`dco/c++`: Derivative Code by Overloading in C++

Introduction and Summary of Features

Uwe Naumann
Klaus Leppkes
Johannes Lotz

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2016-08
RWTH Aachen University | Department of Computer Science | July 2016

`dco/c++`: Derivative Code by Overloading in C++ Introduction and Summary of Features

Uwe Naumann
Klaus Leppkes
Johannes Lotz

Lehr- und Forschungsgebiet Informatik 12:
Software and Tools for Computational Engineering
RWTH Aachen University, Aachen, Germany
info@stce.rwth-aachen.de

Zusammenfassung Boosted by advanced type genericity and support for template metaprogramming techniques the role of C++ as the preferred language for large-scale numerical simulation in Computational Science, Engineering and Finance has been strengthened over recent years. Algorithmic Differentiation of numerical simulations and algorithmic adjoint methods in particular have seen substantial growth in interest due to increased requirement for gradient-based techniques in high dimensions in the context of parameter sensitivity analysis and calibration, uncertainty quantification, and nonlinear optimization. Modern software tools for (adjoint) Algorithmic Differentiation in C++ make heavy use of modern C++ features aiming for increased computational efficiency and decreased memory requirement. The `dco/c++` tool presented in this report aims to take Algorithmic Differentiation in C++ one step further by focussing on derivatives of arbitrary order, support for shared-memory parallelism and powerful and intuitive user interfaces in addition to competitive computational performance. Its algorithmic and software quality has made `dco/c++` the tool of choice in many industrial and academic projects. A proper introduction of the tool has been missing so far. This report aims to fill this gap. Its focus is on giving an overview of the functionality offered by `dco/c++` and a more detailed discussion of some features. A feature summary is given in the appendix.

1 Yet Another AD Tool?

Driven by the growing complexity of nonlinear optimization [52], data analysis/machine learning [2] and general inverse problems [42] in Computational Science, Engineering and Finance Algorithmic Differentiation (AD) [18,34] and its adjoint mode in particular has seen a substantial increase in popularity over recent years. Software tools for AD can be classified into source code transformation (e.g, OpenAD [48], TAF/TAC++ [51], Tapenade [23]) and overloading (e.g, Adept [24], ADOL-C [16], CppAD [3]) tools. The former typically result in better performing derivative code while the latter turn out to be easier to develop and maintain in the context of the ongoing evolution of programming languages and computer architectures. Hybrid approaches have been investigated including the NAG AD Compiler for `Fortran` [38] which combines elements of source transformation at the level of the compiler's internal representation with overloading techniques provided by a run time support library. Approaches that are independent of the programming language have also been explored including AD transformations on universal intermediate formats such as XAIF [25] used by OpenAD and AD applied to assembly code [11]. A large collection of AD software tools can be found on the AD community's web portal `www.autodiff.org` in addition to an extensive bibliography and links to related workshops/conferences.

As of today there is no source transformation tool offering full support for the C++ programming language. This gap is mostly due to the outstanding complexity and (type) genericity of C++. It is filled by overloading solutions. The `dco/c++` AD tool falls into the same category.

Most modern AD tools for C++ make extensive use of template metaprogramming techniques in order to defer certain semantic transformations to compile time (e.g, the preaccumulation of local gradients of right-hand sides of scalar assignments by means of expression templates [41,24,**?**]) yielding a hybrid source transformation/overloading approach. The gap in terms of performance between pure source transformation and this hybrid method for C++ is decreasing. Further progress in the development of the C++ programming language and corresponding compiler technology can be expected to enforce this tendency. Hence we consider a combination of overloading and template metaprogramming supported by multithreading and powerful application programming interfaces (API) as the method of choice for AD tool support in C++.

`dco/c++` has been applied successfully to a number of relevant numerical simulation codes including applications from computational fluid dynamics [45,46], chemical engineering [19,20], athmospheric science [47,31], and computational finance[1] [36]. A proper introduction of the tool has been missing so far. This report aims to fill this gap. Its focus is on giving an overview of the functionality offered by `dco/c++` in Sec. 2 and a more detailed discussion of selected unique features that turned out extremely useful in actual applications in Sec. 3.

The development of `dco/c++` has been driven by requirements due to the previously outlined target codes. Financial applications in particular pose a number of challenges shaping the set of functionalities and the design of the user interface. Most features provided by modern C++ can be dealt with. Experience with other AD overloading tools suggests that this is not generally the case. Indeed we are not aware of any modern C++ construct that `dco/c++` cannot handle. However as of today we cannot claim full coverage of all features by our regression test suite either.

A distinguishing feature of `dco/c++` is the transparency of its internal representation (also referred to as the *tape*) for computation of adjoints. Through a well-designed API users have the opportunity to modify arbitrary details (data dependencies, local partial derivatives, modes of differentiation). This level of flexibility has proven crucial for the design of robust and efficient adjoint solutions in a real-world setting. It facilitates building up libraries of user-defined/domain-specific intrinsics, inclusion of manually derived adjoint code and of approximations of derivatives of black boxes, the handling of non-differentiability/discontinuity through smoothing, and integration of advanced preaccumulation and checkpointing techniques. References on the use of individual features can be found in most of the previously cited articles on applications of `dco/c++`.

2 Basics

AD tools in general and `dco/c++` in particular target implementations of multivariate vector functions

$$\mathbf{y} = F(\mathbf{x}),\ \mathbf{x} \in \mathbb{R}^n,\ \mathbf{y} \in \mathbb{R}^m \tag{1}$$

as computer programs. The mathematical formulation of F in Eqn (1) does not account for aliasing and overwriting of program variables. A more realistic formulation of the targeted

[1] `dco/c++` is used by several tier-one investment banks under strict non-disclosure agreements.

numerical simulation programs is

$$\begin{pmatrix} \mathbf{z} \\ \tilde{\mathbf{z}} \\ \mathbf{y} \\ \tilde{\mathbf{y}} \end{pmatrix} := F(\mathbf{x}, \tilde{\mathbf{x}}, \mathbf{z}, \tilde{\mathbf{z}}) \ ,$$

where $\mathbf{x} \in \mathbb{R}^{d_\mathbf{x}}$, $\tilde{\mathbf{x}} \in \mathbb{R}^{d_{\tilde{\mathbf{x}}}}$, $\mathbf{z} \in \mathbb{R}^{d_\mathbf{z}}$, $\tilde{\mathbf{z}} \in \mathbb{R}^{d_{\tilde{\mathbf{z}}}}$, $\mathbf{y} \in \mathbb{R}^{d_\mathbf{y}}$, $\tilde{\mathbf{y}} \in \mathbb{R}^{d_{\tilde{\mathbf{y}}}}$, and $n = d_\mathbf{x} + d_\mathbf{z}$, $m = d_\mathbf{z} + d_\mathbf{y}$. Equality ($=$) is replaced by assignment ($:=$) in the sense of an imperative programming language such as C++. In addition to pure inputs $(\mathbf{x}, \tilde{\mathbf{x}})$ and pure outputs $(\mathbf{y}, \tilde{\mathbf{y}})$ there are program variables serving as both $(\mathbf{z}, \tilde{\mathbf{z}})$ with their input values potentially overwritten by the given implementation of F. We distinguish between *active* $(\mathbf{x}, \mathbf{z}, \mathbf{y})$ and *passive* $(\tilde{\mathbf{x}}, \tilde{\mathbf{z}}, \tilde{\mathbf{y}})$ variables. Standard AD terminology refers to variables with structurally non-zero derivatives as active. Variables are referred to as passive otherwise. See [22] for details from a static program analysis perspective.

Obviously, signatures of real-world numerical simulations in C++ can become arbitrarily complicated involving, for example, pointers, references to instances of complex class hierarchies, and type-generic arguments. However, most conceptual challenges in AD (and in adjoint AD in particular) can be attributed to aliasing and overwriting. In the following we write $\mathbf{y} := F(\mathbf{x})$ when referring to given implementations of Eqn (1). We call them primal numerical simulation programs or simply primals. Parts of $\mathbf{x}$ and $\mathbf{y}$ can be aliased (yielding $\mathbf{z}$) unless stated otherwise. The numerical programs under consideration are assumed to be k times continuously differentiable whenever derivative models of up to order k are discussed. Note that mere differentiability of the underlying function F does not imply differentiability of the given implementation. For example, the primal

```
float F(float x) { if (x==0) return 0; return x; }
```

implements the continuously differentiable function $y = F(x) = x$ with unit derivative everywhere. An algorithmically differentiated version yields a vanishing derivative at $x = 0$.

We use notation from [34] for the description of first and higher derivative models. Let $J = \frac{dF}{d\mathbf{x}}(\mathbf{x}')$ for a given $\mathbf{x}' \in \mathbb{R}^n$. A matrix-free projection of $J \in \mathbb{R}^{m \times n}$ in direction $\mathbf{v} \in \mathbb{R}^n$ is denoted as $\langle J, \mathbf{v} \rangle \equiv J \cdot \mathbf{v}$. Such directional derivatives (tangents for short) are implemented by first-order tangent versions of $\mathbf{y} := F(\mathbf{x})$. The latter can be generated by tangent (also: forward) mode AD.

A matrix-free projection of J in direction $\mathbf{u} \in \mathbb{R}^m$ is denoted as $\langle \mathbf{u}, J \rangle \equiv J^T \cdot \mathbf{u}$. Such adjoint derivatives (adjoints for short) are implemented by first-order adjoint versions of $\mathbf{y} := F(\mathbf{x})$, which can be generated by adjoint (also: reverse) mode AD.

The projection notation generalizes naturally to second- and higher-order tangents and adjoints of sufficiently often continuously differentiable numerical simulations. Let therefore $H = \frac{d^2F}{d\mathbf{x}^2}(\mathbf{x}')$ for a given $\mathbf{x}' \in \mathbb{R}^n$. The 3-tensor $H \in \mathbb{R}^{m \times n \times n}$ is an m-vector of symmetric $(n \times n)$-matrices. Its first-order matrix-free projection in direction $\mathbf{u} \in \mathbb{R}^m$ is denoted as $\langle \mathbf{u}, H \rangle$ and yields a symmetric $(n \times n)$-matrix as a linear combination of the m vector elements. Multiplication of the latter with a vector $\mathbf{v} \in \mathbb{R}^n$ yields a second-order matrix-free projection of H in directions $\mathbf{u} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^n$ as $\langle \mathbf{u}, H, \mathbf{v} \rangle = \langle \langle \mathbf{u}, H \rangle, \mathbf{v} \rangle$.

Alternatively, the 3-tensor H can be regarded as an n-vector of $(m \times n)$-matrix. Consequently, its first-order matrix-free projection in direction $\mathbf{v} \in \mathbb{R}^n$ is denoted as $\langle H, \mathbf{v} \rangle$ and yields an $(m \times n)$-matrix as a linear combination of the n vector elements. Multiplication of the latter with a vector $\mathbf{w} \in \mathbb{R}^n$ yields a second-order matrix-free projection of H in directions $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ as $\langle H, \mathbf{v}, \mathbf{w} \rangle = \langle \langle H, \mathbf{v} \rangle, \mathbf{w} \rangle$. Symmetry within H implies $\langle H, \mathbf{v}, \mathbf{w} \rangle = \langle H, \mathbf{w}, \mathbf{v} \rangle$ (commutativity of tangent projection) $\langle \langle \mathbf{u}, H \rangle, \mathbf{v} \rangle = \langle \mathbf{u}, \langle H, \mathbf{v} \rangle \rangle$ (associativity of tangent and adjoint projections) $\langle \langle \mathbf{u}, H \rangle, \mathbf{v} \rangle = \langle \mathbf{v}, \langle \mathbf{u}, H \rangle \rangle$ (equivalence of second-order tangent and adjoint projections).

Matrix-free projections of third and higher derivative tensors follow naturally. `dco/c++` supports tangents and adjoints of first and higher order through recursive template instantiation. Arbitrary combinations of matrix-free tangent and adjoint projections can be computed, which makes `dco/c++` well suited for illustration of the mathematical concepts behind AD in a classroom environment.

2.1 Sample Numerical Programs

Our choice of sample applications is driven by three requirements.

1. Examples should be representative for real-world applications featuring practically relevant code and data flow patterns exhibited by many numerical simulations.
2. The implementation should be simple enough to make a detailed discussion of the source code feasible.
3. The simulations should be scalable in terms of computational cost to allow for run time comparison of various scenarios.

Development and maintenance of `dco/c++` as well as of further AD software solutions (see Sec. 4) is driven by the Numerical Algorithms Group Ltd. (NAG)[2], Oxford, UK in collaboration with the Software and Tools for Computational Engineering (STCE) group[3] at RWTH Aachen University, Aachen, Germany. A user guide with details on the full range of functionalities of `dco/c++` can be found on

nag.co.uk/content/nag-and-algorithmic-differentiation .

The numerous example programs referred to in the following can be found on the `dco/c++` *research website*

www.stce.rwth-aachen.de/research/software/dco/cpp.

Reproduction of the numerical results requires a (trial) license for `dco/c++`. Licensing and distribution is organised by NAG.

Burgers Equation We consider the numerical solution of the 1D Burgers Equation [7]

$$\frac{dy}{dt} = v\frac{d^2y}{dx^2} - y\frac{dy}{dx} \tag{2}$$

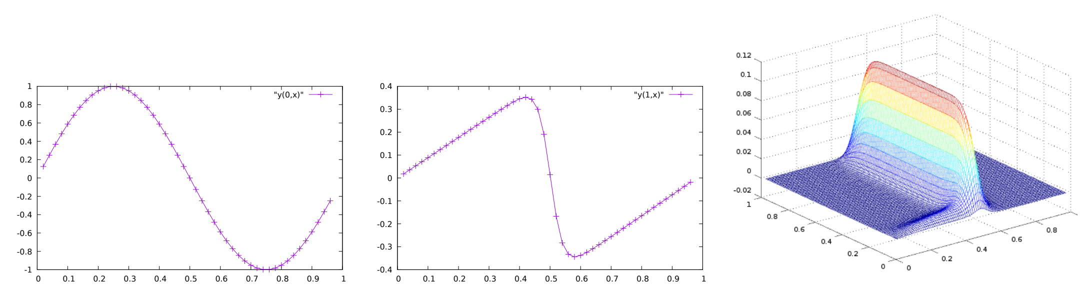
over the unit square defined by $t \in [0, 1]$ and $x = [0, 1]$. For a given initial condition on the state $y = y(t, x)$, for example, $y(0, x) = \sin(2\pi x)$, and fixed vanishing boundary conditions we use central finite differences combined with upwinding in space and backward finite differences in time yielding an implicit Euler scheme implemented as

[2] nag.co.uk
[3] www.stce.rwth-aachen.de

Listing 1.1. Type-generic primal Burgers code

```
1 // ... global passive read-only data
2
3 template <typename T>
4 void burgers(vector<T>& y) {
5   for (int j=0;j<m;j++) {
6     vector<T> yp=y;
7     newton(yp,y);
8   }
9 }
```

with type-generic discrete state vector y[4] of size n holding the discrete initial condition $\mathbf{y}^0 \in \mathbb{R}^n$ (see Fig. 1(a)) as input and the approximate discrete solution $\mathbf{y}^m \in \mathbb{R}^n$ (see Fig. 1(b)) as output for given viscosity `v` and `m` time steps performed. Individual Euler steps are computed as solutions of nonlinear systems using Newton's method in L1:7;[5] see Sec. 3.1 for further details. Experiments are run for the shock-free scenario `v=0.01`. All passive read-only data (`n`, `m`, `v`) is declared globally yielding easier to follow source code listings due to simplifed signatures of the routines called. Refer to [39] and to references therein for a more detailed discussion of the Burgers equation.



(a) (b) (c)

Abbildung 1. Burgers equation: Initial condition $\mathbf{y}^0 \in \mathbb{R}^n$ (a), solution $\mathbf{y}^m \in \mathbb{R}^n$ (b) and Jacobian of the solution with respect to the initial condition within the interior domain.

To illustrate the features offered by `dco/c++` we consider derivatives of the approximate discrete solution with respect to the discrete initial condition, including projections of the Jacobian matrix

$$J = \frac{d\mathbf{y}^m}{d\mathbf{y}^0} \in \mathbb{R}^{n\times n}$$

and of the Hessian tensor

$$H = \frac{d^2\mathbf{y}^m}{d{\mathbf{y}^0}^2} \in \mathbb{R}^{n\times n\times n} \ .$$

For example, Fig. 1 (c) shows a 3D-plot of the whole Jacobian. Finite difference (FD) approximation of derivatives is used as a method for potential validation of derivatives returned by

[4] Depending on the context we use the mathematical, e.g, $\mathbf{y}$, and source code listing notations, e.g, `y` interchangeably.

[5] We reference line i in Listing k by Lk:i. Sequences of lines i to j in Listing k are denoted by Lk:i–j. Sets of nonconsecutive lines $i_1, ..., i_j$ in Listing k are denoted by Lk:$i_1, \ldots, i_j$.

dco/c++ whenever possible. Closely matching values typically indicate success. The negation of this statement does not apply in general due to the well-known numerical instability of FD in finite precision floating point arithmetic. We observe good correspondence between first derivatives obtained by double precision AD and FD for all our test problems. Higher precision arithmetic is used[6] to obtain reliable FD approximations of higher derivatives.

When comparing run times of the various derivative codes we typically report values for their computational cost (run time) relative to an optimized implementation of the primal. For example,

$$\mathcal{R} = \frac{\mathrm{Cost}(F_{(1)})}{\mathrm{Cost}(F)} \tag{3}$$

denotes the relative run time of an adjoint version $F_{(1)}$ of a primal F. All tests are performed on an Intel Xeon E5-2630 with 128GB of main memory and running Linux. For $n = 10^3$ and $m = 10^4$ the run time of the primal Burgers code is $1.0s$.

Reported results of run time measurements should be considered as qualitative. Experience shows that they can be sensitive to the choice of computation environment including hardware specifications, system software and compiler version. Qualitative statements can be expected to remain valid.

LIBOR Market Model As a second case study we consider the LIBOR[7] market model introduced in [6] and used in [12] as illustration of the benefits of adjoint AD for simulations in finance. Over recent years adjoint AD has gained significant importance in computational finance driven mainly by increasing gradient sizes in the context of XVA calculations and documented by a large number of related publications, e.g, [40,32].

The LIBOR sample code simulates the payoff $P \in \mathbb{R}$ of a portfolio of swaptions with given swap rates und maturities. Swaps of the floating forward rate $L \in \mathbb{R}^n$ and a given fixed swap rate are considered. The primal shown in Listing 1.2 is generic in the data type of the active in- (L) and outputs (P) (L2:3–4). Again, all passive read-only data (n,m,p, LIBOR interval, volatility, maturities, swap rates) is declared globally. We use dco/c++ to compute the gradient of the payoff with respect to the initial LIBOR rates.

Listing 1.2. Type-generic primal LIBOR code

```
1  // ... global passive read-only data
2
3  template<typename T>
4  void libor(const vector<T>& L, T& P,
5             const vector<vector<double>>& Z) {
6    T Ps=0;
7    for (int j=0;j<p;j++) {
8      vector<T> Lc(L);
9      path_calc(j,Lc,Z);
10     portfolio(Lc,P);
11     Ps+=P;
12   }
13   P=Ps/p;
```

[6] We use ARPREC [1] developed at Lawrence Berkeley National Laboratory.

[7] London Interbank Offered Rate

```
14 }
```

Monte Carlo simulation with a normally distributed random variable $Z \in \mathbb{R}^{p \times m}$ performs p path calculations evolving L for m time steps. Path simulations (L2:9) are performed on local copies (L2:8) of the initial libor rates. The payoff of each scenario is evaluated (L2:10) followed by summing up individual payoffs (L2:11) for subsequent averaging over all paths (L2:13).

Refer to [13] for further discussion of the mathematical details behind the LIBOR market model. All numerical results obtained by our implementation were validated against the implementation used in [12] and available from Giles' website[8] at the University of Oxford, UK. The run time of $p = 10^4$ primal Monte Carlo path simulations is $1.9s$.

2.2 First-Order Tangents

In tangent mode directional derivatives are computed alongside with function values as

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(1)} \end{pmatrix} := F^{(1)}\left(\mathbf{x}, \mathbf{x}^{(1)}\right) \equiv \begin{pmatrix} F(\mathbf{x}) \\ \left\langle \frac{dF}{d\mathbf{x}}(\mathbf{x}), \mathbf{x}^{(1)} \right\rangle \end{pmatrix} . \tag{4}$$

We mark first-order tangent versions of program variables by the superscript (1). To implement Eqn. (4) a generic tangent 1st-order scalar type (`gt1s`) is provided by `dco/c++`. Its use is illustrated in Listing 1.3.

[8] `people.maths.ox.ac.uk/gilesm/codes/libor_AD`

Listing 1.3. First-order tangents with `dco/c++`

```
1  #include <iostream>
2  #include <vector>
3  #include <cmath>
4  using namespace std;
5
6  #include "burgers.h"
7
8  #include "dco.hpp"
9  typedef dco::gt1s<double>::type DCO_T;
10
11 int main() {
12   vector<DCO_T> y(n);
13   for (int i=1;i<n-1;i++) y[i]=sin(2*PI*i/n);
14   dco::derivative(y[24])=1;
15   burgers(y);
16   vector<double> yp(dco::value(y)), dydy_v(dco::derivative(y));
17   // ... output and return
18 }
```

All `dco/c++` data types, overloaded arithmetic operations and support functions are declared in the C++ header file `dco.hpp` included in L1.3:8. They are members of the namespace `dco`. Differentiation modes are generic with respect to their base types. Instantiation with a non-derivative type such as `double` yields a first derivative mode. AD by overloading is enabled by switching the types of all active program variables to the data type `type` associated with each mode. We use the shortcut `DCO_T` for derivative types provided by `dco/c++` (L1.3:9).

The primal in `burgers.h` included in L1.3:6 is fully generic with respect to the type of its active variables; here the state `y` declared as a C++ vector of type `DCO_T` in L1.3:12 and initialized in L1.3:13. `PI` implements a sufficiently accurate approximation of π. All passive read-only variables are declared globally in `burgers.h` as outlined in Sec. 2.1. The initial directional derivative is set equal to the 25^{th} Cartesian basis vector[9] in $\mathbb{R}^n$ (L1.3:14). A single overloaded evaluation of the primal in L1.3:15 yields the 25^{th} column of the Jacobian alongside with the final primal state. Both are stored in corresponding vector variables in L1.3:16. Their values are printed for subsequent validation. Special read/write access routines for function values (`dco::value(...)`) and derivatives (`dco::derivative(...)`) are provided.

The entire Jacobian can be accumulated column-wise by letting the initial tangent range over the Cartesian basis in $\mathbb{R}^n$ yielding a computational cost of $O(n) \cdot \text{Cost}(F)$. The cost of a single tangent evaluation relative to an optimized primal typically ranges between 1.5 and 2.5 depending on specifics of the given implementation. A generic tangent 1^{st}-order vector type (`gt1v`) is provided by `dco/c++` to compute several directional derivatives simultaneously. Its use is illustrated in the following by the computation of the gradient for the LIBOR case study. Improvements in run time can be expected due to avoiding repeated evaluation of partial derivatives of all operations and better support for vectorization provided by the hardware and system software. For the given problem size we observe a reduction of the run time for accumulation of the Jacobian of the Burgers case study by a factor of three when

[9] Vector entries in C++ are interpreted as offsets into arrays. Hence the i^{th} element carries index $i-1$.

switching to gt1v mode. The corresponding code and further relevant examples are collected on the dco/c++ research website.

To avoid missunderstanding due to overloaded meanings of the term “derivative” in computational finance derivatives are referred to as *greeks* and denoted by corresponding greek letters. For example, in the LIBOR case study the gradient of the payoff with respect to the initial LIBOR rates is also known as *delta*. Its accumulation in tangent mode requires n directional derivatives in the corresponding Cartesian basis directions to be evaluated. For $n = 80$ and $p = 10^4$ Monte Carlo paths, the scalar (gt1s) version takes more than 136s while the corresponding vector (gt1v) version completes the same job in less than 60s.

Listing 1.4. Gradient for LIBOR in first-order tangent vector mode

```
1  // ... stdlib
2
3  #include "libor.h"
4
5  #include "dco.hpp"
6  typedef dco::gt1v<double,n>::type DCO_T;
7
8  int main() {
9    vector<DCO_T> L(n,0.05); DCO_T P=0;
10   srand(0); default_random_engine generator(0);
11   normal_distribution<double> distribution(0.0,1.0);
12   vector<vector<double>> Z(p,vector<double>(m));
13   for (int j=0; j<p;j++)
14     for (int i=0;i<m;i++)
15       Z[j][i]=0.3+distribution(generator);
16   for (int i=0;i<n;i++) dco::derivative(L[i])[i]=1;
17   libor(L,P,Z);
18   vector<double> dPdL(n,0);
19   for (int i=0;i<n;i++) dPdL[i]=dco::derivative(P)[i];
20   // ... output and return
21 }
```

Randon numbers are generated in L1.4:10–15. The length of the derivative vector is passed as a compile-time parameter to the gt1v template (L1.4:6). Here we choose to set it equal to the number of directional derivatives to be evaluated yielding implicitly a product of the gradient with the identity in $\mathbb{R}^n$. The optimal choice of this compile-time parameter depends on hardware specifics and may require some experiments. Vectors of length n are returned by the derivative access routine in vector mode. Offset dereferencing is used to access the individual components; see L1.4:16 and L1.4:19.

The accumulation of Jacobians as sequences of directional derivatives in tangent (vector) mode turns out to be trivially parallel. Multithreaded (using OpenMP) tangent versions of the Burgers code can be found on the dco/c++ research website.

2.3 First-Order Adjoints

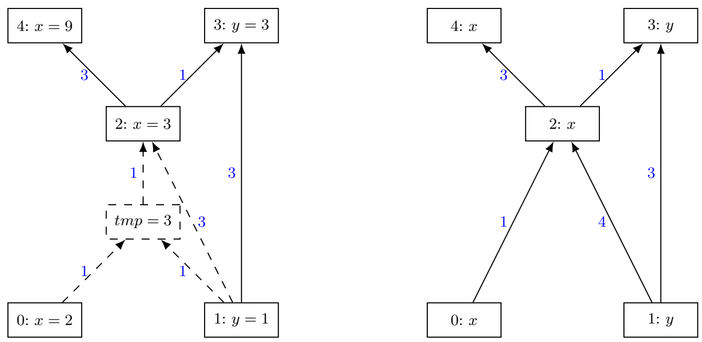
A first-order adjoint version of $\mathbf{y} := F(\mathbf{x})$ augments the primal computation with incrementation of given adjoints $\mathbf{x}_{(1)} \in \mathbb{R}^n$ of the inputs $\mathbf{x}$ with the product of the transposed Jacobian with a given vector of adjoints $\mathbf{y}_{(1)} \in \mathbb{R}^m$ of the outputs $\mathbf{y}$ as

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{x}_{(1)} \end{pmatrix} := F_{(1)}\left(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)}\right) \equiv \begin{pmatrix} F(\mathbf{x}) \\ \mathbf{x}_{(1)} + \left\langle \mathbf{y}_{(1)}, \frac{dF}{d\mathbf{x}}(\mathbf{x}) \right\rangle \end{pmatrix} . \tag{5}$$

First-order adjoint versions of program variables are marked by subscript (1). The entire

$x := 2$
$y := 1$
$x := (x + y) \cdot y$
$y := x \cdot y$
$x := 3 \cdot x$

(a)



(b) (c)

$$\begin{pmatrix} x^4_{(1)} \\ y^3_{(1)} \\ x^2_{(1)} \\ y^1_{(1)} \\ x^0_{(1)} \end{pmatrix} := \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 3 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 3 \\ 12 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 3 \\ \mathbf{12} \\ \mathbf{3} \end{pmatrix}$$

(d)

$$\begin{pmatrix} x^4_{(1)} \\ y^3_{(1)} \\ x^2_{(1)} \\ y^1_{(1)} \\ x^0_{(1)} \end{pmatrix} := \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \\ 3 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \\ 7 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \\ \mathbf{7} \\ \mathbf{1} \end{pmatrix}$$

(e)

Abbildung 2. Internal representation used by `dco/c++` in adjoint mode for given example (a): tape without statement-level gradient preaccumulation (b); tape with statement-level gradient preaccumulation (c); (comma-separated) evolution of vector of adjoints for computation of second row of Jacobian (d); evolution of vector of adjoints for computation of first row of Jacobian (e);

Jacobian can be accumulated row-wise by letting the adjoints of the primal results range over the Cartesian basis in $\mathbb{R}^m$ yielding a computational cost of $O(m) \cdot \text{Cost}(F)$. A single algorithmic adjoint evaluation typically exhibits a relative run time more than three optimized primal function evaluations depending on the specifics of the given implementation. The reduction of the run time overhead is one of the dominating challenges in adjoint AD. Insufficient memory resources may result in failure to evaluate algorithmic adjoints. Realistically this scenario is more common than one would like. Naive application of adjoint mode to practically relevant problems is almost certainly going to exceed the given memory bound. User expertise is required to make adjoints work in general. A feasible solution can always be constructed.

A simple example ($n = m = 2$) illustrating the algorithm employed by `dco/c++` to implement Eqn. (5) is shown in Fig. 2. A tape (Fig. 2 (c)) is recorded storing information on data dependences (visualized as a directed acyclic graph; dag) and local partial derivatives (visualized as edge labels). We use statement-level gradient preaccumulation implemented efficiently by template metaprogramming [41]. The dag of the statement $x := (x + y) \cdot y$ marked by dashed lines in Fig. 2 (b) is replaced by the local gradient yielding the final tape in Fig. 2 (c). For given adjoints of the active outputs (instances of y and x associated with vertices 3 and 4, and their adjoints denoted as $y^3_{(1)}$ and $x^4_{(1)}$, respectively) interpretation of the tape over an associated vector of adjoints yields a linear combination of the rows of the Jacobian. In Fig. 2 (e) and (d) we illustrate the propagation of both Cartesian basis vectors in $\mathbb{R}^2$ as a comma-separated list of states of the vector of adjoints. During a last-in-first-out traversal of the vertices v^i in the dag (e.g, $i = 4, \ldots, 0$) adjoints associated with all predecessors are incremented with the product of $v^i_{(1)}$ with the local partial derivative labelling the corresponding edge. The resulting Jacobian entries are highlighted.

The distinction between tape and associated vector of adjoints enables separation of sequentially (tape) and nonsequentially (vector of adjoints) accessed data. Recording typically dominates the computational effort in comparison to interpretation as supported by results in Sec. 2.7. Moreover decomposition of the internal data structure allows allocation of several vectors of adjoints and their parallel interpretation as described in Sec. 3.2.

To implement Eqn. (5) `dco/c++` provides a generic adjoint 1$^{\text{st}}$-order scalar type (`ga1s`) defined in `dco.hpp`. Refer to Listing 1.5 for illustration in the context of the Burgers case study.

Listing 1.5. First-order adjoints with `dco/c++`

```
1  // ... stdlib
2
3  #include "burgers.h"
4
5  #include "dco.hpp"
6  typedef dco::ga1s<double> DCO_M;
7  typedef DCO_M::type DCO_T;
8  typedef DCO_M::tape_t DCO_TAPE_T;
9
10 int main() {
11   // ... L3:12-13
12   DCO_M::global_tape=DCO_TAPE_T::create();
13   DCO_M::global_tape->register_variable(y);
14   vector<DCO_T> yc(y);
15   burgers(yc);
16   DCO_M::global_tape->register_output_variable(yc[25]);
17   dco::derivative(yc[25])=1.;
18   DCO_M::global_tape->interpret_adjoint();
19   vector<double> v_dydy(dco::derivative(y));
20   cerr << dco::size_of(DCO_M::global_tape) << "B" << endl;
21   DCO_TAPE_T::remove(DCO_M::global_tape);
22   // ... output and return
23 }
```

As in tangent mode, all active program variables need to be redeclared. The new type (shortcut: `DCO_T`) is defined as part of the adjoint mode (`DCO_M`) over a passive base type (here: `double`); L1.5:6–7. A tape type is associated with adjoint mode (`DCO_TAPE_T`; L1.5:8). An instance is created in L1.5:12 to record all information required for the evaluation of Eqn. (5). Recording is triggered by overloaded operations on previously recorded arguments. Hence, all independent inputs need to be recorded (also: registered with the tape) explicitly (L1.5:13). Results of overloaded operations are recorded automatically.

All independent inputs (`y`) must be read-only in order to ensure correct access to their adjoints in L1.5:19. The evolution of the state is therefore performed on a copy `yc` (L1.5:14). The value of the final state can be extracted from `yc` following the call to the overloaded primal (L1.5:15). Active outputs need to be registered explicitly (L1.5:16) to ensure correct computation of their adjoints despited possible reuse in subsequent computations. This scenario is not illustrated by the given example, where line 16 could in fact be omitted. Nevertheless we advise users of `dco/c++` to register active outputs in order to avoid potential trouble in less obvious situations.

We chose to compute the gradient of the 26^{th} entry of the final state with respect to the initial state by setting the adjoint final state equal to the corresponding Cartesian basis vector (L1.5:17). Derivative components of active program variables of `dco/c++` adjoint type are guaranteed to be equal to zero prior to their first use. Interpretation of the tape in L1.5:18 yields adjoints with machine accuracy. Function values match the ones obtained in tangent mode. The adjoints are entries of the 26^{th} row the Jacobian which intersects with the 25^{th} column computed in Sec. 2.2 in its 25^{th} element. Numerical results can be validated by running the tangent and adjoint versions of the Burgers case study provided in the `dco/c++` research website. The size of the tape in bytes can be recovered for diagnostics (L1.5:20). Deallocation of the tape requires calling of a dedicated routine (L1.5:21).

The current version of `dco/c++` supports three kinds of tapes. A “blob tape” allocates a specified amount of main memory for recording and interpretation at maximum speed. It is up to the user to ensure that sufficient tape memory is allocated; an exception is raised otherwise. Improved robustness comes with the “chunk tape”. It allocates chunks of main memory of specified size up to the limit of the physical memory available. A slight run time overhead is induced by chunk management. Chunks can be written to and read from hard disc when using a “file tape.” The resulting increase in tape memory comes at the expense of further decrease in computational efficiency. However it allows “brute force” evaluation of adjoints of larger problem instances at no extra development cost. This feature proves advantageous for debugging during the development and for validation of more sophisticated solutions.

In Listing 1.5 a global tape was used. Thread-safe adjoint simulations require thread-local tapes supported by `dco/c++` through its `ga1sm` mode allowing for multiple tapes to be allocated. Built-in varied (also: forward activity) analysis [22] can help reduce the size of the tapes. An adjoint vector mode is also available. Refer to the `dco/c++` user guide for further information.

For the LIBOR case study a single evaluation of the adjoint consisting of tape recording (L1.6:9–12) and interpretation (L1.6:13–14) gives the entire gradient extracted in L1.6:15.

Listing 1.6. Gradient for LIBOR in first-order adjoint mode

```
1  // ... stdlib
2
3  #include "libor.h"
4  // ... L5:6-8
5
6  int main() {
7    vector<DCO_T> L(n,0.05); DCO_T P=0;
8    // L4:10-15
9    DCO_M::global_tape=DCO_TAPE_T::create();
10   DCO_M::global_tape->register_variable(L);
```

```
  libor(L,P,Z);
  DCO_M::global_tape->register_output_variable(P);
  dco::derivative(P)=1;
  DCO_M::global_tape->interpret_adjoint();
  vector<double> dPdL(dco::derivative(L));
  DCO_TAPE_T::remove(DCO_M::global_tape);
  // ... output and return
}
```

For $n = 80$ and $p = 10^4$ Monte Carlo paths the above returns the gradient after less than 3s yielding a speedup by a factor of roughly 30 compared to tangent vector mode.

In Fig. 3 we compare (total) run times for gradients of the midpoint of the solution of Burgers' case study with respect to the initial condition. The relative computational costs of all three tangent versions considered scale linearly with the number of active inputs while the relative cost of the adjoint remains essentially constant.

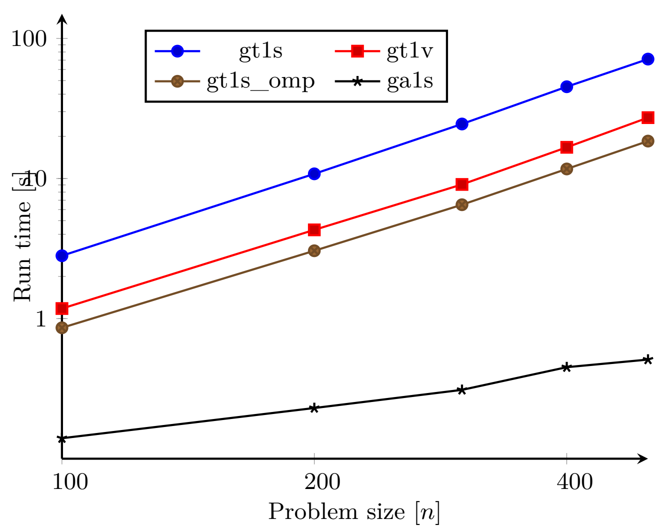


Abbildung 3. Race: We compare run times (in seconds) for gradients of the midpoint of the solution of Burgers' case study with respect to the initial condition for $m = 1000$ implicit Euler steps and $n = 100, \ldots, 500$ spatial grid points. Four `dco/c++` modes are considered: Scalar tangent mode (gt1s), vector tangent mode with vector length n (gt1v), parallel scalar tangent mode with OpenMP on four threads (gt1s_omp), scalar adjoint mode (ga1s).

Similar to tangent mode `dco/c++` features a generic adjoint 1[st]-order vector data type (`ga1v`). The accumulation of Jacobians as sequences of adjoints in adjoint (vector) mode is also trivially parallel. See Sec. 3.2 for discussion of a corresponding solution with `dco/c++`.

2.4 Second-Order Tangents

Application of tangent mode to a first-order tangent code yields a second-order tangent code for evaluating

$$
\begin{aligned}
\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(2)} \\ \mathbf{y}^{(1)} \\ \mathbf{y}^{(1,2)} \end{pmatrix} &:= F^{(1,2)}\left(\mathbf{x}, \mathbf{x}^{(2)}, \mathbf{x}^{(1)}, \mathbf{x}^{(1,2)}\right) \\
&\equiv \begin{pmatrix} F(\mathbf{x}) \\ \left\langle \frac{dF}{d\mathbf{x}}(\mathbf{x}), \mathbf{x}^{(2)} \right\rangle \\ \left\langle \frac{dF}{d\mathbf{x}}(\mathbf{x}), \mathbf{x}^{(1)} \right\rangle \\ \left\langle \frac{d^2F}{d\mathbf{x}^2}(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \right\rangle + \left\langle \frac{dF}{d\mathbf{x}}(\mathbf{x}), \mathbf{x}^{(1,2)} \right\rangle \end{pmatrix} .
\end{aligned}
\tag{6}
$$

Tangent versions of program variables due to the application of tangent mode to first derivative code are marked by superscipt (2). We set $v^{(1)(2)} \equiv v^{(1,2)}$. When implementing Eqn. (6) with `dco/c++` both the computation of the function value and of the first directional derivative are augmented with their respective directional derivatives yielding two first derivatives $\mathbf{y}^{(1)}$ and $\mathbf{y}^{(2)}$ and a second derivative $\mathbf{y}^{(1,2)}$ alongside the function value $\mathbf{y}$. Extraction of pure second derivative information requires $\mathbf{x}^{(1,2)} = 0$ on input. Individual entries $\mathbf{y}^{(1,2)} \in \mathbb{R}^m$ of the Hessian can be obtained by setting $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ equal to the corresponding Cartesian basis vectors yielding a computational cost of $O(n^2) \cdot \mathrm{Cost}(F)$ for accumulation of the whole Hessian.

To illustrate the use of `dco/c++` in second-order tangent mode we compute

$$
\frac{d^2\mathbf{y}^m}{dy_{24}^0 dy_{25}^0} \in \mathbb{R}^n
$$

for the Burgers case study in Listing 1.7.

Listing 1.7. Second-order tangents with `dco/c++`

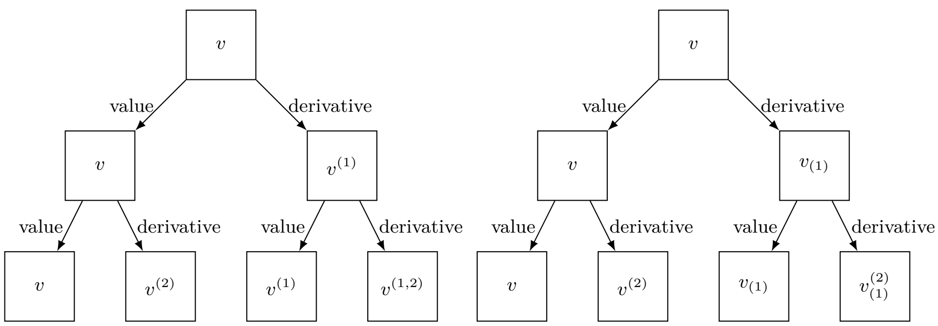
```cpp
// ... stdlib

#include "burgers.h"

#include "dco.hpp"
typedef dco::gt1s<double>::type DCO_BT;
typedef dco::gt1s<DCO_BT>::type DCO_T;

int main() {
  // ... L3:12-13
  dco::value(dco::derivative(y[24]))=1;
  dco::derivative(dco::value(y[25]))=1;
  burgers(y);
  vector<double> ddydyy_v_v(dco::derivative(dco::derivative(y)));
  // ... output and return
}
```

A dco/c++ first-order tangent type is defined over a base type DCO_BT. Setting this base type equal to a first-order tangent type over `double` yields a second-order tangent type (L1.7:6–7). The data access pattern is illustrated in Fig. 4 (a). Recursive template instantiation results in a data type with four elements of type `double`. Access to them is provided by corresponding nested calls to `dco::value(...)` and `dco::derivative(...)`. For example, a call of `dco::derivative`(v) on a variable v of second-order tangent type returns its derivative component $v^{(1)}$. Calling `dco::derivative`($v^{(1)}$) on this variable of first-order tangent type yields $v^{(1,2)}$. The value of $v^{(1)}$ can be extracted by calling `dco::value`($v^{(1)}$). In Fig. 4 the order of a variable is determined by its distance from the leaf nodes of the tree. For example, the intermediate v represents a first-order tangent type while its successor with the same label is the actual value ("0^{th} derivative").



(a) (b)

Abbildung 4. Data access in second-order (tangent over) tangent (a) and (tangent over) adjoint (b) types provided by dco/c++

Consequently, $y^{(2)}$ is set equal to the 25^{th} Cartesian basis vector in L1.7:11 followed by setting $y^{(1)}$ equal to the 26^{th} Cartesian basis vector in L1.7:12. The overloaded primal called in L1.7:13 yields $y^{(1,2)} = \frac{d^2\mathbf{y}^m}{dy^0_{24}dy^0_{25}}$ stored in a vector of matching size n in L1.7:14. Code for the accumulation of the whole Hessian tensor is easily derived from Listing 1.7.

As one of the *greeks* the Hessian of the payoff with respect to the initial LIBOR rates is also referred to as *gamma*. Its accumulation shown in Listing 1.8 takes $\binom{n+1}{2}$ evaluations of the primal model overloaded for second-order tangent dco/c++ types.

Listing 1.8. Hessian of LIBOR in second-order tangent mode

```
// ... stdlib

#include "libor.h"
// ... L7:5-7

int main() {
  vector<DCO_T> L(n,0.05); DCO_T P=0;
  // L4:10-15
  vector<vector<double> > ddPdLL(n,vector<double>(n,0));
  for (int i=0;i<n;i++) {
    dco::value(dco::derivative(L[i]))=1;
    for (int j=0;j<=i;j++) {
```

```
13        dco::derivative(dco::value(L[j]))=1;
14        libor(L,P,Z);
15        ddPdLL[i][j]=ddPdLL[j][i]
16                    =dco::derivative(dco::derivative(P));
17        dco::derivative(dco::value(L[j]))=0;
18      }
19      dco::value(dco::derivative(L[i]))=0;
20    }
21    // ... output and return
22  }
```

With $\mathbf{x} \equiv L \in \mathbb{R}^n$ the input directions $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ are set to range independently over the Cartesian basis vectors in $\mathbb{R}^n$, respectively (L1.8:11,19 and L1.8:13,17). All derivative components of the second-order tangent variables are guaranteed to be equal to zero following construction (L1.8:7). Symmetry of the Hessian is exploited (L1.8:12,15–16). The overloaded LIBOR model is evaluated for each of the $\sum_{k=1}^{n} k = \binom{n+1}{2}$ relevant combinations of $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$. With $y \equiv P \in \mathbb{R}$ the resulting second-order tangent projection $y^{(1,2)}$ of the Hessian contains $h_{i,j} = h_{j,i}$ (L1.8:15). Resetting $x_j^{(2)} = 0$ (L1.8:17) and $x_i^{(1)} = 0$ (L1.8:19) ensures correct (re)seeding with Cartesian basis vectors. Note that $\mathbf{x} \equiv L$ is not modified by the `libor` function which makes selective resetting of its derivative components feasible. If $\mathbf{x}$ was modified, then all derivative components of all its entries would have to be reset to zero to ensure correct seeding prior to each overloaded call to `libor`.

For the given scenario (see Sec. 2.2) the total run time of Hessian accumulation in second-order tangent mode adds up to more than 4 hours. A similar run time is observed when using central finite difference approximation in `double` precision. Combinations of `gt1s` and `gt1v` types are possible. Multithreading can be applied to speed up the computation. Still the overall run time remains unsatisfactory.

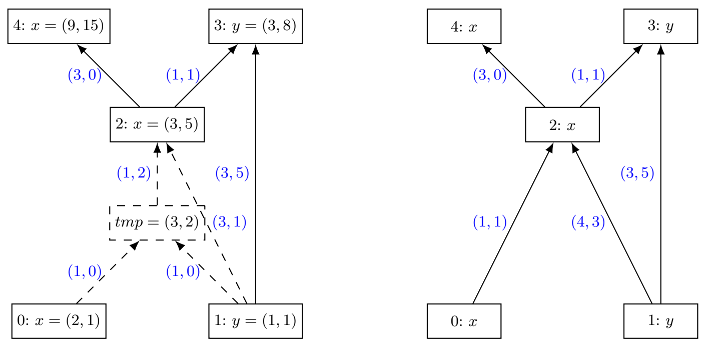
2.5 Second-Order Adjoints

Application of tangent mode to a first-order adjoint code yields a second-order adjoint code for evaluating

$$
\begin{aligned}
\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(2)} \\ \mathbf{x}_{(1)} \\ \mathbf{x}_{(1)}^{(2)} \end{pmatrix} &:= F_{(1)}^{(2)}\left(\mathbf{x}, \mathbf{x}^{(2)}, \mathbf{x}_{(1)}, \mathbf{x}_{(1)}^{(2)}, \mathbf{y}_{(1)}, \mathbf{y}_{(1)}^{(2)}\right) \\
&\equiv \begin{pmatrix} F(\mathbf{x}) \\ \left\langle \frac{dF}{d\mathbf{x}}(\mathbf{x}), \mathbf{x}^{(2)} \right\rangle \\ \mathbf{x}_{(1)} + \left\langle \mathbf{y}_{(1)}, \frac{dF}{d\mathbf{x}}(\mathbf{x}) \right\rangle \\ \mathbf{x}_{(1)}^{(2)} + \left\langle \mathbf{y}_{(1)}, \frac{d^2F}{d\mathbf{x}^2}(\mathbf{x}), \mathbf{x}^{(2)} \right\rangle + \left\langle \mathbf{y}_{(1)}^{(2)}, \frac{dF}{d\mathbf{x}}(\mathbf{x}) \right\rangle \end{pmatrix}
\end{aligned}
\tag{7}
$$

As before, we use the superscript (2) to mark tangent versions of program variables due to the application of tangent mode to a first derivative code (here: first-order adjoint code). To implement Eqn. (7) with `dco/c++` the computation of the function value and of the first-order adjoint are augmented with their first derivatives yielding $\mathbf{y}^{(2)}$ and $\mathbf{x}_{(1)}^{(2)}$ as directional derivatives of $\mathbf{y}$ and $\mathbf{x}_{(1)}$ in direction $\mathbf{x}^{(2)}$, respectively. Extraction of pure second derivative information from $\mathbf{x}_{(1)}^{(2)}$ requires $\mathbf{x}_{(1)}^{(2)} = \mathbf{y}_{(1)}^{(2)} = 0$ on input. Individual columns ($\mathbf{x}_{(1)}^{(2)} \in \mathbb{R}^n$) of the Hessian are obtained by setting $\mathbf{y}_{(1)}$ and $\mathbf{x}^{(2)}$ equal to the corresponding Cartesian basis vectors yielding a computational cost of $O(m \cdot n) \cdot \text{Cost}(F)$ for accumulation of the whole Hessian. Fig. 5 illustrates second-order adjoint mode implemented as tangents of adjoints as

$$
\begin{aligned}
x &:= 2\\
y &:= 1\\
x &:= (x+y)\cdot y\\
y &:= x \cdot y\\
x &:= 3 \cdot x
\end{aligned}
$$



(a) (b) (c)

$$
\begin{pmatrix} x^4_{(1)} \\ y^3_{(1)} \\ x^2_{(1)} \\ y^1_{(1)} \\ x^0_{(1)} \end{pmatrix} := \begin{pmatrix} (1,0) \\ (1,0) \\ (0,0) \\ (0,0) \\ (0,0) \end{pmatrix}, \begin{pmatrix} (1,0) \\ (1,0) \\ (3,0) \\ (0,0) \\ (0,0) \end{pmatrix}, \begin{pmatrix} (1,0) \\ (1,1) \\ (4,1) \\ (3,5) \\ (0,0) \end{pmatrix}, \begin{pmatrix} (1,0) \\ (1,1) \\ (4,1) \\ (19,\mathbf{21}) \\ (4,\mathbf{5}) \end{pmatrix}
$$

Abbildung 5. Internal representation used by `dco/c++` in second-order adjoint mode for given example (a): tape without statement-level gradient preaccumulation (b); tape with statement-level gradient preaccumulation (c); (comma-separated) evolution of vector of first- and second-order adjoints for computation of Hessian projection (below horizontal line)

in Eqn. (7) for the same simple example used in Fig. 2. Directional derivatives in direction $(x^{(2)}, y^{(2)})^T$ are stored alongside function values and local partial derivatives yielding value pairs labelling the vertices and edges in the dag shown in Fig. 5 (b). Statement-level preaccumulation yields the local gradient for $x := (x+y)\cdot y$ and the product of the local 2×2 Hessian with the vector $(x^{(2)}, y^{(2)})^T$. For given adjoints $y^3_{(1)}$ and $x^4_{(1)}$ of the active outputs and assuming vanishing second-order adjoints interpretation of the tape yields a second-order projection of the Hessian tensor in directions $(x_{(1)}, y_{(1)})^T$ and $(x^{(2)}, y^{(2)})^T$. The evolution of the corresponding vector of first- and second-order adjoints is shown in the bottom part of Fig. 5. During a last-in-first-out traversal of the vertices v^i Entries of the resulting Hessian projection are highlighted.

To illustrate the use of `dco/c++` in second-order adjoint mode we compute

$$
\frac{d^2 y^m_{25}}{dy^0_{24} d\mathbf{y}^0} \in \mathbb{R}^n
$$

for the Burgers case study in Listing 1.9.

Listing 1.9. Second-order adjoints with `dco/c++`

```
// ... stdlib
```

```
2
3 #include "burgers.h"
4
5 #include "dco.hpp"
6 typedef dco::gt1s<double>::type DCO_BT;
7 typedef dco::ga1s<DCO_BT> DCO_M;
8 typedef DCO_M::type DCO_T;
9 typedef DCO_M::tape_t DCO_TAPE_T;
10
11 int main() {
12   // ... L3:12-13
13   DCO_M::global_tape=DCO_TAPE_T::create();
14   DCO_M::global_tape->register_variable(y);
15   dco::derivative(dco::value(y[24]))=1;
16   vector<DCO_T> yc(y);
17   burgers(yc);
18   DCO_M::global_tape->register_output_variable(yc[25]);
19   dco::derivative(yc[25])=1.;
20   DCO_M::global_tape->interpret_adjoint();
21   vector<double> v_ddydyy_v(dco::derivative(dco::derivative(y)));
22   DCO_TAPE_T::remove(DCO_M::global_tape);
23   // ... output and return
24 }
```

The first-order adjoint version of the Burgers case study is overloaded for a first-order tangent base type over `double` in L1.9:6–7. Creation of the tape and registration of the initial state as active input is similar to the first-order adjoint (L1.9:13–14). The direction $\mathbf{x}^{(2)}$ is set equal to the 25$^{\text{th}}$ Cartesian basis vector in L1.9:15. Recording of the tape (L1.9:17) is again performed on a copy `yc` of the state (L1.9:16) to ensure correct access to the first- and second-order adjoints of the initial state in L1.9:23. Both the function evaluation and its derivative in direction $\mathbf{x}^{(2)}$ are recorded. The adjoint final state is set equal to the 26$^{\text{th}}$ Cartesian basis vector (L1.9:19) prior to interpretation of the tape (L1.9:20), that is, propagation of adjoints of the function evaluation and of its directional derivative. Access to the individual elements of second-order adjoint variables follows the same logic as the second-order tangent version. It is illustrated in Fig. 4 (b). Within the Hessian tensor

$$H = (h)^k_{i,j} \equiv \frac{d^2\mathbf{y}^m}{d\mathbf{y}^0 d\mathbf{y}^0} \in \mathbb{R}^{n\times n\times n}$$

the second-order tangent and adjoint results intersect in element $h^{25}_{24,25}$. Correctness of the numerical results can be verified by running the sample codes provided on the `dco/c++` research website.

Accumulation of the entire Hessian for the LIBOR example takes n evaluations of the second-order adjoint routine with $\mathtt{P}_{(1)} \equiv y_{(1)} = 1$ and $\mathtt{L}^{(2)} \equiv \mathbf{x}^{(2)}$ ranging over the Cartesian basis vectors in $\mathbb{R}^n$ as shown in Listing 1.10.

Listing 1.10. Hessian of LIBOR in second-order adjoint mode

```
// ... stdlib
```

```
2
3  #include "libor.h"
4  // ... L9:5-9
5
6  int main() {
7    vector<DCO_T> L(n,0.05); DCO_T P=0;
8    // L4:10-15
9    DCO_M::global_tape=DCO_TAPE_T::create();
10   DCO_M::global_tape->register_variable(L);
11   DCO_TAPE_POSITION_T tpos=DCO_M::global_tape->get_position();
12   vector<vector<double> > ddPdLL(n,vector<double>(n,0));
13   for(int j=0;j<n;j++) {
14     dco::derivative(dco::value(L[j]))=1;
15     libor(L,P,Z);
16     DCO_M::global_tape->register_output_variable(P);
17     dco::value(dco::derivative(P))=1;
18     DCO_M::global_tape->interpret_adjoint_to(tpos);
19     for(int i=0;i<n;i++) {
20       ddPdLL[i][j]=dco::derivative(dco::derivative(L[i]));
21       dco::derivative(L[i])=0;
22     }
23     dco::derivative(dco::value(L[j]))=0;
24     DCO_M::global_tape->reset_to(tpos);
25   }
26   DCO_TAPE_T::remove(DCO_M::global_tape);
27   // ... output and return
28 }
```

Both creation of the tape (L1.10:9) and registration of the active inputs (L1.10:10) are performed once followed by n recordings and corresponding interpretations for the different directions $\mathbf{x}^{(2)}$ (L1.10:14). Once allocated tape memory should be shared amongst the recordings. Repeated registration of the same inputs should be avoided. Hence, `dco/c++` allows to store the tape position of type

```
typedef DCO_TAPE_T::position_t DCO_TAPE_POSITION_T
```

after registration of the active inputs (L1.10:11) in order to restart taping from this position (L1.10:24). Optionally, interpretation can be stopped at this position (L1.10:18) to avoid unnecessary visits of tape locations that respresent program variables and have no effect on the propagation of adjoints due to missing arguments.

Subsequent recordings and interpretations require correct re-initialization of certain variables on the right-hand side of Eqn. (7), namely $\mathbf{x}_{(1)}^{(2)}$ (L1.10:21) and $\mathbf{x}^{(2)}$ (L1.10:23). Note that both $\mathbf{x}_{(1)}^{(2)}$ and $\mathbf{x}_{(1)}$ are set equal to zero in L1.10:21 ensuring optionally correct first-order adjoints in $\mathbf{x}_{(1)}$ for all n iterations.

As alternatives to the "tangents of adjoints" approach second-order adjoints can be computed as *"adjoints of tangents"* as well as *"adjoints of adjoints"*; examples can be found on the `dco/c++` research website. While both alternatives turn out to be mathematically equivalent to the "tangents of adjoints" approach the latter requires handling of nested tapes yielding mostly suboptimal run time performance. See [28] for further studies of the various combinations and for an example where *adjoints of adjoints* outperforms its competitors.

2.6 Higher-Order Tangents and Adjoints

Recursive nesting of first derivative types allows seamless extension of most dco/c++ solutions to arbitrary order of differentiation. Despite the fact, that higher-order tangent versions of first-order adjoints are the preferred option for computing higher-order adjoints in almost all cases arbitrary combinations of tangent and adjoint types are possible. For example, third-order adjoint mode can be implemented as an adjoint version of a second-order adjoint model derived as a tangent version of a first-order adjoint yielding

$$\begin{aligned}
\mathbf{y} &:= F(\mathbf{x}) \\
\mathbf{y}^{(2)} &:= \left\langle \frac{dF}{d\mathbf{x}}(\mathbf{x}), \mathbf{x}^{(2)} \right\rangle \\
\mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \left\langle \mathbf{y}_{(1)}, \frac{dF}{d\mathbf{x}}(\mathbf{x}) \right\rangle \\
\mathbf{x}_{(1)}^{(2)} &:= \mathbf{x}_{(1)}^{(2)} + \left\langle \mathbf{y}_{(1)}, \frac{d^2F}{d\mathbf{x}^2}(\mathbf{x}), \mathbf{x}^{(2)} \right\rangle + \left\langle \mathbf{y}_{(1)}^{(2)}, \frac{dF}{d\mathbf{x}}(\mathbf{x}) \right\rangle \\
\mathbf{x}_{(3)} &:= \mathbf{x}_{(3)} + \left\langle \mathbf{y}_{(3)}, \frac{dF}{d\mathbf{x}}(\mathbf{x}) \right\rangle + \left\langle \mathbf{y}_{(3)}^{(2)}, \frac{d^2F}{d\mathbf{x}^2}(\mathbf{x}), \mathbf{x}^{(2)} \right\rangle + \left\langle \mathbf{x}_{(1,3)}, \frac{d^2F}{d\mathbf{x}^2}(\mathbf{x}), \mathbf{x}^{(2)} \right\rangle \\
&\quad + \left\langle \mathbf{x}_{(1,3)}, \mathbf{y}_{(1)}, \frac{dF^2}{d\mathbf{x}^2}(\mathbf{x}) \right\rangle + \left\langle \mathbf{x}_{(1,3)}^{(2)}, \mathbf{y}_{(1)}, \frac{d^3F}{d\mathbf{x}^3}(\mathbf{x}), \mathbf{x}^{(2)} \right\rangle \\
\mathbf{x}_{(3)}^{(2)} &:= \mathbf{x}_{(3)}^{(2)} + \left\langle \mathbf{y}_{(3)}^{(2)}, \frac{dF}{d\mathbf{x}}(\mathbf{x}) \right\rangle + \left\langle \mathbf{x}_{(1,3)}^{(2)}, \mathbf{y}_{(1)}, \frac{d^2F}{d\mathbf{x}^2}(\mathbf{x}) \right\rangle \\
\mathbf{y}_{(1,3)} &:= \mathbf{y}_{(1,3)} + \left\langle \mathbf{x}_{(1,3)}, \frac{dF}{d\mathbf{x}}(\mathbf{x}) \right\rangle + \left\langle \mathbf{x}_{(1,3)}^{(2)}, \frac{d^2F}{d\mathbf{x}^2}(\mathbf{x}), \mathbf{x}^{(2)} \right\rangle \\
\mathbf{y}_{(1,3)}^{(2)} &:= \mathbf{y}_{(1,3)}^{(2)} + \left\langle \mathbf{x}_{(1,3)}^{(2)}, \frac{dF}{d\mathbf{x}}(\mathbf{x}) \right\rangle
\end{aligned}$$

Projections of the third derivative tensor in directions $\mathbf{y}_{(1)}$, $\mathbf{x}^{(2)}$, and $\mathbf{x}_{(1,3)}^{(2)}$ can be computed. An implementation of a corresponding third-order adjoint with dco/c++ can be found on the dco/c++ research website.

2.7 Performance

Performance tuning for AD software is typically focussed on first-order adjoint mode which poses the main challenges. Scalar tangent mode can be expected to be implemented efficiently in virtually all cases. Preaccumulation based on template metaprogramming as employed by dco/c++ typically has a positive effect on vector tangent mode. The chosen vector length and parallelization strategy often turn out to have an even stronger impact. Hence, when evaluating a given vector tangent mode AD solution its quality is more likely to be dependent on the skills of the user of the given AD software rather than on the software itself. Experience shows that this claim holds even more for nontrivial real-world first- and higher-order adjoint solutions requiring substantial user expertise in terms of understanding the data dependencies within the primal (e.g, for checkpointing or preaccumulation), its mathematics (e.g, for handling implicit functions) or the compute environment (e.g, for inclusion of GPUs[10] into a CPU-based adjoint). Fair comparison of individual AD tools becomes extremely difficult if not practially impossible in such cases.

Basic performance of adjoint AD software should be measured in terms of the relative run time $\mathcal{R}$ (see Eqn. (3)) and the amount of memory occupied by the internally stored data (e.g, tape and vector(s) of adjoints in case of `dco/c++`). Reliable comparison among different tools is only possible for relatively small test cases whose memory requirement stays within the given bounds on the available main memory. They should be run in basic adjoint mode (inspired by the "hello world" adjoint examples from the various user guide) not including any "user tricks." Having said this, such performance comparisons are of only limited benefit when it comes to dealing with real-world scenarios. While they might indicate that a given AD tool yields very efficient adjoint code at a local scale the challenge of effective use of the wide range of AD methods by the user of the tool remains dominant. This observation implies that while local performance of AD tools is important their flexibility with respect to diverse application scenarios is even more so. For example, a run time factor of two gained on local sections of the target code is easily lost by choosing the wrong preaccumulation or checkpointing strategies. A suboptimal API of an AD tool may complicate effective use of such methods due to restricted access to the internal representation or due to unnecessary copying of data. `dco/c++` has proven to address such issues effectively in a number of projects. Some commerical users in particular have gone through extensive test periods before committing to `dco/c++`.

In the following we investigate relative run times of basic first-order adjoints generated with `dco/c++` and we comment on results obtained with other AD tools. Target primals are selected according to the previous discussion, that is, their adjoints can be evaluated on our target computer within the given limits on the main memory. The following test problems are part of the `dco/c++` test suite:

3D cross-frame field (CoMISo) This code is part of a method for constructing a 3D cross-frame field, a 3D extension of the 2D cross-frame field as applied to surfaces in applications such as quadrangulation and texture synthesis [26]. It consists of approximately 80 straight lines generated by Maple™ [33] with common subexpression optimization switched on. The code is part of a test case of the software package CoMISo [5].

Lax-Wendroff (LW) and Toon (Toon) Both test problems are described in further detail in [24], where they were used to test performance of the AD tool Adept. Both solve the one-dimensional advection equation

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x}$$

with state u, time t, spacial coordinate x, and the initial state x_0 as parameter. The equation is solved on an equidistant grid with either Lax-Wendroff [27] (linear) or Toon [44] (nonlinear) schemes. The discretized cost functional is given as $f(\mathbf{x}_0) = \|\mathbf{x}_e(\mathbf{x}_0)\|$ with final state $\mathbf{x}_e$.

Burgers' equation with forward Euler time stepping (Burgers(F)) This code implements the problem from Sec. 2.1 using explicit time stepping.

LIBOR market model (LIBOR) This is the code from Sec. 2.1.

[10] Graphics Processing Units

Burgers' equation with backward Euler time stepping (Burgers(B)) This is the code from Sec. 2.1.

In Fig. 6 we show measurements for three different `dco/c++` configurations: blob tape, chunk tape, and with multiple tape support. Blob and chunk tapes were described in Sec. 2.3. Multiple tape support allows for a thread-safe use of several tapes concurrently yielding a larger memory footprint for each program variable, since an additional reference to the owning tape is required. Multiple tapes can be either chunk or blob tapes. We use blob tapes in our measurements.

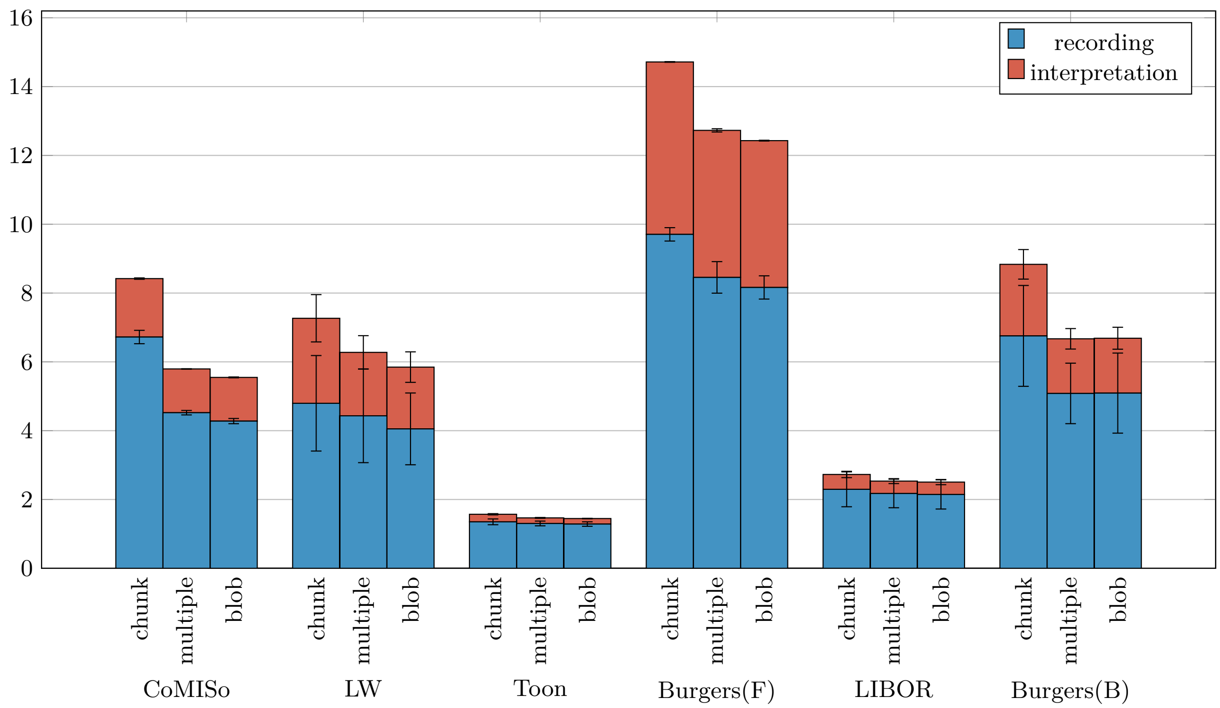


Abbildung 6. This figure shows the run time ratio $\mathcal{R}$ for the listed test problems for different configurations of `dco/c++`. The blue part corresponds to the tape recording time while the red part visualizes tape interpretation time.

Our in-house performance test framework constantly measures run time and memory consumption for other AD overloading tools including ADOL-C[11], CppAD[12], tapescript[13], and Adept.[14] The results indicate `dco/c++` to be the fastest tool, followed by Adept (10% $\sim$ 250% slower) and CppAD (200% $\sim$ 500% slower). ADOL-C and tapescript turn out to be substantially slower on average. For example, on the Burgers(B) case study, our measurements for `dco/c++` (blob tape) suggest a relative run time of less than 7. Adept is second fastest at a relative run time of 19 followed by CppAD (40), ADOL-C (96) and tapescript (109). Qualitatively we were able to reproduce the observations from [24] for LW and Toon. The

[11] projects.coin-or.org/ADOL-C
[12] www.coin-or.org/CppAD/
[13] github.com/compatibl/tapescript
[14] www.met.reading.ac.uk/clouds/adept/

lowest memory consumption is exhibited by `dco/c++` (blob tape) followed by ADOL-C and Adept (30% $\sim$ 250% increase). CppAD and tapescript require even more memory.

3 Selected Special Features

While Sec. 2 dealt with the basic functionality provided by `dco/c++` we discuss in the following selected special and partially unique features that turned out particularly beneficial in actual applications. Three important aspects of algorithmic adjoints are addressed:

1. *Flexibility of user interaction with* `dco/c++` *adjoints*; Users may want to or even have to deviate from the standard `dco/c++` approach to the evaluation of adjoints (taping + interpretation). For example, source may be missing for part of the primal making numerical approximation by finite differences and integration into the adjoint data flow a feasible alternative. An interface for inclusion of external adjoints is presented in Sec. 3.1 in the context of the Burgers case study. It is used to replace the *algorithmic adjoint* Newton solver for the implicit Euler step by a *symbolic adjoint* version as described in detail in [37].
2. *Multithreading*; With virtually all modern computer architectures supporting shared memory parallelism tread-safe implementations of adjoints become increasingly relevant. Two scenarios are discussed in Sec. 3.2:
   (a) Numerical simulations running in a multithreaded shared memory environment require adjoint versions. The use of thread-local tapes is discussed in the context of the LIBOR test case in Sec. 3.2.
   (b) The evaluation of several adjoints at a given point (for a given tape) can be done in parallel using several threads over separate thread-local vectors of adjoints as shown in Sec. 3.2 using the Burgers case study.
3. *Tape compression by preaccumulation*; In most cases the often prohibitive size of the tape is the main limiting factor for applicability of basic `dco/c++` adjoints as introduced in Sec. 2.3. In addition to various checkpointing schemes which can be implemented using the external adjoint interface (see Sec. 3.1) `dco/c++` offers an easy-to-use preaccumulation interface presented in Sec. 3.3.

3.1 External Adjoint Interface

This section introduces the external adjoint interface provided by `dco/c++`. It enables seamless interaction with a `dco/c++` adjoint through implementation of custom adjoints for selected parts of the primal computation and their integration into the corresponding tape. The external adjoint interface has proven crucial for the construction of robust and efficient adjoints for various applications [46,31,30]. Its design is driven by the chain rule of differential calculus. Support is provided for various relevant target scenarios including checkpointing, preaccumulation, approximate adjoints for black boxes, symbolic adjoints for implicit functions as well as generalization to higher-order adjoints.

Some formalism is required to introduce the external adjoint interface properly. For notational convenience we assume all elementals to map from the entire memory space of the program $(v_{1-n}, \ldots, v_q)$ onto itself. A similar approach is taken in [18] and [17].

The primal program for computing a multivariate vector function $F : \mathbb{R}^n \to \mathbb{R}^m$ as $\mathbf{y} = F(\mathbf{x})$ yields an elemental decomposition

$$\mathbf{v}^i = \Phi^i(\mathbf{v}^{i-1}), \quad \Phi^i : \mathbb{R}^{n+q} \to \mathbb{R}^{n+q} \text{ for } i = 1, \ldots, q$$

and $\mathbf{v}^0 = (x^0, \ldots, x^{n-1}, 0, \ldots, 0)$, $\mathbf{v}^q = (x^0, \ldots, x^{n-1}, v^1, \ldots, v^p, y^0, \ldots, y^{m-1})$. Consequently, $\mathbf{x} = P_n \cdot \mathbf{v}^0$ and $\mathbf{y} = \mathbf{v}^q \cdot Q_m^T$ for linear operators $P_n = (I_{n\times n}, 0_{n\times q}) \in \mathbb{R}^{n\times(n+q)}$ and $Q_m = (0_{m\times(n+p)}, I_{m\times m}) \in \mathbb{R}^{m\times(n+q)}$ extracting the first n and last m entries of a vector in $\mathbb{R}^{n+q}$, respectively. The identity in $\mathbb{R}^k$ is $I_{k\times k}$ and $0_{k\times l}$ denotes a matrix of all zeros in $\mathbb{R}^{k\times l}$.

The adjoint program evaluates the adjoint elemental decomposition

$$\mathbf{x}_{(1)} := \mathbf{x}_{(1)} + \left\langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \right\rangle \ ,$$

where

$$\left\langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \right\rangle = P_n \cdot \Phi^1_{(1)}(\mathbf{x}, \Phi^2_{(1)}(\mathbf{v}^1, \ldots \Phi^q_{(1)}(\mathbf{v}^{q-1}, \mathbf{v}^q_{(1)}) \ldots))$$

and for given $\mathbf{v}^q_{(1)} = (0, \ldots, 0, y^0_{(1)}, \ldots, y^{m-1}_{(1)})$ assuming availability of adjoint elementals

$$\mathbf{v}^{i-1}_{(1)} = \Phi^i_{(1)}(\mathbf{v}^{i-1}, \mathbf{v}^i_{(1)}) \equiv \nabla \Phi^i(\mathbf{v}^{i-1})^T \cdot \mathbf{v}^i_{(1)} \text{ for } i = q, \ldots, 1.$$

By default the adjoint elemental decomposition is generated homogeneously with `dco/c++`. Special treatment of certain elementals (e.g, Φ^k) may become desirable or even essential, for example, to ensure feasibility of the memory requirement by checkpointing or preaccumulation [9], to exploit the implicit function theorem [3], to handle nonsmoothness [15] or even discontinuity, or to integrate parts of the computation running on a different compute platform (e.g, GPU) [14]. The resulting *gaps* (the missing tape of Φ^k in the *adjoint context* (the tape of F) need to be filled by custom versions of $\Phi^k_{(1)}$ yielding $\left\langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \right\rangle$ as

$$P_n \cdot (\nabla \Phi^1(\mathbf{x})^T \ldots \overbrace{\nabla \Phi^k(\mathbf{v}_{k-1})^T \underbrace{(\nabla \Phi^{k+1}(\mathbf{v}^k)^T \ldots (\nabla \Phi^q(\mathbf{v}^{q-1})^T \cdot \mathbf{v}^q_{(1)}) \ldots)}_{\mathbf{v}^k_{(1)}}}^{\mathbf{v}^{k-1}_{(1)}}) \ .$$

An API needs to be provided allowing for

$$\mathbf{v}^{k-1}_{(1)} := \Phi^k_{(1)}(\mathbf{v}^{k-1}, \mathbf{v}^k_{(1)}) \equiv \nabla \Phi^k(\mathbf{v}_{k-1})^T \cdot \mathbf{v}^k_{(1)}$$

to be evaluated based on custom required data to be recorded by an appropriately augmented primal version of Φ^k. For example, checkpointing the given implementation of Φ^k requires its input arguments to be stored in order to allow context-free reevaluation of Φ^k. The adjoint $\Phi^k_{(1)}$ restores the argument checkpoint followed by an augmented primal evaluation of Φ^k (e.g, generation of a local tape) and propagation of the adjoints (e.g, interpretation of the local tape). Moreover, communication with the context needs to be established by enabling access to in- and outputs of Φ^k and to the adjoints of all active arguments.

Our upcoming discussion of the external adjoint interface of `dco/c++` replaces the algorithmic adjoint of the Newton solver inside of the Burgers case study with its symbolic adjoint version. The general concept behind adjoint solvers for systems of nonlinear equations is illustrated in Fig. 7 based on prior work in [37]. An algorithmic adjoint (solid arrows) version of the implicitly defined function $\mathbf{x}(\mathbf{p})$ is generated with `dco/c++` by recording a tape of the Newton iterations followed by its interpretation yielding a correct adjoint of the approximate primal solution $\tilde{\mathbf{x}}$ with respect to $\mathbf{p}$. Finite differences (dotted arrow) may be able to validate this adjoint at a much higher computational cost. Alternatively, a symbolic adjoint (dashed arrows) results from differentiating the residual at the solution $\mathbf{x}^*$ followed by solving the

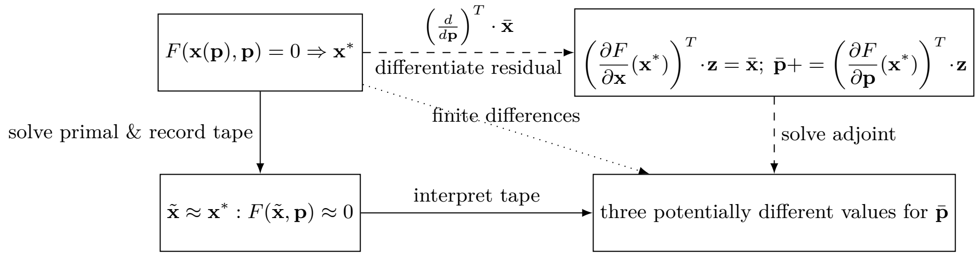


Abbildung 7. Kinds of differentiation: Algorithmic (solid arrows), approximate (dotted arrow), symbolic (dashed arrows)

adjoint equation. Solution of the latter at the given approximate primal solution $\tilde{\mathbf{x}}$ yields yet another approximation of the adjoint. The computational cost can be reduced as well as the memory requirement. See [37] and references therein for further details on symbolic adjoint nonlinear solvers.

Our sample code applies the above to the solution of Burgers' equation. Let therefore $g(\mathbf{y})$ denote the right-hand side of the ordinary differential equation resulting from spatial discretization of Burgers' equation as outlined in Sec. 2.1. Implicit Euler integration with time step Δt and given initial condition $\mathbf{y}^0$ computes iterates $\mathbf{y}^k$ for $k=1,...,\Delta t^{-1}$ as solutions of the system of nonlinear equations

$$f(\mathbf{y}^k,\mathbf{y}^{k-1})\equiv\mathbf{y}^k-\mathbf{y}^{k-1}-\Delta t\cdot g(\mathbf{y}^k)=0\ . \tag{8}$$

The new state $\mathbf{y}^k$ is parameterized by $\mathbf{y}^{k-1}$. It is computed by an implementation of Newton's method.

The basic `dco/c++` adjoint records all Newton iterations on the tape for later interpretation when computing the algorithmic adjoint of the implicit Euler scheme. Alternatively, symbolic differentiation of Eqn (8) at the solution $\mathbf{y}^{k^*}$ with respect to $\mathbf{y}^{k-1}$ yields

$$\frac{df(\mathbf{y}^{k^*},\mathbf{y}^{k-1})}{d\mathbf{y}^{k-1}}=\frac{\partial f(\mathbf{y}^{k^*},\mathbf{y}^{k-1})}{\partial\mathbf{y}^k}\cdot\frac{d\mathbf{y}^{k^*}}{d\mathbf{y}^{k-1}}+\underbrace{\frac{\partial f(\mathbf{y}^{k^*},\mathbf{y}^{k-1})}{\partial\mathbf{y}^{k-1}}}_{:=-I}=0\ .$$

We denote total and partial derivatives by "d" and "∂", respectively. Transposal and multiplication with

$$\mathbf{z}=-\left(\frac{\partial f(\mathbf{y}^{k^*},\mathbf{y}^{k-1})}{\partial\mathbf{y}^k}\right)^{-T}\cdot\mathbf{y}^k_{(1)}$$

from the right yields

$$\mathbf{y}^{k-1}_{(1)}=\left(\frac{d\mathbf{y}^{k^*}}{d\mathbf{y}^{k-1}}\right)^T\mathbf{y}^k_{(1)}=\left(\frac{\partial f(\mathbf{y}^{k^*},\mathbf{y}^{k-1})}{\partial\mathbf{y}^{k-1}}\right)^T\cdot\mathbf{z}=-I\cdot\mathbf{z}=-\mathbf{z}\ .$$

Hence the computation of $\mathbf{y}^{k-1}_{(1)}$ amounts to the solution of the linear system

$$\left(\frac{\partial f(\mathbf{y}^{k^*},\mathbf{y}^{k-1})}{\partial\mathbf{y}^k}\right)^T\cdot\mathbf{z}=\mathbf{y}^k_{(1)}\ . \tag{9}$$

The external adjoint interface of `dco/c++` requires its users to provide two versions for the section of the code (w.l.o.g. wrapped into a function call) subject to non-standard treatment. For the Burgers case study the call to an instance of the nonlinear solver

```
template<typename T>
void newton(const vector<T>& yp, vector<T>& y) { ... }
```

computing `y` as a function of `yp` is replaced by the following specialization for `T=DCO_T` to be called at time of recording:

Listing 1.11. Specialization of `newton` to be called during recording

```
1  template<>
2  void newton(const vector<DCO_T>& yp, vector<DCO_T>& y) {
3    DCO_TAPE_T* tape=dco::tape(yp);
4    DCO_EAO_T* D=tape->create_callback_object<DCO_EAO_T>();
5    vector<DCO_BT> ypv=D->register_input(yp);
6    vector<DCO_BT> yv=dco::value(y);
7    newton(ypv,yv);
8    D->write_data(yv);
9    y=D->register_output(yv);
10   tape->insert_callback(newton_adjoint,D);
11 }
```

Each active variable stores a pointer to its tape extracted in L1.11:3 and required for the creation of a callback object of external adjoint object type

```
typedef DCO_M::external_adjoint_object_t DCO_EAO_T
```

in L1.11:4. The callback object holds all information necessary for the evaluation of the local adjoint including references to local in- and outputs (L1.11:5,9) as well as required data (here the approximate solution of the nonlinear system; L1.11:8). The actual Newton iterations are performed passively (L1.11:7). Both `ypv` returned in L1.11:5 and `yv` hold values of type `DCO_BT=double`. The external adjoint object is inserted into the tape along with a pointer to the callback function (here `newton_adjoint`) to be called by the interpreter when reaching the current position in the tape (L1.11:10).

The interpreter expects an implementation of `newton_adjoint`, e.g,

Listing 1.12. Adjoint version of `newton` to be called during interpretation

```
1  void newton_adjoint(DCO_EAO_T* D) {
2    const vector<DCO_BT>& y=D->read_data<vector<DCO_BT>>();
3    vector<DCO_BT> ya(y.size()); D->get_output_adjoint(ya);
4    vector<DCO_BT> A((y.size()-2)*3+4,0);
5    dfdy(y,A,/*transpose=*/true);
6    LU(A); FS(A,ya); BS(A,ya);
7    D->increment_input_adjoint(ya);
8  }
```

The approximate solution of the nonlinear system is recovered (L1.12:2) followed by extracting adjoints of the local results from the enclosing tape (L1.12:3). Evaluation of the local adjoint amounts to solving the linear system in Eqn. (9) (L1.12:6–7) with the transposed tridiagonal system matrix computed in L1.12:5. The solution is used to increment the adjoint inputs (L1.12:7).

The main driver remains unchanged; see Listing 1.5. For the given scenario (see Sec. 2.2) we observe a speedup by a factor of 3.5 on our target computer. The tape size is reduced by a factor of roughly 30.

Seamless transition to second-order adjoints is supported. It amounts to instantiation of the above code with the second-order adjoint `dco/c++` data type introduced in Sec. 2.5 and linkage with the corresponding second-order adjoint driver. The implementation can be found on the `dco/c++` research website. For example, for $n = 500$ and $m = 1000$ the memory requirement of a second-order adjoint computation is reduced from more than 17GB to less than 400MB. A decrease in run time by a factor of three can be observed. Refer to the `dco/c++` research website for access to the sample code.

3.2 Multithreading

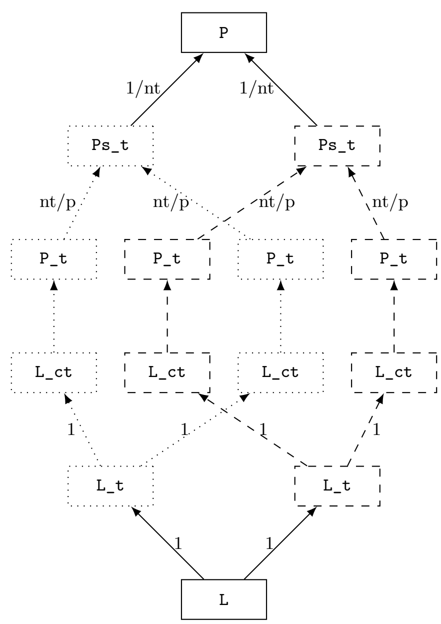


Abbildung 8. Parallel adjoints for LIBOR case study: Distribution of individual paths to the two threads is marked by dotted and dashed lines, respectively. For `nt` threads the `j`-th path is assigned to the thread with thread id `j%nt` (L1.13:23), that is, paths 0 and 2 are evaluted by thread 0 and paths 1 and 3 are evaluted by thread 1. Known local partial derivatives are attached to the corresponding edges, for example, the partial derivatives of the final values of the thread-local `Ps_t` with respect to the are known to be equal to pathwise payoffs `P_t` are known to be equal to nt/p for `p` paths evaluated by `nt` threads. The partial derivatives of the final payoff `P` with respect to the `Ps_t` are equal to `1/nt` yielding according to the chain rule `1/p` as the partial derivatives of `P` with respect to the pathwise payoffs `P_t`. Similarly, the copy operations `L` → `L_t` and `L_t` → `Lc_t` yield unit partial derivatives, respectively.

Adjoints for Multithreading The LIBOR case study features a high degree of concurrency due to mutually independent Monte Carlo path simulations. It lends itself to the exploitation of shared memory parallelism using OpenMP. While a multithreaded implementation of the primal is rather straight forward the corresponding adjoint requires more careful treatment.

Fig. 8 depicts a graphical representation of the data flow for a simplified scenario involving four Monte Carlo paths to be evaluated by two threads. This setup represents a special case of the implementation shown in Listing 1.13. Thread-local copies `L_t` of the common input `L` are generated (L1.13:18). Read-only inputs are required for correct access to the adjoint results as outlined previously. Hence each path is evaluated on a local copy `Lc_t` (L1.13:24–25) yielding a local payoff `P_t`. Thread-local sums `Ps_t` over the pathwise payoffs are built (L1.13:29) followed by averaging (L1.13:32) and summation over all threads to obtain an estimate of the primal payoff `P` (L1.13:34).

Listing 1.13. Thread-parallel recording of multiple tapes

```
1  // ... L6:1-3
2
3  #include "dco.hpp"
4  typedef dco::ga1sm<double> DCO_M;
5  typedef DCO_M::type DCO_T;
6  typedef DCO_M::tape_t DCO_TAPE_T;
7
8  #include <omp.h>
9
10 void libor(vector<double>& L, double& P, vector<double>& dPdL,
11            const vector<vector<double>>& Z) {
12   int nt=omp_get_max_threads();
13   P=0;
14   #pragma omp parallel
15   {
16     int tid=omp_get_thread_num();
17     DCO_TAPE_T *tape=DCO_TAPE_T::create();
18     vector<DCO_T> L_t(n,0); for (int i=0;i<n;i++) L_t[i]=L[i];
19     tape->register_variable(L_t);
20     DCO_T P_t=0; double Ps_t=0;
21     DCO_TAPE_POSITION_T tpos=tape->get_position();
22     for (int j=0;j<p;j++) {
23       if(j%nt!=tid) continue;
24       vector<DCO_T> Lc_t(L_t);
25       path_calc(j,Lc_t,Z); portfolio(Lc_t,P_t);
26       tape->register_output_variable(P_t);
27       dco::derivative(P_t)=1./p;
28       tape->interpret_adjoint();
29       Ps_t+=dco::value(P_t);
30       tape->reset_to(tpos);
31     }
32     Ps_t/=p;
33     #pragma omp atomic
34     P+=Ps_t;
35     for (int i=0;i<n;i++) {
36       #pragma omp atomic
37       dPdL[i]+=dco::derivative(L_t[i]);
```

```
38      }
39      DCO_TAPE_T::remove(tape);
40    }
41  }
42
43  // ... main() calls libor(...)
```

It remains to compute adjoints of P_t with respect to Lc_t for all paths. Pair-wise independence of the paths yields mutually independent adjoints. Each thread allocates a local tape (L1.13:17). Support for multiple tapes is provided by dco/c++ in ga1sm mode (L1.13:4) enabling thread-safe implementations of tape-based adjoints. Thread-local active inputs L_t are registered with the tape (L1.13:19) followed by recording individual paths (L1.13:24–26) and immediate interpretation (L1.13:28) [21] for adjoint local payoffs set equal to 1/p (L1.13:27). Subsequent recordings use the same tape memory as a result of resetting the tape pointer to the position following the local active inputs (L1.13:21,30). The latter are incremented by repeated interpretations yielding correct adjoints for a sequence of paths. Both the reductions of the final payoff (L1.13:34) and of its gradient with respect to the initial LIBOR rates (L1.13:37) require atomic handling due to potential race conditions.

The relative simplicity of the given implementation is due to the Monte Carlo section not being followed by further computation on the payoff P. Adjoints of the path-local payoffs are known to be equal to 1/p at compile time. They do not depend on adjoints to be computed prior to their evaluation. Otherwise checkpointing would have to be applied in order to delay the adjoint Monte Carlo simulation until after the adjoint payoff is available. The read-only initial LIBOR rates L can be used eliminating the need for additional checkpointing memory. Checkpointing may also become necessary in case of more complex individual path simulations whose tape sizes may exceed to available memory resources.

The run time of basic adjoint mode is 2.6s based on a tape of size 1.8GB. While the run time is not reduced significantly when using pathwise taping, the size of the tape is reduced to 40KB. Shared memory parallelization using four threads increases the tape size by four. A speedup of about three can be observed. Seamless transition to second- (and higher-)order adjoint modes is guaranteed; see corresponding example on the dco/c++ research website.

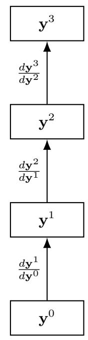
Multithreading for Adjoints Multithreading can also be applied to several concurrent interpretations of the same tape. dco/c++ supports the allocation of multiple thread-local vectors of adjoints sharing a single, sequentially recorded tape. As an example we consider the computation of several inner rows of the Jacobian of the Burgers case study illustrated in Fig. 9 for the two inner rows of a 4×4 Jacobian. Obviously, adjoint mode would only be used in practice if the number of rows turned out to be substantially lower than the number of active inputs resulting from the given spatial discretization scheme. Fig. 9 (b) shows a representation of the tape to be interpreted twice using vectors of adjoints shown in Fig. 9 (c) and (d). The two interpretations can be performed concurrently by two threads with adjoint final states set equal to the second and third Cartesian basis vectors, respectively.

Our implementation in Listing 1.14 uses a band width bw defined in burgers.h to select the 2·bw target rows within the Jacobian. All relevant code is restricted to the main driver function.

Listing 1.14. Thread-parallel evaluation of multiple adjoints for single tape

$$A = \left(\begin{array}{cccc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ \hline a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ \hline a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{array}\right)$$

(a)



(b)

$$\begin{pmatrix} a_{1,0} \\ a_{1,1} \\ a_{1,2} \\ a_{1,3} \\ \vdots \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

(c)

$$\begin{pmatrix} a_{2,0} \\ a_{2,1} \\ a_{2,2} \\ a_{2,3} \\ \vdots \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

(d)

Abbildung 9. Parallel accumulation of two inner rows of the Jacobian $A \in \mathbb{R}^{n\times n}$ (a) for the Burgers case study with $n = 4$, $m = 3$ and using two threads: Tape of evolution of state with preaccumulated local Jacobians of individual time steps (b); second row computed by first thread using vector of scalar adjoints (c); third row computed by second thread using vector of scalar adjoints (d).

```
// ... L5:1-8

#include <omp.h>

int main() {
  omp_set_num_threads(nt);
  vector<DCO_AT> y(n,0);
  for (int i=1;i<n-1;i++) y[i]=sin(2*PI*i/n);
  DCO_AM::global_tape=DCO_TAPE_T::create();
  DCO_AM::global_tape->register_variable(y);
  vector<DCO_AT> yc(y);
  burgers(yc);
  for(int i=n/2-bw;i<n/2+bw;i++)
    DCO_AM::global_tape->register_output_variable(yc[i]);
  vector<vector<double>> M_dydy(2*bw,vector<double>(n));
#pragma omp parallel
  {
    int tid=omp_get_thread_num();
    dco::adjoint_vector<DCO_TAPE_T,double,2*bw/nt>
                                          av(DCO_AM::global_tape);
    for(int i=n/2-bw,j=0;i<n/2+bw;i++) {
      if(i%nt!=tid) continue;
      av.derivative(yc[i])[j]=1;
      j++;
    }
    av.interpret_adjoint();
    for(int i=n/2-bw,k=0;i<n/2+bw;i++) {
      if(i%nt!=tid) continue;
      for(int j=1;j<n-1;j++)
        M_dydy[i-n/2+bw][j]=av.derivative(y[j])[k];
```

```
31        k++;
32      }
33    }
34    DCO_TAPE_T::remove(DCO_AM::global_tape);
35    // ... output and return
36  }
```

Declaration and initialization of the initial state in (L1.14:7–8) is followed by the recording of the tape as previously discussed (L1.14:9–14). The target rows of the Jacobian are stored in an appropriately declared matrix (L1.14:15). Within the parallel section (L1.14:16–33) each of the `nt` threads allocates 2·`bw`/`nt` vectors of adjoints linked to the single global tape and with elements of type `double` (L1.14:19–20). Adjoint final states are set equal to the Cartesian basis vectors yielding the corresponding row of the Jacobian (L1.14:21–25) by thread-local tape interpretation (L1.14:26). The results are stored (L1.14:27–32)) prior to leaving the parallel section and deallocation of the global tape (L1.14:34).

For `bw` $= 32$, $n = 500$, and $m = 1000$ basic adjoint mode takes 12s. Shared memory parallelization with four threads yields a speedup of roughly three. Ongoing investigations in the context of larger use cases are expected to provide further insight into the tuning of multithreading applied to separate vectors of adjoints. Transition to second and higher order is straight forward as illustrated by examples on the `dco/c++` research website.

3.3 Tape Compression by Preaccumulation

The main challenge faced by all users of algorithmic adjoint software including users of `dco/c++` is the often infeasible memory requirement of methods for implementing the reversal of the data flow, for example, by a tape combined with a vector of adjoints. Checkpointing is probably the preferred method for limiting the memory footprint at the expense of additional computation. Corresponding support is provided by `dco/c++`, for example, through its external adjoint interface; see Sec. 3.1. Alternatively, preaccumulation of local Jacobians can help to ensure feasibility of an adjoint solution.

`dco/c++` offers various ways to replace certain sections of the tape with the corresponding local Jacobian including its external adjoint interface and direct insertion of local partial derivatives into the tape not discussed in detail in this report; see `dco/c++` user guide for further details. The following solution to preaccumulation has been developed as part of an ongoing effort to simplify the user interface to `dco/c++` wherever possible. Isolated (free of side effects) parts of the tape can be replaced with their corresponding local Jacobians by using only a few instructions as illustrated in Listing 1.15 for the LIBOR case study. Specific modifications are limited to the `libor` routine. The enclosing driver program remains unchanged; see Listing 1.6.

Listing 1.15. Reduction of tape size through preaccumulation of local Jacobians

```
void libor(const vector<DCO_T>& L, DCO_T& P,
           const vector<vector<double>>& Z) {
  DCO_T Ps=0;
  DCO_M::jacobian_preaccumulator_t jp(dco::tape(L));
  for (int j=0;j<p;j++) {
    jp.start();
    vector<DCO_T> Lc(L);
```

```
8      path_calc(j,Lc,Z); portfolio(Lc,P);
9      jp.register_output(P);
10     jp.finish();
11     Ps+=P;
12   }
13   P=Ps/p;
14 }
```

In Listing 1.15 local gradients of path-local payoffs `P` with respect to the path-local copies `Lc` of the initial LIBOR rates are preaccumulated. The corresponding local tapes of the entire path calculation and associated evaluation of the portfolio (L1.15:8) are replaced by a single gradient, respectively, yielding a substantial decrease in overall tape size. Therefore a Jacobian preaccumulator object needs to be created for the target tape (L1.15:4). A pointer to the latter can be extracted from any active variable by using the `dco::tape` routine. For each path preaccumulation is initiated by setting a start position (L1.15:6). Local active outputs need to be registered explicitly following the recording of the local tape (L1.15:9). The actual preaccumulation is triggered by a call to the `finish` member function (L1.15:10) of the Jacobian preaccumulator object. Interpretation of the local tape with adjoints of its m outputs set equal to the Cartesian basis vectors in $\mathbb{R}^m$ (here: adjoint of scalar local output set equal to one) replaces the local tape with the local Jacobian (here: gradient).

Preaccumulation applied to a basic adjoint of the LIBOR case study results in a tape of size 28MB. The run time of the corresponding gradient computation is 2.7s. Pathwise taping combined with preaccumulation reduces the tape size to 1.6KB while no significant improvement in run time can be observed. Shared memory parallelization with four threads reduces the overall run time by a factor of two at the expense of an increase in memory requirement by a factor of four. Again, the transition to second and higher order does not pose any conceptual challenges. Corresponding sample codes can be found on the `dco/c++` research website.

4 Above and Beyond `dco/c++`

Integration of (adjoint) AD into a nontrivial numerical simulation software environment remains a demanding effort. The benefits in terms of feasibility of derivative-based methods for parameter sensitivity analysis and calibration, large-scale nonlinear optimization and uncertainty quantification typically outweigh the investment. However, it must be recognized that this investment is not a one-off exercise. AD has a significant impact on software development and maintenance procedures. Sensitivity information can and should be included into unit and regression test hierarchies. Coding guidelines may have to be adapted to ensure robustness of new versions of the code base with respect to its augmented semantics.

Taking all this into account, the level of professionalism expected from an AD software has risen over recent years. We have been investing into a state of the art `dco/c++` software engineering environment in order to meet these expectations as formulated by both commerical and academic partners. Crucial elements include cross-platform overnight builds[15], a unit and regression test suite, extensive user documentation and quality assurance mechanisms implemented in collaboration with our partners at NAG.

[15] see www.stce.rwth-aachen.de/buildbot/dco

`dco/c++` forms the basis for a number of extensions targeting other programming models and languages. Inspired by earlier efforts to handle `Fortran` by providing a suitable wrapper to an underlying C++ solution (ADOL-F [43]) `dco/fortran` has been developed as a `Fortran` front-end to `dco/c++`. One of its main target applications is the NAG Library. In collaboration with NAG an AD version of the NAG Library is under development including algorithmic adjoints based on `dco/fortran` as well as symbolic adjoints of implicit functions (e.g, [non]linear equation solvers) and hybrid adjoint routines combining both algorithmic and symbolic elements. `dco/fortran` is also used to derive and maintain adjoint versions of Telemac [49] and ICON [50] based on prior work on the NAG AD Fortran Compiler [38]. Preliminary studies for other programming languages include `dco/matlab` and `dco/python`. Both tools are currently used in a purely experimental regime.

Working with numerous partners in academia and industry we have been confronted with requests to extend algorithmic adjoint capabilities to GPUs. The traditional approach of allocating substantial amounts of (tape) memory dynamically turned out to be infeasible for massively parallel accelerators featuring a relatively low amount of main memory compared to their computational peak performance. Analysis of the technical challenges resulted in the concept of *meta adjoint programming* implemented by `dco/map` [29]. A domain-specific language is combined with custom preprocessing of the primal to yield highly efficient adjoint code on both CPUs and GPUs. Coupling of CPUs and GPUs is supported through combination of `dco/c++` and `dco/map`. First applications of `dco/map` show highly promising results.

5 Conclusion

Algorithmic adjoint methods for large-scale gradient-based numerical simulation and incorporating symbolic as well as approximate approaches wherever appropriate or necessary can be expected to play an increasingly important role in Computational Science, Engineering and Finance. While not being the easiest programming language to master C++ features a degree of flexibility and semantic richness which is likely to make it the first choice for a large number of ongoing and future large-scale and long-term simulation software development efforts. Software for Algorithmic Differentiation of C++ code will remain a fundamental element of the numerical simulation toolbox.

The `dco/c++` software is a central ingredient of numerous ongoing academic and commerical projects. Its proven robustness, support for post C++11 standards, efficiency, innovation and sutainability represents the basis for further development addressing substantial challenges within an ever changing computational environment. Ongoing work includes extensions toward vector and matrix derivative types, implementation of *adjoint code design patterns* [35] and further support for parallelism. However, these improvements will not eliminate the need for user knowledge. A perfect algorithmic adjoint remains the result of a powerful tool applied by an expert user.

A Feature Summary

We list all features of the current release of `dco/c++` without further explanation. Contact

info@stce.rwth-aachen.de

to find out more.

- stable
  - generic first- and higher-order tangents (scalar mode / vector mode)
  - generic first- and higher-order adjoints (scalar mode / vector mode)
  - blob tape, chunk tape, file tape
  - activity analysis for recording
  - thread-safety by support for multiple tapes
  - external adjoint interface, i.e. support for user-callbacks
  - adjoint code module interface
  - direct tape manipulation (user-defined gradients)
  - tape compression (tape-based preaccumulation of gradients/Jacobians)
  - Adjoint MPI support
  - vector of adjoints separated from tape; benefits:
    - arbitrary data type for adjoint vector
    - vector and scalar modes with same data type (no re-instantiation of code base)
    - parallel use of same tape w/ multiple adjoint vectors (e.g. using OpenMP)
  - first- and second-order sparsity pattern detection
  - vector adjoint mode with multiple tape support: ga1vm
  - `dco/map`: meta adjoint programming (different piece of software)
  - C++11 support
- experimental
  - adjoint code generation during overloading (full unrolling)
  - debugging using a combined data type: finite differences vs. tangents vs. adjoints; benefits:
    - automatic check of externally implemented adjoints (tangent/adjoint identity)
    - weak discontinuity detection (using finite differences)
    - control flow discontinuity detection
- prototyped (outlook for v4.0)
  - vectorization support (AVX2, AVX-512)
  - C++11 support
  - code instrumentation for bidirectional dataflow analysis, debugging during tape interpretation, determination of numerical intensive kernels
  - Linux: HPC version using low-level memory access; benefits:
    - active datatype occupies same memory as passive datatype: simple decay possible; zero-copy when calling passive kernels
    - fixed size adjoint memory (mem(adjoints) = mem(primals)); only tape grows (sequential write/read only $\rightarrow$ well suited for writing/reading to/from file)
  - Linux: OpenMP support w/ automatic spawning and reducing of per-thread tapes
  - Linux: fully automatic checkpointing support

Literatur

1. D. Bailey, Y. Hida, X. Li, and B. Thompson. Arprec: An arbitrary precision computation package. Technical report, 2002.
2. A. Baydin, B. Pearlmutter, and A. Radul. Automatic differentiation in machine learning: A survey. *CoRR*, abs/1502.05767, 2015.
3. B. Bell and J. Burke. Algorithmic differentiation of implicit functions and optimal values. In *[4]*, pages 67–77. Springer, 2008.
4. C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, number 64 in Lecture Notes in Computational Science and Engineering (LNCSE). Springer, 2008.
5. D. Bommes, H. Zimmer, and L. Kobbelt. Practical mixed-integer optimization for geometry processing. In *Curves and Surfaces*, Lecture Notes in Computer Science, pages 193–206. Springer, 2012.
6. A. Brace, D. Gatarek, and M. Musiela. The market model of interest rate dynamics. *Mathematical Finance*, 7:127–147, 1997.
7. J. Burgers. Mathematical examples illustrating relations occurring in the theory of turbulent fluid motion. *Verhandelingen der Koninklijke Nederlandse Akademie van Wetenschappen, Afdeeling Natuurkunde*, 2(17):1–53, 1939.
8. G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science. Springer, 2002.
9. M. Fagan and A. Carle. Reducing reverse-mode memory requirements by using profile-driven checkpointing. *Future Generation Computer Systems*, 21(8):1380–1390, 2005.
10. S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, editors. *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2012.
11. D. Gendler, U. Naumann, and B. Christianson. Automatic differentiation of Assembler code. In *Proceedings of the IADIS International Conference on Applied Computing*, pages 431–436. IADIS, 2007.
12. M. Giles and P. Glasserman. Smoking adjoints: Fast Monte Carlo Greeks. *Risk*, pages 88–92, January 2006.
13. P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2003.
14. F. Gremse, A. Hoefter, L. Razik, F. Kiessling, and U. Naumann. GPU-accelerated adjoint algorithmic differentiation. *Computer Physics Communications*, 200:300–311, 2016.
15. A. Griewank. On stable piecewise linearization and generalized algorithmic differentiation. *Optimization Methods and Software*, 28(6):1139–1178, 2013.
16. A. Griewank, D. Juedes, and J. Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.
17. A. Griewank and U. Naumann. Accumulating Jacobians as chained sparse matrix products. *Mathematical Programming*, 95(3):555–571, 2003.
18. A. Griewank and A. Walther. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation, Seocnd Edition*. Number OT105 in Other Titles in Applied Mathematics. SIAM, 2008.
19. R. Hannemann, W. Marquardt, U. Naumann, and B. Gendler. Discrete first- and second-order adjoints and automatic differentiation for the sensitivity analysis of dynamic models. *Procedia Computer Science*, 1(1):297 – 305, 2010.
20. R. Hannemann-Tamás, J. Tillack, M. Schmitz, M. Förster, J. Wyes, K. Nöh, E. von Lieres, U. Naumann, W. Wiechert, and W. Marquardt. First- and second-order parameter sensitivities of a metabolically and isotopically non-stationary biochemical network model. In *Electronic Proceedings of the 9th International Modelica Conference, Munich, Sep 3-5, 2012*. Modelica Association, 2012.
21. L. Hascoët, S. Fidanova, and C. Held. Adjoining independent computations. In *[8]*, Computer and Information Science, chapter 35, pages 299–304. Springer, New York, NY, 2002.
22. L. Hascoët, U. Naumann, and V. Pascual. To-Be-Recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems*, 21:1401–1417, 2005.
23. L. Hascoët and V. Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software*, 39(3):20:1–20:43, 2013.
24. R. Hogan. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Transactions on Mathematical Software*, 40(4):26:1–26:24, jun 2014.
25. P. Hovland, U. Naumann, and B. Norris. An XML-based platform for semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications*, pages 530–538. ACTA Press, 2002.

26. J. Huang, Y. Tong, H. Wei, and H. Bao. Boundary aligned smooth 3d cross-frame field. *ACM Trans. Graph.*, 30(6):143:1–143:8, 2011.
27. P. Lax and B. Wendroff. Systems of conservation laws. *Communications on Pure and Applied mathematics*, 13(2):217–237, 1960.
28. J. Lotz. *Hybrid Approaches to Adjoint Code Generation with* `dco/c++`. PhD thesis, RWTH Aachen University, 2016.
29. J. Lotz, K. Leppkes, U. Naumann, and J. du Toit. Meta adjoint programming in C++. Technical Report AIB-2017-07, Department of Computer Science, RWTH Aachen University, 2017.
30. J. Lotz, U. Naumann, R. Hannemann-Tamás, T. Ploch, and A. Mitsos. Higher-order discrete adjoint ODE solver in C++ for dynamic optimization. *Procedia Computer Science*, 51:256–265, 2015.
31. J. Lotz, U. Naumann, and J. Ungermann. Hierarchical algorithmic differentiation: A case study. In *[10]*, pages 187–196. Springer, 2012.
32. D. Lu. *The XVA of Financial Derivatives: CVA, DVA and FVA Explained*. Springer, 2016.
33. M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 10 Programming Guide*. Maplesoft, 2005.
34. U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.* Number SE24 in Software, Environments, and Tools. SIAM, 2012.
35. U. Naumann. Adjoint code design patterns. In *Seventh International Conference on Algorithmic Differentiation, Oxford, UK*, 2016. Extended abstract. Full paper under review.
36. U. Naumann and J. du Toit. Adjoint algorithmic differentiation tool support for typical numerical patterns in computational finance. *Journal of Computational Finance*, 2016. To appear.
37. U. Naumann, J. Lotz, K. Leppkes, and M. Towara. Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations. *ACM Transactions on Mathematical Software*, 41:1–26, 2015.
38. U. Naumann and J. Riehme. A differentiation-enabled Fortran 95 compiler. *ACM Transactions on Mathematical Software*, 31(4):458–474, December 2005.
39. A. Noack and A. Walther. Adjoint concepts for the optimal control of Burgers equation. *Comput. Optim. Appl.*, 36(1):109–133, 2007.
40. A. Pfadler. Computing sensitivities of CVA using adjoint algorithmic differentiation. Master's thesis, University of Oxford, 2015.
41. E. Phipps and R. Pawlowski. Efficient expression templates for operator overloading-based automatic differentiation. In *[10]*, volume 87 of *Lecture Notes in Computational Science and Engineering*, pages 309–319. Springer, Berlin, 2012.
42. M. Sambridge, P. Rickwood, N. Rawlinson, and S. Sommacal. Automatic differentiation in geophysical inverse problems. 170:1 – 8, 07 2007.
43. D. Shiriaev, A. Griewank, and J. Utke. A user guide to ADOL–F: Automatic differentiation of Fortran codes. Tech. Report IOKOMO–04–1995, TU Dresden, Dept. of Mathematics, 1996.
44. O. Toon, R. Turco, D. Westphal, R. Malone, and M. Liu. A multidimensional model for aerosols: Description of computational analogs. *Journal of the Atmospheric Sciences*, 45(15):2123–2144, 1988.
45. M. Towara and U. Naumann. A discrete adjoint model for OpenFOAM. *Procedia Computer Science*, 18:429–438, 2013.
46. M. Towara, M. Schanen, and U. Naumann. MPI-parallel discrete adjoint OpenFOAM. *Procedia Computer Science*, 51:19–28, 2015.
47. J. Ungermann, J. Blank, J. Lotz, K. Leppkes, Lars Hoffmann, T. Guggenmoser, M. Kaufmann, P. Preusse, U. Naumann, and M. Riese. A 3-d tomographic retrieval approach with advection compensation for the air-borne limb-imager GLORIA. *Atmospheric Measurement Techniques*, 4(11):2509–2529, 2011.
48. J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch. OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, 34(4):18:1–18:36, July 2008.
49. C. Villaret, R. Kopmann, J. Riehme, D. Wyncoll, U. Merkel, and U. Naumann. First-order uncertainty analysis using algorithmic differentiation of the Telemac-2D/Sisyphe morphodynamic model. *Computers & Geosciences*, 90(B):144–151, 2015.
50. A. Vlasenko, P. Korn, J. Riehme, and U. Naumann. Estimation of data assimilation error: A shallow-water model study. *Monthly Weather Review*, 142:2502–2520, 2014.
51. M. Voßbeck, R. Giering, and T. Kaminski. Development and first applications of TAC++. In *[4]*, pages 187–197. Springer, 2008.
52. A. Wächter and L. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57, 2006.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

http://aib.informatik.rwth-aachen.de/

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: biblio@informatik.rwth-aachen.de

2015-01 * Fachgruppe Informatik: Annual Report 2015
2015-02 Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications
2015-05 Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity
2015-06 Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"
2015-07 Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon
2015-08 Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization
2015-09 Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models
2015-11 Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus
2015-12 Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic
2015-13 Andreas Polzer: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2015-14 Niloofar Safiran and Uwe Naumann: Symbolic vs. Algorithmic Differentiation of GSL Integration Routines
2016-01 * Fachgruppe Informatik: Annual Report 2016
2016-02 Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems
2016-03 Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs
2016-04 Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution
2016-05 Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization
2016-06 Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud
2016-07 Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis
2016-08 Klaus Leppkes, Johannes Lotz, and Uwe Naumann: Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features
2016-09 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic
2016-10 Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering
2017-01 * Fachgruppe Informatik: Annual Report 2017
2017-02 Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity

2017-04 Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE
2017-05 Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems
2017-06 Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy
2017-07 Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++
2017-08 Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts
2017-09 Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing
2018-01 * Fachgruppe Informatik: Annual Report 2018
2018-02 Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2018-03 Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices
2018-04 Andreas Ganser: Operation-Based Model Recommenders

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.