasible memory peak requirements . The N evaluations of the Euler - Maruyama integrator (g) lead to the explosion in tape size .

To tackle this problem , during the forward run , rather than recording evaluations of the Euler - Maruyama integrator to the tape , checkpointing can be done as follows :

A checkpoint of the output $(\boldsymbol{u}, \widetilde{\boldsymbol{u}})$ of f_1 and is made and an external function *g_make_gap* is provided which computes g with standard floating-point data types (eg double) instead of `dco/c++` adjoint types. This effectively creates a gap in the tape for each evaluation of g.



(a)



g_fill_gap
$\mathbf{u}'_{1(1)}$
$\mathbf{v}'_{1(1)}$
$[\frac{\partial \mathbf{v}'_1}{\partial \mathbf{u}'_1}]$
[1]
$\mathbf{v}_{1(1)}$
$[\frac{\partial y}{\partial \mathbf{v}_1}]$
$\mathbf{x}_{(1)}$
$\mathbf{u}_{(1)}$
$[\frac{\partial \mathbf{u}}{\partial \mathbf{x}}]$
$y_{(1)}$
g_fill_gap
$\mathbf{u}'_{N(1)}$
$\mathbf{v}'_{N(1)}$
$[\frac{\partial \mathbf{v}'_N}{\partial \mathbf{u}'_N}]$
$\mathbf{v}_{N(1)}$
$[\frac{\partial y}{\partial \mathbf{v}_N}]$

(b)

[1]

Once *g_make_gap* has computed g it registers the outputs $(\boldsymbol{v}_i\prime, \widetilde{\boldsymbol{v}_i})$ with the EAO which inserts $\boldsymbol{v}_i\prime$ into new `dco/c++` adjoint types $\boldsymbol{v}_i$ and registers them with the tape.

Therefore the active outputs $\boldsymbol{v}_i\prime$ of g are of type double , while the active inputs $\boldsymbol{v}_i$ of f_2 are `dco/c++` first-order adjoint types. Consequently f_1 and f_2 record values to the tape while the calls to g do not.

Interpretation of the tape for a given input adjoint $y_{(1)}$ starts as normal. The tape interpreter computes all the adjoints

$$\boldsymbol{v}_{i(1)} := \boldsymbol{v}_{i(1)} + \frac{\partial y^T}{\partial v_i} \cdot y_{(1)} = \frac{\partial y^T}{\partial v_i} \cdot y_{(1)}$$

for i = 1, . . . , N where the equality follows since $v_{i(1)}$ is initialized to zero. For each gap in the tape the interpreter then calls the user defined adjoint function *g__fill__gap* which must compute

$$\boldsymbol{u}_{(1)} := \boldsymbol{u}_{(1)} + \frac{\partial \boldsymbol{v}_{i'}^{T}}{\partial \boldsymbol{u}} . \boldsymbol{v}_{i(1)'} = \boldsymbol{u}_{(1)} + \frac{\partial \boldsymbol{v}_{i}^{T}}{\partial \boldsymbol{u}} . \boldsymbol{v}_{i(1)}$$

where the equality follows since vi was created from $\boldsymbol{v}_{i'}$ by assignment. The easiest way to do this is to use `dco/c++` itself. The checkpoint is restored, the value $\boldsymbol{u}_{i'}$ is inserted into new `dco/c++` first-order adjoint data types u i , and g is computed recording values onto a (conceptually new) tape. This tape is seeded with the input adjoint $\boldsymbol{v}_{i(1)}$ and is interpreted, yielding a local adjoint $\boldsymbol{u}_{i(1)}$ which is then added to the running sum $\boldsymbol{u}_{(1)} := \boldsymbol{u}_{(1)} + \boldsymbol{u}_{i(1)}$. Once this has been done for all gaps, the interpreter can compute the adjoint $\boldsymbol{x}_{(1)} := \boldsymbol{x}_{(1)} + \frac{\partial \boldsymbol{u}^{T}}{\partial \boldsymbol{x}} . \boldsymbol{u}_{(1)}$ of f_1 which completes the adjoint calculation.

The method described above fills only one gap at a time. Since the evaluations of g are mutually independent , several gaps can be filled in parallel, leading to a parallel adjoint calculation.

The checkpointing scheme presented above trades computation for memory: each sample path is effectively computed twice in order to complete the adjoint calculation. Although the increase in computation does not scale linearly with the decrease in peak memory requirements meaning the trade-off is overall advantageous .

8 Performance Comparison

n	`mc/primal`	`mc/cfd`	`mc/a1s`	`mc/a1s_ensemble`	$\mathcal{R}$
10	0.3s	6.1s	1.8s (2GB)	1.3s (1.9MB)	4.3
22	0.4s	15.7s	- (> 3GB)	2.3s (2.2MB)	5.7
34	0.5s	29.0s	- (> 3GB)	3.0s (2.5MB)	6.0
62	0.7s	80.9s	- (> 3GB)	5.1s (3.2MB)	7.3
142	1.5s	423.5s	- (> 3GB)	12.4s (5.1MB)	8.3
222	2.3s	1010.7s	- (> 3GB)	24.4s (7.1MB)	10.6

[1]

Table 1: Run times and peak memory requirements as a function of gradient size n for `dco/c++` first-order adjoint code vs. central finite differences for the Monte Carlo kernel from Section 4.1. Naive first-order adjoints for n>=22 required too much memory to run. The relative computational cost R is given for mc/a1s ensemble. Although theoretically constant, R is sensitive to specifics such as compiler flags, memory hierarchy and cache sizes, and level of optimization of the primal code.

n	`pde/primal`	`pde/cfd`	`pde/a1s`	`pde/a1s_checkpointing`	$\mathcal{R}$
10	0,3s	6.5s	- (> 3GB)	5,2s (205MB)	17.3
22	0,5s	19.6s	- (> 3GB)	8,3s (370MB)	16.6
34	0,6s	37.7s	- (> 3GB)	11,6s (535MB)	19.3
62	1,0s	119.5s	- (> 3GB)	18,7s (919MB)	18.7
142	2,6s	741.2s	- (> 3GB)	39s (2GB)	15.0
222	4,1s	1857.3s	- (> 3GB)	60s (3GB)	14.6

[1]

Table 2: Run time and peak memory requirements as a function of the gradient size n of the naive and checkpointed first-order adjoint codes vs. central finite differences. The checkpointing used is equidistant (every 10th time step). The naive adjoint ran out of memory even for the smallest problem size. The relative computational cost R is given for pde/a1s checkpointing.

References

[1] Uwe Naumann and Jacques du Toit. *Adjoint Algorithmic Differentiation Tool Support for Typical Numerical Patterns in Computational Finance*. 2014.

[2] AutoDiff Home Page
`http://www.autodiff.org/`

[3] NAG : `dco/c++` download page
`https://www.nag.com/content/downloads-dco-c-versions`

[4] Reverse-mode automatic differentiation: a tutorial
`https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation`

[5] Algorithmic Differentiation The Art of Differentiating Computer Programms
`https://www.stce.rwth-aachen.de/teaching/lectures/oxford2017`