Adjoint Algorithmic Differentiation Tool Support for Typical Numerical Patterns in Computational Finance

Uwe Naumann* and Jacques du Toit

RWTH Aachen University, Germany and
Numerical Algorithms Group (NAG) Ltd., UK

Abstract

We demonstrate the flexibility and ease of use of C++ algorithmic differentiation (AD) tools based on overloading to numerical patterns (kernels) arising in computational finance. While adjoint methods and AD have been known in the finance literature for some time, there are few tools capable of handling and integrating with the C++ codes found in production. Adjoint methods are also known to be very powerful but to potentially have infeasible memory requirements. We present several techniques for dealing with this problem and demonstrate them on numerical kernels which occur frequently in finance. We build the discussion around our own AD tool `dco/c++` which is designed to handle arbitrary C++ codes and to be highly flexible, however the sketched concepts can certainly be transferred to other AD solutions including in-house tools. An archive of the source code for the numerical kernels as well as all the AD solutions discussed can be downloaded from an accompanying website. This includes documentation for the code and `dco/c++`. Trial licences for `dco/c++` are available from NAG.

1 Introduction and Motivation

Ever since the work of Black, Scholes and Merton, the pricing and risk management of contingent claims have been closely tied to the calculation of mathematical derivatives. To hedge a claim in the complete market framework it is sufficient to know the derivative of the price with respect to the underlying, and consequently among quantitative finance practitioners the term "risk" has become more or less synonymous with differentiation.

Unfortunately the vast majority of financial problems do not admit closed form solutions. Often all we have is a computer program which calculates an approximate solution. Obtaining mathematical derivatives becomes impossible and we have to resort to numerical techniques, the simplest (and most popular) of which is finite differences. The problem is that finite differencing is often computationally expensive (for large gradients), and for higher derivatives it can become arbitrarily inaccurate.

Algorithmic differentiation (AD) [Griewank and Walter, 2008, Naumann, 2012] is a numerical technique which gives exact mathematical derivatives of a given computer program. No approximations are made: the given computer program is differentiated exactly, to any order, and to machine accuracy. In addition adjoint AD (AAD) can compute gradients at a computational cost which is independent of the size of the gradient, and is typically within a (small) constant factor of the cost of running the original program. For this reason AD (and AAD in particular) has generated considerable interest in many areas of Computational Science and Engineering; see, for example, the proceedings of the last three international conferences on AD [Bischof et al., 2008, Bücker et al., 2005, Forth et al., 2012]. The main problem with AAD is that

*corresponding author; E-mail: `naumann@stce.rwth-aachen.de`

it typically requires huge amounts of memory unless some care is taken. A large part of this paper is therefore devoted to discussing techniques for reducing this memory requirement.

Although AD has been known in the finance community for some time now [Giles and Glasserman, 2006, Capriotti, 2011], adoption has been relatively slow. The most common reasons that finance practitioners cite for this are

(i) unfamiliarity with the method,

(ii) inability to implement AD "by hand" across a large code base,

(iii) fears of running out of memory when doing adjoint AD,

(iv) uncertainty on how to apply AD to a particular financial problem.

The aim of this paper is to address these issues. The most serious objection raised is (ii). Doing AD by hand on any production size code is simply not feasible and is incompatible with modern software engineering strategies. For production code one has to use software tools. These tools are distinguished by how easy they are to use, which languages and language features they support, how flexible they are, and how efficient the resulting derivative code is. The tool we propose is `dco/c++` (derivative computation by overloading) developed at RWTH Aachen University in collaboration with NAG. `dco/c++` has been applied successfully to industrial C/C++ codes numbering in the millions of lines and is easy to use, efficient and highly flexible. We will give a fuller description of `dco/c++` later on.

Our approach to (iv) above has been to identify several common "numerical patterns" (computational kernels) which appear frequently in financial codes and which in some sense are basic building blocks: evolutions (sequential loops), ensembles (parallel loops), linear solvers, root finders and optimization. This paper addresses the first two kernels, leaving the last three to a subsequent publication. Around these first two kernels we have written small financial applications (test applications) to place the kernels in their proper context, and we apply AD to these test applications. We demonstrate various strategies which exploit particular features of the kernels to reduce the memory requirements and/or computational cost.

As will be clear from our discussion, AD is essentially a transformation of source code. The source for the entire application therefore has to be available. This is a problem for codes which depend on closed source third party libraries. In Section 6 below we briefly address various topics arising from our discussion, including options for handling closed source libraries within the context of `dco/c++`.

An archive of all the source code for these test applications and the AD strategies we have applied is available from

`http://www.nag.com/adtoolsforfinance/sourcematerials`

The code is documented (including documentation for `dco/c++`) and its structure closely follows the mathematical presentation in this paper. Readers can study the code to see the details of implementing the AD strategies we discuss, and can also use the trial copy of `dco/c++` included in the archive on their own codes. Trial licences for `dco/c++` can be obtained from NAG via the download link.

2 AD in a Nutshell

AD is a semantic transformation[1] of a given computer code called the *primal code* or *primal function*: in addition to computing the primal function value, the transformed code also computes the derivatives of the primal function with respect to a specified set of parameters.

[1] I.e. changes the meaning

Consider a computer implementation of a function F mapping $\mathbb{R}^{n+\tilde{n}}$ into $\mathbb{R}^{m+\tilde{m}}$. We assume that we are interested in derivatives of the first m outputs of F (the *active outputs*) with respect to the first n inputs (the *active inputs*). The second $\tilde{m}$ outputs and the second $\tilde{n}$ inputs of F are termed the *passive outputs* and *passive inputs* respectively. For example, an active output may be the Monte Carlo price of an option while a passive output may be the corresponding confidence interval (see Section 3). An active input may be the initial asset price S_0, while a passive input may be the set of random numbers used in the Monte Carlo simulation.

Without loss of generality and to keep the notation simple, we restrict the discussion to scalar active outputs i.e. $m = 1$.[2] We therefore consider multivariate functions of the type

$$F : \mathbb{R}^n \times \mathbb{R}^{\tilde{n}} \to \mathbb{R} \times \mathbb{R}^{\tilde{m}}, \quad (y, \tilde{\mathbf{y}}) = F(\mathbf{x}, \tilde{\mathbf{x}}), \tag{1}$$

where we assume that F and its computer implementation are differentiable up to the required order [3]. We are interested in (semi-)automatic ways to generate the vector of all partial derivatives of the active output y with respect to the active inputs $\mathbf{x}$, that is the gradient

$$\nabla F = \nabla F(\mathbf{x}, \tilde{\mathbf{x}}) \equiv \left(\frac{\partial y}{\partial x_i} \right)_{i=0,\ldots,n-1} \in \mathbb{R}^n, \tag{2}$$

along with the values of all active and passive outputs as functions of the active and passive inputs. Similarly, we look for the Hessian of all second partial derivatives of y with respect to $\mathbf{x}$

$$\nabla^2 F = \nabla^2 F(\mathbf{x}, \tilde{\mathbf{x}}) \equiv \left(\frac{\partial^2 y}{\partial x_j \partial x_i} \right)_{j,i=0,\ldots,n-1} \in \mathbb{R}^{n \times n}. \tag{3}$$

2.1 Tangent Mode AD

In the following we use the notation from [Naumann, 2012]. *Forward (also: tangent) mode AD* yields a tangent version $F^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^{\tilde{n}} \to \mathbb{R} \times \mathbb{R} \times \mathbb{R}^{\tilde{m}}$ of the primal function F. The tangent code is a function $(y, y^{(1)}, \tilde{\mathbf{y}}) := F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}, \tilde{\mathbf{x}})$ where

$$y^{(1)} := \nabla F(\mathbf{x}, \tilde{\mathbf{x}})^T \cdot \mathbf{x}^{(1)} \tag{4}$$

is the directional derivative of F in the direction $\mathbf{x}^{(1)}$. We use superscripts $^{(1)}$ to denote first-order tangents. The operator := represents the assignment of imperative programming languages, not to be confused with equality = in the mathematical sense. The entire gradient can be calculated entry by entry with n runs of the tangent code through a process known as *seeding* and *harvesting*: the vector $\mathbf{x}^{(1)}$ in (4) is successively set equal to each of the Cartesian basis vectors in $\mathbb{R}^n$ (it is *seeded*), the tangent code is run and the corresponding gradient entry is *harvested* from $y^{(1)}$. The gradient is computed with machine accuracy while the computational cost is $O(n) \cdot \mathrm{Cost}(F)$, the same as that of a finite difference approximation.

2.2 Adjoint Mode AD

Reverse (also: adjoint) mode AD yields an adjoint version

$$F_{(1)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^{\tilde{n}} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}^{\tilde{m}} \times \mathbb{R}^n \times \mathbb{R}$$

[2] The modifications for $m > 1$ are straightforward.

[3] Differentiability is a crucial prerequisite for (algorithmic) differentiation. Non-differentiability at selected points can be handled by smoothing techniques as well as by combinations of AD with local finite difference approximations. A detailed discussion of these issues is beyond the scope of this article. Its results remain valid under the presence of points of non-differentiability despite the fact that generalized derivative information needs to be dealt with. Refer to [Griewank, 2013] for recent work on extensions of AD to non differentiable numerical simulation code.

of the primal function F. The adjoint code is a function $(y, \tilde{\mathbf{y}}, \mathbf{x}_{(1)}, y_{(1)}) := F_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \tilde{\mathbf{x}}, y_{(1)})$ where

$$\begin{aligned} \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \nabla F(\mathbf{x}, \tilde{\mathbf{x}})^T \cdot y_{(1)} \\ y_{(1)} &:= 0 \end{aligned} . \tag{5}$$

The adjoint code therefore increments given adjoints $\mathbf{x}_{(1)}$ of the active inputs with the product of the gradient and a given adjoint $y_{(1)}$ of the active output. The adjoint output is reset to zero in order to ensure correct incrementation of adjoints of previously used (in the primal code) values which are represented by the same program variable y. See [Naumann, 2012] for details. We use subscripts $_{(1)}$ to denote first-order adjoints. Initializing (seeding) $\mathbf{x}_{(1)} = \mathbf{0}$ and $y_{(1)} = 1$ yields the gradient in $\mathbf{x}_{(1)}$ from a single run of the adjoint code. Again, the gradient is computed with machine accuracy. The computational cost no longer depends on the size of the gradient n.

2.3 Second Derivatives

The second derivative is the first derivative of the first derivative. Second derivative code can therefore be obtained by any of the four combinations of forward and reverse modes. In *forward-over-forward mode AD* forward mode is applied to the tangent code from Section 2.1 above. The resulting model has three vectors that are seeded, and the full Hessian can be harvested from n^2 runs of the second-order tangent code. The computational cost is of the same order as that of a second-order finite difference approximation. Obviously, the accuracy of the latter is typically far from satisfactory, particularly for calculations performed in single precision floating-point arithmetic. For further details of forward-over-forward mode AD we refer to [Naumann, 2012].

Symmetry of the Hessian implies mathematical equivalence of the three remaining second-order adjoint modes: see [Naumann, 2012] for details. We therefore only consider one of them. In *forward-over-reverse mode AD* forward mode is applied to the adjoint code from Section 2.2 above, yielding a second-order adjoint version of the primal code. The model has 4 inputs that are seeded and yields the full Hessian in n runs of the second-order adjoint code. The reduction in computational complexity due to the initial application of adjoint mode to the primal code is therefore carried over to the second-order adjoint. Sparsity of the Hessian can and should be exploited [Gebremedhin et al., 2005]. Lastly, we note that a Hessian-vector product can be computed in one run of the second-order adjoint code. This can be of interest if the Hessian is known to be strongly diagonally dominant and only the diagonal is desired.

2.4 Higher Derivatives

Higher-order AD codes can be built by recursively applying tangent or adjoint modes to the primal code to build up the derivative calculation. For further details see [Naumann, 2012].

2.5 Efficiency and Memory Requirements

The efficiency (or cost) of AD codes is usually measured as the ratio

$$\mathcal{R} = \frac{\text{Runtime of given AD code}}{\text{Runtime of primal code}} \tag{6}$$

For first order adjoint AD the ratio $\mathcal{R}$ typically ranges between 1 and 100 depending on mathematical and structural properties of the primal code and how well these are exploited within the given AD solution. Compared to forward mode AD or finite differences, $\mathcal{R}$ gives an indication of how large the gradient must be before adjoint methods become attractive. For `dco/c++` this ratio is usually less than 15 and often less than 10, making adjoint methods interesting even for small gradients.

The main complication in adjoint AD is that the primal code effectively has to be run backwards. A full exposition of this fact is given in [Naumann, 2012] and is beyond the scope of this article. Running the primal code backwards means that certain intermediate values computed by the primal code must be made available in reverse order (*data flow reversal*). This can rapidly result in excessive memory requirements for all but very simple numerical simulations.

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. The tape is generated and populated when the adjoint code is run forward, and the tape is then *interpreted* (played back) to reverse the data flow and compute the adjoint. Consequently, adjoint code (and especially tape-based adjoint code) exhibits a possibly infeasible memory requirement to be dealt with when applying AD in practice. We will see how `dco/c++` deals with these issues in Section 4.1 below.

3 Numerical Patterns and Test Applications

The numerical patterns we address in this paper are *ensembles* (parallel loops) in the context of Monte Carlo simulation and *evolutions* (sequential loops) in the context of solvers for parabolic PDEs. The symbolic differentiation of direct solvers of linear systems and of iterative root finders and optimizers in the context of AD implementation using `dco/c++` will be dealt with in a subsequent publication.

To illustrate how to approach these numerical patterns from an AD perspective we have implemented two solutions to a simple pricing problem. A related discussion is presented in [W. Xu, 2014]. The distinguishing feature of our work is the support of the formal techniques by a ready to use software tool and publication of a reference implementation.

3.1 Monte Carlo Pricing

Monte Carlo needs no introduction. From the point of view of AD, and in particular when computing adjoints, Monte Carlo methods can lead to problems since the memory requirements scale more or less linearly with the number of sample paths. It is important to be able to control the memory use without affecting performance.

As a test application we consider a simple European call option written on an underlying driven by a local volatility process. Let $S = (S_t)_{t\geq 0}$ be the solution to the SDE

$$dS_t = rS_t\,dt + \sigma\big(\log(S_t), t\big)S_t\,dW_t \tag{7}$$

where $W = (W_t)_{t\geq 0}$ is a standard Brownian motion, $r > 0$ is the risk free interest rate and σ is the local volatility function. The price of the call option is then given by

$$V = e^{-rT}\mathbb{E}(S_T - K)^+ \tag{8}$$

for maturity $T > 0$ and strike $K > 0$.

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. While we could have treated such a case it would have resulted in a slightly more complex test application than needed. Instead, to keep the code simple, we chose to represent σ as

$$\sigma(x, t) = g(x) \cdot t \tag{9}$$

where $g : \mathbb{R} \mapsto \mathbb{R}_+$ is given by a Padé approximation

$$g(x) = \frac{p_m(x)}{q_n(x)} = \frac{a_0 + a_1x + \cdots + a_mx^m}{b_0 + b_1x + \cdots + b_nx^n} \tag{10}$$

for p_n and q_m polynomials of order n and m respectively. The approximation is chosen so that g is positive and has a reasonable shape over all likely values of x. We wish to note that this choice of σ was motivated simply by a desire to keep the code simple: the AD techniques can handle any method of representing σ.

To compute the call price V from (8) above we apply a simple Euler–Maruyama scheme to the log process $X_t = \log(S_t)$ which satisfies the SDE

$$dX_t = \big(r - \tfrac{1}{2}\sigma^2(X_t, t)\big)dt + \sigma(X_t, t)\, dW_t. \tag{11}$$

Setting $\Delta = T/M$ for some integer M and defining a sequence of Monte Carlo time steps $t_i = i\Delta$ for $i = 1, 2, \ldots, M$, we set

$$X_{t_{i+1}} = X_{t_i} + \big(r - \tfrac{1}{2}\sigma^2(X_{t_i}, t_i)\big)\Delta + \sigma(X_{t_i}, t_i)\sqrt{\Delta}Z_i \tag{12}$$

where each Z_i is a standard normal random number and $X_{t_0} = \log(S_0)$. N sample paths of the log process are generated and used in a Monte Carlo integral of (8) to estimate the price V and to obtain a confidence interval. We use AD to compute sensitivities of V with respect to the input parameters $K, T, r, S_0, a_0, \ldots, a_m$ and $b_0, \ldots, b_n$. By varying the order of the Padé approximation we can easily add more input parameters to the model.

3.2 PDE Methods

PDE methods are often preferred over Monte Carlo where tractability permits. The core of any parabolic PDE solver is a time marching evolution of "state", namely the value of the solution at that point in time. Moving from one time step to the next typically involves solving a linear system, for example in implicit or mixed (eg, Crank-Nicholson) methods.

From the point of view of adjoint AD this time marching evolution of state can be problematic since the memory requirements for the state scale more or less linearly with the number of time steps. We therefore need to constrain this memory use. In addition, approaching the linear solver in a naive way (this will be clarified later) will lead to memory and computational requirements for the solver that are $O(n^3)$. By exploiting the structure of the solver we can reduce both these to $O(n^2)$, which for large dense systems can be a dramatic improvement. Although our stiffness matrix will end up being tridiagonal, we note that large dense matrices often arise in partial integro-differential equations (PIDEs) associated with discontinuous processes (see, eg, [Cont and Tankov, 2004, Matache et al., 2004]). Dense linear systems also feature in other areas such as computing nearest correlation matrices (see, eg, [Borsdorf and Higham, 2010, Qi and Sun, 2006]) or in non-linear optimization. It is therefore useful to know how to handle the linear solver correctly.

Our test application consists of solving the same problem described in Section 3.1 above using a Crank-Nicholson scheme. In particular, we take the SDE (12) where σ is given by (9) and (10) and consider the pricing problem (8). To recast the pricing problem in a PDE setting we extend the value V into a value function $V : \mathbb{R} \times [0, T] \to \mathbb{R}_+$ given by

$$V(x, t) = e^{-r(T-t)}\mathbb{E}_{x,t}\big(e^{X_T} - K\big)^+ \tag{13}$$

where $\mathbb{P}_{x,t}$ is the measure under which the Markov process X starts at time $t \in [0, T]$ at the value $x \in \mathbb{R}$, and $\mathbb{E}_{x,t}$ denotes expectation with respect to $\mathbb{P}_{x,t}$. Standard results from the theory of Markov processes then show that V satisfies the parabolic PDE

$$\begin{aligned} 0 = \frac{\partial}{\partial t}V(x, t) + \big(r - \tfrac{1}{2}\sigma^2(x, t)\big)\frac{\partial}{\partial x}V(x, t) \qquad & (14) \\ + \tfrac{1}{2}\sigma^2(x, t)\frac{\partial^2}{\partial x^2}V(x, t) - rV(x, t) \quad \text{for } (x, t) \in \mathbb{R} \times [0, T), & \\ (e^x - K)^+ = V(x, T) \quad \text{for all } x \in \mathbb{R}. \qquad & (15) \end{aligned}$$

To these we add the asymptotic boundary conditions

$$\lim_{x\to-\infty} V(x,t) = 0 \quad \text{for all } t \in [0,T], \tag{16}$$

$$\lim_{x\to\infty} V(x,t) = e^{-r(T-t)}(e^x - K) \quad \text{for all } t \in [0,T]. \tag{17}$$

The system is solved by a Crank-Nicholson scheme as described in [Andersen and Brotherton-Ratcliffe, 2000]. We use AD to compute the same sensitivities as before.

4 AAD by Overloading in C++

Ideally AD should be implemented by source transformation, thus gaining full access to static program analysis and optimizing compiler technology. Source transformation by hand turns out to be tedious and highly error-prone for all but the simplest codes. Unfortunately, the source transformation tool support for C++ is still very rudimentary. Existing tools [Hascoët and Pascual, 2013, Schmitz et al., 2011, Voßbeck et al., 2008] deliver satisfactory results for codes written in (a subset of) C, however they fail to handle nontrivial C++ programs. Operator overloading combined with meta-programming techniques represent a useful alternative since they allow AD to be applied to any C++ code. If an overloading AD tool for C++ is to be considered state of the art, it should have the following features:

- It should provide first- and higher-order tangents and adjoints by instantiating a generic (templated) primal code with corresponding derivative types. Optimizations applied to the primal or derivative code of some order should be generically reusable by higher-order derivative code. For example, by switching the *base type* (see also Section 4.2) of an optimized first-order adjoint (perhaps featuring optimized checkpointing schemes and/or symbolically differentiated kernels) from a floating-point data type to a first-order derivative type, an efficient second-order adjoint code should be obtained.

- The cost $\mathcal{R}$ from (6) of a first-order adjoint code should be minimal – ideally less than 10 if the tape fits entirely into main memory. An aggressively optimized primal code will make this goal harder to achieve than a suboptimal one. This should be taken into account whenever looking at run time reports for adjoints. In any case, such reports are of limited use unless $\mathcal{R}$ is stated.

- The memory requirement of the tape generated by first-order adjoint code should be minimal. Ideally, the recorded data is limited to the essential items while low-cost compression techniques and cache optimized data layouts are employed.

One could argue that these issues have always been on the agenda of the various academic AD tool development efforts. Nevertheless the performance (both run time and memory requirement in adjoint mode) of different overloading tools varies significantly. Numerous in-house experiments as well as comparisons run by others indicate that `dco/c++` is more than just competitive, often exhibiting the lowest run time and the smallest tape size. However we would like to emphasise two additional issues which we found to be significantly more important when working with practitioners in computational finance:

1. Adjoints of nontrivial simulation codes are rarely obtained in a fully automatic fashion. Checkpointing, user-defined adjoints and other advanced techniques (discussed below) require flexible and intuitive access to the internal data generated by the AD tool. Absence of such access is likely to restrict the use of the tool to small and mid-sized primal codes. A complete quantitative finance library will probably not be a feasible target code.

2. It is crucial that a derivative code can be maintained easily. For this reason it is highly desirable to integrate the AD tool fully into the (complex) software platform already in use, since this greatly simplifies development, maintenance and deployment. Such deep integration requires the AD tool to have state of the art software engineering methodology, thorough testing, systematic bug tracking and elimination, version management and professional support. Adopting adjoint AD as a central component in a software development agenda will necessarily increase the overall complexity of the code base, however as far as possible this should be kept to a minimum.

`dco/c++` has been developed with these issues in mind. It has been used efficiently in distributed and shared memory parallel settings [Schanen et al., 2012], has been coupled with accelerators (GPUs) [Du Toit et al., 2014], and is actively used by a number of Tier-1 investment banks as well as several other industry sectors (including automotive and aerospace).

4.1 Introducing `dco/c++`

`dco/c++` is an operator overloading AD tool. In Section 2.5 we discussed a number of challenges that such tools face (the main being memory requirements), and we now take a closer look at how `dco/c++` deals with them. When used in adjoint mode, `dco/c++` must store a representation of the primal calculation to the tape at runtime. Execution of a generic (templated) primal code instantiated with the first order adjoint `dco/c++` data type increases the run time by a factor of up to 2 even if no tape is recorded. Recording of the tape into main memory adds another factor of 2 to 6 depending on the structure of the primal code[4]. By default the tape is a dynamically growing structure with chunks of memory allocated as needed, but there are options for statically sized tapes as well (which requires knowing how big the tape must be). Interpretation of the tape is typically very efficient adding another factor of < 1.5 due to the cache optimized data layout. On today's CPU-based workstations the run time overhead $\mathcal{R}$ induced by a first order adjoint code generated with `dco/c++` can typically be expected to lie between 5 and 15 if the tape fits into main memory. If the size of the tape exceeds the available main memory, which will most often be the case in practice, then additional measures need to be taken possibly resulting in a further increase or reduction of $\mathcal{R}$. Possibilities include checkpointing techniques for evolutions (sequential loops) and ensembles (parallel loops), as well as symbolically differentiated numerical kernels (eg, solvers for linear systems). These techniques are discussed below and we show how they can be integrated into `dco/c++`'s tape-based adjoint computation. The key enabling feature for all these advanced approaches is *user defined adjoint functions* which are discussed in Section 4.3. Further methods for improving efficiency and scalability of adjoint code generated with `dco/c++` include parallel recording of tapes, use of accelerators (eg, GPUs), and advanced pre-accumulation techniques. Some of these topics are briefly introduced in Section 6, however a full discussion is beyond the scope of this article.

4.2 Using `dco/c++`

We illustrate basic use of `dco/c++` with reference to the pricing problem from Section 3.1. The code accompanying this paper which solves the pricing problem has the same structure as presented below.

Suppose there are generic (templated) data types for active inputs and outputs, and regular (non-templated) data types for passive inputs and outputs:

[4] `dco/c++`'s assignment-level compression algorithm makes the recording cost depend on the length of a right-hand side of an assignment.

```
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;
};
```

LocalVolSurface is a generic type representing the local volatility surface and contains the parameters of the Padé approximation (10), 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
    );
```

mapping inputs XP and X to outputs YP and Y.

4.2.1 First-order tangent mode

To enable tangent first-order AD the active input and output types are instantiated with the `dco/c++` first-order tangent data type dco::gt1s<**double**>::type over base type **double**. The user then seeds and harvests the first-order tangent code, as is shown in the following code listing which computes $\partial V/\partial S_0$ and $\partial V/\partial r$:

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

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

  dco::derivative(X.S0)=1;
  price(X,XP,Y,YP);
  cout << "Y=" << dco::value(Y.V) << endl;
  cout << "dY/dX.S0=" << dco::derivative(Y.V) << endl;

  dco::derivative(X.S0)=0;
  dco::derivative(X.r)=1;
  price(X,XP,Y,YP);
  cout << "dY/dX.r=" << dco::derivative(Y.V) << endl;
```

Calling dco::derivative in line 10 is equivalent to setting $\mathbf{x}^{(1)}$ in (4) to the Cartesian basis vector corresponding to $\partial V/\partial S_0$. The option value is obtained in line 12 and the derivative value in line 13. Lines 15–16 seed $\mathbf{x}^{(1)}$ with the Cartesian basis vector for $\partial V/\partial r$. The primal code has to be run again before the derivative can be harvested in line 18.

4.2.2 First-order adjoint mode

First-order adjoint mode uses the same generic data types and primal function described above. 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).

```
1  #include "dco.hpp"
2  typedef dco::ga1s<double> DCO_MODE;
3  typedef DCO_MODE::type DCO_TYPE;
4  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
5  DCO_TAPE_TYPE* & DCO_TAPE_POINTER=DCO_MODE::global_tape;
6
7     ACTIVE_INPUTS<DCO_TYPE> X;
8     PASSIVE_INPUTS XP;
9     ACTIVE_OUTPUTS<DCO_TYPE> Y;
10    PASSIVE_OUTPUTS YP;
11
12    DCO_TAPE_POINTER = DCO_TAPE_TYPE::create();
13    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);
```

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. Note that the full gradient is obtained from one run of the adjoint code. Lastly memory allocated by the tape is released in line 28.

4.3 User-Defined Adjoints: Mind the Gap (in the Tape)

By default `dco/c++` in adjoint mode records intermediate calculations to the tape as the primal code is run forward, and then runs the calculation backwards when the tape is interpreted. Any additional information or structure a user wishes to exploit in an adjoint calculation necessarily has to interfere with this process: the user must stop `dco/c++` doing what it would normally do and instead has to show it how to do something more sophisticated. The result is *gaps* in `dco/c++`'s tape.

Conceptually, the following happens. Consider a section g in a primal code

$$F : (\mathbf{x}, \tilde{\mathbf{x}}) \xrightarrow{f_1} (\mathbf{u}, \tilde{\mathbf{u}}) \xrightarrow{g} (\mathbf{v}, \tilde{\mathbf{v}}) \xrightarrow{f_2} (y, \tilde{\mathbf{y}}) \,. \tag{18}$$

that a user wishes to treat specially (eg a Monte Carlo loop). The forward run starts by computing f_1 and recording values to the tape. When it reaches g the user must interrupt `dco/c++` and must compute g without storing values to the tape, thereby creating a gap. At the end of this section control is handed back to `dco/c++` which computes f_2, populating the tape as normal. This process is illustrated in Figure 1. Note that the way to "interrupt" `dco/c++` when computing g and avoid recording to the tape is to not use `dco/c++` data types. Therefore g must be computed with standard floating-point data types, denoted by $\mathbf{u}'$ and $\mathbf{v}'$ in Figure 1.

When the tape is interpreted, `dco/c++` starts by computing the adjoint of f_2. It then reaches the gap in the tape and calls a *user-defined adjoint function* which must compute the adjoint of g using the chain rule of differential calculus. Once this is done the user hands this adjoint back to `dco/c++` which uses this to compute the adjoint of f_1, thus completing the interpretation of the tape.

User-defined adjoint functions are registered with `dco/c++` through its *external function interface*. `dco/c++` provides a *factory* (see [Gamma et al., 1995]) which issues external adjoint objects (EAOs) that are associated with the tape, thus ensuring C++ exception safety and avoiding memory leaks. The active inputs $\mathbf{u}$ and active outputs $\mathbf{v}$ of the function $(\mathbf{v}, \tilde{\mathbf{v}}) = g(\mathbf{u}, \tilde{\mathbf{u}})$ above are registered with the EAO. Additionally, the passive inputs $\tilde{\mathbf{u}}$ and outputs $\tilde{\mathbf{v}}$ can also be stored in the EAO, along with any other data required for the computation of the gap's Jacobian $\nabla g \equiv \frac{\partial \mathbf{v}}{\partial \mathbf{u}}$.

The EAO is created and populated during the forward run by the user's implementation of the function g, called, eg, g_make_gap. The user-defined adjoint function which fills the gap, called, eg, g_fill_gap , needs to be registered with the EAO. When the tape is interpreted, `dco/c++` begins by computing the adjoint $\mathbf{v}_{(1)}$ of f_2. The interpreter then reaches the gap and calls g_fill_gap which must compute adjoints $\mathbf{u}_{(1)} := \mathbf{u}_{(1)} + (\nabla \mathbf{v})^T \cdot \mathbf{v}_{(1)}$ of the active inputs of g. These are then passed back to `dco/c++` which completes the interpretation of the tape to get $\mathbf{x}_{(1)} = \mathbf{x}_{(1)} + (\nabla F)^T \cdot y_{(1)}$.

The external function interface is the preferred option for incorporating user knowledge into a `dco/c++` adjoint AD solution. In Section 5 we discuss techniques which arise in the context of our test applications: exploiting independence in Monte Carlo simulations, and trading memory for floating-point operations through checkpointing. However, the external function interface facilitates the implementation of a variety of other targeted approaches to AAD with `dco/c++` some of which are briefly discussed in Section 6.

We have not yet addressed *how* the user fills the gap and computes the adjoint of g in g_fill_gap . While analytic results (see Section 6), hand-written adjoints or source transformation techniques are all options, it turns out that one can use `dco/c++` itself to fill the gap. This will be discussed in Section 5 below.

5 Ensembles and Evolutions

We show how `dco/c++`'s external function interface allows us to reduce the memory footprint of our test applications substantially. This is done by exploiting particular structural properties in each application.

We wish to note that the techniques discussed below are not optional refinements: in most production codes they are absolutely *essential* in order to get a tape-based adjoint code to run at all, even on large memory machines.

5.1 Ensembles: Exploiting Independence in Monte Carlo Simulations

We consider the first-order adjoint version of the Monte Carlo application in Section 3.1. Since the memory requirements of the tape scale more or less linearly with the number of sample paths,

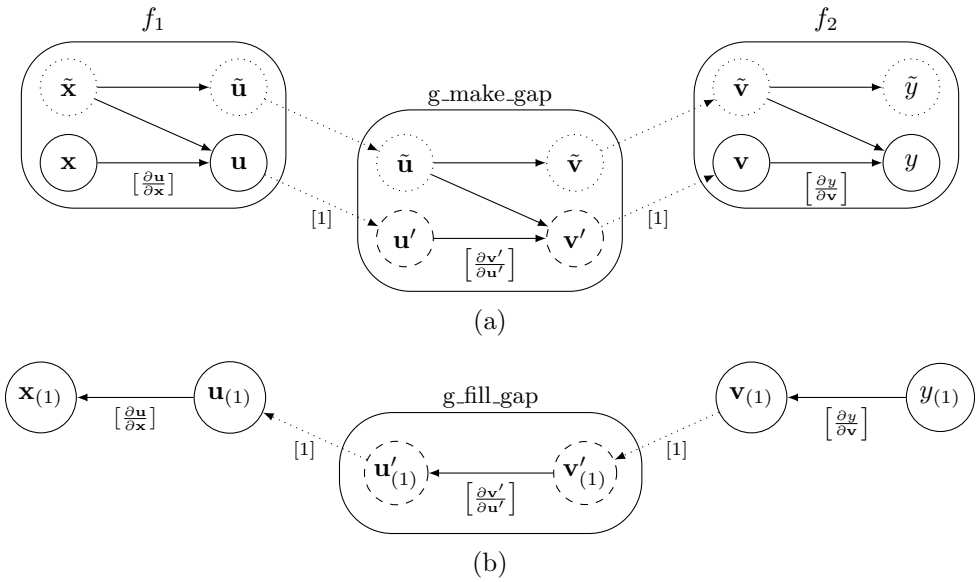

Figure 1: External Function Interface of `dco/c++`: Subfigure (a) illustrates the tape generation discussed in Section 4.3. The active part of the computation (marked by solid circles) is taped. Directed edges are labelled with partial derivatives of outputs w.r.t. inputs (unless zero). Dotted edges are assignments from the tape into the external function and back, yielding unit partial derivatives. Subfigure (b) illustrates the tape interpretation discussed in Section 4.3. Active computations are handled by `dco/c++` while g_fill_gap must fill the gap in the tape.

a naive (automatic) approach to generating adjoint code with `dco/c++` rapidly leads to infeasible peak memory requirements. The general concept of exploiting independence (parallelism) for generating efficient serial adjoints was discussed in [Hascoët et al., 2002].

A *checkpoint* is a set of data which is stored (either to disk or to memory) at a point during a computation, and which allows the computation to be restarted (at some later time) from that point. A Monte Carlo simulation is typically a function of the form

$$F \;:\; (\mathbf{x},\tilde{\mathbf{x}}) \xrightarrow{f_1} (\mathbf{u},\tilde{\mathbf{u}})\,,\; \begin{pmatrix} (\mathbf{u},\tilde{\mathbf{u}}_1) \xrightarrow{g} (\mathbf{v}_1,\tilde{\mathbf{v}}_1) \\ (\mathbf{u},\tilde{\mathbf{u}}_2) \xrightarrow{g} (\mathbf{v}_2,\tilde{\mathbf{v}}_2) \\ \vdots \\ (\mathbf{u},\tilde{\mathbf{u}}_N) \xrightarrow{g} (\mathbf{v}_N,\tilde{\mathbf{v}}_N) \end{pmatrix},\; (\mathbf{v}_1,\ldots,\mathbf{v}_N,\tilde{\mathbf{v}}_1,\ldots,\tilde{\mathbf{v}}_N) \xrightarrow{f_2} (y,\tilde{\mathbf{y}})\,. \tag{19}$$

In our context g is the Euler–Maruyama integrator (12) and $\tilde{\mathbf{u}}_i$ represents sample-path-specific passive inputs such as the random numbers (it includes $\tilde{\mathbf{u}}$ from f_1). The outputs $\mathbf{v}_i$ represent the N sample paths, and f_2 is the payoff function.

The explosion in tape size comes from the N evaluations of g. Therefore in the forward run, the evaluations of g must not record to the tape. The way to do this is to make a *checkpoint* of the output $(\mathbf{u},\tilde{\mathbf{u}})$ of f_1 and provide an external function g_make_gap 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. This process is illustrated in Figure 2 (a). Consider the ith call to the function g_make_gap. It requests an external adjoint object (EAO), registers the active input $\mathbf{u}$ with the EAO and receives it's value $\mathbf{u}'_i$ which is of type **double**[5]. The reason for

[5]The passive input $\tilde{\mathbf{u}}$ is also stored in the checkpoint; it is already of type **double**

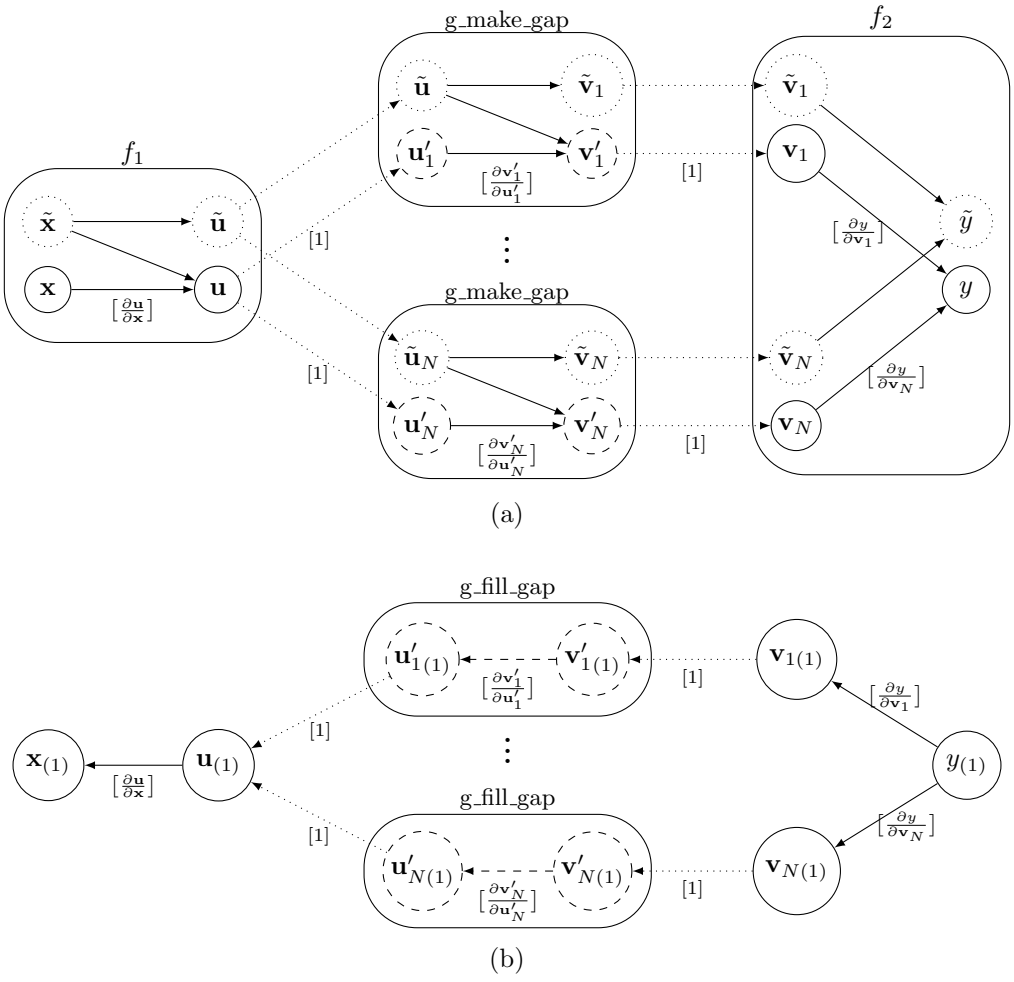

Figure 2: Illustration of the adjoint ensemble discussed in Section 5.1. Subfigure (a) depicts the forward (recording) run. The active part of the computation (marked by solid circles) is taped. Subfigure (b) denotes the adjoint calculation. Edges are labelled with partial derivatives (unless zero). Dotted edges are assignments from tape to external function and vice versa.

the subscript i will become clear when we discuss the reverse pass: for now note that all the $\mathbf{u}'_i$s are equal to the (passive) value of $\mathbf{u}$. Once g_make_gap has computed g it registers the outputs $(\mathbf{v}'_i, \tilde{\mathbf{v}}_i)$ with the EAO which inserts $\mathbf{v}'_i$ into new `dco/c++` adjoint types[6] $\mathbf{v}_i$ and registers them with the tape. Therefore the active outputs $\mathbf{v}'_i$ of g in (19) are of type **double**, while the active inputs $\mathbf{v}_i$ 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 don't.

Note that each call to g_make_gap creates *the same* checkpoint $(\mathbf{u}, \tilde{\mathbf{u}})$. Although not necessarily a problem this is somewhat inefficient. There are obvious ways to improve this, which are done in the code accompanying this paper.

Interpretation of the tape (illustrated in Figure 2 (b)) for a given input adjoint $y_{(1)}$ starts as normal. The tape interpreter computes all the adjoints

$$\mathbf{v}_{i(1)} := \mathbf{v}_{i(1)} + \frac{\partial y}{\partial \mathbf{v}_i}^T \cdot y_{(1)} = \frac{\partial y}{\partial \mathbf{v}_i}^T \cdot y_{(1)} \tag{20}$$

for $i = 1, \ldots, N$ where the equality follows since $\mathbf{v}_{i(1)}$ 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

$$\mathbf{u}_{(1)} := \mathbf{u}_{(1)} + \frac{\partial \mathbf{v}'_i}{\partial \mathbf{u}}^T \cdot \mathbf{v}'_{i(1)} = \mathbf{u}_{(1)} + \frac{\partial \mathbf{v}_i}{\partial \mathbf{u}}^T \cdot \mathbf{v}_{i(1)} \tag{21}$$

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

The method described above fills only one gap at a time. Mutual independence of the evaluations of g allows for several gaps to be filled in parallel, which yields a parallel adjoint calculation. It is also clear that the checkpointing scheme we present trades computation for memory: each sample path is effectively computed twice in order to complete the adjoint calculation.

Table 1 compares peak memory requirements and elapsed run times for the solutions in `mc/a1s` and `mc/a1s_ensemble` for $N = 10,000$ and $M = 360$ against central finite differences on our reference computer with 3GB of main memory available and swapping to disk disabled. The naive approach (`mc/a1s`) without checkpointing yields infeasible memory requirements for gradient sizes $n \geq 22$. Exploiting concurrency in `mc/a1s_ensemble` yields adjoints with machine precision at the expense $\mathcal{R} < 10$ primal function evaluations for all gradient sizes.

Table 2 shows the accuracy of gradient entries computed via finite differences (forward and central) compared with AD. The first entry ($i = 0$) has the best accuracy, the last entry ($i = 4$) has the worst, while the middle entry is representative of the average accuracy of the finite difference gradients. Figures are for the smallest problem (gradient size $n = 10$).

5.2 Evolutions: Multiple Checkpoints to Control Memory

An evolution is a serial loop, typically representing some kind of iterative method. In the context of the parabolic PDE considered in Section 3.2, the iteration arises as the payoff is evolved backwards in time by iteratively solving systems of linear equations. This can be depicted as

$$F : (\mathbf{x}, \tilde{\mathbf{x}}) \xrightarrow{f_1} (\mathbf{u}_0, \tilde{\mathbf{u}}_0) \xrightarrow{g} (\mathbf{u}_1, \tilde{\mathbf{u}}_1) \xrightarrow{g} \cdots \xrightarrow{g} (\mathbf{u}_N, \tilde{\mathbf{u}}_N) \xrightarrow{f_2} (y, \tilde{\mathbf{y}}) \tag{22}$$

[6] Only active outputs are converted to `dco/c++` types.

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

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 3.1. Naive first-order adjoints for $n \geq 22$ required too much memory to run. The relative computational cost $\mathcal{R}$ is given for `mc/a1s_ensemble`. Although theoretically constant, $\mathcal{R}$ is sensitive to specifics such as compiler flags, memory hierarchy and cache sizes, and level of optimization of the primal code.

i	mc/ffd	mc/cfd	mc/t1s	mc/a1s_ensemble
0	0.982097033091	0.982097084181	**0.9820970831594**85	**0.9820970831594**84
7	-0.0716705322265	-0.071666955947	**-0.07166605682464**82	**-0.07166605682464**84
4	0.34617424011**2**	0.34613132476**8**	**0.34612682023939**18	**0.34612682023939**24

Table 2: Accuracy of selected (ith) forward and central finite difference gradient entries vs. AD for the Monte Carlo code with scenario n=10. Top row shows best case (rel. err. $\approx$ 1e-8), bottom row worst case (rel. err. $\approx$ 6e-5) while middle row is a representative value (rel. err. $\approx$ 2e-5). These finite difference values are rather better than one often observes in practice.

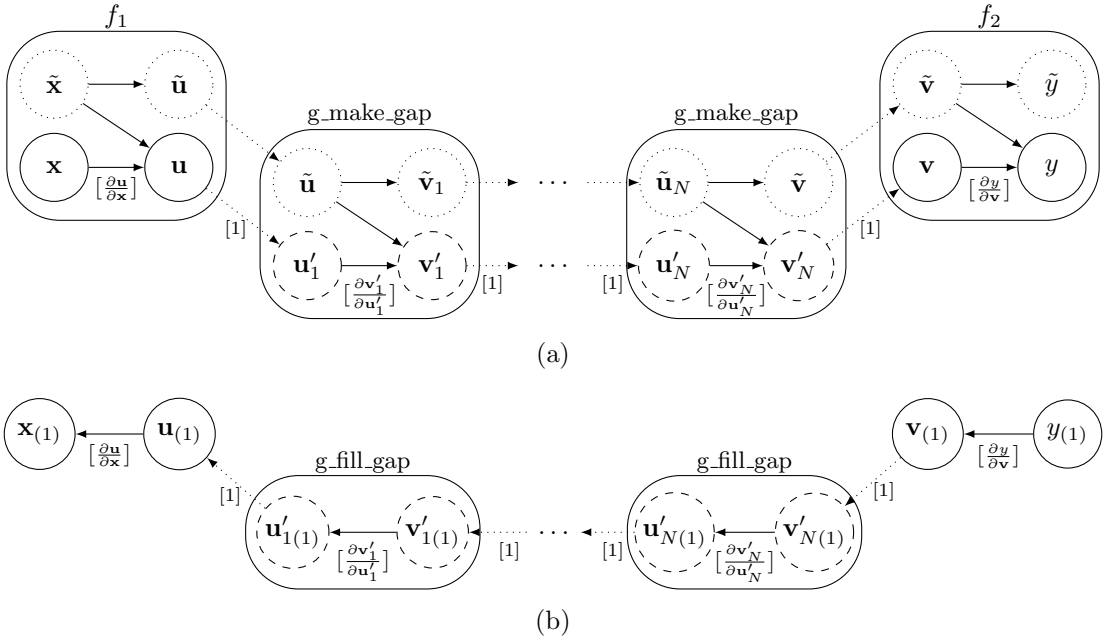

Figure 3: Illustration of the adjoint evolution discussed in Section 5.2. Line types (solid, dotted) have the same meanings as before. Subfigure (a) depicts the forward (recording) run and subfigure (b) the adjoint calculation. In general each function g_make_gap could be different, though in the PDE solver we discuss they are all the same.

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

Table 3: 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 $\mathcal{R}$ is given for `pde/a1s_checkpointing`. Again, this theoretically constant value is typically rather sensitive to specifics of the target computer architecture and software stack.

where each g represents the solution of a linear system[7]. As mentioned before, the tape's memory requirements scale more or less linearly with the number of time steps N. The only difference between an evolution and the ensemble discussed in the previous section is that in the ensemble each function g started with the same active input $\mathbf{u}$, whereas in (22) the active output of one iteration becomes the active input of the next iteration. We can therefore apply exactly the same ideas from the previous section and checkpoint the values $(\mathbf{u}_i, \tilde{\mathbf{u}}_i)$ at the start of each iteration for $i = 0, \ldots, N-1$. This is illustrated in Figure 3. Each function g is run *passively* (ie, not using `dco/c++` data types) so that there is a gap in the tape for each function g. When the tape is interpreted to compute the adjoint we do the same as before. First the adjoint $\mathbf{u}_{N(1)}$ of f_2 is computed. Next the checkpoint for $(\mathbf{u}_{N-1}, \tilde{\mathbf{u}}_{N-1})$ is restored and inserted into new `dco/c++` first-order adjoint types. The last (Nth) iteration is run actively, the tape is seeded with $\mathbf{u}_{N(1)}$ and the adjoint $\mathbf{u}_{(N-1)(1)}$ is computed. Then the checkpoint for $(\mathbf{u}_{N-2}, \tilde{\mathbf{u}}_{N-2})$ is restored and the same is done. This is repeated until the adjoint $\mathbf{u}_{0(1)}$ of the payoff is computed, after which the interpretation of the tape can be completed.

As noted in the previous section, each checkpoint introduces additional computations but saves memory. If one iteration doesn't use too much memory, then one can potentially checkpoint only every 10th or 20th iteration, thereby speeding up the adjoint calculation. Checkpointing techniques have been investigated extensively in the literature. The problem of minimizing the adjoint cost by placing checkpoints subject to a given upper bound on the available memory is known to be NP-complete [Naumann, 2008, Naumann, 2009]. Binomial checkpointing solves this problem for evolutions where the number of iterations is known in advance [Griewank, 1992]. Parallel approaches have been developed in order to exploit concurrency in the re-evaluation of parts of the computation from the stored checkpoints [Lehmann and Walther, 2002].

The sample code in `pde/a1s_checkpointing/` illustrates the use of the external function interface for implementing an equidistant checkpointing scheme for adjoint first-order AD. Table 3 compares peak memory requirements and elapsed run times for the solutions in `pde/a1s` and `pde/a1s_checkpointing` for $N = 10,000$ spatial grid points and $M = 360$ time steps on our reference computer with 3GB of main memory available and swapping to disk disabled. The naive (automatic) approach (`pde/a1s`) with no checkpointing yields infeasible memory requirements for all gradients of size $n \geq 10$. Equidistant checkpointing of every tenth time step in `pde/a1s_checkpointing` yields adjoints with machine precision at the expense of $\mathcal{R} \approx 15$ primal function evaluations. Implementing an optimal (binomial) checkpointing scheme, such as given by `revolve` [Griewank and Walther, 2000], can be expected to give further improvements.

The accuracy of finite differences compared with AD values is more or less the same as those

[7] In the general case each g in (22) could be a different function g_i, however for simplicity we have presented the case corresponding to Crank–Nicholson solver used in Section 3.2.

shown in Table 2.

6 Further Topics

We close with a brief discussion of some further topics which are applicable to AD in general or to `dco/c++` in particular. Their in-depth coverage in the context of AD solutions based on `dco/c++` will be the subject of upcoming publications.

6.1 Domain-Specific Adjoint AD Intrinsics

As we have seen, `dco/c++`'s external function interface allows one to define a mapping between a primal function and a user-defined adjoint function. This allows one to provide sets of highly optimized adjoint versions (featuring, eg, hand-written code or manually optimized output from an AAD source transformation tool) of functions that can be considered as the state of the art in a given field. For this to work well, the primal implementations of these functions would need to remain stable over the medium to long term so that the overheads of development, optimization and testing pay off due to frequent reuse. Entire libraries of user-defined `dco/c++` intrinsics are built in practice supporting a highly modular software engineering approach to adjoint sensitivity analysis and optimization.

6.2 Local Bumping and Closed Source Library Dependencies

Discontinuous and/or non-differentiable functions do occur in several financial settings. By definition, these functions cannot be differentiated in any classical sense of the term. Local bumping (finite differences) may be an option for obtaining some form of generalized / globalized sensitivity information for such functions. While this approach would no longer fit into the category of accurate differentiation, such generalized sensitivity information can still be integrated into an exact adjoint through user-defined adjoint functions as discussed above. It is up to the user to ensure that this "mix" makes some mathematical sense. An alternative approach to discontinuities/non-smoothness can be based on smoothing. The resulting continuously differentiable approximation of the primal model can then be differentiated with `dco/c++`.

Local bumping also allows a way to deal with closed source library dependencies. It is clear that for AD to work in general, the entire source code including any dependencies must be available. When parts of the source are closed, these functions can still be handled by local bumping. The NAG Library[8] is currently the only commercial maths library which is fully integrated with `dco/c++`. This means that `dco/c++` treats a growing set of functions in the NAG Library as *intrinsic* and can compute exact tangents and adjoints for them. Programs which use NAG components can therefore be handled as if the entire source code were available.

6.3 (Direct) Linear Solvers

The key to reducing the computational and memory requirements of direct linear solvers

$$A \cdot \mathbf{s} = \mathbf{b} \tag{23}$$

is to consider the linear system mathematically and then derive the analytic adjoint symbolically. Consider this system as a function of some auxiliary scalar input variable $z \in \mathbb{R}$ (generalization to the multivariate case turns out to be straight forward). Differentiating (23) gives $\frac{\partial A}{\partial z} \cdot \mathbf{s} + A \cdot \frac{\partial \mathbf{s}}{\partial z} = \frac{\partial \mathbf{b}}{\partial z}$. Denoting $\mathbf{s}^{(1)} = \frac{\partial \mathbf{s}}{\partial z}$ we obtain (see, for example, [Giles, 2008])

$$A \cdot \mathbf{s}^{(1)} = \mathbf{b}^{(1)} - A^{(1)} \cdot \mathbf{s} \tag{24}$$

[8] Available from www.nag.co.uk

where $A^{(1)} = \frac{\partial A}{\partial z}$ and $\mathbf{b}^{(1)} = \frac{\partial \mathbf{b}}{\partial z}$ are the (input) partial derivatives of A and $\mathbf{b}$ with respect to z. Therefore in tangent mode AD the derivative of the solution $\mathbf{s}$ w.r.t. z can be obtained by solving (24) where $A^{(1)}$ and $\mathbf{b}^{(1)}$ must already be available (obtained, eg, by application of `dco/c++` in first-order tangent mode to $A = A(z)$ and $\mathbf{b} = \mathbf{b}(z)$).

Similar arguments can be used [Giles, 2008, Naumann and Lotz, 2012] to show that the adjoint projections $A_{(1)}$ and $\mathbf{b}_{(1)}$ for a given input adjoint $\mathbf{s}_{(1)}$ in (23) satisfy

$$\begin{aligned} A^T \cdot \mathbf{b}_{(1)} &= \mathbf{s}_{(1)} \\ A_{(1)} &= -\mathbf{b}_{(1)} \cdot \mathbf{s}^T \end{aligned} \tag{25}$$

where the input adjoint must be available (eg, by application of `dco/c++` in first-order adjoint mode to the computation of $f = f(\mathbf{s})$ following the solution of the linear system in the primal code). There are two things to note from these results:

- There are analytic formulae (24) and (25) for the tangents $\mathbf{s}^{(1)}$ and adjoints $A_{(1)}$ and $\mathbf{b}_{(1)}$ of the linear system (23), so there is no need to use AD to compute these.
- The factorization of A computed by the primal solver can be reused in (24) and (25) for the solution of the equations.

The computational cost of the derivative calculation can thus be reduced significantly, eg, from $O(n^3)$ to $O(n^2)$ for a dense system, since the derivatives can be computed with matrix-vector products. For the adjoint calculation a similar statement applies to the memory requirements of the tape: if AD were used to compute the adjoint of (23), then $O(n^3)$ values would need to be stored to the tape. Now all that is needed is to store the factorization of A and the solution vector $\mathbf{s}$ which is $O(n^2)$ in size.

Note that the approach above assumes availability of the exact primal solution $\mathbf{s}$ of the linear system. This requirement is unlikely to be satisfied by iterative solutions since a higher accuracy of the primal solution (ie, $\mathbf{s}$) is needed in order to get the required accuracy of the sensitivities. For example, converging to 10 or 12 significant digits may be sufficient for a double precision adjoint calculation, although this would need to be verified in the given application.

The effect of the computational and memory savings in the Crank–Nicholson context of Section 3.2 is rather muted since the matrix A is tridiagonal and hence very sparse. We therefore chose not to include the implementation of the discussion above in the source code archive since it gains us nothing in our PDE kernel. However there are many applications in finance where large dense systems are handled, eg. in PIDEs, nearest correlation matrix calculations and interior point optimization methods to name but a few.

6.4 Root Finding

Differentiation of the parameterized system of nonlinear equations $F(\mathbf{x}, \boldsymbol{\lambda}) = 0$ at the solution $\mathbf{x} = \mathbf{x}^*$ with respect to the parameters $\boldsymbol{\lambda}$ yields

$$\frac{dF}{d\boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) = \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) + \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}} = 0$$

and hence

$$\frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}} = -\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^{-1} \cdot \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}). \tag{26}$$

The computation of the directional derivative

$$\mathbf{x}^{(1)} = \frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}} \cdot \boldsymbol{\lambda}^{(1)} = -\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^{-1} \cdot \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \boldsymbol{\lambda}^{(1)} \tag{27}$$

amounts to the solution of the linear system

$$\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \mathbf{x}^{(1)} = -\frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \boldsymbol{\lambda}^{(1)} \tag{28}$$

the right-hand side of which can be obtained by a single evaluation of the tangent mode of F. The direct solution of (28) requires the $n \times n$ Jacobian $\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})$ which is preferably computed in tangent mode so that sparsity can be exploited.

Transposing (26) gives $\left(\frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}}\right)^T = -\left(\frac{\partial F}{\partial \boldsymbol{\lambda}}\right)^T \cdot \left(\frac{\partial F}{\partial \mathbf{x}}\right)^{-T}$ and hence

$$\boldsymbol{\lambda}_{(1)} := \boldsymbol{\lambda}_{(1)} + \left(\frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}}\right)^T \cdot \mathbf{x}_{(1)} = \boldsymbol{\lambda}_{(1)} - \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^{-T} \cdot \mathbf{x}_{(1)}. \tag{29}$$

Consequently the adjoint solver needs to solve the linear system

$$\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \mathbf{z} = -\mathbf{x}_{(1)} \tag{30}$$

followed by a single call of the adjoint model of F seeded with the solution $\mathbf{z}$ which gives

$$\boldsymbol{\lambda}_{(1)} = \boldsymbol{\lambda}_{(1)} + \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \mathbf{z}.$$

The approach outlined above assumes availability of the exact primal solution of the nonlinear system. Again, a higher accuracy of the primal is necessary in order to get the desired accuracy of the adjoint.

6.5 Optimization

An unconstrained optimization problem can be regarded as root finding for the first-order optimality condition $\frac{\partial}{\partial \mathbf{x}} F(\mathbf{x}, \boldsymbol{\lambda}) = 0$. Differentiating at the solution $\mathbf{x} = \mathbf{x}^*$ with respect to $\boldsymbol{\lambda}$ gives

$$\frac{\partial}{\partial \boldsymbol{\lambda}} \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}) = \frac{\partial^2 F}{\partial \boldsymbol{\lambda} \partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}) + \frac{\partial^2 F}{\partial \mathbf{x}^2}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) = 0$$

so that we obtain

$$\frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) = -\frac{\partial^2 F}{\partial \mathbf{x}^2}(\mathbf{x}, \boldsymbol{\lambda})^{-1} \cdot \frac{\partial^2 F}{\partial \boldsymbol{\lambda} \partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}). \tag{31}$$

As before, computing the directional derivative $\mathbf{x}^{(1)} = \frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}} \cdot \boldsymbol{\lambda}^{(1)}$ amounts to solving a linear system

$$-\frac{\partial^2 F}{\partial \mathbf{x}^2}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \mathbf{x}^{(1)} = \frac{\partial^2 F}{\partial \boldsymbol{\lambda} \partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \boldsymbol{\lambda}^{(1)} \tag{32}$$

the right-hand side of which can be obtained by a single evaluation of the second-order adjoint version of F. The direct solution of (32) requires the $n \times n$ Hessian $\frac{\partial^2 f}{\partial \mathbf{x}^2}(\mathbf{x}, \boldsymbol{\lambda})$, which is preferably computed with second-order adjoint AD while exploiting potential sparsity.

Transposing (31) gives $\left(\frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}}\right)^T = -\left(\frac{\partial^2 F}{\partial \boldsymbol{\lambda} \partial \mathbf{x}}\right)^T \cdot \left(\frac{\partial^2 F}{\partial \mathbf{x}^2}\right)^{-T}$ yielding the first-order adjoint model

$$\boldsymbol{\lambda}_{(1)} := \boldsymbol{\lambda}_{(1)} + \frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \mathbf{x}_{(1)} = \boldsymbol{\lambda}_{(1)} - \frac{\partial^2 F}{\partial \boldsymbol{\lambda} \partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \frac{\partial^2 F}{\partial \mathbf{x}^2}(\mathbf{x}, \boldsymbol{\lambda})^{-T} \cdot \mathbf{x}_{(1)}.$$

Computing $\boldsymbol{\lambda}_{(1)}$ amounts to solving the linear system $\frac{\partial^2 F}{\partial \mathbf{x}^2}^T \cdot \mathbf{z} = -\mathbf{x}_{(1)}$ followed by a single call of the second-order adjoint model of F at the solution $\mathbf{z}$ to obtain

$$\boldsymbol{\lambda}_{(1)} := \boldsymbol{\lambda}_{(1)} + \frac{\partial^2 F}{\partial \boldsymbol{\lambda} \partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \mathbf{z}.$$

See [Henrard, 2014] for a discussion of this method in the context of calibration. Generalizations for constrained optimization problems can be derived naturally, eg, by treating the KKT[9] system according to Section 6.4.

[9] Karush–Kuhn–Tucker

n	`mc/socfd`	`mc/t2s_t1s`	`mc/t2s_a1s`	`mc/t2s_a1s_ensemble`
10	6s	4s	2s (316MB)	2s (0.5MB)
22	35s	23s	8s (778MB)	8s (1.0MB)
34	98s	66s	18s (1.2GB)	18s (1.5MB)
62	8min	5.3min	62s (2.3GB)	53s (2.5MB)
142	2.4hrs	52min	- (> 3GB)	5.3min (5.7MB)
222	6.1hrs	3.2hrs	- (> 3GB)	13.8min (8.7MB)

Table 4: Run time and peak memory requirements as a function of gradient size n for computing the entire Hessian of the Monte Carlo code. Second order central differences are compared with forward AD, naive adjoint-based AD and a checkpointed adjoint code. Only $N = 1000$ sample paths were used to keep finite difference run times manageable.

6.6 Pre-Accumulation of Local Jacobian

Consider the adjoint calculation of a function

$$F : (\mathbf{x}, \tilde{\mathbf{x}}) \xrightarrow{f_1} (\mathbf{u}, \tilde{\mathbf{u}}) \xrightarrow{g} (\mathbf{v}, \tilde{\mathbf{v}}) \xrightarrow{f_2} (y, \tilde{\mathbf{y}}) \tag{33}$$

where $\mathbf{u} \in I\!R^n$ and $\mathbf{v} \in I\!R^m$ so that the Jacobian $\nabla g \in I\!R^{m \times n}$. The maximum $m \times n$ memory needed to store the matrix ∇g is often much less than the amount of memory needed to *tape* g, especially if g involves a lot of computation. The problem is compounded if a large amount of memory is also needed to tape f_2, leading to the overall tape size potentially exceeding the available memory. In this case memory can be freed if the Jacobian for g is *pre-accumulated* locally during the forward run (using, eg, `dco/c++` in adjoint mode, or in tangent mode if $n \leq \mathcal{R} \cdot m$ for $\mathcal{R}$ the adjoint cost) and stored, effectively creating a gap in the tape for g. When the tape is interpreted, the gap is filled by computing $\mathbf{u}_{(1)} := \mathbf{u}_{(1)} + (\nabla g)^T \cdot \mathbf{v}_{(1)}$ using the stored matrix.

Consider now the case when the active output of F is a vector $\mathbf{y} \in I\!R^p$. The Jacobian ∇F is now a matrix and the adjoint model of F is $\mathbf{x}_{(1)} := \mathbf{x}_{(1)} + (\nabla F)^T \cdot \mathbf{y}_{(1)}$. To compute the entire Jacobian we seed $\mathbf{y}_{(1)}$ with the Cartesian basis vectors in $I\!R^p$, which means running the tape interpreter p times, once for each column in the Jacobian. Having the pre-accumulated Jacobian ∇g available avoids having to essentially re-compute it each time the tape interpreter is run, reducing the computational cost of computing the Jacobian.

7 Conclusion

Adjoint AD is an extremely powerful numerical technique. In order to use it in production C++ codes, flexible tools are needed which allow a user sufficient control over the adjoint computation so that the memory and computational costs can be kept in check. `dco/c++` was designed specifically to allow such control, and this paper (and the accompanying code) presents techniques for controlling these costs within the context of computational finance. The tools and techniques presented here are applicable across financial mathematics: sensitivities for xVA calculations, portfolio optimization, calibration, hedge portfolios and more can all be obtained while keeping memory use within limits.

As mentioned in Section 2 higher derivatives pose no conceptual problem. Recursive instantiation with `dco/c++` types yields derivatives of arbitrary orders, memory problems still feature for methods built on adjoints, and the techniques presented in this paper can be used to control these. For completeness Table 4 presents run times and memory requirements for computing the entire Hessian for the Monte Carlo code from Section 3.1.

`dco/c++` can also be combined with accelerators such as GPUs[10]: see e.g. [Du Toit et al., 2014] and [Gremse et al., 2014]. `dco/c++` in tangent mode has full support for CUDA and can be used

[10] Although there is no published work on using `dco/c++` with Intel Xeon Phi, `dco/c++` supports Intel's LEO and

"out the box." While `dco/c++` in adjoint mode does work on these devices, using it effectively is more challenging since memory is constrained and memory accesses have to be orchestrated carefully. In addition race conditions may arise when updating adjoints of shared variables. Tapes on host and device are in separate memory spaces, and each thread on the device requires its own tape which often requires more memory than is available. For these reasons, `dco/c++` in adjoint mode is typically not used on the device itself but only on the host portion of the code.

On the device itself there are two main options. The first, as in [Gremse et al., 2014], is to narrow the focus to a single problem domain, the setting considered there being medical image reconstruction. Algorithms in this field can frequently be broken down into sequences of linear algebra operations. These can be differentiated analytically (as in Section 6.3 above) and the resulting GPU kernels can be incorporated as `dco/c++` intrinsics (as in Section 6.1).

Although this leads to device code which is memory bandwidth bound (since kernels are small and are often matrix-vector operations), this approach works well in the problem domain considered. It is rather restrictive, however, and does not translate well to computational finance where many numerical kernels cannot be decomposed easily into sequences of linear algebra operations. Efficient code can be obtained by writing accelerated adjoint kernels by hand as in [Du Toit et al., 2014], however this is hardly ideal for primal codes which may change frequently. A source transformation tool (eg, [Hascoët and Pascual, 2013]) could be used provided that the target code can be handled. The other option involves extending `dco/c++`'s type system using C++11 features so that adjoints for blocks of *straight-line code*[11] can be generated at compile time by the platform C++11 compiler. No writes to the tape are therefore generated for that block of code. This is a topic of ongoing research and should ease the process of generating efficient adjoint kernels for accelerators.

8 Acknowledgements

The authors would like to thank a senior quantitative analyst Claudia Yastremiz for many suggestions and helpful discussions which ultimately led to this paper, and to a fuller understanding of the challenges any AD tool faces when operating on a large bank's quant library.

References

[Andersen and Brotherton-Ratcliffe, 2000] Andersen, L. B. G. and Brotherton-Ratcliffe, R. (2000). The equity option volatility smile: an implicit finite difference approach. *Journal of Computational Finance*.

[Bischof et al., 2008] Bischof, C., Bücker, M., Hovland, P., Naumann, U., and Utke, J., editors (2008). *Advances in Automatic Differentiation*, volume 64 of *Lecture Notes in Computational Science and Engineering*. Springer.

[Borsdorf and Higham, 2010] Borsdorf, R. and Higham, N. J. (2010). A preconditioned (Newton) algorithm for the nearest correlation matrix. *IMA Journal of Numerical Analysis 30(1)*.

[Bücker et al., 2005] Bücker, M., Corliss, G., Hovland, P., Naumann, U., and Norris, B., editors (2005). *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*. Springer.

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

can be used on the Xeon Phi in both tangent and adjoint mode. The general discussion about difficulties with adjoint calculations applies to this platform as well.

[11] Code with no branches or function calls.

[Cont and Tankov, 2004] Cont, R. and Tankov, P. (2004). *Financial Modelling with Jump Processes*. Chapman & Hall.

[Du Toit et al., 2014] Du Toit, J., Lotz, J., and Naumann, U. (2014). Algorithmic differentiation of a gpu accelerated application. *NAG Technical Report No. TR2/14*.

[Forth et al., 2012] Forth, S., Hovland, P., Phipps, E., Utke, J., and Walther, A., editors (2012). *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*. Springer.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns*. Addison-Wesley.

[Gebremedhin et al., 2005] Gebremedhin, A., F., M., and Pothen, A. (2005). What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705.

[Giles, 2008] Giles, M. (2008). Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In *[Bischof et al., 2008]*, pages 35–44. Springer.

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

[Gremse et al., 2014] Gremse, F., Theek, B., Kunjachan, S., Lederle, W., Pardo, A., Barth, S., Lammers, T., Naumann, U., and Kiessling, F. (2014). Absorption reconstruction improves biodistribution assessment of fluorescent nanoprobes using hybrid fluorescence-mediated tomography. *Theranostics 4, 960-971.*

[Griewank, 1992] Griewank, A. (1992). Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54.

[Griewank, 2013] Griewank, A. (2013). On stable piecewise linearization and generalized algorithmic differentiation. *Optimization Methods and Software*, 28(6):1139–1178.

[Griewank and Walter, 2008] Griewank, A. and Walter, A. (2008). *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation (2. Edition)*. SIAM, Philadelphia.

[Griewank and Walther, 2000] Griewank, A. and Walther, A. (2000). Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45. Also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.

[Hascoët et al., 2002] Hascoët, L., Fidanova, S., and Held, C. (2002). Adjoining independent computations. In Corliss, G., Faure, C., Griewank, A., Hascoët, L., and Naumann, U., editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 35, pages 299–304. Springer, New York, NY.

[Hascoët and Pascual, 2013] Hascoët, L. and Pascual, V. (2013). The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software*, 39(3):20:1–20:43.

[Henrard, 2014] Henrard, M. (2014). Adjoint algorithmic differentiation: calibration and implicit function theorem. *Journal of Computational Finance*. To appear.

[Lehmann and Walther, 2002] Lehmann, U. and Walther, A. (2002). The implementation and testing of time-minimal and resource-optimal parallel reversal schedules. In Sloot, P. M. A., Tan, C. J. K., Dongarra, J. J., and Hoekstra, A. G., editors, *Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, volume 2330 of *Lecture Notes in Computer Science*, pages 1049–1058, Berlin. Springer.

[Matache et al., 2004] Matache, A. M., von Petersdorff, T., and Schwab, C. (2004). Fast deterministic pricing of options on lévy-driven assets. *M2AN. Mathematical Modelling and Numerical Analysis, 38*.

[Naumann, 2008] Naumann, U. (2008). Call tree reversal is NP-complete. In *[Bischof et al., 2008]*, pages 13–22. Springer.

[Naumann, 2009] Naumann, U. (2009). DAG reversal is NP-complete. *Journal of Discrete Algorithms*, 7:402–410.

[Naumann, 2012] Naumann, U. (2012). *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation*. Number 24 in Software, Environments, and Tools. SIAM.

[Naumann and Lotz, 2012] Naumann, U. and Lotz, J. (2012). Algorithmic differentiation of numerical methods: Tangent-linear and adjoint direct solvers for systems of linear equations. Technical Report AIB-2012-10, LuFG Inf. 12, RWTH Aachen.

[Qi and Sun, 2006] Qi, H. and Sun, D. (2006). A quadratically convergent Newton method for computing the nearest correlation matrix. *SIAM Journal of Matrix Analysis and Applications 29(2)*.

[Schanen et al., 2012] Schanen, M., Foerster, M., Lotz, J., Leppkes, K., and Naumann, U. (2012). Adjoining hybrid parallel code. In et al., B. T., editor, *Proceedings of the Fifth International Conference on Engineering Computational Technology*, pages 1–19. Civil-Comp Press.

[Schmitz et al., 2011] Schmitz, M., Hannemann-Tamas, R., Gendler, B., Förster, M., Marquardt, W., and Naumann, U. (2011). Software for higher-order sensitivity analysis of parametric DAEs. *SNE*, 22(3-4):163–168.

[Voßbeck et al., 2008] Voßbeck, M., Giering, R., and Kaminski, T. (2008). Development and first applications of TAC++. In *[Bischof et al., 2008]*, pages 187–197. Springer.

[W. Xu, 2014] W. Xu, X. Chen, T. C. (2014). The efficient application of automatic differentiation for computing gradients in financial applications. *Journal of Computational Finance*. To appear.