# dco/c++ User Guide

version 3.8.1

The Numerical Algorithms Group Ltd. (NAG)
30 St Giles'
Oxford OX1 3LE, United Kingdom

www: www.nag.com
contact email: info@nag.co.uk

and

Informatik 12
Software and Tools for Computational Engineering (STCE)
Department of Computer Science
RWTH Aachen University
D-52056 Aachen, Germany

www: www.stce.rwth-aachen.de
contact email: naumann@stce.rwth-aachen.de

# Copyright Statement

Produced by NAG and STCE.

Version 3.8.1 released November 2022

# Preface

`dco/c++` is a highly flexible and efficient implementation of first- and higher-order tangent and adjoint Algorithmic Differentiation (AD) by operator overloading in C++. It combines a cache-optimized internal representation generated with the help of C++ expression templates with an intuitive and powerful application programmer interface (API). `dco/c++` has been applied successfully to a growing number of numerical simulations in the context of, for example, large-scale parameter calibration and shape optimization.

Let us assume that you regularly run numerical simulations of a scalar objective $y$ (for example, the price of a financial product) depending on $n$ uncertain parameters $\mathbf{x} \in \mathbb{R}^n$ (for example, market volatilities). Suppose that a single run of the given implementation of the (pricing) function $y = f(\mathbf{x})$ as a C++ program takes one minute on the available computer. In addition to the value $y$ you might be interested in first derivatives of $y$ with respect to all elements of $\mathbf{x}$. Finite difference approximation of the corresponding $n$ gradient entries requires $O(n)$ evaluations of $f$. Their accuracy suffers from truncation and/or numerical effects due to cancellation and rounding in finite precision floating-point arithmetic. Moreover, if, for example, $n = 1000$, then the approximation of the gradient will take at least 1000 minutes (more than 16 hours). In adjoint mode `dco/c++` can be expected to take less than 20 (often less than 10 minutes) minutes for the accumulation of the same gradient with machine accuracy.

AD can be implemented manually, that is, given an implementation of an arbitrary objective function AD experts should be able to write a corresponding adjoint version the runtime of which is likely to undercut that of tool-based solution. This process can be tedious, error-prone, and extremely hard to debug. More importantly, it does not meet basic requirements for modern software engineering such as sustainability and maintainability. Each modification in the original code implies the need for corresponding modifications in the adjoint. To keep both codes consistent will become at least challenging.

`dco/c++` has been designed for use with large-scale real-world simulation code. Robust and efficient adjoints of arbitrary complexity can be evaluated within the available memory resources through combinations of checkpointing, symbolic differentiation of implicit functions, and integration of adjoint source code into a `dco/c++` adjoint. Users are encouraged to build libraries of optimized domain-specific higher-level intrinsics to be linked with `dco/c++` adjoints. The adjoint callback interface of `dco/c++` facilitates use of accelerators (such as GPUs) as well as the coupling with established and highly optimized numerical libraries. A growing set of methods from the NAG Library are supported as intrinsics requiring linkage with the separate NAG AD Library.

**Disclaimer:** This User Guide is driven by examples. Many of them are self-explanatory. Others may require more in-depth descriptions of the semantics behind the `dco/c++` syntax. Further information will be added to upcoming versions of this document.

This User Guide targets readers with a very good understanding of AD. See [7] for an introduction. More advanced issues are discussed in [4]. The AD community maintains the web portal `www.autodiff.org` with an extensive bibliography on the subject.

# Contents

**1 Summary of Features: v3.8.1** **6**

**2 General Remarks / Essentials** **12**

2.1 C++ Standard . . . . . . . . . . 12
2.2 Memory Allocation . . . . . . . . . . 12
2.3 Parallelization . . . . . . . . . . 12
2.4 Performance . . . . . . . . . . 13
2.5 Type Casting . . . . . . . . . . 13
2.6 File I/O . . . . . . . . . . 13
2.7 Feature Selection . . . . . . . . . . 13
2.8 Coupling with Other Template Libraries . . . . . . . . . . 15

**3 General Functions and Traits** **17**

**4 Features by Example** **21**

4.1 First-order Tangent Mode . . . . . . . . . . 21
4.2 First-order Adjoint Mode . . . . . . . . . . 23
4.3 Combined First-order Tangent and Adjoint Mode . . . . . . . . . . 25
4.4 Vector Type . . . . . . . . . . 28
4.5 Vectorized Tangent Type . . . . . . . . . . 30
4.6 Vectorized Adjoint Type . . . . . . . . . . 32
4.7 First-order Adjoint Mode: Blob/Chunk Tapes . . . . . . . . . . 34
4.8 First-order Adjoint Mode: File Tape . . . . . . . . . . 35
4.9 First-order Adjoint Mode: Tape Data Types . . . . . . . . . . 36
4.10 Second-order Tangent Mode . . . . . . . . . . 37
4.11 Second-order Adjoint Mode . . . . . . . . . . 39
4.12 Approximate Second-order Adjoint Mode . . . . . . . . . . 41
4.13 Third-order Tangent Mode . . . . . . . . . . 44
4.14 Third-order Adjoint Mode . . . . . . . . . . 48
4.15 First-order Tangent Vector Mode . . . . . . . . . . 51
4.16 First-order Adjoint Vector Mode . . . . . . . . . . 53

4.17 Adjoint First-order Scalar Mode: Multiple Tapes . . . 55
4.18 First-order Adjoint Vector Mode: Multiple Tapes . . . 55
4.19 Binary Compatible Passive Type: gbcp . . . 57
4.20 First-order Adjoint Mode: Basic Checkpointing . . . 60
4.21 Second-order Adjoint Mode: Basic Checkpointing . . . 63
4.22 First-order Adjoint Mode: Equidistant Checkpointing of Evolutions . . . 66
4.23 Second-order Adjoint Mode: Equidistant Checkpointing of Evolutions . . . 68
4.24 First-order Adjoint Mode: Recursive Checkpointing of Evolutions . . . 70
4.25 Second-order Adjoint Mode: Recursive Checkpointing of Evolutions . . . 75
4.26 First-order Adjoint Mode: Checkpointing Ensembles . . . 78
4.27 Second-order Adjoint Mode: Checkpointing Ensembles . . . 80
4.28 First-order Adjoint Mode: Embedding Adjoint Source Code . . . 82
4.29 Second-order Adjoint Mode: Embedding First-order Adjoint Code . . . 84
4.30 First-order Adjoint Vector Mode: Embedding Adjoint Source Code . . . 87
4.31 First-order Adjoint Mode: Local Preaccumulation Using First-order Tangent Code 90
4.32 Second-order Adjoint Mode: Local Preaccumulation Using First-order Tangent Code 92
4.33 Adjoint Mode: Edge Insertion for Preaccumulation of Local Gradients . . . 94
4.34 First-order Adjoint Mode: User-Defined Adjoints . . . 95
4.35 Second-order Adjoint Mode: User-Defined First-order Adjoints . . . 97
4.36 First-order Adjoint Mode: Local Jacobian Preaccumulation . . . 99
4.37 Second-order Adjoint Mode: Local Jacobian Preaccumulation . . . 101
4.38 First-order Adjoint Mode: Multiple Adjoint Vectors . . . 103
4.39 Modulo Adjoint Propagation . . . 105
4.40 Optimization . . . 105
4.41 Sparse Tape Interpretation . . . 107
4.42 First-order Adjoint Mode: Custom Tape Interpreation Hook . . . 108
4.43 Logging . . . 111
4.44 Explicit Type Cast to Specified Type . . . 112
4.45 Huge Pages . . . 112
4.46 Thread-Parallel Taping: Monte Carlo . . . 113

**A Status of User Interface 115**
A.1 Gold . . . 115
A.2 Silver . . . 115
A.3 Bronze . . . 122
A.4 Undefined . . . 124

**B Case Study: Diffusion 126**
B.1 Purpose . . . 126
B.2 Function . . . 140

B.3 Observations . . . . . 140
B.4 Gradient . . . . . 141
B.5 Hessian . . . . . 153

**C Case Study: Race 166**

C.1 Purpose . . . . . 166
C.2 Gradient . . . . . 167
C.3 Hessian . . . . . 170

# Chapter 1

# Summary of Features: v3.8.1

We consider implementations of multivariate vector functions

$$f : D^n \times D^{n'} \to D^m \times D^{m'} : (\mathbf{y}, \mathbf{y}') = f(\mathbf{x}, \mathbf{x}')$$

as computer programs over some base data type $D$ (for example, single or higher precision floating-point data, intervals, convex/concave relaxations, vectors/ensembles).[1] The $n$ active (also: independent) and $n'$ passive inputs are mapped onto $m$ active (also: dependent) and $m'$ passive outputs. The given implementation is assumed to be $k$ times continuously differentiable at all points of interest implying the existence and finiteness of the Jacobian

$$\nabla f = \nabla_{\mathbf{x}} f(\mathbf{x}, \mathbf{x}') \equiv \frac{\partial \mathbf{y}}{\partial \mathbf{x}}(\mathbf{x}, \mathbf{x}') \in D^{m \times n},$$

the Hessian

$$\nabla^2 f = \nabla^2 f(\mathbf{x}, \mathbf{x}') \equiv \frac{\partial^2 \mathbf{y}}{\partial \mathbf{x}^2}(\mathbf{x}, \mathbf{x}') \in D^{m \times n \times n},$$

if $k \geq 2$, and of potentially higher derivative tensors

$$\nabla^k f = \nabla^k f(\mathbf{x}, \mathbf{x}') \equiv \frac{\partial^k \mathbf{y}}{\partial \mathbf{x}^k}(\mathbf{x}, \mathbf{x}') \in D^{m \times n \times_k \overset{\dots}{\text{times}} \times n}.$$

We denote

$$\nabla^k f = \left( \left[ \nabla^k f \right]_i^{j_1, \dots, j_k} \right)_{i=0,\dots,m-1}^{j_1,\dots,j_k=0,\dots,n-1},$$

and we use $*$ to denote the entire range of an index. For example, $[\nabla f]_i^*$ denotes the $i$th row and $[\nabla f]_*^j$ the $j$th column of the Jacobian, respectively. Algorithmic differentiation is implemented by the overloading of elemental functions including the built-in functions and operators of C++ as well as user-defined higher-level elemental functions.

## 1.1 First-order Tangent Mode

### Generic first-order scalar tangents

Generic first-order scalar tangent mode enables the computation of products of the Jacobian with vectors $\mathbf{x}^{(1)} \in D^n$

$$\mathbf{y}^{(1)} = \nabla f \cdot \mathbf{x}^{(1)} \in D^m$$

through provision of a generic first-order scalar tangent data type over arbitrary base data types $D$.

[1] We assume that the arithmetic inside $f$ is completely defined (through overloading of the *elemental functions*; see below) for variables from $D$.

#### Generic first-order vector tangents

Generic first-order vector tangent mode enables the computation of products of the Jacobian with matrices $X^{(1)} \in D^{n \times l}$

$$Y^{(1)} = \nabla f \cdot X^{(1)} \in D^{m \times l}$$

through provision of a generic first-order vector tangent data type over arbitrary base data types $D$.

#### Preaccumulation through use of expression templates

Expression templates enable the generation of statically optimized gradient code at the level of individual assignments which can be beneficial for vector tangent mode.

## 1.2 First-order Adjoint Mode

#### Generic first-order scalar adjoints

Generic first-order scalar adjoint mode enables the computation of products of the transposed Jacobian with vectors $\mathbf{y}_{(1)} \in D^m$

$$\mathbf{x}_{(1)} = \nabla f^T \cdot \mathbf{y}_{(1)} \in D^n$$

through provision of a generic first-order scalar adjoint data type over arbitrary base data types $D$.

#### Generic first-order vector adjoints

Generic first-order vector adjoint mode enables the computation of products of the transposed Jacobian with matrices $Y_{(1)} \in D^{m \times l}$

$$X_{(1)} = \nabla f^T \cdot Y_{(1)} \in D^{n \times l}$$

through provision of a generic first-order scalar adjoint data type over arbitrary base data types $D$.

#### Global "blob" tape

Memory of the specified size is allocated and used for storing the tape without bound checks.

#### Global "chunk" tape

The tape grows in chunks up to the physical memory bound.

#### Global "file" tape (v3.2)

Chunks are written to and read from disk.

#### Thread-local tape (v3.4)

The global tapes are now thread-local by default allowing an easier switch to multi-threading.

### Distinct data types for values, partial derivatives and adjoints with all kinds of tapes (v3.2)

Data types for values, partial derivatives and adjoints can be set individually when defining the differentiation mode.

### (Thread-)local tapes

Multiple “blob” or “chunk” tapes can be allocated, for example, to implement thread-safe adjoints.

### Preaccumulation through use of expression templates

Expression templates enable the generation of statically optimized gradient code at the level of individual assignments. This preaccumulation can result in a decrease in tape memory requirement.

### Repeated evaluation of tape

Tapes can be recorded at a given point and interpreted repeatedly.

### Adjoint callback interface

The adjoint callback interface supports

- checkpointing
- symbolic adjoints of numerical methods
- combinations of tangent and adjoint modes
- preaccumulation of local derivative tensors
- creation of collections of domain-specific higher-level elemental functions

### User-defined local Jacobians interface

Externally preaccumulated partial derivatives can be inserted directly into the tape for use within subsequent interpretations.

### User-driven preaccumulation of local Jacobians (v3.2)

An easy-to-use interface for robust preaccumulation of local Jacobians is provided. Aggressive reduction of the tape size is facilitated.

### Multiple adjoint vectors for a single tape (v3.2)

Several (concurrent) interpretations of the same tape are enabled through separate (thread-local) adjoint vectors.

### Modulo Adjoint Propagation (v3.3)

The vector of adjoints is compressed by analysing the maximum number of required distinct adjoint memory locations. During interpretation, adjoint memory, which is no longer required, is overwritten and thus reused by indexing through modulo operations. This feature is especially useful for iterative algorithms (e.g. time iterations). The required memory for the vector of adjoints usually stays constant, independent of the number of iterations. Combined with the use of disk tape, almost arbitrarily sized tapes can be generated, which might be especially of interest for prototyping or validation purposes.

### Sparse Tape Interpretation (v3.3)

The adjoint interpretation of the tape can omit propagation along edges when the corresponding adjoint to be propagated is zero. This might be of use when NaNs or Infs occur as local partial derivatives (e.g. when computing square root of zero), but this local result is only used for subsequent computations which are not relevant for the overall output. This feature might have a performance impact on tape interpretation and should therefore be considered for debugging configuration only.

## 1.3 Second- and Higher-order Tangent Mode

### Generic second-order scalar tangents

Instantiation of the generic first-order scalar tangent data type with the generic first-order scalar tangent data type over a non-derivative base data type yields the second-order scalar tangent data type. It enables the computation of scalar projections of the Hessian $\nabla^2 f \in D^{m\times n\times n}$ in its two domain dimensions of length $n$ as

$$\mathbf{y}^{(1,2)} = \langle \nabla^2 f, \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \rangle \equiv \left( {\mathbf{x}^{(1)}}^T \cdot \left[\nabla^2 f\right]_i^{*,*} \cdot \mathbf{x}^{(2)} \right)_{i=0,\ldots,m-1} \in D^m$$

for $\mathbf{x}^{(1)}, \mathbf{x}^{(2)} \in D^n$ over arbitrary base data types $D$. The computational cost of accumulating the whole Hessian over $D$ becomes $O(n^2)\cdot \text{Cost}(f)$. with both $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ ranging independently over the Cartesian basis vectors in $D^n$.

### Generic second-order vector tangents

Instantiation of the generic first-order vector tangent data type with the generic first-order vector tangent data type over a non-derivative base data type yields the second-order scalar tangent data type. It enables the computation of vector projections of the Hessian $\nabla^2 f \in D^{m\times n\times n}$ in its two domain dimensions of length $n$ as

$$Y^{(1,2)} = \langle \nabla^2 f, X^{(1)}, X^{(2)} \rangle \equiv \left( {X^{(1)}}^T \cdot \left[\nabla^2 f\right]_i^{*,*} \cdot X^{(2)} \right)_{i=0,\ldots,m-1} \in D^{m\times l_1 \times l_2}$$

for $X^{(1)} \in D^{n\times l_1}$, $X^{(2)} \in D^{n\times l_2}$ over arbitrary base data types $D$. The computational cost of accumulating the whole Hessian over $D$ remains equal to $O(n^2)\cdot\text{Cost}(f)$ with both $X^{(1)}$ and $X^{(2)}$ set equal to the identities in $D^n$.

### Other generic second-order tangents

Instantiation of the generic first-order vector tangent data type with the generic first-order scalar tangent data type over a non-derivative base data type yields a second-order tangent data type.

Similarly, instantiation of the generic first-order scalar tangent data type with the generic first-order vector tangent data type over a non-derivative base data type yields a second-order tangent data type.

### Generic third- and higher-order tangents

Instantiation of tangent types with $k$th-order tangent types yields $(k+1)$th-order tangent types.

## 1.4 Second- and Higher-order Adjoint Mode

### Generic second-order scalar adjoints

Instantiation of the generic first-order scalar adjoint data type with the generic first-order scalar tangent data type over a non-derivative base data type yields a second-order scalar adjoint data type. It enables the computation of scalar projections of the Hessian $\nabla^2 f \in D^{m\times n\times n}$ in its image and one of its domain dimensions as

$$\mathbf{x}_{(1)}^{(2)} = \langle \mathbf{y}_{(1)}, \nabla^2 f, \mathbf{x}^{(2)} \rangle \equiv \left( \mathbf{y}_{(1)}^T \cdot \left[\nabla^2 f\right]_*^{j,*} \cdot \mathbf{x}^{(2)} \right)_{j=0,\dots,n-1} \in D^n$$

for $\mathbf{y}_{(1)} \in D^m$ and $\mathbf{x}^{(2)} \in D^n$ over arbitrary base data types $D$. The computational cost of accumulating the whole Hessian over $D$ becomes $O(m \cdot n) \cdot \text{Cost}(f)$. with $\mathbf{y}_{(1)}$ and $\mathbf{x}^{(2)}$ ranging over the Cartesian basis vectors in $D^m$ and $D^n$, respectively.

### Generic second-order vector adjoints

Instantiation of the generic first-order vector adjoint data type with the generic first-order vector tangent data type over a non-derivative base data type yields a second-order vector adjoint data type. It enables the computation of vector projections of the Hessian $\nabla^2 f \in D^{m\times n\times n}$ in its image and one of its domain dimensions as

$$X_{(1)}^{(2)} = \langle Y_{(1)}, \nabla^2 f, X^{(2)} \rangle \equiv \left( Y_{(1)}^T \cdot \left[\nabla^2 f\right]_*^{j,*} \cdot X^{(2)} \right)_{j=0,\dots,n-1} \in D^{l_1\times n\times l_2}$$

for $Y_{(1)} \in D^{m\times l_1}$ and $X^{(2)} \in D^{n\times l_2}$ over arbitrary base data types $D$. The computational cost of accumulating the whole Hessian over $D$ remains equal to $O(m \cdot n) \cdot \text{Cost}(f)$. with $Y_{(1)}$ and $X^{(2)}$ set equal to the identities in $D^m$ and $D^n$, respectively.

### Other generic second-order adjoints

Instantiation of the generic first-order vector adjoint data type with the generic first-order scalar tangent data type over a non-derivative base data type yields a second-order adjoint data type. Similarly, instantiation of the generic first-order scalar adjoint data type with the generic first-order vector tangent data type over a non-derivative base data type yields a second-order adjoint data type.

Symmetry of the Hessian in its two domain dimensions yields the following additional second-order adjoint data types: Instantiation of a generic first-order tangent data type with a generic first-order adjoint data type over a non-derivative base data type yields a second-order adjoint data type for computing

$$\langle \mathbf{y}_{(2)}^{(1)}, \nabla^2 f, \mathbf{x}^{(1)} \rangle \equiv \left( \mathbf{y}_{(2)}^{(1)^T} \cdot \left[\nabla^2 f\right]_*^{j,*} \cdot \mathbf{x}^{(1)} \right)_{j=0,\dots,n-1} \in D^n.$$

Similarly, instantiation of a generic first-order adjoint data type with a generic first-order adjoint data type over a non-derivative base data type yields a second-order adjoint data type for computing

$$\langle \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^2 f \rangle \equiv \left( \mathbf{y}_{(1)}^T \cdot \left[\nabla^2 f\right]_{*}^{j,*} \cdot \mathbf{x}_{(1,2)} \right)_{j=0,\ldots,n-1} \in D^n .$$

### Generic third- and higher-order adjoints

Instantiation of tangent types with $k$th-order adjoint types yields $(k+1)$th-order adjoint types. Similarly, instantiation of adjoint types with $k$th-order tangent or adjoint types yields $(k+1)$th-order adjoint types.

## 1.5 Additional Functionality

### Vectorized (AVX) tangent and adjoint modes (v3.4)

For making use of vectorization, a respective vector data type can be used as base type for tangent and adjoint modes.

### Debugging capabilities (v3.4)

For enabling internal debugging checks, a compile time flag was added.

# Chapter 2

# General Remarks / Essentials

## 2.1 C++ Standard

dco/c++ is fully compliant with C++11. If you need C++98 support, please contact NAG.

## 2.2 Memory Allocation

- Default configuration uses a *blob tape*, i.e. no dynamic growth of memory during recording. Selection of a *chunk tape* is done at precompile time by the preprocessor (see next section).
- Allocated memory is not initialized.
- Under Linux, huge pages are supported. Those might enhance performance immensely (we've seen up to 30%). It can be switched on/off via the `tape_options` object. See Sec. 4.45.

The following environment variables define the size of the blob tape.

- The environment variable `DCO_MEM_RATIO` defines the ratio of available physical memory that should be allocated (default: 0.1).
- The environment variable `DCO_MAX_ALLOCATION` defines the maximal allocation size to be used in kilobytes.

## 2.3 Parallelization

- All tangent types are thread-safe.
- `dco::ga1s<T>`, `dco::ga1v<T>`, etc. are by default thread-safe due to use of a thread-local global tape. If using multiple threads, please make sure you never mix variables from different threads. See section 2.7 on how to disable thread-local global tapes.
- For explicit thread safety, you can use multiple tape versions: `dco::ga1sm<T>`, `dco::ga1vm<T>`, etc.

## 2.4 Performance

- Make sure (target specific) compiler optimization is switched on.
- Inlining is important. On the other hand, the increase in compilation time might not be acceptable in your project. For more agressive inlining, use the define `DCO_AGRESSIVE_INLINE`. This is defined by default under Linux and undefined by default under Windows (compilation time increases immensely there). Use `DCO_NO_AGRESSIVE_INLINE` to disable under Linux.

## 2.5 Type Casting

Usually, it is not desired to cast a `dco/c++` type to a non-dco type (e.g. **`double`**), since `dco/c++` is then unable to compute derivatives of that data flow. Nonetheless, there are cases, where a cast is actually what you want to do (e.g. cast to **`int`** for an index calculation). We therefore allow explicit cast operations to specified data type (implicit casts are not allowed). Use carefully. See Sec. 4.44.

*This is only available in C++11, since the implementation uses the* ***`explicit`*** *keyword on conversion functions.*

## 2.6 File I/O

By using `dco/c++` data types the binary layout of all data structures used throughout the code changes. This should be considered when using binary input/output, e.g. to files or streams. To avoid having to handle each call to e.g. `fscanf` individually, we ship respective implementations within the **`namespace`** `dco`. Those implementations only forward the underlying data to the respective calls (`fprinf, sprintf, fscanf, sscanf`). Make sure you are using the original calls, if non-wrapped binary I/O is needed, i.e. use for example `::sscanf(...)` (global namespace) explicitly.

*This is only available in C++11, since the implementation uses variadic template arguments.*

In addition, **`operator`**`<<` is overloaded and forwards the respective operator on the value component.

## 2.7 Feature Selection

Definition of the following preprocessor variables select the respective feature.

- `DCO_DEBUG` (default: OFF): Improved safety through enhanced checking, e.g, overflow checks (has performance impact).
- `DCO_CHUNK_TAPE` (default: undefined): Switches to *chunk tape*. The default chunk size is 128MB.
- `DCO_NO_THREADLOCAL` (default: undefined): Switches thread-local global tapes off (i.e. no use of `thread_local` keyword).
- `DCO_FORCE_THREADLOCAL_KEYWORD` (default: undefined): Forces the usage of `thread_local` keyword for the global tape. Due to bugs in compiler implementation of `thread_local` keyword (Intel Compiler on Windows) by default we may use other keywords to define global tape as thread-local (e.g., `__declspec(thread)` on Windows).

- `DCO_LOG_MAX_LEVEL` (default: $-1$): Defines the maximal and default logging level. If $-1$, logging is optimized out completely. See 4.43 for further details.
- `DCO_AUTO_SUPPORT` (default: undefined): When using `dco/c++` with 'auto'-keyword (C++11), please make sure setting this variable to avoid dangling references to local stack variables. Possible performance implications: might slow down recording when working with long right-hand sides.
- `DCO_TAPE_ACTIVITY` (default: defined): If tape activity check is enabled, dco only records tape for operations which depend on registered variables.
- `DCO_GT1S_ACTIVITY` (default: undefined): If defined, tangents are propagated only when the tangent components are nonzero. If it is undefined, NaNs, occurring in parts of the code that does not depend on the inputs, will be propagated to the output tangents.
- `DCO_TAPE_USE_LONG_INT` (default: undefined): Deprecated. Use `DCO_TAPE_USE_INT64` instead.
- `DCO_TAPE_USE_INT64` (default: undefined): Switches internal counter from 'int' to 64 bit integer (Windows: `__int64`, Linux:`int64_t`). This is usually required if recording a huge amount of tape when writing to disk. Required when number of assignments greater than $2^{31}-1$.
- `DCO_STD_COMPATIBILITY` (default: undefined): Will instruct `dco/c++` to import the overloaded standard math functions (e.g. ceil, floor, min, max, ...) into the namespace `std`. In addition, `numeric_limits` is specialized and included into the namespace `std`.
- `DCO_INTEROPERABILITY_BOOST_INTERVAL` (default: undefined): Linux and post C++11 only. Will instruct `dco/c++` to include the BOOST header `boost/numeric/interval.hpp`. In addition, a handful of functions and structs are specialized to support `dco/c++` data types as base types. `DCO_STD_COMPATIBILITY` is defined implicitly, since it's required.
- `DCO_BITWISE_MODULO` (default: undefined): The adjoint vector size is rounded up to the next power of two for enabling a faster modulo operation (bitwise &). See Ch. 4.39 for details.
- `DCO_SKIP_WINDOWS_H_INCLUDE` (default: undefined): Under Windows, `windows.h` is included by default. This is required for getting the total physical memory size (for automatic blob tape size calculation). If define is set, `windows.h` is not included and when using the blob tape, you must define the environment variable `DCO_MAX_ALLOCATION` (see Sec. 2.2).
- `DCO_USE_INTRINSIC_COMPLEX_PRIMALS` (default: undefined): Since it is impossible to overload builtin complex arithmetic, primal values might differ when either using `std::complex` template implementation or the reimplementation we did in `dco/c++`. We have, however, a type, which falls back to the builtin arithmetic. This requires the computation to be performed twice, but delivers exactly the same primals as with passive types.
- `DCO_AGRESSIVE_INLINE` (default: platform dependent): See Sec. 2.4.
- `DCO_NO_AGRESSIVE_INLINE` (default: undefined): See Sec. 2.4.
- `DCO_NO_INTERMEDIATES` (default: undefined): Removes use of template expressions; each operation now returns an active type as opposed to an expression type. Negative impact on performance expected (run time- as well as memory-wise).

## 2.8 Coupling with Other Template Libraries

### Standard Template Library (STL)

Conceptually, `dco/c++` data types can be used as template arguments for all STL functions and classes. Specific remarks follow:

- `std::complex`
  Unfortunately, there are some issues with the implementation of `std::complex` under Windows. Therefore, a specialized implementation of `std::complex` for `dco/c++` types is used. In addition, the builtin arithmetic is not used by default for `dco/c++` types. This might lead to differences in the primal values. To fall back to the builtin arithmetic, `DCO_USE_INTRINSIC_COMPLEX_PRIMALS` can be defined. This requires an additional passive evaluation of the operations, though.

- `std::common_type`
  (23.15.7.6) in the C++ standard allows specialization of `std::common_type`. This is specialized for all `dco/c++` types. The following lambda is an example on how to use this. The auto-deduced return type is different for the two branches. Therefore, a common type of both possible return types needs to be used.

```
auto f = [](auto && x, auto && y)
    -> typename std::common_type<decltype(x), decltype(y)>::type {
  if   (x < 0) return x;
  else         return y;
};
```

### Boost Interval Arithmetic Library

To obtain semi-local derivative information with `dco/c++` the `interval<T,Policies>` type from the Boost Interval Arithmetic Library[1] can be used as a base type of `dco/c++` types.

For this to work, the `DCO_INTEROPERABILITY_BOOST_INTERVAL` needs to be defined.

Instead of using the default policies, you should specify the rounding and checking policies to enable transcendental and trigonometric functions. As an example the following type can be used.

```
boost::numeric::interval<double, boost::numeric::interval_lib::policies<
  boost::numeric::interval_lib::save_state<
    boost::numeric::interval_lib::rounded_transc_std<double> >,
  boost::numeric::interval_lib::checking_base<double> > >
```

The derivative component of variable x can be seeded by `dco::derivative(x).assign(1,1)`.

To obtain the lower and upper bound of the interval derivative `dco::derivative(y).lower()` and `dco::derivative(y).upper()` can be used.

It is also possible to nest the other way round, i.e. instantiate the interval type with `dco/c++` types as template parameter.

### Eigen Library

As Eigen states on its webpage[2], for supporting custom data types as template argument, `Eigen::NumTraits` needs to be specialized. This is done within `dco.hpp`. Make sure having included Eigen

[1] https://www.boost.org/doc/libs/1_62_0/libs/numeric/interval/doc/interval.htm
[2] https://eigen.tuxfamily.org/dox/TopicCustomizing_CustomScalar.html

headers before including `dco.hpp`. `dco/c++` will automatically switch on the `DCO_STD_COMPATIBILITY` define in this case as well.

Note that the specialization of `NumTraits` is only tested for the latest Eigen version at the time of writing, which currently is 3.3.9. You can enforce use of the `dco/c++` `NumTraits` specialization for other Eigen versions by providing the `DCO_EXT_EIGEN_IGNORE_VERSION` define. In case you want to prevent `dco/c++` from providing the specialization, you can use the `DCO_EXT_NO_EIGEN` define.

Since both – `dco/c++` and Eigen – implement expression templates, there can be compilation errors depending on the underlying user code. If this is the case, you can disable the `dco/c++` template expressions using the `DCO_NO_INTERMEDIATES` define.

There is a dedicated project to optimize algorithmic differentiation within Eigen called Eigen-AD. `dco/c++` provides corresponding implementations to Eigen-AD, allowing symbolical treatment of the Eigen dense solvers and matrix-matrix multiplication. Refer to the Eigen-AD[3] page for further details.

### Stan Math Library

When using `dco/c++` types as template arguments for the Stan Math Library, there are a couple of specializations and overloads required. Please contact `support@nag.co.uk`.

[3] https://gitlab.stce.rwth-aachen.de/stce/eigen-ad

# Chapter 3

# General Functions and Traits

This section lists a couple of functions and traits which are general in the sense, that they are available for all differentiation modes (i.e. `gt1s<...>, ga1s<..., ...`) and respective types.

## 3.1 Functions

### 3.1.1 `dco::value`

#### Definition

```
template <typename T> [const] RET<T>& dco::value([const] T&)
```

#### Description

This function works with all types. If `T` is a `dco/c++` type, a reference to the value component is returned. Otherwise, a reference to the type itself is returned. `RET<T>` is deduced respectively.

### 3.1.2 `dco::derivative`

#### Definition

```
template <typename T> [const] RET<T>& dco::derivative([const] T&)
```

#### Description

This function works with all types. If `T` is a `dco/c++` type, it returns a reference to the derivative component. Otherwise, a respective zero is returned *by value*. `RET<T>` is deduced respectively.

### 3.1.3 `dco::tangent`

#### Definition

```
template <typename T> [const] RET<T>& dco::tangent([const] T&)
```

#### Description

This function works with all types. If `T` is a `dco/c++` tangent type, it returns a reference to the tangent component. Otherwise, a respective zero is returned *by value*. `RET<T>` is deduced respectively.

### 3.1.4 `dco::adjoint`

#### Definition

```
template <typename T> [const] RET<T>& dco::adjoint([const] T&)
```

#### Description

This function works with all types. If `T` is a `dco/c++` adjoint type, it returns a reference to the adjoint component. Otherwise, a respective zero is returned *by value*. `RET<T>` is deduced respectively.

### 3.1.5 `dco::passive_value`

#### Definition

```
template <typename T> [const] RET<T>& dco::passive_value([const] T&)
```

#### Description

This function works with all types. If `T` is a `dco/c++` type, a reference to the passive value component is returned; different to `dco::value` only for higher order types. Otherwise (not a `dco/c++` type), a reference to the type itself is returned. `RET<T>` is deduced respectively.

### 3.1.6 `dco::tape`

#### Definition

```
template <typename T> TAPE<T>* dco::tape(const T&)
```

#### Description

This function works with all types. If `T` is a `dco/c++` adjoint type, it returns a pointer to the underlying tape, `NULL` if not registered in case of multiple tape support. Otherwise (not a `dco/c++` type) a `NULL` is returned as **`void*`**. `TAPE<T>` is deduced respectively.

### 3.1.7 `dco::tape_index`

#### Definition

```
template <typename T> TAPE_IDX<T> dco::tape_index(const T&)
```

### Description

This function works with all types. If `T` is a `dco/c++` adjoint type, it returns the tape index, `0` if not registered. Otherwise (not a `dco/c++` type) `0` is returned *by value*. `TAPE_IDX<T>` is deduced respectively.

### 3.1.8 `dco::size_of`

#### Definition

A) `template <typename T> size_t dco::size_of(const T&)`
B) `template <typename T> size_t dco::size_of(const T&, int)`

#### Description

A+B) This function works with all types and returns its size. If `T` is a user-defined type, a specialization of the corresponding struct (`trait_size_of`) is required (see Sec. 3.2).

B) If `T` is a tape pointer, a configuration can be added via the `enum TAPE::size_of_mode`:

```
enum size_of_mode {
  //** include the currently used memory of the internal stack
  //**    (the stack holds the graph structure: edges and weights)
  size_of_stack = 1,
  //** include the allocated memory of the internal stack
  //**    (especially for blob tape, this might be much larger)
  size_of_allocated_stack = 2,
  //** include the allocated memory of the internal vector of adjoints
  size_of_internal_adjoint_vector = 4,
  //** include the memory occupied by the written checkpoints
  //**   for this to work with user-defined datatypes,
  //**   trait_size_of is required to be specialized
  size_of_checkpoints = 8,
  size_of_default = size_of_stack | size_of_internal_adjoint_vector
};
```

Of course, the various modes can be combined as, e.g.:

```
    size_of(tape, TAPE_T::size_of_allocated_stack | TAPE_T::
        size_of_internal_adjoint_vector);
```

### 3.1.9 Allocation / Deallocation Functions

#### Definition

```
    void* dco::alloc(size_t size,
                     bool use_huge_pages = false);
    void  dco::dealloc(void*);
```

#### Description

Allocate `size` bytes of uninitialized storage. `use_huge_pages` is currently only available under Linux and uses `mmap` to allocate huge pages (see Ch. 4.45). The memory allocated with `dco::`

alloc needs to be deallocated using dco::dealloc function. Returns 0 if unable to allocate requested memory.

## 3.2 Traits

### 3.2.1 dco::mode

#### Definition

```
template <typename T> struct mode;
typename mode::value_t;
typename mode::passive_t;
typename mode::derivative_t;
typename mode::tape_t;
typename mode::local_gradient_t;
typename mode::external_adjoint_object_t;
typename mode::jacobian_preaccumulator_t;
bool mode::is_dco_type;
bool mode::is_adjoint_type;
bool mode::is_tangent_type;
int  mode::order;
```

#### Description

This trait works with all types. In case T is a dco/c++ type, the respective types and booleans are set. Otherwise, most types are set to void, apart from value_t and passive_t, which are set to T. This trait is very useful for template specializations.

### 3.2.2 dco::trait_size_of

#### Definition

```
template <typename T> struct trait_size_of {
  size_t get(const T&);
};
```

#### Description

This trait is implemented for all dco/c++ types as well as fundamental C++ types. In case the external adjoint data object has stored user-defined types, the user needs to implement a specialization of this trait.

# Chapter 4

# Features by Example

## 4.1 First-order Tangent Mode

Example program can be found here: `examples/gt1s`

### 4.1.1 Purpose

The `dco/c++` data type `gt1s<DCO_BASE_TYPE>::type` implements tangent first-order scalar mode. The first-order tangent version

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(1)} \end{pmatrix} := f^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$$

resulting from the application of `dco/c++` to a multivariate vector function $f : \mathbb{R}^n \to \mathbb{R}^m$, $\mathbf{y} = f(\mathbf{x})$, computes

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{y}^{(1)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1)} \rangle \equiv \frac{\partial f}{\partial \mathbf{x}} \cdot \mathbf{x}^{(1)}. \end{aligned} \tag{4.1}$$

### 4.1.2 Example

We consider the following implementation of a function $f : \mathbb{R}^4 \to \mathbb{R}^2$ :

```
template <typename T> std::vector<T> f(const std::vector<T>& x) {
  std::vector<T> y(2);
  T v = tan(x[2] * x[3]);
  T w = x[1] - v;
  y[0] = x[0] * v / w;
  y[1] = y[0] * x[1];
  return y;
}
```

The main function computes the Jacobian matrix using (4.1) by iterating a loop over the Cartesian basis vector.

```
int main() {
  // Using first-order tangent mode.
  constexpr std::size_t n = 4, m = 2;
  using mode_t = dco::gt1s<double>;
  using type = mode_t::type;

  // Inputs.
  std::vector<type> x(n, 1.0);

  // Iterate over the Cartesian basis vectors in R^n.
  for (std::size_t i = 0; i < n; ++i) {

    // Seed i-th Cartesian basis vector to the input tangents.
    dco::derivative(x[i]) = 1.0;

    // Propagate tangents from inputs to outputs, i.e. perform
    // implicit Jacobian vector product.
    auto y = f(x);

    // Reset input tangent.
    dco::derivative(x[i]) = 0.0;

    if (i == 0) {
      // Print output values (just once on first iteration).
      std::cout << "y = [ ";
      for (auto const& yi : y) {
        std::cout << yi << " ";
      }
      std::cout << "]" << std::endl;
    }

    // Print Jacobian.
    std::cout << "dy / dx[" << i << "] = [ ";
    for (std::size_t j = 0; j < m; ++j) {
      std::cout << dco::derivative(y[j]) << " ";
    }
    std::cout << "]" << std::endl;
  }
}
```

The following output is generated:

```
y = [ -2.79402 -2.79402 ]
dy / dx[0] = [ -2.79402 -2.79402 ]
dy / dx[1] = [ -5.01252 -7.80654 ]
dy / dx[2] = [ 11.025 11.025 ]
dy / dx[3] = [ 11.025 11.025 ]
```

### 4.1.3 Concepts introduced in this Section

#### dco.hpp

dco/c++ header to be included.

#### DCO_BASE_TYPE

Variable instantiation type for generic AD modes.

`gt1s<DCO_BASE_TYPE>`

Generic first-order tangent mode.

`DCO_MODE`

Generic AD mode; for example `typedef gt1s<DCO_BASE_TYPE> DCO_MODE`.

`gt1s<DCO_BASE_TYPE>::type`

Generic first-order tangent type.

`DCO_TYPE`

Generic AD type; for example `typedef gt1s<DCO_BASE_TYPE>::type DCO_TYPE`.

`value`

Returns reference to value of argument.

`derivative`

Returns reference to derivative of argument.

## 4.2 First-order Adjoint Mode

Example program can be found here: `examples/ga1s`

### 4.2.1 Purpose

The `dco/c++` type `ga1s<DCO_BASE_TYPE>::type` implements adjoint first-order scalar mode using a global tape.

The first-order adjoint version

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{x}_{(1)} \end{pmatrix} := f_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{y}, \mathbf{y}_{(1)})$$

resulting from the application of `dco/c++` to a multivariate vector function $f : \mathbb{R}^n \to \mathbb{R}^m$, $\mathbf{y} = f(\mathbf{x})$, computes

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)} \cdot \frac{\partial f}{\partial \mathbf{x}} \rangle \equiv \mathbf{x}_{(1)} + \left(\frac{\partial f}{\partial \mathbf{x}}\right)^T \cdot \mathbf{y}_{(1)}. \end{aligned} \tag{4.2}$$

### 4.2.2 Example

For the same function $f : \mathbb{R}^4 \to \mathbb{R}^2$ used in Ch. 4.1 the main program computes the Jacobian using (4.2).

```
int main() {
  // Using first-order adjoint mode.
  constexpr std::size_t n = 4, m = 2;
  using mode_t = dco::ga1s<double>;
  using type = mode_t::type;

  // Inputs.
  std::vector<type> x(n, 1.0);

  dco::smart_tape_ptr_t<mode_t> tape;
  tape->register_variable(x);

  // Record tape for f. This is only required once!
  auto y = f(x);

  tape->register_output_variable(y);

  // Print output values.
  std::cout << "y = [ ";
  for (auto const& yi : y) {
    std::cout << yi << " ";
  }
  std::cout << "]" << std::endl;

  // Iterate over the Cartesian basis vectors in R^m.
  for (std::size_t j = 0; j < m; ++j) {
    // Seed j-th Cartesian basis vector to the output adjoints.
    dco::derivative(y[j]) = 1.0;

    // Interpret the tape for j-th output.
    tape->interpret_adjoint();

    // Print Jacobian.
    std::cout << "dy[" << j << "] = [ ";
    for (std::size_t i = 0; i < n; ++i) {
      std::cout << dco::derivative(x[i]) << " ";
    }
    std::cout << "]" << std::endl;

    // Reset all adjoints to zero.
    tape->zero_adjoints();
  }
}
```

The following output is generated:

```
y = [ -2.79402 -2.79402 ]
dy[0] = [ -2.79402 -5.01252 11.025 11.025 ]
dy[1] = [ -2.79402 -7.80654 11.025 11.025 ]
```

### 4.2.3 Concepts introduced in this Section

**`ga1s<DCO_BASE_TYPE>`**

Generic first-order adjoint scalar mode.

**`ga1s<DCO_BASE_TYPE>::type`**

Generic first-order adjoint scalar type.

**`DCO_TAPE_TYPE`**

Type of tape associated with `DCO_MODE`.

**`dco::smart_tape_ptr_t<DCO_MODE>`**

Smart pointer to global tape associated with `DCO_MODE`.

**`DCO_TAPE_TYPE::create`**

Tape creator returns pointer to tape as `DCO_TAPE_TYPE*`. Not required when using `dco::smart_tape_ptr` .

**`DCO_TAPE_TYPE::register_variable`**

Creates entry for argument (independent variable) in associated tape.

**`DCO_TAPE_TYPE::register_output_variable`**

Marks argument as a dependent variable in associated tape.

**`DCO_TAPE_TYPE::interpret_adjoint`**

Adjoint interpretation of entire tape.

**`DCO_TAPE_TYPE::remove`**

Deallocates global tape. Not required when using `dco::smart_tape_ptr`.

## 4.3 Combined First-order Tangent and Adjoint Mode

Example program can be found here: `examples/gtas`

### 4.3.1 Purpose

The `dco/c++` type `gtas<DCO_BASE_TYPE>::type` implements a combined tangent and adjoint first-order scalar mode using a global tape. See respective sections for documentation of tangent and adjoint modes. Be aware, that the accessor functions `dco::tangent` and `dco::adjoint` have to be used instead of `dco::derivative`. Main benefit is, that only one instantiation of the code is required to be able to compute tangents and adjoints. Downside is larger memory footprint per program variable.

### 4.3.2 Example

For the same function $f : \mathbb{R}^4 \to \mathbb{R}^2$ used in Ch. 4.1 the main program computes the Jacobian using tangents and adjoints.

```

int main() {
  constexpr std::size_t n = 4, m = 2;
  using mode_t = dco::gtas<double>;
  using type = mode_t::type;

  // Inputs.
  std::vector<type> x(n, 1.0);

  // Using first-order adjoint mode.
  dco::smart_tape_ptr_t<mode_t> tape;
  tape->register_variable(x);

  // Record tape for f. This is only required once!
  auto y = f(x);

  tape->register_output_variable(y);

  // Print output values.
  std::cout << "y = [ ";
  for (auto const& yi : y) {
    std::cout << yi << " ";
  }
  std::cout << "]" << std::endl;

  // Iterate over the Cartesian basis vectors in R^m.
  for (std::size_t j = 0; j < m; ++j) {
    // Seed j-th Cartesian basis vector to the output adjoints.
    dco::adjoint(y[j]) = 1.0;

    // Interpret the tape for j-th output.
    tape->interpret_adjoint();

    // Print Jacobian.
    std::cout << "dy[" << j << "] = [ ";
    for (std::size_t i = 0; i < n; ++i) {
      std::cout << dco::adjoint(x[i]) << " ";
    }
    std::cout << "]" << std::endl;

    // Reset all adjoints to zero.
    tape->zero_adjoints();
  }

  // Using first-order tangent mode.
  // Iterate over the Cartesian basis vectors in R^n.
  for (std::size_t i = 0; i < n; ++i) {

    // Seed i-th Cartesian basis vector to the input tangents.
    dco::tangent(x[i]) = 1.0;

    // Propagate tangents from inputs to outputs, i.e. perform
    // implicit Jacobian vector product.
    y = f(x);

    // Reset input tangent.
    dco::tangent(x[i]) = 0.0;

    if (i == 0) {
      // Print output values (just once on first iteration).
      std::cout << "y = [ ";
      for (auto const& yi : y) {
        std::cout << yi << " ";
      }
      std::cout << "]" << std::endl;
    }

    // Print Jacobian.
    std::cout << "dy / dx[" << i << "] = [ ";
    for (std::size_t j = 0; j < m; ++j) {
      std::cout << dco::tangent(y[j]) << " ";
    }
    std::cout << "]" << std::endl;
  }
}
```

### 4.3.3 Concepts introduced in this Section

**`gtas<DCO_BASE_TYPE>`**

Generic first-order combined tangent and adjoint scalar mode.

**`gtas<DCO_BASE_TYPE>::type`**

Generic first-order combined tangent and adjoint scalar type.

**`tangent`**

Returns reference to tangent of argument.

**`adjoint`**

Returns reference to adjoint of argument.

## 4.4 Vector Type

Example program can be found here: `examples/gv`

### 4.4.1 Purpose

The `dco/c++` data type `gv<DCO_BASE_TYPE, VECTOR_SIZE>::type` implements a vector data type primarily useful for SSE/AVX vectorization. This section describes the basic usage of the vector type; this type can then be used as base type for tangent and adjoint `dco/c++` types (see next chapters).

We assume mutually independent loop iterations. Each iteration computes one element in the output vector, i.e.

$$\mathbf{y}_i = f(\mathbf{x}_i, \lambda)$$

with active input vector $\mathbf{x} \in R^n$, active output vector $\mathbf{y} \in R^n$, and passive inputs $\lambda$. A very famous example would be a Monte Carlo simulation.

Instead of iterating over all indices with scalar data, we can now only iterate $\frac{s}{n}$ times, for $s =$ `VECTOR_SIZE`, computing $s$ elements at once (assuming $n$ is a multiple of $s$).

When doing vector evaluations of a function, data dependent control flow gets more complicated, since the data flow for individual vector elements could potentially diverge. At least for simple cases, this can be handled by comparison operators returning bitsets (i.e. for each vector element it holds the respective boolean), which can then be used to mask out computed values; see the Example for more details.

#### Remarks

Make sure you enable SSE/AVX/AVX-512 if supported by CPU and compiler. You might have to use `-march=native` (for `g++` or `clang++`) or `/arch=AVX2` (MSVC) and `-O3` or `-Ofast` as compilation flags.

### 4.4.2 Example

#### Without Diverging Control Flow

We consider the following implementation of a branch-free function $f : \mathbb{R} \to \mathbb{R}$ :

```
template <typename T> void f(const T& x, T& y) {
  T v = sin(x * x);
  T w = -cos(v);
  y = v * w;
}
```

The driver computes the elements of $\mathbf{y}$ by running a vectorized version of $f$.

```
typedef double DCO_BASE_TYPE;
static const unsigned long VECTOR_SIZE = 8;
typedef gv<DCO_BASE_TYPE, VECTOR_SIZE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

void driver(const vector<double>& x, vector<double>& y) {
  const size_t n = x.size();
  DCO_TYPE vec_x, vec_y;
  for (auto range : dco::subranges<VECTOR_SIZE>(n)) {
    for (auto i : range) {
      vec_x[i.sub_index] = x[i.global_index];
    }
    f(vec_x, vec_y);
    for (auto i : range) {
      y[i.global_index] = vec_y[i.sub_index];
    }
  }
```

The main program initializes the input variables with random numbers followed by calling the driver and printing the results.

```

int main() {
  const int m = 12, n = 12;
  cout.precision(5);
  vector<double> x(n), y(m);
  for (int i = 0; i < n; i++) {
    x[i] = sin(static_cast<double>(i + 1));
  }
  driver(x, y);
```

The following output is generated:

```
y[0]=-0.516941929775134
y[1]=-0.15305648476807
[...]
y[10]=-0.220197556962565
y[11]=-0.357012857533553
```

### With Diverging Control Flow

Now we consider the following implementation of a function $f : \mathbb{R} \to \mathbb{R}$ :

```
template <typename T> void f(const T& x, T& y) {
  T v = sin(x * x);
  T w = -cos(v);
  y = v * w;
  if (y < -0.5)
    y += sin(exp(x));
}
```

This can be rewritten by using the mentioned bitset and a masking function.

```
template <typename T> void f(const T& x, T& y) {
  T v = sin(x * x);
  T w = -cos(v);
  y = v * w;
  y += dco::vmask(y < -0.5, sin(exp(x)));
}
```

Driver and main function are unaltered compared to the previous section.

The following output is generated:

```
y[0]=0.21746389190323
```

```
y[1]=-0.153056484768117
[...]
y[10]=-0.22019755696259
y[11]=-0.357012857533558
```

### 4.4.3 Concepts introduced in this Section

**`gv<DCO_BASE_TYPE, VECTOR_SIZE>`**

Generic vector mode; individual values of type `DCO_BASE_TYPE`; with `VECTOR_SIZE` elements.

**`operator[]`**

```
DCO_BASE_TYPE [const]& gv<DCO_BASE_TYPE, VECTOR_SIZE>::operator[]
        (unsigned int i) [const]
```

Returns reference to ith element of vector.

**`template <size_t m> subrange_t<m> subranges<m>(n)`**

Returns an iterable, which can in turn be used to iterate over the subranges. The range $[0, n)$ is equally divided into subranges of size $m$.

#### Comparison Operator

```
bitset<VECTOR_SIZE> operator>(gv<...>::type const& x, gv<...>::type const& y)
```

Returns bitset of size `VECTOR_SIZE` with ith element set to `1`, if comparison operator of ith values returns `true`; `0` otherwise. This is implemented for all comparison operators.

**`vmask(...)`**

```
gv<FLOAT_T,VECTOR_SIZE> vmask(std::bitset<VECTOR_SIZE> const& mask,
                              gv<FLOAT_T,VECTOR_SIZE>  const& x)
```

Return a copy, with ith element of vector `x` to `0` if ith element of `mask` is `0`; otherwise, doesn't touch the element. Overloads for `bool` and scalar types exist.

## 4.5 Vectorized Tangent Type

Example program can be found here: `examples/gv_gt1s`

### 4.5.1 Purpose

The vector type can be used as base type for `gt1s`.

#### Remarks

Please see remark in Sec. 4.4.1.

### 4.5.2 Example

We consider the same examples as in Sec. 4.4.2.

The main program initializes the input variables with random numbers and sets input tangents followed by calling the driver and printing the results.

```

int main() {
  const int m = 12, n = 12;
  cout.precision(5);
  vector<double> xv(n), xt(n), yv(m), yt(m);
  for (int i = 0; i < n; i++) {
    xv[i] = sin(static_cast<double>(i + 1));
    xt[i] = 1;
  }
  driver(xv, xt, yv, yt);
  for (int i = 0; i < m; i++)
```

The driver needs to initialized values and tangents of vectorized types correspondingly.

```
static const unsigned long VECTOR_SIZE = 8;
typedef gv<double, VECTOR_SIZE> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef gt1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

void driver(const vector<double>& xv, vector<double>& xt, vector<double>& yv,
            vector<double>& yt) {
  const size_t n = xv.size();
  DCO_TYPE vec_x, vec_y;
  for (auto range : dco::subranges<VECTOR_SIZE>(n)) {
    for (auto i : range) {
      value(vec_x)[i.sub_index] = xv[i.global_index];
      derivative(vec_x)[i.sub_index] = xt[i.global_index];
    }
    f(vec_x, vec_y);
    for (auto i : range) {
      yv[i.global_index] = value(vec_y)[i.sub_index];
      yt[i.global_index] = derivative(vec_y)[i.sub_index];
    }
  }
}
#else
typedef double DCO_BASE_TYPE;
```

#### Without Diverging Control Flow

The following output is generated:

```
y[0]=-0.516941929775134
[...]
y[11]=-0.357012857533553
y^{(1)}[0]=-0.517789731589026
[...]
y^{(1)}[11]=-0.907555941961519
```

#### With Diverging Control Flow

Driver and main function are unaltered compared to the previous section.

The following output is generated:

```
y[0]=0.21746389190323
[...]
y[11]=-0.357012857533553
y^{(1)}[0]=-2.09022771607514
[...]
y^{(1)}[11]=-0.907555941961519
```

## 4.6 Vectorized Adjoint Type

Example program can be found here: `examples/gv_ga1s`

### 4.6.1 Purpose

The vector type can be used as base type for `ga1s`. When recording a tape, the individual loop iterations are not allowed to share active inputs.

#### Remarks

Vector type is currently only supported when using *blob tape*.

Please see remark in Sec. 4.4.1.

### 4.6.2 Example

We consider the same examples as in Sec. 4.4.2.

The main program initializes the input variables with random numbers and sets input and output adjoints followed by calling the driver and printing the results.

```

int main() {
  const int m = 12, n = 12;
  cout.precision(5);
  vector<double> xv(n), xa(n), yv(m), ya(m);
  for (int i = 0; i < n; i++) {
    xv[i] = sin(static_cast<double>(i + 1));
    xa[i] = 1;
  }
  for (int i = 0; i < m; i++)
    ya[i] = 1;
  driver(xv, xa, yv, ya);
```

The driver needs to initialize values and adjoints of vectorized types correspondingly. In addition, the correct global tape needs to be used.

```
typedef gv<double, VECTOR_SIZE> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

void driver(const vector<double>& xv, vector<double>& xa, vector<double>& yv,
            vector<double>& ya) {
  dco::smart_tape_ptr_t<DCO_MODE> tape;

  const size_t n = xv.size();
  DCO_TYPE vec_x, vec_y;
  for (auto range : dco::subranges<VECTOR_SIZE>(n)) {
    for (auto i : range) {
      value(vec_x)[i.sub_index] = xv[i.global_index];
    }
    tape->register_variable(vec_x);
    f(vec_x, vec_y);
    tape->register_output_variable(vec_y);
    for (auto i : range) {
      yv[i.global_index] = value(vec_y)[i.sub_index];
      derivative(vec_y)[i.sub_index] = ya[i.global_index];
      derivative(vec_x)[i.sub_index] = xa[i.global_index];
    }
    tape->interpret_adjoint();
    for (auto i : range) {
      ya[i.global_index] = derivative(vec_y)[i.sub_index];
      xa[i.global_index] = derivative(vec_x)[i.sub_index];
    }
    tape->reset();
  }
}
```

### Without Diverging Control Flow

The following output is generated:

```
y[0]=-0.516941929775134
[...]
y[11]=-0.357012857533553
x_{(1)}[0]=0.482210268410974
[...]
x_{(1)}[11]=0.0924440580384813
```

### With Diverging Control Flow

Driver and main function are unaltered compared to the previous section.

The following output is generated:

```
y[0]=0.21746389190323
[...]
y[11]=-0.357012857533553
x_{(1)}[0]=-1.09022771607514
[...]
x_{(1)}[11]=0.0924440580384813
```

## 4.7 First-order Adjoint Mode: Blob/Chunk Tapes

### 4.7.1 Purpose

*Blob tapes* dynamically allocate an area of memory of specified size under the assumption that the given target computation can be recorded within these bounds. Alternatively, an exception is thrown, if running out of bounds.

*Chunk tapes* dynamically allocate chunks of memory of specified size. The chunk size is specified at runtime when creating the tape; default chunk size is 128MB. Filled chunks result in allocation of new chunks up to the system memory bound. Corresponding bound checks are performed.

For enabling the chunk tape, compile with preprocessor define `DCO_CHUNK_TAPE`, i.e. with `g++`:

```
g++ main.cpp -DDCO_CHUNK_TAPE
```

All tapes (first- and higher-order, global and local) are switched to chunk tape.

### 4.7.2 Example

The user interfaces to both tape types are similar differing only in the syntax and semantic of setting the tape and chunk sizes, respectively.

```
...
  dco::tape_options o;
  o.set_chunk_size_in_byte(1024); // for chunk tape
  o.set_blob_size_in_mbyte(1); // for blob tape
  o.deallocation_on_reset() = false; // for chunk tape
  dco::smart_tape_ptr_t<DCO_M> tape(o);
  std::cout << o.chunk_size_in_byte() << std::endl;
...
```

A `tape_options` object allows for the chunk size to be specified, for example, one kilobyte. It is passed as an argument to the tape creation routine. If blob tape is used, the blob tape size will be one megabyte in this example. In addition, when using the chunk tape, the chunks are by default deallocated the moment the tape gets reset to a smaller position. In the example above, this is disabled, i.e., the chunks are kept in memory. This is especially useful, if another recording is likely to happen which reuses these chunks again.

### 4.7.3 Concepts introduced in this Section

**`dco::tape_options`**

Tape configuration object.

**`size_t dco::tape_options::chunk_size_in_byte()`**

Get chunk size in bytes.

**`void dco::tape_options::set_chunk_size_in_kbyte(double)`**

Set chunk size in kilobytes.

**void dco::tape_options::set_chunk_size_in_mbyte(double)**

Set chunk size in megabytes.

**void dco::tape_options::set_chunk_size_in_gbyte(double)**

Set chunk size in gigabytes.

**size_t dco::tape_options::blob_size_in_byte()**

Get blob size in bytes.

**void dco::tape_options::set_blob_size_in_kbyte(double)**

Set blob size in kilobytes.

**void dco::tape_options::set_blob_size_in_mbyte(double)**

Set blob size in megabytes.

**void dco::tape_options::set_blob_size_in_gbyte(double)**

Set blob size in gigabytes.

**bool& dco::tape_options::deallocation_on_reset()**

Get/set option 'deallocation_on_reset'. Please see example for description.

**DCO_TAPE_TYPE::create**

Tape creator allows for `tape_options` object to be passed as an argument; returns pointer to tape as `DCO_TAPE_TYPE*`.

## 4.8 First-order Adjoint Mode: File Tape

### 4.8.1 Purpose

*File tapes* are chunk tapes. Compile with preprocessor define `DCO_CHUNK_TAPE`, i.e. with `g++`:

```
g++ main.cpp -DDCO_CHUNK_TAPE
```

The chunk tape dynamically allocates chunks of memory of specified size. The chunk size is specified at runtime when creating the tape; default chunk size is 128MB. Filled chunks result in offloading to disk followed by creation of new chunks within the previously allocated memory. Corresponding bound checks are performed.

Chunks are read from disk during tape interpretation. All corresponding files are deleted when the tape is removed.

Second- and higher-order adjoint work correspondingly.

Remark: When using the `size_of` function to get the size of the tape, offloaded chunks are counted as well.

### 4.8.2 Example

The user interface to file tapes is similar to chunk tapes differing only in the internal handling of the chunks as outlined above.

```
...
  dco::tape_options o;
  o.write_to_file()=true;
  o.set_chunk_size_in_byte(1024);
  dco::smart_tape_ptr_t<DCO_M> tape(o);
  std::cout << o.chunk_size_in_byte() << std::endl;
...
```

For example, a tape of an overall size of 3.5kB results in a total of four chunks, three of which are offloaded to disk during recording.

### 4.8.3 Concepts introduced in this Section

The functionality of chunk tapes is extended to make use of the available disk memory in addition to the main memory.

## 4.9 First-order Adjoint Mode: Tape Data Types

### 4.9.1 Purpose

Individual types for function values, partial derivatives and adjoints can be specified allowing, for example, tape recording in a lower precision followed by propagation of adjoints in interval arithmetic assuming that an appropriate interval data type is available.

See Ch. 4.2 for general information on first-order adjoint mode.

### 4.9.2 Example

#### Example Text

Replace lines 7–10 in the driver in Sec. 4.46.2 with

```
typedef long double DCO_VALUE_TYPE;
typedef double DCO_PARTIAL_TYPE;
typedef float DCO_ADJOINT_TYPE;
typedef ga1s<DCO_VALUE_TYPE,DCO_PARTIAL_TYPE,DCO_ADJOINT_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
```

When printing the results in the main program, the expected accuracy is taken into account. Eighteen significant digits can be expected for the function values while the accuracy of the adjoints is reduced to six significant digits.

```
#include <iostream>
#include <vector>
```

```
using namespace std;

#include "driver.hpp"

int main() {
  const int n=4, m=2;
  vector<double> xv(n), xa(n), yv(m), ya(m);
  for (int i=0;i<n;i++) { xv[i]=1; xa[i]=1; }
  for (int i=0;i<m;i++) ya[i]=1;
  driver(xv,xa,yv,ya);
  cout.precision(18);
  for (int i=0;i<m;i++)
    cout << "y[" << i << "]=" << yv[i] << endl;
  cout.precision(6);
  for (int i=0;i<n;i++)
    cout << "x_{(1)}[" << i << "]=" << xa[i] << endl;
  return 0;
}
```

**Example Results**

The following output is generated:

```
y[0]=-2.79401891249194989
y[1]=-2.79401891249194989
x_{(1)}[0]=-4.58804
x_{(1)}[1]=-11.8191
x_{(1)}[2]=23.0501
x_{(1)}[3]=23.0501
```

### 4.9.3 Concepts introduced in this Section

**`ga1s<DCO_VALUE_TYPE,DCO_PARTIAL_TYPE,DCO_ADJOINT_TYPE>`**

Generic first-order adjoint scalar mode with separate data types for function values `DCO_VALUE_TYPE`, partial derivative (`DCO_PARTIAL_TYPE`) and adjoints (`DCO_ADJOINT_TYPE`).

## 4.10 Second-order Tangent Mode

Example program can be found here: `examples/gt2s_gt1s`

### 4.10.1 Purpose

The second-order tangent version

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(1)} \\ \mathbf{y}^{(2)} \\ \mathbf{y}^{(1,2)} \end{pmatrix} := f^{(1,2)}(\mathbf{x}, \mathbf{x}^{(2)}, \mathbf{x}^{(1)}, \mathbf{x}^{(1,2)})$$

resulting from the application of dco/c++ to a multivariate vector function $f : \mathbb{R}^n \to \mathbb{R}^m$, $\mathbf{y} = f(\mathbf{x})$, computes

$$\begin{aligned}\mathbf{y} &:= f(\mathbf{x})\\ \mathbf{y}^{(2)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(2)} \rangle\\ \mathbf{y}^{(1)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1)} \rangle\\ \mathbf{y}^{(1,2)} &:= \langle \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \rangle + \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1,2)} \rangle .\end{aligned} \tag{4.3}$$

### 4.10.2 Example

For the same function $f : \mathbb{R}^4 \to \mathbb{R}^2$ used in Ch. 4.1 the driver computes (4.3) for $\mathbf{x} \hat{=}$ xv, $\mathbf{x}^{(1)} \hat{=}$ xt1, $\mathbf{x}^{(2)} \hat{=}$ xt2, $\mathbf{x}^{(1,2)} \hat{=}$ xt1t2, $\mathbf{y} \hat{=}$ yv, $\mathbf{y}^{(1)} \hat{=}$ yt1, $\mathbf{y}^{(2)} \hat{=}$ yt2, and $\mathbf{y}^{(1,2)} \hat{=}$ yt1t2.

```
#include <iostream>
#include "dco.hpp"

using namespace std;
using namespace dco;
typedef gt1s<double> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef gt1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

#include "f.hpp"

void driver(const vector<double>& xv, const vector<double>& xt1,
            const vector<double>& xt2, const vector<double>& xt1t2,
            vector<double>& yv, vector<double>& yt1, vector<double>& yt2,
            vector<double>& yt1t2) {
  const size_t n = xv.size(), m = yv.size();
  vector<DCO_TYPE> x(n), y(m);
  for (size_t i = 0; i < n; i++) {
    value(value(x[i])) = xv[i];
    derivative(value(x[i])) = xt1[i];
    value(derivative(x[i])) = xt2[i];
    derivative(derivative(x[i])) = xt1t2[i];
  }
  f(x, y);
  for (size_t i = 0; i < m; i++) {
    yv[i] = passive_value(y[i]);
    yt1[i] = derivative(value(y[i]));
    yt2[i] = value(derivative(y[i]));
    yt1t2[i] = derivative(derivative(y[i]));
  }
}
```

The main program initializes all variables on the right-hand side of (4.3) followed by calling the driver and printing the results.

```
#include <iostream>
#include <vector>
using namespace std;
```

```

#include "driver.hpp"

int main() {
  const int n = 4, m = 2;
  cout.precision(5);
  vector<double> xv(n), xt1(n), xt2(n), xt1t2(n);
  vector<double> yv(m), yt1(m), yt2(m), yt1t2(m);
  for (int i = 0; i < n; i++) {
    xv[i] = 1;
    xt1[i] = 1;
    xt2[i] = 1;
    xt1t2[i] = 1;
  }
  driver(xv, xt1, xt2, xt1t2, yv, yt1, yt2, yt1t2);
  for (int i = 0; i < m; i++)
    cout << "y[" << i << "]=" << yv[i] << endl;
  for (int i = 0; i < m; i++)
    cout << "y^{(1)}[" << i << "]=" << yt1[i] << endl;
  for (int i = 0; i < m; i++)
    cout << "y^{(2)}[" << i << "]=" << yt2[i] << endl;
  for (int i = 0; i < m; i++)
    cout << "y^{(1,2)}[" << i << "]=" << yt1t2[i] << endl;
}
```

The following output is generated:

```
y[0]=-2.79401891249195
y[1]=-2.79401891249195
y^{(1)}[0]=14.2435494001203
y^{(1)}[1]=11.4495304876283
y^{(2)}[0]=14.2435494001203
y^{(2)}[1]=11.4495304876283
y^{(1,2)}[0]=-149.94964806237
y^{(1,2)}[1]=-124.256568174621
```

### 4.10.3 Concepts introduced in this Section

**`passive_value`**

Returns reference to passive value (value of value for second derivative types) of argument.

## 4.11 Second-order Adjoint Mode

Example program can be found here: `examples/gt2s_ga1s`

### 4.11.1 Purpose

The second-order adjoint version

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(2)} \\ \mathbf{x}_{(1)} \\ \mathbf{x}_{(1)}^{(2)} \end{pmatrix} := f_{(1)}^{(2)}(\mathbf{x}, \mathbf{x}^{(2)}, \mathbf{x}_{(1)}, \mathbf{x}_{(1)}^{(2)}, \mathbf{y}, \mathbf{y}^{(2)}, \mathbf{y}_{(1)}, \mathbf{y}_{(1)}^{(2)})$$

resulting from the application of dco/c++ to a multivariate vector function $f : \mathbb{R}^n \to \mathbb{R}^m$, $\mathbf{y} = f(\mathbf{x})$, computes

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{y}^{(2)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(2)} \rangle \\ \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \frac{\partial f}{\partial \mathbf{x}} \rangle \\ \mathbf{x}_{(1)}^{(2)} &:= \mathbf{x}_{(1)}^{(2)} + \langle \mathbf{y}_{(1)}^{(2)}, \frac{\partial f}{\partial \mathbf{x}} \rangle + \langle \mathbf{y}_{(1)}, \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(2)} \rangle . \end{aligned} \tag{4.4}$$

### 4.11.2 Example

For a similar function $f : \mathbb{R}^4 \to \mathbb{R}$ used in Ch. 4.1 (the two outputs are just added up) the main program computes the Hessian.

```
// This example demostrates basic use of 2nd order adjoint mode. It computes the
// nxn Hessian by letting the input tangents iterate over the
// Cartesian basis vectors in R^n.
#include "dco.hpp"

// Implementation of y = f(x), which we differentiate in main.
// n = 4, m = 1.
// NOTE: We've change the function to only have a single output.
template <typename T> T f(const std::vector<T>& x) {
  std::vector<T> y(2);
  T v = tan(x[2] * x[3]);
  T w = x[1] - v;
  y[0] = x[0] * v / w;
  y[1] = y[0] * x[1];
  return y[0] + y[1];
}

int main() {
  // Using second-order adjoint mode.
  constexpr std::size_t n = 4;
  using mode_t = dco::ga1s<dco::gt1s<double>::type>;
  using type = mode_t::type;

  // Inputs.
  std::vector<type> x(n, 1.0);

  dco::smart_tape_ptr_t<mode_t> tape;

  // Iterate over the Cartesian basis vectors in R^m.
  for (std::size_t j = 0; j < n; ++j) {
    tape->register_variable(x);

    // Seed j-th Cartesian basis vector to the output adjoints.
    dco::derivative(dco::value(x[j])) = 1.0;

    // Record tape for f. This is only required once!
    auto y = f(x);

    // Print output value.
    if (j == 0) {
      std::cout << "y = " << y << std::endl;
    }

    tape->register_output_variable(y);
    dco::derivative(y) = 1.0;

    // Interpret the tape for j-th output.
    tape->interpret_adjoint();

    // Print Hessian line by line.
    std::cout << "ddy[" << j << "] = [ ";
    for (std::size_t i = 0; i < n; ++i) {
      std::cout << dco::derivative(dco::derivative(x[i])) << " ";
    }
    std::cout << "]" << std::endl;

    // Reset all adjoints to zero.
    tape->reset();
    dco::derivative(dco::value(x[j])) = 0.0;
  }
}
```

```
y = -5.58804
ddy[0] = [ 0 -12.8191 22.0501 22.0501 ]
ddy[1] = [ -12.8191 -45.9953 112.192 112.192 ]
ddy[2] = [ 22.0501 112.192 -202.333 -180.283 ]
ddy[3] = [ 22.0501 112.192 -180.283 -202.333 ]
```

### 4.11.3 Concepts introduced in this Section

None.

## 4.12 Approximate Second-order Adjoint Mode

Example program can be found here: `examples/fd_ga1s`

### 4.12.1 Purpose

The approximate second-order adjoint version

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(2)} \\ \mathbf{x}_{(1)} \\ \mathbf{x}_{(1)}^{(2)} \end{pmatrix} := f_{(1)}^{(2)}(\mathbf{x}, \mathbf{x}^{(2)}, \mathbf{x}_{(1)}, \mathbf{x}_{(1)}^{(2)}, \mathbf{y}, \mathbf{y}^{(2)}, \mathbf{y}_{(1)}, \mathbf{y}_{(1)}^{(2)})$$

resulting from the application of finite differences to a dco/c++ first-order adjoint version of a multivariate vector function $f : \mathbb{R}^n \to \mathbb{R}^m$, $\mathbf{y} = f(\mathbf{x})$, computes

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{y}^{(2)} &:\approx \langle f(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\ \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \frac{\partial f}{\partial \mathbf{x}} \rangle \\ \mathbf{x}_{(1)}^{(2)} &:\approx \mathbf{x}_{(1)}^{(2)} + \langle \mathbf{y}_{(1)}^{(2)}, \frac{\partial f}{\partial \mathbf{x}} \rangle + \langle \mathbf{y}_{(1)}, \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(2)} \rangle . \end{aligned} \tag{4.5}$$

### 4.12.2 Example

```
#include <vector>
#include "dco.hpp"

using namespace std;
using namespace dco;

typedef double DCO_BASE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "f.hpp"

void driver(const vector<double>& xv, vector<double>& xa, vector<double>& yv,
            vector<double>& ya) {
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  size_t n = xv.size(), m = yv.size();
  vector<DCO_TYPE> x(n), y(m);
  for (size_t i = 0; i < n; i++) {
    x[i] = xv[i];
    tape->register_variable(x[i]);
  }
  f(x, y);
  for (size_t i = 0; i < m; i++) {
    tape->register_output_variable(y[i]);
    yv[i] = value(y[i]);
    derivative(y[i]) = ya[i];
  }
  for (size_t i = 0; i < n; i++)
    derivative(x[i]) = xa[i];
  tape->interpret_adjoint();
  for (size_t i = 0; i < n; i++)
```

```
    xa[i] = derivative(x[i]);
  for (size_t i = 0; i < m; i++)
    ya[i] = derivative(y[i]);
}

#include <cfloat>

void fd_driver(const vector<double>& xv, vector<double>& xa,
               vector<double>& xa1t2x, vector<double>& yv, vector<double>& yt2x,
               vector<double>& ya) {
  size_t n = xv.size(), m = yv.size();
  vector<double> h(n), xap(n), xvp(n), yvp(m), yap(m);
  for (size_t i = 0; i < m; i++)
    yap[i] = ya[i];
  for (size_t i = 0; i < n; i++) {
    xap[i] = xa[i];
    h[i] = (xv[i] == 0) ? sqrt(DBL_EPSILON) : sqrt(DBL_EPSILON) * abs(xv[i]);
    xvp[i] = xv[i] + h[i];
  }
  driver(xv, xa, yv, ya);
  driver(xvp, xap, yvp, yap);
  for (size_t i = 0; i < m; i++)
    yt2x[i] = (yvp[i] - yv[i]) / h[i];
  for (size_t i = 0; i < n; i++)
    xa1t2x[i] = (xap[i] - xa[i]) / h[i];
}
```

```
#include <iostream>
#include <vector>
using namespace std;

#include "driver.hpp"

int main() {
  const int n = 4, m = 2;
  cout.precision(5);
  vector<double> xv(n), xa(n), xa1t2x(n), yv(m), yt2x(m), ya(m);
  for (int i = 0; i < n; i++) {
    xv[i] = 1;
    xa[i] = 1;
  }
  for (int i = 0; i < m; i++)
    ya[i] = 1;
  fd_driver(xv, xa, xa1t2x, yv, yt2x, ya);
  for (int i = 0; i < m; i++)
    cout << "y[" << i << "]=" << yv[i] << endl;
  for (int i = 0; i < n; i++)
    cout << "x_{(1)}[" << i << "]=" << xa[i] << endl;
  for (int i = 0; i < m; i++)
    cout << "y^{(2)}[" << i << "]~=" << yt2x[i] << endl;
  for (int i = 0; i < n; i++)
    cout << "x_{(1)}^{(2)}[" << i << "]~=" << xa1t2x[i] << endl;
}
```

The following output is generated

```
y[0]=-2.79401891249195
y[1]=-2.79401891249195
x_{(1)}[0]=-4.5880378249839
x_{(1)}[1]=-11.8190644542335
x_{(1)}[2]=23.050091083483
x_{(1)}[3]=23.050091083483
y^{(2)}[0]~=14.2435482144356
y^{(2)}[1]~=11.4495295286179
x_{(1)}^{(2)}[0]~=31.2811151146889
x_{(1)}^{(2)}[1]~=165.569020390511
x_{(1)}^{(2)}[2]~=-248.374695301056
x_{(1)}^{(2)}[3]~=-248.374695301056
```

### 4.12.3 Concepts introduced in this Section

None.

## 4.13 Third-order Tangent Mode

Example program can be found here: `examples/gt3s_gt2s_gt1s`

### 4.13.1 Purpose

The second-order tangent version

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(3)} \\ \mathbf{y}^{(1)} \\ \mathbf{y}^{(1,3)} \\ \mathbf{y}^{(2)} \\ \mathbf{y}^{(2,3)} \\ \mathbf{y}^{(1,2)} \\ \mathbf{y}^{(1,2,3)} \end{pmatrix} := f^{(1,2,3)}(\mathbf{x}, \mathbf{x}^{(3)}, \mathbf{x}^{(2)}, \mathbf{x}^{(2,3)}, \mathbf{x}^{(1)}, \mathbf{x}^{(1,3)}, \mathbf{x}^{(1,2)}\mathbf{x}^{(1,2,3)})$$

of a multivariate vector function $f : \mathbb{R}^n \to \mathbb{R}^m$, $\mathbf{y} = f(\mathbf{x})$, computes

$$
\begin{aligned}
\mathbf{y} &:= f(\mathbf{x}) \\
\mathbf{y}^{(3)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(3)} \rangle \\
\mathbf{y}^{(2)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(2)} \rangle \\
\mathbf{y}^{(2,3)} &:= \langle \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle + \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(2,3)} \rangle \\
\mathbf{y}^{(1)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1)} \rangle \\
\mathbf{y}^{(1,3)} &:= \langle \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(1)}, \mathbf{x}^{(3)} \rangle + \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1,3)} \rangle \\
\mathbf{y}^{(1,2)} &:= \langle \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \rangle + \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1,2)} \rangle \\
\mathbf{y}^{(1,2,3)} &:= \langle \frac{\partial^2 f}{\partial \mathbf{x}^3}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle + \langle \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(1,3)}, \mathbf{x}^{(2)} \rangle + \langle \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(1)}, \mathbf{x}^{(2,3)} \rangle \\
&\quad + \langle \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(1,2)}, \mathbf{x}^{(3)} \rangle + \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1,2,3)} \rangle .
\end{aligned}
$$

### 4.13.2 Example

```
#include <iostream>
#include "dco.hpp"

using namespace std;
using namespace dco;

typedef gt1s<double> DCO_BASE_BASE_MODE;
typedef DCO_BASE_BASE_MODE::type DCO_BASE_BASE_TYPE;
typedef gt1s<DCO_BASE_BASE_TYPE> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef gt1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

#include "f.hpp"

void driver(const vector<double>& xv, const vector<double>& xt3,
            const vector<double>& xt1, const vector<double>& xt1t3,
            const vector<double>& xt2, const vector<double>& xt2t3,
            const vector<double>& xt1t2, const vector<double>& xt1t2t3,
            vector<double>& yv, vector<double>& yt3, vector<double>& yt1,
            vector<double>& yt1t3, vector<double>& yt2, vector<double>& yt2t3,
            vector<double>& yt1t2, vector<double>& yt1t2t3) {
  const size_t n = xv.size(), m = yv.size();
  vector<DCO_TYPE> x(n), y(m);
  for (size_t i = 0; i < n; i++) {
    value(value(value(x[i]))) = xv[i];
    value(value(derivative(x[i]))) = xt1[i];
    value(derivative(value(x[i]))) = xt2[i];
    derivative(value(value(x[i]))) = xt3[i];
    value(derivative(derivative(x[i]))) = xt1t2[i];
    derivative(derivative(value(x[i]))) = xt2t3[i];
    derivative(value(derivative(x[i]))) = xt1t3[i];
    derivative(derivative(derivative(x[i]))) = xt1t2t3[i];
  }
  f(x, y);
  for (size_t j = 0; j < m; j++) {
    yt1t2t3[j] = derivative(derivative(derivative(y[j])));
    yt2t3[j] = derivative(derivative(value(y[j])));
    yt1t3[j] = derivative(value(derivative(y[j])));
    yt1t2[j] = value(derivative(derivative(y[j])));
    yt1[j] = value(value(derivative(y[j])));
    yt2[j] = value(derivative(value(y[j])));
    yt3[j] = derivative(value(value(y[j])));
    yv[j] = value(value(value(y[j])));
  }
}
```

```
#include <iostream>
#include <vector>
using namespace std;

#include "driver.hpp"
```

```

int main() {
  const int n = 4, m = 2;
  cout.precision(5);
  vector<double> x(n), xt3(n), xt1(n), xt1t3(n), xt2(n), xt2t3(n), xt1t2(n),
      xt1t2t3(n);
  vector<double> y(m), yt3(m), yt1(m), yt1t3(m), yt2(m), yt2t3(m), yt1t2(m),
      yt1t2t3(m);
  for (int i = 0; i < n; i++)
    x[i] = xt3[i] = xt1[i] = xt1t3[i] = xt2[i] = xt2t3[i] = xt1t2[i] =
        xt1t2t3[i] = 1;
  driver(x, xt3, xt1, xt1t3, xt2, xt2t3, xt1t2, xt1t2t3, y, yt3, yt1, yt1t3,
         yt2, yt2t3, yt1t2, yt1t2t3);
  for (int j = 0; j < m; j++)
    cout << "y[" << j << "]=" << y[j] << endl;
  for (int j = 0; j < m; j++)
    cout << "y^{(3)}[" << j << "]=" << yt3[j] << endl;
  for (int j = 0; j < m; j++)
    cout << "y^{(1)}[" << j << "]=" << yt1[j] << endl;
  for (int j = 0; j < m; j++)
    cout << "y^{(1,3)}[" << j << "]=" << yt1t3[j] << endl;
  for (int j = 0; j < m; j++)
    cout << "y^{(2)}[" << j << "]=" << yt2[j] << endl;
  for (int j = 0; j < m; j++)
    cout << "y^{(2,3)}[" << j << "]=" << yt2t3[j] << endl;
  for (int j = 0; j < m; j++)
    cout << "y^{(1,2)}[" << j << "]=" << yt1t2[j] << endl;
  for (int j = 0; j < m; j++)
    cout << "y^{(1,2,3)}[" << j << "]=" << yt1t2t3[j] << endl;
}
```

The following output is generated:

```
y[0]=-2.79401891249195
y[1]=-2.79401891249195
y^{(3)}[0]=14.2435494001203
y^{(3)}[1]=11.4495304876283
y^{(1)}[0]=14.2435494001203
y^{(1)}[1]=11.4495304876283
y^{(1,3)}[0]=-149.94964806237
y^{(1,3)}[1]=-124.256568174621
y^{(2)}[0]=14.2435494001203
y^{(2)}[1]=11.4495304876283
y^{(2,3)}[0]=-149.94964806237
y^{(2,3)}[1]=-124.256568174621
y^{(1,2)}[0]=-149.94964806237
y^{(1,2)}[1]=-124.256568174621
y^{(1,2,3)}[0]=2486.48615431459
y^{(1,2,3)}[1]=2079.36785832784
```

### 4.13.3 Concepts introduced in this Section

None.

## 4.14 Third-order Adjoint Mode

Example program can be found here: `examples/gt3s_gt2s_ga1s`

### 4.14.1 Purpose

The third-order adjoint version

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

of a multivariate vector function $f : \mathbb{R}^n \to \mathbb{R}^m$, $\mathbf{y} = f(\mathbf{x})$, computes

$$\begin{aligned}
\mathbf{y} &:= f(\mathbf{x}) \\
\mathbf{y}^{(3)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(3)} \rangle \\
\mathbf{y}^{(2)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(2)} \rangle \\
\mathbf{y}^{(2,3)} &:= \langle \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle + \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(2,3)} \rangle \\
\mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \frac{\partial f}{\partial \mathbf{x}} \rangle \\
\mathbf{x}_{(1)}^{(3)} &:= \mathbf{x}_{(1)}^{(3)} + \langle \mathbf{y}_{(1)}^{(3)}, \frac{\partial f}{\partial \mathbf{x}} \rangle + \langle \mathbf{y}_{(1)}, \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(3)} \rangle \\
\mathbf{x}_{(1)}^{(2)} &:= \mathbf{x}_{(1)}^{(2)} + \langle \mathbf{y}_{(1)}^{(2)}, \frac{\partial f}{\partial \mathbf{x}} \rangle + \langle \mathbf{y}_{(1)}, \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(2)} \rangle \\
\mathbf{x}_{(1)}^{(2,3)} &:= \mathbf{x}_{(1)}^{(2,3)} + \langle \mathbf{y}_{(1)}^{(2,3)}, \frac{\partial f}{\partial \mathbf{x}} \rangle + \langle \mathbf{y}_{(1)}^{(2)}, \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(3)} \rangle + \langle \mathbf{y}_{(1,2)}, \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(2)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}, \frac{\partial^3 f}{\partial \mathbf{x}^3}, \mathbf{x}^{(2)}, \mathbf{x}(3) \rangle + \langle \mathbf{y}_{(1)}, \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(2,3)} \rangle .
\end{aligned}$$

### 4.14.2 Example

```
#include <iostream>
#include <vector>
#include "dco.hpp"

using namespace std;
using namespace dco;

typedef gt1s<double> DCO_BASE_BASE_MODE;
typedef DCO_BASE_BASE_MODE::type DCO_BASE_BASE_TYPE;
typedef gt1s<DCO_BASE_BASE_TYPE> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "f.hpp"

void driver(const vector<double>& xv, const vector<double>& xt3,
            const vector<double>& xt2, const vector<double>& xt2t3,
            vector<double>& xa1, vector<double>& xa1t3, vector<double>& xa1t2,
            vector<double>& xa1t2t3, vector<double>& yv, vector<double>& yt3,
            vector<double>& yt2, vector<double>& yt2t3, vector<double>& ya1,
            vector<double>& ya1t3, vector<double>& ya1t2,
            vector<double>& ya1t2t3) {
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  const size_t n = xv.size(), m = yv.size();
  vector<DCO_TYPE> x(n), y(m);
  for (size_t i = 0; i < n; i++) {
    tape->register_variable(x[i]);
    value(value(value(x[i]))) = xv[i];
    derivative(value(value(x[i]))) = xt3[i];
    value(derivative(value(x[i]))) = xt2[i];
    derivative(derivative(value(x[i]))) = xt2t3[i];
  }
  f(x, y);
  for (size_t i = 0; i < n; i++) {
    value(value(derivative(x[i]))) = xa1t3[i];
    derivative(value(derivative(x[i]))) = xa1t3[i];
    value(derivative(derivative(x[i]))) = xa1t2[i];
    derivative(derivative(derivative(x[i]))) = xa1t2t3[i];
  }
  for (size_t i = 0; i < m; i++) {
    yv[i] = value(value(value(y[i])));
    yt3[i] = derivative(value(value(y[i])));
    yt2[i] = value(derivative(value(y[i])));
    yt2t3[i] = derivative(derivative(value(y[i])));
    tape->register_output_variable(y[i]);
    derivative(derivative(derivative(y[i]))) = ya1t2t3[i];
    value(derivative(derivative(y[i]))) = ya1t2[i];
    derivative(value(derivative(y[i]))) = ya1t3[i];
    value(value(derivative(y[i]))) = ya1[i];
  }
  tape->interpret_adjoint();
  for (size_t i = 0; i < n; i++) {
    xa1t2t3[i] = derivative(derivative(derivative(x[i])));
    xa1t2[i] = value(derivative(derivative(x[i])));
    xa1t3[i] = derivative(value(derivative(x[i])));
    xa1[i] = value(value(derivative(x[i])));
  }
  for (size_t i = 0; i < m; i++) {
    ya1t2t3[i] = derivative(derivative(derivative(y[i])));
    ya1t2[i] = value(derivative(derivative(y[i])));
    ya1t3[i] = derivative(value(derivative(y[i])));
    ya1[i] = value(value(derivative(y[i])));
  }
}
```

```
#include <iostream>
#include <vector>
using namespace std;

#include "driver.hpp"

int main() {
  const int n = 4, m = 2;
  vector<double> x(n), xt3(n), xa1(n), xa1t3(n), xt2(n), xt2t3(n), xa1t2(n),
      xa1t2t3(n);
  vector<double> y(m), yt3(m), ya1(m), ya1t3(m), yt2(m), yt2t3(m), ya1t2(m),
      ya1t2t3(m);
  // initialization of inputs
  for (int i = 0; i < n; i++)
    x[i] = xt3[i] = xt2[i] = xt2t3[i] = xa1[i] = xa1t3[i] = xa1t2[i] =
        xa1t2t3[i] = 1;
  for (int j = 0; j < m; j++)
    ya1[j] = ya1t3[j] = ya1t2[j] = ya1t2t3[j] = 1;
  // driver
  driver(x, xt3, xt2, xt2t3, xa1, xa1t3, xa1t2, xa1t2t3, y, yt3, yt2, yt2t3,
         ya1, ya1t3, ya1t2, ya1t2t3);
  // results
  for (int j = 0; j < m; j++)
    cout << "y[" << j << "]=" << y[j] << endl;
  for (int j = 0; j < m; j++)
    cout << "y^{(3)}[" << j << "]=" << yt3[j] << endl;
  for (int i = 0; i < n; i++)
    cout << "x_{(1)}[" << i << "]=" << xa1[i] << endl;
  for (int i = 0; i < n; i++)
    cout << "x_{(1)}^{(3)}[" << i << "]=" << xa1t3[i] << endl;
  for (int j = 0; j < m; j++)
    cout << "y^{(2)}[" << j << "]=" << yt2[j] << endl;
  for (int j = 0; j < m; j++)
    cout << "y^{(2,3)}[" << j << "]=" << yt2t3[j] << endl;
  for (int i = 0; i < n; i++)
    cout << "x_{(1)}^{(2)}[" << i << "]=" << xa1t2[i] << endl;
  for (int i = 0; i < n; i++)
    cout << "x_{(1)}^{(2,3)}[" << i << "]=" << xa1t2t3[i] << endl;
  for (int j = 0; j < m; j++)
```

```
    cout << "y_{(1)}[" << j << "]=" << ya1[j] << endl;
  for (int j = 0; j < m; j++)
    cout << "y_{(1)}^{(3)}[" << j << "]=" << ya1t3[j] << endl;
  for (int j = 0; j < m; j++)
    cout << "y_{(1)}^{(2)}[" << j << "]=" << ya1t2[j] << endl;
  for (int j = 0; j < m; j++)
    cout << "y_{(1)}^{(2,3)}[" << j << "]=" << ya1t2t3[j] << endl;
}
```

The following output is generated:

```
y[0]=-2.79402
y[1]=-2.79402
y^{(3)}[0]=14.2435
y^{(3)}[1]=11.4495
x_{(1)}[0]=-4.58804
x_{(1)}[1]=-11.8191
x_{(1)}[2]=23.0501
x_{(1)}[3]=23.0501
x_{(1)}^{(3)}[0]=26.6931
x_{(1)}^{(3)}[1]=153.75
x_{(1)}^{(3)}[2]=-225.325
x_{(1)}^{(3)}[3]=-225.325
y^{(2)}[0]=14.2435
y^{(2)}[1]=11.4495
y^{(2,3)}[0]=-149.95
y^{(2,3)}[1]=-124.257
x_{(1)}^{(2)}[0]=26.6931
x_{(1)}^{(2)}[1]=153.75
x_{(1)}^{(2)}[2]=-225.325
x_{(1)}^{(2)}[3]=-225.325
x_{(1)}^{(2,3)}[0]=-273.206
x_{(1)}^{(2,3)}[1]=-2503.41
x_{(1)}^{(2,3)}[2]=3686.08
x_{(1)}^{(2,3)}[3]=3686.08
y_{(1)}[0]=1
y_{(1)}[1]=1
y_{(1)}^{(3)}[0]=1
y_{(1)}^{(3)}[1]=1
y_{(1)}^{(2)}[0]=1
y_{(1)}^{(2)}[1]=1
y_{(1)}^{(2,3)}[0]=1
y_{(1)}^{(2,3)}[1]=1
```

### 4.14.3 Concepts introduced in this Section

None.

## 4.15 First-order Tangent Vector Mode

Example program can be found here: `examples/gt1v`

### 4.15.1 Purpose

The first-order vector tangent version

$$\begin{pmatrix} \mathbf{y} \\ Y^{(1)} \end{pmatrix} := f^{(1)}(\mathbf{x}, X^{(1)})$$

of a multivariate vector function $f : \mathbb{R}^n \to \mathbb{R}^m$, $\mathbf{y} = f(\mathbf{x})$, computes

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ Y^{(1)} &:= \frac{\partial f}{\partial \mathbf{x}} \cdot X^{(1)}. \end{aligned}$$

### 4.15.2 Example

```cpp
// This example demostrates basic use of tangent vector mode. It
// computes the mxn Jacobian by seeding the input tangents to the
// identity matrix in R^n.
#include "dco.hpp"

// Implementation of y = f(x), which we differentiate in main.
// n = 4, m = 2.
template <typename T> std::vector<T> f(const std::vector<T>& x) {
  std::vector<T> y(2);
  T v = tan(x[2] * x[3]);
  T w = x[1] - v;
  y[0] = x[0] * v / w;
  y[1] = y[0] * x[1];
  return y;
}

int main() {
  // Using first-order tangent vector mode.
  constexpr std::size_t n = 4, m = 2;
  using mode_t = dco::gt1v<double, n>;
  using type = mode_t::type;

  // Inputs.
  std::vector<type> x(n, 1.0);

  // Seed identity matrix to the input tangents.
  for (std::size_t i = 0; i < n; ++i) {
    dco::derivative(x[i])[i] = 1.0;
  }

  // Evaluate function only once. Implicitly performs Jacobian matrix
  // product, the matrix being the identity in R^n.
  auto y = f(x);

  // Print output values.
  std::cout << "y = [ ";
  for (auto const& yi : y) {
    std::cout << yi << " ";
  }
  std::cout << "]" << std::endl;

  // Print Jacobian.
  for (std::size_t i = 0; i < n; ++i) {
    std::cout << "dy / dx[" << i << "] = [ ";
    for (std::size_t j = 0; j < m; ++j) {
      std::cout << dco::derivative(y[j])[i] << " ";
    }
    std::cout << "]" << std::endl;
  }
}
```

The following output is generated:

```
y = [ -2.79402 -2.79402 ]
dy / dx[0] = [ -2.79402 -2.79402 ]
dy / dx[1] = [ -5.01252 -7.80654 ]
dy / dx[2] = [ 11.025 11.025 ]
dy / dx[3] = [ 11.025 11.025 ]
```

### 4.15.3 Concepts introduced in this Section

**`derivative`**

Returns reference to vector of derivatives of argument; individual elements of returned object can be accessed through `DCO_BASE_TYPE& operator[](int);`.

## 4.16 First-order Adjoint Vector Mode

Example program can be found here: `examples/ga1v`

### 4.16.1 Purpose

The first-order vector adjoint version

$$\begin{pmatrix} \mathbf{y} \\ X_{(1)} \end{pmatrix} := f_{(1)}(\mathbf{x}, X_{(1)}, \mathbf{y}, Y_{(1)})$$

resulting from the application of dco/c++ to a multivariate vector function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, $\mathbf{y} = f(\mathbf{x})$, computes

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ X_{(1)} &:= X_{(1)} + \langle Y_{(1)} \cdot \frac{\partial f}{\partial \mathbf{x}} \rangle \equiv X_{(1)} + \left(\frac{\partial f}{\partial \mathbf{x}}\right)^T \cdot Y_{(1)}. \end{aligned} \tag{4.6}$$

### 4.16.2 Example

```
// This example demostrates basic use of adjoint vector mode. It
// computes the mxn Jacobian by seeding the output adjoints to the
// identity matrix in R^m.
#include "dco.hpp"

```

```
// Implementation of y = f(x), which we differentiate in main.
// n = 4, m = 2.
template <typename T> std::vector<T> f(const std::vector<T>& x) {
  std::vector<T> y(2);
  T v = tan(x[2] * x[3]);
  T w = x[1] - v;
  y[0] = x[0] * v / w;
  y[1] = y[0] * x[1];
  return y;
}

int main() {
  // Using first-order adjoint vector mode.
  constexpr std::size_t n = 4, m = 2;
  using mode_t = dco::ga1v<double, m>;
  using type = mode_t::type;

  // Inputs.
  std::vector<type> x(n, 1.0);

  dco::smart_tape_ptr_t<mode_t> tape;
  tape->register_variable(x);

  // Record tape for f. This is only required once!
  auto y = f(x);

  tape->register_output_variable(y);

  // Seed identity matrix in the output adjoints.
  for (std::size_t i = 0; i < m; ++i) {
    dco::derivative(y[i])[i] = 1.0;
  }
  // Interpret the tape. Also only once! Implicitly performs the
  // transpoed Jacobian matrix product, the matrix being the identity
  // in R^m.
  tape->interpret_adjoint();

  // Print output values.
  std::cout << "y = [ ";
  for (auto const& yi : y) {
    std::cout << yi << " ";
  }
  std::cout << "]" << std::endl;

  // Print Jacobian.
  for (std::size_t j = 0; j < m; ++j) {
    std::cout << "dy[" << j << "] = [ ";
    for (std::size_t i = 0; i < n; ++i) {
      std::cout << dco::derivative(x[i])[j] << " ";
    }
    std::cout << "]" << std::endl;
  }
}
```

The following output is generated:

```
y = [ -2.79402 -2.79402 ]
dy[0] = [ -2.79402 -5.01252 11.025 11.025 ]
dy[1] = [ -2.79402 -7.80654 11.025 11.025 ]
```

### 4.16.3 Concepts introduced in this Section

None.

## 4.17 Adjoint First-order Scalar Mode: Multiple Tapes

Example program can be found here: `examples/ga1sm`

### 4.17.1 Purpose

The `dco/c++` type `ga1sm<DCO_BASE_TYPE>::type` enables the use of multiple tapes in adjoint first-order scalar mode. Apart from the mode, the example is identical to Ch. 4.2.

### 4.17.2 Example

For the same function $f : \mathbb{R}^4 \to \mathbb{R}^2$ used in Ch. 4.1, the following output is generated:

```
y = [ -2.79402 -2.79402 ]
dy[0] = [ -2.79402 -5.01252 11.025 11.025 ]
dy[1] = [ -2.79402 -7.80654 11.025 11.025 ]
```

### 4.17.3 Concepts introduced in this Section

**`ga1sm<DCO_BASE_TYPE>`**

Generic adjoint first-order scalar mode enabling use of multiple tapes.

**`ga1sm<DCO_BASE_TYPE>::type`**

Generic adjoint first-order scalar type enabling use of multiple tapes.

## 4.18 First-order Adjoint Vector Mode: Multiple Tapes

Example program can be found here: `examples/ga1vm`

### 4.18.1 Purpose

The `dco/c++` type `ga1vm<DCO_BASE_TYPE>::type` enables the use of multiple tapes in adjoint first-order vector mode.

### 4.18.2 Example

This example assumes access to the header-only version of dco/c++.

```
#include <vector>
#include "dco.hpp"

using namespace std;
using namespace dco;

const std::size_t l = 5;

typedef double DCO_BASE_TYPE;
typedef ga1vm<DCO_BASE_TYPE, l> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "f.hpp"

void driver(const vector<double>& xv, vector<vector<double>>& xa,
            vector<double>& yv, vector<vector<double>>& ya) {
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  size_t n = xv.size(), m = yv.size();
  vector<DCO_TYPE> x(n), y(m);
  for (size_t i = 0; i < n; i++) {
    x[i] = xv[i];
    tape->register_variable(x[i]);
  }
  f(x, y);
  for (std::size_t i = 0; i < m; i++) {
    tape->register_output_variable(y[i]);
    yv[i] = value(y[i]);
    for (std::size_t j = 0; j < l; j++)
      derivative(y[i])[j] = ya[i][j];
  }
  for (size_t i = 0; i < n; i++) {
    for (std::size_t j = 0; j < l; j++)
      derivative(x[i])[j] = xa[i][j];
  }
  tape->interpret_adjoint();
  for (size_t i = 0; i < n; i++) {
    for (std::size_t j = 0; j < l; j++)
      xa[i][j] = derivative(x[i])[j];
  }
  for (size_t i = 0; i < m; i++) {
    for (std::size_t j = 0; j < l; j++)
      ya[i][j] = derivative(y[i])[j];
  }
}
```

```
#include <iostream>
#include <vector>
using namespace std;

#include "driver.hpp"
```

```

int main() {
  const int n = 4, m = 5;
  cout.precision(5);
  vector<double> xv(n), yv(m);
  vector<vector<double>> xa(n, vector<double>(m)), ya(m, vector<double>(m));
  for (int i = 0; i < n; i++) {
    xv[i] = 1;
    for (int j = 0; j < m; j++)
      xa[i][j] = 0;
  }
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < m; j++)
      ya[i][j] = 0;
    ya[i][i] = 1;
  }
  driver(xv, xa, yv, ya);
  for (int i = 0; i < m; i++)
    cout << "y[" << i << "]=" << yv[i] << endl;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      cout << "x_{(1)}[" << i << "][" << j << "]=" << xa[i][j] << endl;
}
```

The following output is generated:

```
y[0]=-2.79401891249195
y[1]=-2.79401891249195
x_{(1)}[0][0]=-2.79401891249195
x_{(1)}[0][1]=-2.79401891249195
x_{(1)}[1][0]=-5.01252277087075
x_{(1)}[1][1]=-7.8065416833627
x_{(1)}[2][0]=11.0250455417415
x_{(1)}[2][1]=11.0250455417415
x_{(1)}[3][0]=11.0250455417415
x_{(1)}[3][1]=11.0250455417415
```

### 4.18.3 Concepts introduced in this Section

None.

## 4.19 Binary Compatible Passive Type: gbcp

Example program can be found here: `examples/gbcp`

### 4.19.1 Purpose

This type can be used to turn parts of an adjoint computation passive, i.e. no tape activity is performed for objects of this type. Declaring a gbcp type of an adjoint type results in equal type sizes (=binary compatibility), while the gbcp type only provides access to the value object of the active type. A gbcp type can safely be cast to its value type when performing passive computations. Chaining gbcp types can be used to access any lower order of a higher order active type.

Note that usage of the gbcp type requires the compiler to have C++11 capabilities.

### 4.19.2 Example

```
template <typename T, typename U> void f(const vector<T>& x, vector<U>& y) {
  T v = tan(x[2] * x[3]);
  T w = x[1] - v;
  y[0] = x[0] * v / w;
  y[1] = y[0] * x[1];
}
```

```
#include <vector>
#include "dco.hpp"

using namespace std;
using namespace dco;

typedef double DCO_BASE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "f.hpp"

void driver(const vector<double>& xv, vector<double>& xa, vector<double>& yv,
            vector<double>& ya, vector<double>& yp) {
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  size_t n = xv.size(), m = yv.size();
  vector<DCO_TYPE> x(begin(xv), end(xv)), y(m);
  tape->register_variable(x);
  f(x, y);

  std::vector<double> py(n);

  typedef gbcp<DCO_TYPE>::type DCO_GBCP_TYPE;
  auto const& px = create_gbcp<vector<DCO_GBCP_TYPE>>(x);
  f(px, py);

  for (size_t i = 0; i < m; i++) {
    tape->register_output_variable(y[i]);
    yv[i] = value(y[i]);
    derivative(y[i]) = ya[i];
  }
  for (size_t i = 0; i < n; i++)
    derivative(x[i]) = xa[i];

  tape->write_to_dot();
  tape->interpret_adjoint();
  for (size_t i = 0; i < n; i++)
    xa[i] = derivative(x[i]);
  for (size_t i = 0; i < m; i++)
    ya[i] = derivative(y[i]);
42   for (size_t i = 0; i < m; i++)
43     yp[i] = py[i];
44 }
```

```
#include <iostream>
#include <vector>
using namespace std;

#include "driver.hpp"

int main() {
  const size_t n = 4, m = 2;
  cout.precision(5);
  vector<double> xv(n), xa(n), yv(m), ya(m), yp(m);
  for (size_t i = 0; i < n; i++) {
    xv[i] = 1;
    xa[i] = 1;
  }
  for (size_t i = 0; i < m; i++)
    ya[i] = 1;
  driver(xv, xa, yv, ya, yp);
  for (size_t i = 0; i < m; i++)
    cout << "y[" << i << "]=" << yv[i] << endl;
  for (size_t i = 0; i < m; i++)
    cout << "yp[" << i << "]=" << yp[i] << endl;
  for (size_t i = 0; i < n; i++)
    cout << "x_{(1)}[" << i << "]=" << xa[i] << endl;
}
```

### 4.19.3 Example Text

A gbcp pointer array named `px` which provides passive access to the input vector of active types `x` is created in line 28 of the driver. The array points to the exact same memory space of `x`, but can be used in computations together with the passive type (line 30). Implicit casting to the passive type is supported and done for example in line 10 of the `f()` implementation.

### 4.19.4 Example Results

The following output is generated.

```
y[0]=-2.79401891249195
y[1]=-2.79401891249195
yp[0]=-2.79401891249195
yp[1]=-2.79401891249195
x_{(1)}[0]=-4.5880378249839
x_{(1)}[1]=-11.8190644542334
x_{(1)}[2]=23.050091083483
x_{(1)}[3]=23.050091083483
```

The passive computation returns the same result as the primal of the active computation which is not surprising since the inputs share the same memory addresses.

### 4.19.5 Concepts introduced in this Section

**`gbcp<DCO_ACTIVE_TYPE>::type`**

gbcp type with binary compatibility to `DCO_ACTIVE_TYPE` while providing casting capabilities to the `value_t` of the `DCO_ACTIVE_TYPE`. Note that `DCO_ACTIVE_TYPE` can be another gbcp type; this chained construct results in `gbcp_order=M` for $M$ used gbcp types. The `DCO_ACTIVE_TYPE` of the most inner gbcp type can be an active type of arbitrary order $N$; however, only $0 < M \leq N$ is considered useful. The resulting type is binary compatible to an active type of order $N$, acts passively in computations of orders $M+1, ..., N$ and is active in order $N-M$.

**`create_gbcp<GBCP_TYPE>(ACTIVE_TYPE)`**

Interface to be used for creating a gbcp object of type `GBCP_TYPE` for an object of type `ACTIVE_TYPE`. The return type should be handled by the **`auto`** specifier. Both, references and pointers, are supported:

```
ACTIVE_TYPE x;

auto& ref_px     = dco::create_gbcp<GBCP_TYPE>(x);
auto  pointer_px = dco::create_gbcp<GBCP_TYPE*>(x);

auto& ref_px     = dco::create_gbcp<const GBCP_TYPE>(x);
auto  pointer_px = dco::create_gbcp<const GBCP_TYPE*>(x);
```

**`DCO_MODE::order`**

Integer containing the `order` $N$ of the corresponding active type.

**`DCO_MODE::gbcp_order`**

Integer containing the `gbcp_order` $M$ of the corresponding active type. Only available for gbcp types.

## 4.20 First-order Adjoint Mode: Basic Checkpointing

Example program can be found here: `examples/ga1s_joint`

### 4.20.1 Purpose

This section illustrates the *joint reversal* [4] of a subprogram call. The given solution allows for second and higher derivatives to be computed with minimal implementation effort; see Ch. 4.21.

### 4.20.2 Example

```
// This example demostrates basic checkpointing with dco/c++ for
// first-order adjoint mode. It computes the derivative of the
// scalar function y = f(x) with a standard dco/c++ first-order
// adjoint driver (see main). f(x) consists of three parts. First,
// it computes a cosine of the input, then it calls g, and
// afterwards is computes cosine of the output of g and returns
// that. g is checkpointed in g_make_gap.

#include "dco.hpp"

// g_make_gap uses dco/c++'s checkpointing interface; see
// implementation below.
template <typename T> T g_make_gap(T const& x);

// Implementation of g. It returns the sine of the input.
template <typename T> T g(T const& x) {
  T y = sin(x);
  return y;
}

// Implementation of y = f(x), which we differentiate in main.
template <typename T> T f(T const& x) {
  // Computation preceding function g. Tape is written for this part.
  T y = cos(x);
  // This function will write a checkpoint and computes g only
  // passively, without taping.
  y = g_make_gap(y);
  // Computation succeeding function g. Tape is written for this part.
  y = cos(y);
  return y;
}

int main() {
  // Using first-order adjoint mode.
  using mode_t = dco::ga1s<double>;
  using type = mode_t::type;

  // Input variable.
  type x = 2.1;

  dco::smart_tape_ptr_t<mode_t> tape;
  tape->register_variable(x);

  // Record tape for f; this includes the checkpoint.
  type y = f(x);

  tape->register_output_variable(y);

  dco::derivative(y) = 1.0;
  // Interpret the tape. This will call the fill_gap function (see
  // implementation of g_make_gap below).
  tape->interpret_adjoint();

  std::cout << "y = " << y << std::endl;
  std::cout << "x_{(1)} = " << dco::derivative(x) << std::endl;
}

// Creates a checkpoint and inserts the callback into the tape.
template <typename T> T g_make_gap(T const& x) {
  // Get tape pointer to tape x is registered in (tape).
  auto tape = dco::tape(x);

  // Compute the value of g with doubles (return type of
  // dco::value(x)). The active variable y is initialized with the
  // passive value.
  T y = g(dco::value(x));
  // Register variable y, since it is the output of g_make_gap;
  // succeeding computation is supposed to be taped, i.e. y is an
  // input to everything which comes later.
  tape->register_variable(y);

  // Callback. This lambda captures by copy ([=]) all accessed
  // data. In this case x and y. This callback is called during tape
  // interpretation, so all accessed data needs to be available
  // during tape interpretation time. The tape is used to fill the
  // gap.
  auto fill_gap = [=]() {
    // This is the current position of the tape.
    auto p = tape->get_position();
    // Record tape for g, since it's now executed with active types.
    // The captured input x is used directly.
    T yl = g(x);
    // The local output yl now lives in the tape (has a tape index).
    // It gets seeded with the adjoint of the original output
    // y. dco::derivative(y) holds the adjoint of the computation in
    // f which succeeded the call of g_make_gap.
    dco::derivative(yl) = dco::derivative(y);
    // Propagate adjoints into the inputs x and reset the tape we've
    // recorded in this callback.
    tape->interpret_adjoint_and_reset_to(p);
  };
  // We move the lambda into the tape as callback. No copy required,
  // since fill_gap is not required outside the tape.
  tape->insert_callback(std::move(fill_gap));

  return y;
}
```

The following output is generated:

```
y = 0.885293
x_{(1)} = -0.351344
```

### 4.20.3 Concepts introduced in this Section

**DCO_TAPE_TYPE::iterator_t**

Position in tape.

**DCO_TAPE_TYPE::get_position**

Returns current position in associated tape.

**DCO_TAPE_TYPE::interpret_adjoint_and_reset_to**

Runs tape interpreter and resets current position in tape to argument.

Remark: Make sure you are not using any active variables from the reset tape section afterwards.

**dco::mode<DCO_TYPE>**

DCO_MODE for given DCO_TYPE.

**DCO_MODE::value_t**

Type of value component of variables of type DCO_MODE::type.

**DCO_TAPE_TYPE::insert_callback**

Inserts callback into tape for filling the *gap*. The callback is passed as callable, i.e. either as a lambda or a functor. The user has to ensure that the captured state is constant and still alive / valid during tape interpretation. Easiest way of achieving that is by using capture by copy of the lambda [=]. Be especially careful, if capturing pointers. Use shared_pointers to constant data instead. If you know data is not used anymore at a later stage, you can also make use of move capture for moving it into the lambda. The callable can be copied or moved into the tape, use move semantics (see example) to save memory and avoid the copy. This can be used to save memory. See example source for further comments.

**DCO_TAPE_TYPE::reset_to**

Resets current tape position to argument.

Remark: Make sure you are not using any active variables from the reset tape section afterwards.

**dco::size_of(DCO_TAPE_TYPE*)**

Returns size of tape in memory; external adjoint object data is not included.

## 4.21 Second-order Adjoint Mode: Basic Checkpointing

Example program can be found here: examples/gt2s_ga1s_joint

### 4.21.1 Purpose

This section illustrates the computation of second derivatives for the joint reversal of a subprogram call.

### 4.21.2 Example

```
// This example demostrates basic checkpointing with dco/c++ for
// second-order adjoint mode. It computes the first- and second-order
// derivative of the scalar function y = f(x) with a standard dco/c++
// second-order adjoint driver (see main). f(x) consists of three
// parts. First, it computes a cosine of the input, then it calls g,
// and afterwards is computes cosine of the output of g and returns
// that. g is checkpointed in g_make_gap.

// The example is almost identical to the first-order one. Only the
// driver code changes (see main). Especially all the checkpointing
// logic doesn't have to change.

#include "dco.hpp"

// g_make_gap uses dco/c++'s checkpointing interface; see
// implementation below.
template <typename T> T g_make_gap(T const& x);

// Implementation of g. It returns the sine of the input.
template <typename T> T g(T const& x) {
  T y = sin(x);
  return y;
}

// Implementation of y = f(x), which we differentiate in main.
template <typename T> T f(T const& x) {
  // Computation preceding function g. Tape is written for this part.
  T y = cos(x);
  // This function will write a checkpoint and computes g only
  // passively, without taping.
  y = g_make_gap(y);
  // Computation succeeding function g. Tape is written for this part.
  y = cos(y);
  return y;
}

int main() {
  // Using second-order adjoint mode (forward-over-reverse).
  using mode_t = dco::ga1s<dco::gt1s<double>::type>;
  using type = mode_t::type;

  // Input variable.
  type x = 2.1;

  dco::smart_tape_ptr_t<mode_t> tape;
  tape->register_variable(x);

  // Seeding tangent direction. This is required in addition compared
  // to the first-order code.
  dco::derivative(dco::value(x)) = 1.0;

  // Record tape for f; this includes the checkpoint.
  type y = f(x);

  tape->register_output_variable(y);

  dco::derivative(y) = 1.0;
  // Interpret the tape. This will call the fill_gap function (see
  // implementation of g_make_gap below).
  tape->interpret_adjoint();

  std::cout << "y = " << y << std::endl;
  std::cout << "x_{(1)} = " << dco::derivative(x) << std::endl;
  // This is additional information compared to the first-order code.
  std::cout << "y^{(2)} = " << dco::derivative(dco::value(y)) << std::endl;
  std::cout << "x_{(1)}^{(2)} = " << dco::derivative(dco::derivative(x))
            << std::endl;
}

// Creates a checkpoint and inserts the callback into the tape.
template <typename T> T g_make_gap(T const& x) {
  // Get tape pointer to tape x is registered in (tape).
  auto tape = dco::tape(x);

  // Compute the value of g with doubles (return type of
  // dco::value(x)). The active variable y is initialized with the
  // passive value.
  T y = g(dco::value(x));
  // Register variable y, since it is the output of g_make_gap;
  // succeeding computation is supposed to be taped, i.e. y is an
  // input to everything which comes later.
  tape->register_variable(y);

  // Callback. This lambda captures by copy ([=]) all accessed
  // data. In this case x and y. This callback is called during tape
  // interpretation, so all accessed data needs to be available
  // during tape interpretation time. The tape is used to fill the
  // gap.
  auto fill_gap = [=]() {
    // This is the current position of the tape.
    auto p = tape->get_position();
    // Record tape for g, since it's now executed with active types.
    // The captured input x is used directly.
    T yl = g(x);
    // The local output yl now lives in the tape (has a tape index).
    // It gets seeded with the adjoint of the original output
    // y. dco::derivative(y) holds the adjoint of the computation in
    // f which succeeded the call of g_make_gap.
    dco::derivative(yl) = dco::derivative(y);
    // Propagate adjoints into the inputs x and reset the tape we've
    // recorded in this callback.
    tape->interpret_adjoint_and_reset_to(p);
  };
  // We move the lambda into the tape as callback. No copy required,
  // since fill_gap is not required outside the tape.
  tape->insert_callback(std::move(fill_gap));

  return y;
}
```

The following output is generated:

```
y = 0.885293
x_{(1)} = -0.351344
y^{(2)} = -0.351344
x_{(1)}^{(2)} = -0.132258
```

### 4.21.3 Concepts introduced in this Section

None.

## 4.22 First-order Adjoint Mode: Equidistant Checkpointing of Evolutions

Example program can be found here: `examples/ga1s_cp_loop_equidist`

### 4.22.1 Purpose

This section illustrates the equidistant checkpointing of evolutions.

### 4.22.2 Example

```
// This example demostrates how to checkpoint a loop equidistantly.
// This algorithm can be used e.g., for evolutions. For mutually
// independent loop iterations, see the example "ga1s_ensemble".

#include <iostream>
#include <cmath>

#include "dco.hpp"

// Function that mimics an evolution.
template <typename T> void g(int n, T& x) {
  for (int i = 0; i < n; i++)
    x = sin(x);
}

// Creates a gap for a chunk of size n.
void g_make_gap(int n, dco::ga1s<double>::type& x) {
  using mode_t = dco::ga1s<double>;
  using type = mode_t::type;
  // Count the chunks/gaps used for printing tape sizes.
  static int gap_counter_helper = 1;
  int gap_counter = gap_counter_helper++;

  // Since we are overwritting x, x_in stores the reference
  // (tape index) to access the correct derivative.
  type x_in(x);
  // Compute x passively.
  g(n, dco::value(x));

  // Create a new tape entry for the output of function g.
  dco::tape(x_in)->register_variable(x);

  // For a detailed description of checkpointing see example ga1s_joint.
  auto fill_gap = [=]() {
    type x_tmp = x_in;
    auto p = dco::tape(x_in)->get_position();
    g(n, x_tmp);
    std::cerr << "size of tape after gap" << gap_counter << " = "
              << dco::size_of(dco::tape(x_in)) << "B" << std::endl;
    dco::derivative(x_tmp) = dco::derivative(x);
    dco::tape(x_in)->interpret_adjoint_and_reset_to(p);
  };

  dco::tape(x_in)->insert_callback(std::move(fill_gap));
}

// This function splits the loop in g in chunks of size m.
template <typename T> void equidistant_checkpointing(int n, int m, T& x) {
  for (int i = 0; i < n; i += m)
    g_make_gap(std::min(m, n - i), x);
}

int main() {
  int n = 10;
  int m = 2;
  // Using first-order adjoint mode.
  using mode_t = dco::ga1s<double>;
  using type = mode_t::type;
  std::cout.precision(5);
  // Input:
  type x = 2.1;

  dco::smart_tape_ptr_t<mode_t> tape;
  tape->register_variable(x);

  // For dco/c++ inputs must be read only. Since we are overwritting x, x_in
  // stores the reference (tape index) used to access the correct derivative.
  type x_in = x;
  auto pos = tape->get_position();

  // Original function we want to checkpoint below.
  //    g(n, x);
  // Record tape, checkpointing the loop in chunks of size m.
  equidistant_checkpointing(n, m, x);

  std::cerr << "base = " << dco::size_of(tape) << "B" << std::endl;
  // Seed the output adjoint.
  dco::derivative(x) = 1.0;
  tape->interpret_adjoint_and_reset_to(pos);

  // Print output value.
  std::cout << "x = " << dco::value(x) << std::endl;
  // Print derivative.
  std::cout << "dx = " << dco::derivative(x_in) << std::endl;
}
```

The following output is generated for n=10 and m=2:

```
ts0=0.00025177001953125MB
ts5=0.0003509521484375MB
ts4=0.00030517578125MB
ts3=0.0002593994140625MB
ts2=0.000213623046875MB
ts1=0.0001678466796875MB

x=0.466043407062983
x_{(1)}=-0.0692765151830291
```

### 4.22.3 Concepts introduced in this Section

None.

## 4.23 Second-order Adjoint Mode: Equidistant Checkpointing of Evolutions

Example program can be found here: `examples/gt2s_ga1s_cp_loop_equidist`

### 4.23.1 Purpose

This section illustrates the equidistant checkpointing of evolutions in second-order adjoint mode.

### 4.23.2 Example

```cpp
#include <iostream>
#include <cmath>
#include "dco.hpp"

using namespace std;

using namespace dco;

typedef gt1s<double> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

#include "f.hpp"

void driver(const int n, const int m, double& x, double& xt2, double& xa1,
            double& xt2a1) {
  DCO_TYPE t2s_a1s_x;
  DCO_BASE_TYPE v;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  v = value(t2s_a1s_x);
  value(v) = x;
  derivative(v) = xt2;
  value(t2s_a1s_x) = v;
  tape->register_variable(t2s_a1s_x);
  DCO_TYPE x_indep = t2s_a1s_x;
  auto p = tape->get_position();
  f(n, m, t2s_a1s_x);
  cerr << "base=" << dco::size_of(tape) << "B" << endl;
  v = derivative(t2s_a1s_x);
  value(v) = xa1;
  derivative(v) = xt2a1;
  derivative(t2s_a1s_x) = v;
  tape->interpret_adjoint_and_reset_to(p);
  v = value(t2s_a1s_x);
  x = value(v);
  xt2 = derivative(v);
  v = derivative(x_indep);
  xa1 = value(v);
  xt2a1 = derivative(v);
}
```

```
#include <cassert>
#include <cstdlib>
#include <iostream>
using namespace std;

#include "driver.hpp"

int main() {
  int n = 10;
  int m = 2;
  cout.precision(5);
  double x = 2.1, xa1 = 1.0, xt2 = 1.0, xt2a1 = 0.0;
  driver(n, m, x, xt2, xa1, xt2a1);
  cout << "x=" << x << endl;
  cout << "x^{(2)}=" << xt2 << endl;
  cout << "x_{(1)}=" << xa1 << endl;
  cout << "x_{(1)}^{(2)}=" << xt2a1 << endl;
}
```

The following output is generated:

```
ts0=0.00041961669921875MB
ts5=0.000579833984375MB
ts4=0.0005035400390625MB
ts3=0.00042724609375MB
ts2=0.0003509521484375MB
ts1=0.000274658203125MB

x=0.466043407062983
x^{(2)}=-0.0692765151830291
x_{(1)}=-0.0692765151830291
x_{(1)}^{(2)}=-0.22663755263662
```

### 4.23.3 Concepts introduced in this Section

None.

## 4.24 First-order Adjoint Mode: Recursive Checkpointing of Evolutions

Example program can be found here: `examples/ga1s_cp_loop_rec`

### 4.24.1 Purpose

This section illustrates the recursive (multi-level) checkpointing of evolutions in first-order adjoint mode.

### 4.24.2 Example

```
#include "g_gap.hpp"

template <typename ATYPE>
void f(int from, int to,
       int stride, // max number of consecutive tapings
       ATYPE& x) {
  // g(from,to,stride,x); // for split reversal
  g_make_gap(from, to, stride, x);
}
```

```
#include <stack>
#include "g.hpp"
using namespace std;

enum RUN_MODE { CHECKPOINT_ARGUMENTS_AND_RUN_PASSIVELY, GENERATE_TAPE };

stack<pair<int, DCO_MODE::value_t>> state;

void g_make_gap(int from, int to, int stride, DCO_MODE::type& xg,
                RUN_MODE m = GENERATE_TAPE) {

  if (m == CHECKPOINT_ARGUMENTS_AND_RUN_PASSIVELY) {
    cout << "STORE CHECKPOINT FOR SECTION " << from << " ... " << to - 1
         << endl;

    // write argument checkpoint (FIFO)
    if (state.empty() || from != state.top().first) {
      cout << "PUSHING (" << from << ", " << dco::value(xg) << ")" << endl;
      state.push(make_pair(from, dco::value(xg)));
    }

    // call passive version of f
    cout << "RUN SECTION " << from << " ... " << to - 1 << " PASSIVELY" << endl;
    g(from, to, stride, dco::value(xg));

    // register output x with tape and store its
    // position for retrieval of incoming adjoint required
    // during interpretation
    DCO_MODE::type xg_in(xg);
    dco::tape(xg)->register_variable(xg);

    auto fill_gap = [=]() {
      cout << "top=" << state.top().second << endl;
      cout << "RESTORE CHECKPOINT FOR SECTION " << from << " ... " << to - 1
           << endl;
      DCO_TYPE x = state.top().second;
      dco::tape(xg)->register_variable(x);
      DCO_TYPE x_in = x;
      auto p = dco::tape(xg)->get_position();
      g_make_gap(from, to, stride, x, GENERATE_TAPE);
      dco::derivative(x) = dco::derivative(xg);
      cout << "INTERPRET SECTION " << from << " ... " << to - 1 << endl;
      dco::tape(xg)->interpret_adjoint_and_reset_to(p);
      dco::derivative(xg_in) += dco::derivative(x_in);
      if (to - from <= stride) {
        cout << "poping " << state.top().first << ", " << state.top().second
             << endl;
        state.pop();
      }
    };

    dco::tape(xg)->insert_callback(std::move(fill_gap));

  } else if (m == GENERATE_TAPE) {
    cout << "GENERATE TAPE FOR SECTION " << from << " ... " << to - 1 << endl;
    stringstream s;
    s << from << "GENERATE_TAPE_" << to - 1 << ".dot";
    dco::tape(xg)->write_to_dot(s.str());
    // in taping mode, the interval is subdivided further if
    // its length exceeds the desired stride ...
    if (to - from > stride) {
      g_make_gap(from, from + (to - from) / 2, stride, xg,
                 CHECKPOINT_ARGUMENTS_AND_RUN_PASSIVELY);
      g_make_gap(from + (to - from) / 2, to, stride, xg,
                 CHECKPOINT_ARGUMENTS_AND_RUN_PASSIVELY);
    } else
      // ... or the given number of iterations is performed
      // actively and the corresponding tape is written
      for (int i = from; i < to; i++)
        xg = sin(xg);
  }
  if (dco::size_of(dco::tape(xg)) > max_tape_size)
    max_tape_size = dco::size_of(dco::tape(xg));
}
```

```
#include <vector>
using namespace std;

template <typename AD_TYPE>
void g(int from, int to,
       int stride, // max number of consecutive tapings
       AD_TYPE& x) {
  if (to - from > stride) {
    g(from, from + (to - from) / 2, stride, x);
    g(from + (to - from) / 2, to, stride, x);
  } else
    for (int i = from; i < to; i++)
      x = sin(x);
}
```

```
#include <iostream>
#include <cstdlib>
#include <cassert>
#include "dco.hpp"

using namespace std;
using namespace dco;

typedef ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

#define STATISTICS
#define VERBOSE

#ifdef STATISTICS
dco::mem_long_t max_tape_size;
unsigned int max_checkpoint_size;
#endif
#include "f.hpp"

void driver(const int n, const int stride, double& xv, double& xa1) {
#ifdef STATISTICS
  max_tape_size = 0;
  max_checkpoint_size = 0;
#endif
  DCO_TYPE x = xv;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  tape->register_variable(x);
  DCO_TYPE x_in = x;

  f(0, n, stride, x);

  tape->register_output_variable(x);
  derivative(x) = 1;

#ifdef VERBOSE
  cout << "INTERPRET SECTION: 0 ... " << n - 1 << endl;
#endif
  tape->write_to_dot("driver.dot");
  tape->interpret_adjoint();

  xv = value(x);
  xa1 = derivative(x_in);
#ifdef STATISTICS
  cerr << "maximum tape size=" << max_tape_size * 1024 * 1024 << "Byte" << endl;
  cerr << "maximum checkpoint size=" << max_checkpoint_size * 8 << "Byte"
```

```
      << endl;
#endif
}
```

```
#include <cassert>
#include <cstdlib>
#include <iostream>
using namespace std;

#include "driver.hpp"

int main() {
  cout.precision(5);
  int n = 10;
  int stride = 2;
  double x = 2.1, xa1 = 1.0;
  driver(n, stride, x, xa1);
  cout << "x=" << x << endl;
  cout << "x_{(1)}=" << xa1 << endl;
}
```

The following output is generated for `n=10` and `m=2`:

```
GENERATE TAPE FOR SECTION 0 ... 9
STORE CHECKPOINT FOR SECTION 0 ... 4
PUSHING (0, 2.1)
RUN SECTION 0 ... 4 PASSIVELY
STORE CHECKPOINT FOR SECTION 5 ... 9
PUSHING (5, 0.593714565454065)
RUN SECTION 5 ... 9 PASSIVELY
Evaluation trial version of ADL6A31IC
INTERPRET SECTION: 0 ... 9
top=0.593714565454065
RESTORE CHECKPOINT FOR SECTION 5 ... 9
GENERATE TAPE FOR SECTION 5 ... 9
STORE CHECKPOINT FOR SECTION 5 ... 6
RUN SECTION 5 ... 6 PASSIVELY
STORE CHECKPOINT FOR SECTION 7 ... 9
PUSHING (7, 0.530714839456817)
RUN SECTION 7 ... 9 PASSIVELY
INTERPRET SECTION 5 ... 9
top=0.530714839456817
RESTORE CHECKPOINT FOR SECTION 7 ... 9
GENERATE TAPE FOR SECTION 7 ... 9
STORE CHECKPOINT FOR SECTION 7 ... 7
RUN SECTION 7 ... 7 PASSIVELY
STORE CHECKPOINT FOR SECTION 8 ... 9
PUSHING (8, 0.506149980527331)
RUN SECTION 8 ... 9 PASSIVELY
INTERPRET SECTION 7 ... 9
top=0.506149980527331
RESTORE CHECKPOINT FOR SECTION 8 ... 9
GENERATE TAPE FOR SECTION 8 ... 9
INTERPRET SECTION 8 ... 9
poping 8, 0.506149980527331
```

```
top=0.530714839456817
RESTORE CHECKPOINT FOR SECTION 7 ... 7
GENERATE TAPE FOR SECTION 7 ... 7
INTERPRET SECTION 7 ... 7
poping 7, 0.530714839456817
top=0.593714565454065
RESTORE CHECKPOINT FOR SECTION 5 ... 6
GENERATE TAPE FOR SECTION 5 ... 6
INTERPRET SECTION 5 ... 6
poping 5, 0.593714565454065
top=2.1
RESTORE CHECKPOINT FOR SECTION 0 ... 4
GENERATE TAPE FOR SECTION 0 ... 4
STORE CHECKPOINT FOR SECTION 0 ... 1
RUN SECTION 0 ... 1 PASSIVELY
STORE CHECKPOINT FOR SECTION 2 ... 4
PUSHING (2, 0.75993256745268)
RUN SECTION 2 ... 4 PASSIVELY
INTERPRET SECTION 0 ... 4
top=0.75993256745268
RESTORE CHECKPOINT FOR SECTION 2 ... 4
GENERATE TAPE FOR SECTION 2 ... 4
STORE CHECKPOINT FOR SECTION 2 ... 2
RUN SECTION 2 ... 2 PASSIVELY
STORE CHECKPOINT FOR SECTION 3 ... 4
PUSHING (3, 0.688872566005681)
RUN SECTION 3 ... 4 PASSIVELY
INTERPRET SECTION 2 ... 4
top=0.688872566005681
RESTORE CHECKPOINT FOR SECTION 3 ... 4
GENERATE TAPE FOR SECTION 3 ... 4
INTERPRET SECTION 3 ... 4
poping 3, 0.688872566005681
top=0.75993256745268
RESTORE CHECKPOINT FOR SECTION 2 ... 2
GENERATE TAPE FOR SECTION 2 ... 2
INTERPRET SECTION 2 ... 2
poping 2, 0.75993256745268
top=2.1
RESTORE CHECKPOINT FOR SECTION 0 ... 1
GENERATE TAPE FOR SECTION 0 ... 1
INTERPRET SECTION 0 ... 1
poping 0, 2.1
maximum tape size=528Byte
maximum checkpoint size=32Byte
x=0.466043407062983
x_{(1)}=-0.0692765151830291
```

### 4.24.3 Concepts introduced in this Section

**`my_external_adjoint_object_t`**

Specialization of `DCO_MODE::external_adjoint_object_t`.

## 4.25 Second-order Adjoint Mode: Recursive Checkpointing of Evolutions

Example program can be found here: `examples/gt2s_ga1s_cp_loop_rec`

### 4.25.1 Purpose

This section illustrates the recursive (multi-level) checkpointing of evolutions in second-order adjoint mode.

### 4.25.2 Example

```
#include <iostream>
#include <cmath>
#include "dco.hpp"

using namespace std;

using namespace dco;

typedef gt1s<double> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_ITERATOR_TYPE;

#define VERBOSE

#include "f.hpp"

void driver(const int n, const int stride, double& x, double& xt2, double& xa1,
            double& xt2a1) {
  DCO_TYPE t2s_a1s_x;
  DCO_BASE_TYPE v;
  v = value(t2s_a1s_x);
  value(v) = x;
  derivative(v) = xt2;
  value(t2s_a1s_x) = v;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  tape->register_variable(t2s_a1s_x);
  DCO_TYPE x_indep = t2s_a1s_x;
  DCO_TAPE_ITERATOR_TYPE to_be_reset_to = tape->get_position();
  f<DCO_MODE>(0, n, stride, t2s_a1s_x);
  v = derivative(t2s_a1s_x);
  value(v) = xa1;
  derivative(v) = xt2a1;
  derivative(t2s_a1s_x) = v;
  tape->interpret_adjoint_and_reset_to(to_be_reset_to);
  v = value(t2s_a1s_x);
  x = value(v);
  xt2 = derivative(v);
  v = derivative(x_indep);
  xa1 = value(v);
  xt2a1 = derivative(v);
}
```

```
#include <cassert>
#include <cstdlib>
#include <iostream>
using namespace std;

#include "driver.hpp"

int main() {
  int n = 10;
  int stride = 2;
  cout.precision(5);
  double x = 2.1, xa1 = 1.0, xt2 = 1.0, xt2a1 = 0.0;
  driver(n, stride, x, xt2, xa1, xt2a1);
  cout << "x=" << x << endl;
  cout << "x^{(2)}=" << xt2 << endl;
  cout << "x_{(1)}=" << xa1 << endl;
  cout << "x_{(1)}^{(2)}=" << xt2a1 << endl;
}
```

The following output is generated:

```
GENERATE TAPE FOR SECTION 0 ... 9
STORE CHECKPOINT FOR SECTION 0 ... 4
PUSHING (0, 2.1)
RUN SECTION 0 ... 4 PASSIVELY
STORE CHECKPOINT FOR SECTION 5 ... 9
PUSHING (5, 0.593714565454065)
RUN SECTION 5 ... 9 PASSIVELY
top=0.593714565454065
RESTORE CHECKPOINT FOR SECTION 5 ... 9
GENERATE TAPE FOR SECTION 5 ... 9
STORE CHECKPOINT FOR SECTION 5 ... 6
RUN SECTION 5 ... 6 PASSIVELY
STORE CHECKPOINT FOR SECTION 7 ... 9
PUSHING (7, 0.530714839456817)
RUN SECTION 7 ... 9 PASSIVELY
INTERPRET SECTION 5 ... 9
top=0.530714839456817
RESTORE CHECKPOINT FOR SECTION 7 ... 9
GENERATE TAPE FOR SECTION 7 ... 9
STORE CHECKPOINT FOR SECTION 7 ... 7
RUN SECTION 7 ... 7 PASSIVELY
STORE CHECKPOINT FOR SECTION 8 ... 9
PUSHING (8, 0.506149980527331)
RUN SECTION 8 ... 9 PASSIVELY
INTERPRET SECTION 7 ... 9
top=0.506149980527331
RESTORE CHECKPOINT FOR SECTION 8 ... 9
GENERATE TAPE FOR SECTION 8 ... 9
INTERPRET SECTION 8 ... 9
```

```
poping 8, 0.506149980527331
top=0.530714839456817
RESTORE CHECKPOINT FOR SECTION 7 ... 7
GENERATE TAPE FOR SECTION 7 ... 7
INTERPRET SECTION 7 ... 7
poping 7, 0.530714839456817
top=0.593714565454065
RESTORE CHECKPOINT FOR SECTION 5 ... 6
GENERATE TAPE FOR SECTION 5 ... 6
INTERPRET SECTION 5 ... 6
poping 5, 0.593714565454065
top=2.1
RESTORE CHECKPOINT FOR SECTION 0 ... 4
GENERATE TAPE FOR SECTION 0 ... 4
STORE CHECKPOINT FOR SECTION 0 ... 1
RUN SECTION 0 ... 1 PASSIVELY
STORE CHECKPOINT FOR SECTION 2 ... 4
PUSHING (2, 0.75993256745268)
RUN SECTION 2 ... 4 PASSIVELY
INTERPRET SECTION 0 ... 4
top=0.75993256745268
RESTORE CHECKPOINT FOR SECTION 2 ... 4
GENERATE TAPE FOR SECTION 2 ... 4
STORE CHECKPOINT FOR SECTION 2 ... 2
RUN SECTION 2 ... 2 PASSIVELY
STORE CHECKPOINT FOR SECTION 3 ... 4
PUSHING (3, 0.688872566005681)
RUN SECTION 3 ... 4 PASSIVELY
INTERPRET SECTION 2 ... 4
top=0.688872566005681
RESTORE CHECKPOINT FOR SECTION 3 ... 4
GENERATE TAPE FOR SECTION 3 ... 4
INTERPRET SECTION 3 ... 4
poping 3, 0.688872566005681
top=0.75993256745268
RESTORE CHECKPOINT FOR SECTION 2 ... 2
GENERATE TAPE FOR SECTION 2 ... 2
INTERPRET SECTION 2 ... 2
poping 2, 0.75993256745268
top=2.1
RESTORE CHECKPOINT FOR SECTION 0 ... 1
GENERATE TAPE FOR SECTION 0 ... 1
INTERPRET SECTION 0 ... 1
poping 0, 2.1
x=0.466043407062983
x^{(2)}=-0.0692765151830291
x_{(1)}=-0.0692765151830291
x_{(1)}^{(2)}=-0.22663755263662
```

### 4.25.3 Concepts introduced in this Section

None.

## 4.26 First-order Adjoint Mode: Checkpointing Ensembles

Example program can be found here: `examples/ga1s_ensemble`

### 4.26.1 Purpose

This section illustrates checkpointing of ensembles in first-order adjoint mode. A global tape is used. The given solution allows for second and higher derivatives to be computed with minimal implementation effort; see Ch. 4.27.

### 4.26.2 Example

```
#include "g_gap.hpp"

inline unsigned int prng() {
  static unsigned int seed = 5323;
  seed = 8253729 * seed + 2396403;
  return seed % 32768;
}

template <typename DCO_TYPE>
void f(int n, int m, const DCO_TYPE& x, DCO_TYPE& y) {
  double r;
  DCO_TYPE sum;
  for (int i = 0; i < n; i++) {
    r = prng();
    g_make_gap(m, x, r, y);
    sum += y;
  }
  y = sum / n;
}
```

```
#include <iostream>
using namespace std;
#include "g.hpp"

void g_make_gap(int m, const DCO_TYPE& x, const double& r, DCO_TYPE& yg) {
  static int c = 1;
  int ts = c++;

  g(m, dco::value(x), r, dco::value(yg));
  dco::tape(x)->register_variable(yg);

  auto fill_gap = [=]() {
    auto p = dco::tape(x)->get_position();
    DCO_TYPE y;
    g(m, x, r, y);
    cerr << "ts" << ts << "=" << dco::size_of(dco::tape(x)) << "B" << endl;
    dco::derivative(y) = dco::derivative(yg);
    dco::tape(x)->interpret_adjoint_and_reset_to(p);
  };

  dco::tape(x)->insert_callback(std::move(fill_gap));
}
```

```
template <typename ATYPE>
void g(int m, const ATYPE& x, const double& r, ATYPE& y) {
  y = 0;
  for (int i = 0; i < m; i++)
    y += sin(x + r);
}
```

```
#include "dco.hpp"
using namespace dco;

typedef dco::ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

#include "f.hpp"

void driver(const int n, const int m, const double& xv, double& xa, double& yv,
            const double& ya) {
  DCO_TYPE x = xv, y;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  tape->register_variable(x);
  DCO_TAPE_POSITION_TYPE p = tape->get_position();
  f(n, m, x, y);
  cerr << "base=" << dco::size_of(tape) << "B" << endl;
  yv = value(y);
  tape->register_output_variable(y);
  derivative(y) = ya;
  derivative(x) = xa;
  tape->interpret_adjoint_and_reset_to(p);
  xa = derivative(x);
}
```

```
#include <iostream>
using namespace std;

#include "driver.hpp"

int main() {
  cout.precision(5);
  int n = 10;
  int m = 10;
  double xv = 2.1, xa = 0.0, yv, ya = 1.0;
  driver(n, m, xv, xa, yv, ya);
  cout << "y=" << yv << endl;
  cout << "x_{(1)}=" << xa << endl;
}
```

The following output is generated for n=10 and m=2:

```
ts0=0.00103759765625MB
ts10=0.00146484375MB
```

```
ts9=0.00136566162109375MB
ts8=0.0012664794921875MB
ts7=0.00116729736328125MB
ts6=0.001068115234375MB
ts5=0.00096893310546875MB
ts4=0.0008697509765625MB
ts3=0.00077056884765625MB
ts2=0.00067138671875MB
ts1=0.00058746337890625MB

y=-0.0653545466085305
x_{(1)}=-0.10569862778361
```

### 4.26.3 Concepts introduced in this Section

None.

## 4.27 Second-order Adjoint Mode: Checkpointing Ensembles

Example program can be found here: `examples/gt2s_ga1s_ensemble`

### 4.27.1 Purpose

This section illustrates checkpointing of ensembles in second-order adjoint mode.

### 4.27.2 Example

```cpp
#include "dco.hpp"
using namespace dco;

typedef gt1s<double> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

#include "f.hpp"

void driver(const int n, const int m, const double& xv, const double& xt2,
            double& xa1, double& xa1t2, double& yv, double& yt2,
            const double& ya1, const double& ya1t2) {
  DCO_TYPE x = xv, y;
  derivative(value(x)) = xt2;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  tape->register_variable(x);
  DCO_TAPE_POSITION_TYPE p = tape->get_position();
  f(n, m, x, y);
  cerr << "base=" << dco::size_of(tape) << "B" << endl;
  yv = passive_value(y);
  yt2 = derivative(value(y));
  tape->register_output_variable(y);
  value(derivative(y)) = ya1;
  derivative(derivative(y)) = ya1t2;
  value(derivative(x)) = xa1;
  derivative(derivative(x)) = xa1t2;
  tape->interpret_adjoint_and_reset_to(p);
  xa1 = value(derivative(x));
  xa1t2 = derivative(derivative(x));
}
```

```
#include <cassert>
#include <cstdlib>
#include <iostream>
using namespace std;

#include "driver.hpp"

int main() {
  cout.precision(5);
  int n = 10;
  int m = 10;
  double x = 2.1, xt2 = 1.0, xa1 = 0.0, xa1t2 = 0.0, y, yt2, ya1 = 1.0,
         ya1t2 = 0.0;
  driver(n, m, x, xt2, xa1, xa1t2, y, yt2, ya1, ya1t2);
  cout << "y=" << y << endl;
  cout << "y^{(2)}=" << yt2 << endl;
  cout << "x_{(1)}=" << xa1 << endl;
  cout << "x_{(1)}^{(2)}=" << xa1t2 << endl;
}
```

The following output is generated:

```
ts0=0.001678466796875MB
ts0=0.001678466796875MB
ts10=0.002349853515625MB
ts9=0.0021896362304687 5MB
ts8=0.002029418945312 5MB
ts7=0.0018692016601562 5MB
ts6=0.001708984375MB
ts5=0.0015487670898437 5MB
ts4=0.001388549804687 5MB
ts3=0.0012283325195312 5MB
ts2=0.001068115234375MB
ts1=0.000930786132812 5MB

y=-0.326772733042652
y^{(2)}=-0.528493138918051
x_{(1)}=-0.528493138918051
x_{(1)}^{(2)}=0.326772733042653
```

### 4.27.3 Concepts introduced in this Section

None.

## 4.28 First-order Adjoint Mode: Embedding Adjoint Source Code

Example program can be found here: `examples/ga1s_external_manual`

### 4.28.1 Purpose

This section illustrates the embedding of adjoint source code into a `dco/c++` adjoint.

### 4.28.2 Example

```cpp
#include <vector>
#include "g_gap.hpp"

template <typename DCO_TYPE> void f(std::vector<DCO_TYPE>& x, DCO_TYPE& y) {
  for (unsigned i = 0; i < x.size(); i++)
    x[i] *= x[i];
  // g(x,y);
  g_make_gap(x, y);
  y *= y;
}
```

```cpp
#include <vector>

#include "g.hpp"
#include "g_a1s.hpp"

void g_make_gap(std::vector<DCO_TYPE>& x, DCO_TYPE& y) {
  typedef DCO_MODE::value_t DCO_VALUE_TYPE;

  //** Extract passive value and run primal function.
  vector<DCO_VALUE_TYPE> xv(x.size());
  xv = dco::value(x);
  g(xv, dco::value(y));

  //** Register output variable to continue tape recording after
  //** this function returns.
  dco::tape(x)->register_variable(y);

  //** Filling the gap with the tape callback. Use automatically
  //** captured (checkpointed) x and y to access adjoints.
  auto fill_gap = [=]() {
    vector<DCO_VALUE_TYPE> xa(x.size());
    g_a1s(xa, dco::derivative(y));
    dco::derivative(x) += xa;
  };

  dco::tape(x)->insert_callback(std::move(fill_gap));
}
```

```
#include <vector>

template <typename DCO_TYPE> void g(std::vector<DCO_TYPE> x, DCO_TYPE& y) {
  y = 0;
  for (unsigned i = 0; i < x.size(); i++)
    y += x[i];
}
```

```
#include <vector>

template <typename DCO_TYPE>
void g_a1s(std::vector<DCO_TYPE>& xa1, const DCO_TYPE& ya1) {
  typename std::vector<DCO_TYPE>::iterator i;
  for (i = xa1.begin(); i != xa1.end(); i++)
    *i += ya1;
}
```

```
#include <vector>
#include "dco.hpp"

using namespace std;
using namespace dco;
typedef ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

#include "f.hpp"

void driver(const vector<double>& xv, vector<double>& xa1, double& yv,
            double& ya1) {
  size_t n = xv.size();
  vector<DCO_TYPE> x(n);
  vector<DCO_TYPE> x_in(n);
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  for (size_t i = 0; i < n; i++) {
    tape->register_variable(x[i]);
    value(x[i]) = xv[i];
    derivative(x[i]) = xa1[i];
    x_in[i] = x[i];
  }
  DCO_TYPE y;
  auto p = tape->get_position();
  f(x, y);
  cerr << "ts=" << dco::size_of(tape) << "B" << endl;
  yv = value(y);
  derivative(y) = ya1;
  tape->interpret_adjoint_and_reset_to(p);
  for (size_t i = 0; i < n; i++)
    xa1[i] = derivative(x_in[i]);
}
```

```
#include <iostream>
```

```
#include <cstdlib>
#include <vector>
#include <cmath>
using namespace std;

#include "driver.hpp"

int main() {
  cout.precision(5);
  int n = 5;
  vector<double> x(n), xa1(n);
  double y = 0, ya1;
  for (int i = 0; i < n; i++) {
    x[i] = cos(double(i));
    xa1[i] = 0;
  }
  ya1 = 1;
  driver(x, xa1, y, ya1);
  cout << "y=" << y << endl;
  for (int i = 0; i < n; i++)
    cout << "x_{(1)}[" << i << "]=" << xa1[i] << endl;
}
```

The following output is generated:

```
ts=0.00048065185546875MB

y=8.25091096598783
x_{(1)}[0]=11.489759590862
x_{(1)}[1]=6.20794360081331
x_{(1)}[2]=-4.78142710642441
x_{(1)}[3]=-11.3747757826964
x_{(1)}[4]=-7.51020806182345
```

Calling `g` instead of `g_make_gap` in `f` yields

```
ts=0.0006866455078125MB

y=8.25091096598783
x_{(1)}[0]=11.489759590862
...
```

### 4.28.3 Concepts introduced in this Section

None.

## 4.29 Second-order Adjoint Mode: Embedding First-order Adjoint Code

Example program can be found here: `examples/gt2s_ga1s_external_manual`

### 4.29.1 Purpose

This section illustrates the embedding of first-order adjoint code into a dco/c++ second-order adjoint mode computation.

### 4.29.2 Example

```
#include <vector>
#include "dco.hpp"

using namespace std;
using namespace dco;

typedef gt1s<double> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

#include "f.hpp"

void driver(const vector<double>& xv, const vector<double>& xt2,
            vector<double>& xa1, vector<double>& xa1t2, double& yv, double& yt2,
            double& ya1, double& ya1t2) {
  dco::smart_tape_ptr_t<ga1s<DCO_BASE_TYPE>> tape;
  const size_t n = xv.size();
  vector<DCO_TYPE> x(n), x_in(n);
  DCO_TYPE y;
  for (size_t i = 0; i < n; i++) {
    tape->register_variable(x[i]);
    value(value(x[i])) = xv[i];
    derivative(value(x[i])) = xt2[i];
    x_in[i] = x[i];
  }
  auto p = tape->get_position();
  f(x, y);
  cerr << "ts=" << dco::size_of(tape) << "B" << endl;
  for (size_t i = 0; i < n; i++) {
    value(derivative(x_in[i])) = xa1[i];
    derivative(derivative(x_in[i])) = xa1t2[i];
  }
  yv = value(value(y));
  yt2 = derivative(value(y));
  tape->register_output_variable(y);
  value(derivative(y)) = ya1;
  derivative(derivative(y)) = ya1t2;
  tape->interpret_adjoint_and_reset_to(p);
  for (size_t i = 0; i < n; i++) {
    xa1[i] = value(derivative(x_in[i]));
    xa1t2[i] = derivative(derivative(x_in[i]));
  }
}
```

```
#include <iostream>
```

```
#include <vector>
#include <cmath>
using namespace std;

#include "driver.hpp"

int main() {
  cout.precision(5);
  int n = 5;
  vector<double> x(n), xa1(n), xt2(n), xa1t2(n);
  double y, ya1, yt2, ya1t2;
  for (int i = 0; i < n; i++) {
    x[i] = cos(double(i));
    xt2[i] = 1;
    xa1[i] = 0;
    xa1t2[i] = 0;
  }
  y = 0, yt2 = 0, ya1 = 1;
  ya1t2 = 0;
  driver(x, xt2, xa1, xa1t2, y, yt2, ya1, ya1t2);
  cout << "y=" << y << endl;
  for (int i = 0; i < n; i++)
    cout << "x_{(1)}[" << i << "]=" << xa1[i] << endl;
  cout << "y^{(2)}=" << yt2 << endl;
  for (int i = 0; i < n; i++)
    cout << "x_{(1)}^{(2)}[" << i << "]=" << xa1t2[i] << endl;
  cout << "y_{(1)}=" << ya1 << endl;
  cout << "y_{(1)}^{(2)}=" << ya1t2 << endl;
}
```

The following output is generated:

```
ts=0.00077056884765625MB

y=8.25091096598783
x_{(1)}[0]=11.489759590862
x_{(1)}[1]=6.20794360081331
x_{(1)}[2]=-4.78142710642441
x_{(1)}[3]=-11.3747757826964
x_{(1)}[4]=-7.51020806182345
y^{(2)}=-5.96870775926893
x_{(1)}^{(2)}[0]=7.33391440571752
x_{(1)}^{(2)}[1]=9.24434685449743
x_{(1)}^{(2)}[2]=13.2192014178395
x_{(1)}^{(2)}[3]=15.6040151411881
x_{(1)}^{(2)}[4]=14.2062012854284
y_{(1)}=0
y_{(1)}^{(2)}=0
```

Calling `g` instead of `g_make_gap` in `f` yields

```
ts=0.00109100341796875MB

y=8.25091096598783
x_{(1)}[0]=11.489759590862
```

```
...
y^{(2)}=-5.96870775926893
x_{(1)}^{(2)}[0]=7.33391440571752
...
```

### 4.29.3 Concepts introduced in this Section

None.

## 4.30 First-order Adjoint Vector Mode: Embedding Adjoint Source Code

Example program can be found here: `examples/ga1v_external_manual`

### 4.30.1 Purpose

This section illustrates the embedding of adjoint source code into a dco/c++ vector adjoint.

### 4.30.2 Example

```
#include <vector>
#include "g_gap.hpp"

template <typename DCO_TYPE> void f(std::vector<DCO_TYPE>& x, DCO_TYPE& y) {
  for (unsigned i = 0; i < x.size(); i++)
    x[i] *= x[i];
  // g(x,y);
  g_make_gap(x, y);
  y *= y;
}
```

```
#include <vector>

#include "g.hpp"
#include "g_a1s.hpp"

void g_make_gap(std::vector<DCO_TYPE>& x, DCO_TYPE& y) {
  typedef DCO_MODE::derivative_t DCO_DERIVATIVE_TYPE;
  typedef DCO_MODE::value_t DCO_VALUE_TYPE;

  //** Extract passive value and run primal function.
  vector<DCO_VALUE_TYPE> xv(x.size());
  xv = dco::value(x);
  g(xv, dco::value(y));

  //** Register output variable to continue tape recording after
  //** this function returns.
  dco::tape(x)->register_variable(y);

  //** Filling the gap with the tape callback. Use automatically
  //** captured (checkpointed) x and y to access adjoints.
  auto fill_gap = [=]() {
    vector<DCO_DERIVATIVE_TYPE> xa(x.size(), 0);
    g_a1s(xa, dco::derivative(y));
    dco::derivative(x) += xa;
  };

  dco::tape(x)->insert_callback(std::move(fill_gap));
}
```

```
#include <vector>

template <typename DCO_TYPE> void g(std::vector<DCO_TYPE> x, DCO_TYPE& y) {
  y = 0;
  for (unsigned i = 0; i < x.size(); i++)
    y += x[i];
}
```

```
#include <vector>
template <typename DCO_TYPE>
void g_a1s(std::vector<DCO_TYPE>& xa1, const DCO_TYPE& ya1) {
  for (auto& i : xa1)
    i += ya1;
}
```

```
#include <vector>
#include <array>
#include "dco.hpp"

using namespace std;
#include "driver.hpp"

using namespace dco;
typedef ga1v<double, N> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

#include "f.hpp"

void driver(const vector<double>& xv, vector<array<double, N>>& xa1, double& yv,
            array<double, N>& ya1) {
  size_t n = xv.size();
  vector<DCO_TYPE> x(n);
  vector<DCO_TYPE> x_in(n);
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  for (size_t i = 0; i < n; i++) {
    tape->register_variable(x[i]);
    value(x[i]) = xv[i];
    derivative(x[i]) = xa1[i];
    x_in[i] = x[i];
  }
  DCO_TYPE y;
  auto p = tape->get_position();
  f(x, y);
  cerr << "ts=" << dco::size_of(tape) << "B" << endl;
  yv = value(y);
  derivative(y) = ya1;

  //** The callback is implemented in vectorized fashion, not with
  //** scalars, see g_gap.hpp:21.
  tape->scalar_callback_mode() = false;

  tape->interpret_adjoint_and_reset_to(p);
  for (size_t i = 0; i < n; i++)
    xa1[i] = derivative(x_in[i]);
}
```

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <array>
#include <cmath>
using namespace std;

#include "driver.hpp"

int main() {
  cout.precision(5);
  int n = 5;
  vector<double> x(n);
  vector<array<double, N>> xa1(n, array<double, N>{});
  double y = 0;
  array<double, N> ya1;
  for (int i = 0; i < n; i++) {
    x[i] = cos(double(i));
  }
  for (int i = 0; i < N; i++) {
    ya1[i] = i + 1;
  }
  driver(x, xa1, y, ya1);
  cout << "y=" << y << endl;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      cout << "x_{(1)}[" << i << "][" << j << "]=" << xa1[i][j] << endl;
}
```

The following output is generated:

```
ts=0.00048065185546875MB

y=8.25091096598783
x_{(1)}[0]=11.489759590862
x_{(1)}[1]=6.20794360081331
x_{(1)}[2]=-4.78142710642441
x_{(1)}[3]=-11.3747757826964
x_{(1)}[4]=-7.51020806182345
```

Calling g instead of `g_make_gap` in f yields

```
ts=0.0006866455078125MB
```

```
y=8.25091096598783
x_{(1)}[0]=11.489759590862
...
```

### 4.30.3 Concepts introduced in this Section

**`DCO_TAPE_TYPE::scalar_callback_mode`**

Returns reference to tape member boolean, which switches scalar callback mode on and off. If switched on, external adjoint callbacks are called multiple times, once for each vector element. It is initialized to **`true`** by default.

## 4.31 First-order Adjoint Mode: Local Preaccumulation Using First-order Tangent Code

Example program can be found here: `examples/ga1s_preaccumulation`

### 4.31.1 Purpose

This section illustrates the use of hand-written tangent code for the preaccumulation of local Jacobians embedded into a `dco/c++` first-order adjoint mode computation.

### 4.31.2 Example

```cpp
#include "g_gap.hpp"

template <typename AD_TYPE> void f(int n, AD_TYPE& x) {
  g(n / 3, x);
  g(n / 3, x);
  // g_make_gap(n/3,x);
  g(n - n / 3 * 2, x);
}
```

```cpp
#include "g.hpp"

template <typename DCO_BASE_TYPE>
void gt(int n, DCO_BASE_TYPE& x, DCO_BASE_TYPE& xt1) {
  for (int i = 0; i < n; i++) {
    xt1 = cos(x) * xt1;
    x = sin(x);
  }
}

void g_make_gap(int n, DCO_TYPE& x) {
  typedef DCO_MODE::value_t DCO_VALUE_TYPE;

  DCO_VALUE_TYPE xt = 1.;
  gt(n, dco::value(x), xt);

  DCO_TYPE x_in(x);
  dco::tape(x_in)->register_variable(x);

  auto fill_gap = [=]() { dco::derivative(x_in) += xt * dco::derivative(x); };

  dco::tape(x_in)->insert_callback(std::move(fill_gap));
}
```

```
#include <cmath>
using namespace std;

template <typename ATYPE> void g(int n, ATYPE& x) {
  for (int i = 0; i < n; i++)
    x = sin(x);
}
```

```
#include <iostream>
#include <cmath>
#include "dco.hpp"

using namespace std;
using namespace dco;
typedef ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

#include "f.hpp"

void driver(const int n, double& xv, double& xa) {
  DCO_TYPE x = xv;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  tape->register_variable(x);
  DCO_TYPE x_in = x;
  auto p = tape->get_position();
  f(n, x);
  cerr << "ts=" << dco::size_of(tape) << "B" << endl;
  derivative(x) = xa;
  tape->interpret_adjoint_and_reset_to(p);
  xv = value(x);
  xa = derivative(x_in);
}
```

```
#include <cassert>
#include <cstdlib>
#include <iostream>
using namespace std;

#include "driver.hpp"

int main() {
  cout.precision(5);
  int n = 10;
  double x = 2.1, xa1 = 1.0;
  driver(n, x, xa1);
  cout << "x=" << x << endl;
  cout << "x_{(1)}=" << xa1 << endl;
}
```

The following output is generated:

```
ts=0.000335693359375MB

x=0.466043407062983
x_{(1)}=-0.0692765151830291
```

Calling `g` instead of `g_make_gap` in `f` yields

```
ts=0.00040435791015625MB

x=0.466043407062983
x_{(1)}=-0.0692765151830291
```

### 4.31.3 Concepts introduced in this Section

None.

## 4.32 Second-order Adjoint Mode: Local Preaccumulation Using First-order Tangent Code

Example program can be found here: `examples/gt2s_ga1s_preaccumulation`

### 4.32.1 Purpose

This section illustrates the use of hand-written first-order tangent code for the preaccumulation of a local Jacobian embedded into a dco/c++ second-order adjoint mode computation.

### 4.32.2 Example

```
#include <iostream>
#include <cmath>
#include "dco.hpp"

using namespace std;

using namespace dco;

typedef gt1s<double> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

#include "f.hpp"

void driver(const int n, double& xv, double& xt2, double& xa1, double& xt2a1) {
  DCO_TYPE x;
  passive_value(x) = xv;
  derivative(value(x)) = xt2;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  tape->register_variable(x);
  DCO_TYPE x_in = x;
  auto p = tape->get_position();
  f(n, x);
  cerr << "ts=" << dco::size_of(tape) << "B" << endl;
  xv = passive_value(x);
  xt2 = derivative(value(x));
  value(derivative(x)) = xa1;
  derivative(derivative(x)) = xt2a1;
  tape->interpret_adjoint_and_reset_to(p);
  xa1 = value(derivative(x_in));
  xt2a1 = derivative(derivative(x_in));
}
```

```
#include <cassert>
#include <cstdlib>
#include <iostream>
using namespace std;

#include "driver.hpp"

int main() {
  int n = 10;
  cout.precision(5);
  double x = 2.1, xa1 = 1.0, xt2 = 1.0, xt2a1 = 0.0;
  driver(n, x, xt2, xa1, xt2a1);
  cout << "x=" << x << endl;
  cout << "x^{(2)}=" << xt2 << endl;
  cout << "x_{(1)}=" << xa1 << endl;
  cout << "x_{(1)}^{(2)}=" << xt2a1 << endl;
}
```

The following output is generated:

```
ts=0.00054168701171875MB

x=0.466043407062983
x^{(2)}=-0.0692765151830291
x_{(1)}=-0.0692765151830291
x_{(1)}^{(2)}=-0.22663755263662
```

Calling `g` instead of `g_make_gap` in `f` yields

```
ts=0.00064849853515625MB

x=0.466043407062983
x^{(2)}=-0.0692765151830291
x_{(1)}=-0.0692765151830291
x_{(1)}^{(2)}=-0.22663755263662
```

### 4.32.3 Concepts introduced in this Section

None.

## 4.33 Adjoint Mode: Edge Insertion for Preaccumulation of Local Gradients

Example program can be found here: `examples/ga1s_preaccumulation_gradient_t`

### 4.33.1 Purpose

This section illustrates the use of hand-written code for the preaccumulation of local gradients embedded into a `dco/c++` first-order adjoint mode computation. This interface works exactly the same for higher order modes.

### 4.33.2 Example

```
//** This example shows how to add a custom gradient directly into the
//** tape. No callback is introduced (see ga1s_external_manual for how
//** to use callbacks instead). This interface works exactly the same
//** for higher order modes.
#include "dco.hpp"

//** Custom multiplication function. Returns x*y.
template <typename T>
//** This implementation is only valid for adjoint dco types.
typename std::enable_if<dco::mode<T>::is_adjoint_type, T>::type
my_multiply(T const& x, T const& y) {

  using value_t = typename dco::mode<T>::value_t;

  //** Declare active variable and assign it to function value,
  //** i.e. the multiplication result. The function evaluation should
  //** only use value_t data; avoid active computation. If you need to
  //** use active data, switch the tape off first.
  T z = dco::value(x) * dco::value(y);

  //** Compute partial derivatives (dz/dx and dz/dy). Again, only
  //** value_t data should be used.
  value_t dzdx = dco::value(y);
  value_t dzdy = dco::value(x);

  //** Create local gradient builder; the constructor activates z
  //** (i.e. assigns new tape index).
  typename dco::mode<T>::local_gradient_t z_gradient_builder(z);

  //** Gradient entries are passed element by element. First argument
  //** is the active input variable, second is the partial derivative
  //** w.r.t. that input. The second argument is of type value_t.
  z_gradient_builder.put(x, dzdx);
  z_gradient_builder.put(y, dzdy);

  return z;
}

int main() {
  //** Run first order adjoint. Usual driver code.
  using mode_t = dco::ga1s<double>;
  using type = mode_t::type;

  dco::smart_tape_ptr_t<mode_t> tape;

  //** Declare, initialize, register inputs.
  type x(1.0), y(2.0);
  tape->register_variable(x);
  tape->register_variable(y);

  //** Record tape.
  type z = my_multiply(x, y);

  //** Register output, seed adjoint, interpret tape.
  tape->register_output_variable(z);
  dco::derivative(z) = 1.0;
  tape->interpret_adjoint();

  std::cout << "dz/dx = " << dco::derivative(x) << std::endl;
  std::cout << "dz/dy = " << dco::derivative(y) << std::endl;
}
```

The following output is generated:

```
dz/dx = 2
dz/dy = 1
```

### 4.33.3 Concepts introduced in this Section

**`DCO_MODE::local_gradient_t`**

Local gradient type associated with `DCO_MODE`.

**`DCO_LOCAL_GRADIENT_TYPE::put`**

Adds local gradient entry passed as the second argument with respect to variable passed as the first argument.

## 4.34 First-order Adjoint Mode: User-Defined Adjoints

Example program can be found here: `examples/ga1s_user_defined_adjoint`

### 4.34.1 Purpose

This section illustrates the definitions of custom adjoints by the user in `dco/c++` first-order adjoint mode.

### 4.34.2 Example

```cpp
#include "g_gap.hpp"

template <typename DCO_TYPE> void f(DCO_TYPE p, DCO_TYPE& x) {
  p = exp(p);
  // g(p,x);
  g_make_gap(p, x);
  x = x * p;
}
```

```cpp
#include "g.hpp"

template <typename DCO_TYPE>
void a1s_g(DCO_TYPE& pa, const DCO_TYPE& xv, DCO_TYPE& xa) {
  pa = xa / (2 * xv);
}

void g_make_gap(const DCO_TYPE& p, DCO_TYPE& x) {

  g(dco::value(p), dco::value(x));
  dco::tape(p)->register_variable(x);

  auto fill_gap = [=]() {
    DCO_MODE::value_t pa = 0;
    a1s_g(pa, dco::value(x), dco::derivative(x));
    dco::derivative(p) += pa;
  };

  dco::tape(p)->insert_callback(std::move(fill_gap));
}
```

```cpp
template <class DCO_TYPE> void g(const DCO_TYPE& p, DCO_TYPE& x) {
  const DCO_TYPE eps = 1e-12;
  DCO_TYPE xp = x + 1;
  while (fabs(x - xp) > eps) {
    xp = x;
    x = xp - (xp * xp - p) / (2 * xp);
  }
}
```

```cpp
#include <iostream>
#include "dco.hpp"

using namespace std;
using namespace dco;

typedef ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

#include "f.hpp"

void driver(const double& pv, double& pa, double& xv, const double& xa) {
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  DCO_TYPE x = xv, p = pv;
  tape->register_variable(p);
  f(p, x);
  cerr << "ts=" << dco::size_of(tape) << "B" << endl;
  xv = value(x);
  tape->register_output_variable(x);
  derivative(x) = xa;
  tape->interpret_adjoint();
  pa = derivative(p);
}
```

```
#include <iostream>
using namespace std;

#include "driver.hpp"

int main() {
  double xv = 1, pv = 5, xa = 1, pa = 0;
  driver(pv, pa, xv, xa);
  cout.precision(5);
  cout << "x=" << xv << endl;
  cout << "p_{(1)}=" << pa << endl;
}
```

The following output is generated:

```
ts=0.000160217MB

x=1808.04241445606
p_{(1)}=2712.06362168409
```

Calling `g` instead of `g_make_gap` in `f` yields

```
ts=0.000946045MB

x=1808.04241445606
p_{(1)}=2712.06362168409
```

### 4.34.3 Concepts introduced in this Section

None.

## 4.35 Second-order Adjoint Mode: User-Defined First-order Adjoints

Example program can be found here: `examples/gt2s_ga1s_user_defined_adjoint`

### 4.35.1 Purpose

This section illustrates the definitions of custom first-order adjoints by the user in second-order adjoint mode.

### 4.35.2 Example

```cpp
#include <iostream>
#include "dco.hpp"

using namespace std;
using namespace dco;

typedef gt1s<double> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "f.hpp"

void driver(const double& pv, const double& pt2, double& pa1, double& pa1t2,
            double& xv, double& xt2, const double& xa1, const double& xa1t2) {
  DCO_TYPE p, x;
  passive_value(p) = pv;
  derivative(value(p)) = pt2;
  passive_value(x) = xv;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  tape->register_variable(p);
  f(p, x);
  cerr << "ts=" << dco::size_of(tape) << "B" << endl;
  xv = passive_value(x);
  xt2 = derivative(value(x));
  tape->register_output_variable(x);
  value(derivative(x)) = xa1;
  derivative(derivative(x)) = xa1t2;
  value(derivative(p)) = pa1;
  derivative(derivative(p)) = pa1t2;
  tape->interpret_adjoint();
  pa1 = value(derivative(p));
  pa1t2 = derivative(derivative(p));
}
```

```cpp
#include <cassert>
#include <cstdlib>
#include <iostream>
using namespace std;

#include "driver.hpp"

int main() {
  cout.precision(5);
  double p = 5, pt2 = 1, pa1 = 0, pa1t2 = 0, x = 1, xt2 = 0, xa1 = 1, xa1t2 = 0;
  driver(p, pt2, pa1, pa1t2, x, xt2, xa1, xa1t2);
  cout << "x=" << x << endl;
  cout << "x^{(2)}=" << xt2 << endl;
  cout << "p_{(1)}=" << pa1 << endl;
  cout << "p_{(1)}^{(2)}=" << pa1t2 << endl;
}
```

The following output is generated:

```
ts=0.0002593994140625MB

x=1808.04241445606
x^{(2)}=2712.06362168409
p_{(1)}=2712.06362168409
p_{(1)}^{(2)}=4068.09543252614
```

Calling `g` instead of `g_make_gap` in `f` yields

```
ts=0.00146484375MB

x=1808.04241445606
x^{(2)}=2712.06362168409
p_{(1)}=2712.06362168409
p_{(1)}^{(2)}=4068.09543252614
```

#### 4.35.3 Concepts introduced in this Section

None.

### 4.36 First-order Adjoint Mode: Local Jacobian Preaccumulation

Example program can be found here: `examples/ga1s_jacobian_preaccumulation`

#### 4.36.1 Purpose

The size of the tape is reduced by preaccumulation of local Jacobians. An easy-to-use interface is provided.

#### 4.36.2 Remarks

Registration of outputs throws `std::runtime_error` if the passed variable is not an output.

Nesting the preaccumulator and external adjoint functionality is not supported.

#### 4.36.3 Example

**Example Text**

We consider the following implementation of an implicit Euler scheme for an initial value problem for state `x` with control parameter `p` defined at each time step.

```
template<typename T>
void euler(const int from, const int to, T& x, const vector<T>& p) {
  double dt=1./p.size();
  for (int i=from;i<to;i++) {
    T x_prev=x;
    T f=-dt*p[i]*sin(x*i*dt);
    while (abs(f)>1e-7) {
      x=x-f/(1-dt*p[i]*i*dt*cos(x*i*dt));
      f=x-x_prev-dt*p[i]*sin(x*i*dt);
    }
  }
}
```

The gradient of the approximate solution with respect to p is computed by the following driver.

```
1  #include <iostream>
2  #include <cmath>
3  #include <vector>
4  #include "dco.hpp"
5
6  using namespace std;
7
8  typedef dco::ga1s<double> DCO_MODE;
9  typedef DCO_MODE::type DCO_TYPE;
10 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
11
12 #include "f.h"
13
14 void driver() {
15   const int n = 20, s = 10;
16   const double x0 = 1;
17   vector<DCO_TYPE> p(n, 1);
18   dco::smart_tape_ptr_t<DCO_MODE> tape;
19   for (int i = 0; i < n; i++)
20     tape->register_variable(p[i]);
21   DCO_TYPE x = x0;
22   DCO_MODE::jacobian_preaccumulator_t jp(tape);
23   for (int i = 0; i < n; i += s) {
24     jp.start();
25     euler(i, min(i + s, n), x, p);
26     jp.register_output(x);
27     jp.finish();
28   }
29   tape->register_output_variable(x);
30   dco::derivative(x) = 1;
31   tape->interpret_adjoint();
32   for (int i = n / 2 - 3; i < n / 2 + 3; i++)
33     cout << "dx/dp[" << i << "]=" << dco::derivative(p[i]) << endl;
34 }
```

Local Jacobians (here gradients) of consecutive chunks of s implicit Euler steps are preaccumulated to reduce the size of the tape. A corresponding jacobian_preaccumulator_t object jp is created in line 22. A preaccumulation step is initiated in line 24 causing the following s implicit Euler steps to be recorded followed by preaccumulation of the local Jacobian in adjoint mode. The actual preaccumulation is triggered by registration of all active local outputs (here only x in line 26) and finalization in line 27.

### Example Results

Instead of printing the whole gradient of size 1000 we inspect only selected elements. The following output is generated:

```
dx/dp[497]=0.000636433
dx/dp[498]=0.000637576
dx/dp[499]=0.000638718
dx/dp[500]=0.00063986
dx/dp[501]=0.000641
dx/dp[502]=0.00064214
```

### 4.36.4 Concepts introduced in this Section

**`jacobian_preaccumulator_t`**

Easy to use interface for reduction of tape size through preaccumulation of local Jacobians. The constructor of the Jacobian preaccumulator type expects a pointer to the associated tape as argument.

**`jacobian_preaccumulator_t::start()`**

Start recording of tape used for preaccumulation.

**`jacobian_preaccumulator_t::register_output(DCO_TYPE&)`**

Register active outputs of computation subject to preaccumulation.

**`jacobian_preaccumulator_t::finish()`**

Preaccumulate local Jacobian and integrate into associated tape.

## 4.37 Second-order Adjoint Mode: Local Jacobian Preaccumulation

Example program can be found here: `examples/gt2s_ga1s_jacobian_preaccumulation`

### 4.37.1 Purpose

First-order adjoints using preaccumulation of local Jacobians can be extended to second- (and higher-)order adjoints by simply replacing the passive `DCO_BASE_TYPE` with an active first- (or higher-)order tangent type.

### 4.37.2 Example

The example from Sec. 4.36.3 is extended towards second-order adjoints.

**Example Text**

```
#include <iostream>
#include <cmath>
#include <vector>
#include "dco.hpp"
```

```

using namespace std;

typedef dco::gt1s<double>::type DCO_BASE_TYPE;
typedef dco::ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "f.h"

void driver() {
  const int n = 20, s = 10;
  const double x0 = 1;
  vector<DCO_TYPE> p(n, 1);
  vector<vector<double>> H(n, vector<double>(n, 0));
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  for (int i = 0; i < n; i++)
    tape->register_variable(p[i]);
  DCO_TAPE_TYPE::position_t tpos = tape->get_position();
  for (int j = 0; j < n; j++) {
    DCO_TYPE x = x0;
    dco::derivative(dco::value(p[j])) = 1;
    DCO_MODE::jacobian_preaccumulator_t jp(tape);
    for (int i = 0; i < n; i += s) {
      jp.start();
      euler(i, min(i + s, n), x, p);
      jp.register_output(x);
      jp.finish();
    }
    tape->register_output_variable(x);
    dco::derivative(x) = 1;
    tape->interpret_adjoint();
    for (int i = 0; i < n; i++) {
      H[i][j] = dco::derivative(dco::derivative(p[i]));
      dco::derivative(p[i]) = 0;
    }
    tape->reset_to(tpos);
    dco::derivative(dco::value(p[j])) = 0;
  }
  for (int j = n / 2 - 1; j < n / 2 + 1; j++)
    for (int i = n / 2 - 1; i < n / 2 + 1; i++)
      cout << "ddx/dp[" << j << "]dp[" << i << "]=" << H[j][i] << endl;
}
```

### Example Results

We restrict the output of the Hessian to a few selected entries:

```
ddx/dp[49]dp[49]=4.34577e-05
ddx/dp[49]dp[50]=1.71089e-05
ddx/dp[50]dp[49]=1.71089e-05
ddx/dp[50]dp[50]=4.47534e-05
```

### 4.37.3 Concepts introduced in this Section

Seamless extension of a first-order adjoint with local Jacobian preaccumulation to second (and higher) order is possible.

## 4.38 First-order Adjoint Mode: Multiple Adjoint Vectors

Example program can be found here: `examples/ga1s_multiple_adjoint_vectors`

### 4.38.1 Purpose

Recording of a single tape can be followed by repeated (concurrent) interpretations in separate address spaces implemented by multiple adjoint vectors of potentially different types; see also `DCO_ADJOINT_TYPE` in Ch. 4.9.

### 4.38.2 Example

The implicit Euler case study from Sec. 4.36.3 is considered.

### 4.38.3 Example Text

OpenMP multithreading is used for parallel interpretation of a single tape for the implicit Euler scheme using two adjoint vectors.

```
#include <iostream>
#include <cmath>
#include <omp.h>

#include "dco.hpp"
typedef dco::ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "f.h"

using namespace std;

void driver(float& adjoint_float, double& adjoint_double) {
  const int n = 1000;
  double x0 = 1;
  vector<DCO_TYPE> p(n, 1);
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  for (int i = 0; i < n; i++)
    tape->register_variable(p[i]);
  DCO_TYPE x = x0;
  euler(0, n, x, p);
  tape->register_output_variable(x);

  omp_set_num_threads(2);

#pragma omp parallel
28    {
29      switch (omp_get_thread_num()) {
30      case 0: {
31        dco::adjoint_vector<DCO_TAPE_TYPE, float> av_float(tape);
32        dco::derivative(x, av_float) = 1;
33        av_float.interpret_adjoint();
34        adjoint_float = dco::derivative(p[n / 2], av_float);
35        break;
36      }
37      case 1: {
38        dco::adjoint_vector<DCO_TAPE_TYPE, double> av_double(tape);
39        dco::derivative(x, av_double) = 1;
40        av_double.interpret_adjoint();
41        adjoint_double = dco::derivative(p[n / 2], av_double);
42      }
43      default: {
44      }
45      }
46    }
47  }
```

A single tape is recorded in lines 17-22 prior to entering the parallel regions in line 33. Thread 0 allocates an adjoint vector over `DCO_ADJOINT_TYPE=`**`float`** in lines 37-38. Similarly, an adjoint vector over `DCO_ADJOINT_TYPE=`**`double`** is allocated by thread 1 in lines 45-46. Access to derivative components requires specification of the associated adjoint vector (e.g., lines 41 and 49). Interpretation is called for the individual adjoint vectors in lines 40 und 48 followed by printing the respective results.

### 4.38.4 Example Results

The following output is generated.

```
float dx/dp=0.000639859
double dx/dp=0.00063986
```

Only six significant digits can be expected from the computation in single precision.

### 4.38.5 Concepts introduced in this Section

**`adjoint_vector<DCO_TAPE_TYPE,DCO_ADJOINT_TYPE>`**

Adjoint vector of type `DCO_ADJOINT_TYPE` associated with a tape of type `DCO_TAPE_TYPE`. A given instance of `adjoint_vector` is referred to as `DCO_ADJOINT_VECTOR_TYPE`. The constructor of `adjoint_vector` expects a pointer to the associated tape as an argument.

**`derivative(DCO_T&,DCO_ADJOINT_VECTOR_TYPE&)`**

Derivative access requires specification of the associated adjoint vector.

**`DCO_ADJOINT_VECTOR_TYPE::interpret_adjoint()`**

Interpreters are called directly on the adjoint vector.

## 4.39 Modulo Adjoint Propagation

Example program can be found here: `examples/ga1s_mod`

### 4.39.1 Purpose

For each adjoint mode `X` already shown (i.e. `X` $\in$ `ga1s, ga1sm, ga1v, ga1vm`) there is a mode called `X_mod`, where the required size for the vector of adjoints is potentially much smaller than with the usual mode. The vector of adjoints is compressed by analysing the maximum number of required distinct adjoint memory locations. During interpretation, adjoint memory, which is no longer required, is overwritten and thus reused by indexing through modulo operations.

This feature is especially useful for iterative algorithms (e.g. time iteration). The required memory for the vector of adjoints usually stays constant, independent of the number of iterations.

This mode requires many modulo operations during interpretation. This introduces a slow down. Expect the recording runtime of `X_mod` to be similar to mode `X`.

## 4.40 Optimization

A faster modulo operation can be performed on numbers which are a power of two:

```
int a = rand();
a % 32 == a & 31; // this is true
```

To enable this feature, define `DCO_BITWISE_MODULO` at compile time (see Ch. 2). This will round up the adjoint vector size to the next power of two for exploiting the above optimization. (Thanks to M. Towara)

Please check memory decrease with

```
  dco::size_of(tape, TAPE::size_of_internal_adjoint_vector)}
```

see Sec. 3.1.8.

### 4.40.1 Example

We consider an example for `ga1s_mod`: A simple Euler time stepping for the one-dimensional ordinary differential equation

$$\frac{dx}{dt} = f(x(t,p),t,p) \tag{4.7}$$

with state $x \in \mathbb{R}$, free parameter $p \in \mathbb{R}$ and initial condition $x(0,p) = 1$. Time is discretized equidistantly between 0 and 1 into $n$ time steps. This is implemented in `timestepping.hpp`:

```
#include <cmath>

template <typename T> void f(const int n, T& x, const T& p) {
  double dt = 1. / n, t = 0;
  for (int i = 0; i < n; i++, t += dt)
    x += dt * p * sin(x * t);
}
```

The main routine is given as follows.

```
#include "dco.hpp"

using namespace std;
using namespace dco;

typedef dco::ga1s_mod<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "timestepping.hpp"

int main(int c, char* v[]) {
  int n = 10;
  if (c == 2) {
    n = atoi(v[1]);
  }

  DCO_TYPE x = 1.0, p = 1.0;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  tape->register_variable(p);

  f(n, x, p);

  dco::derivative(x) = 1.0;
  tape->interpret_adjoint();

  cerr << "size of internal adjoint vector = "
       << dco::size_of(tape, DCO_TAPE_TYPE::size_of_internal_adjoint_vector)
       << "b" << endl;
  cerr << "size of tape memory = "
       << dco::size_of(tape, DCO_TAPE_TYPE::size_of_stack) << "b" << endl;

  cout << "dp=" << dco::derivative(p) << endl;
}
```

Output for $n = 10$ is

```
size of internal adjoint vector = 40b
size of tape memory = 300b
```

and for $n = 1000$

```
size of internal adjoint vector = 40b
size of tape memory = 35940b
```

The size of the internal adjoint vector stays constant, while the stack memory grows with the number of time iterations. In combination with the disk tape, this is a very powerful feature.

Remark: The maximum number of required distinct adjoint memory location depends on the *farthest dependency* apart from the registered variables.

Remark: It is required to register the inputs at the very beginning, before recording anything. If variables are registered during recording, their adjoint is only correct at the respective position during interpretation. Use `get_position()` tape member function.

### 4.40.2 Concepts introduced in this Section

`ga1s_mod<DCO_BASE_TYPE>`

Generic first-order adjoint scalar mode using modulo adjoint propagation.

## 4.41 Sparse Tape Interpretation

Example program can be found here: `examples/sparse_interpret`

### 4.41.1 Purpose

The adjoint interpretation of the tape can omit propagation along edges when the corresponding adjoint to be propagated is zero. This might be of use when NaNs or Infs occur as local partial derivatives, but this local result is not used for subsequent computations which are relevant for the overall output. This option should not be used by default since it might hide NaNs or Infs that hint to actual problems in the code (e.g. $\log(x)$ at $x = 0$). If a respective tangent version does not suffer from NaNs or Infs, it is likely that this option resolves the issue for the adjoint. It can be used in the following way:

```
...
  dco::smart_tape_ptr_t<DCO_M> tape;
  // record
  tape->sparse_interpret() = true;
  tape->interpret_adjoint();

  tape->zero_adjoints();
  tape->sparse_interpret() = false;
  tape->interpret_adjoint();
...
```

### 4.41.2 Example

We consider an example for sparse interpretation: A specific path in the program produces NaN local partial derivatives. The overall output is independent of this local variable though; so sparse interpret resolves the problem of NaN propagation.

```
#include "dco.hpp"

typedef dco::ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

int main() {

  dco::smart_tape_ptr_t<DCO_MODE> tape;
  DCO_TYPE x(0.0);

  tape->register_variable(x);

  DCO_TYPE t = sqrt(x * x);
  (void)t; // unused...
  DCO_TYPE y = 2 * x;

  dco::derivative(y) = 1.0;
  tape->interpret_adjoint();

  std::cerr << "Adjoint of x without sparse interpretation: "
            << dco::derivative(x) << std::endl;

  tape->zero_adjoints();
  tape->sparse_interpret() = true;

  dco::derivative(y) = 1.0;
  tape->interpret_adjoint();

  std::cout << "Adjoint of x with sparse interpretation: " << dco::derivative(x)
            << std::endl;
}
```

### 4.41.3 Concepts introduced in this Section

**`TAPE_T::sparse_interpret()`**

Enable and disable sparse interpretation.

## 4.42 First-order Adjoint Mode: Custom Tape Interpreation Hook

Example program can be found here: `examples/ga1s_custom_tape_interpretation_hook`

### 4.42.1 Purpose

This section illustrates the use of a custom callable inserted into the interpreter to monitor edge propagation. In the example, the Jacobian sparsity information is extracted in two ways. Both the indices of inputs with nonzero adjoints and the Jacobian matrix stored in Coordinate Sparse Format are returned. A global tape is used.

### 4.42.2 Example

```
#include <iostream>
#include <vector>

#include "dco.hpp"
// This example program demonstrates how a custom hook can be inserted into
// interpreter to monitor edge propagation.

// Operator of the discretised 1D Poisson equation:
template <typename T> void f(std::vector<T>& x, std::vector<T>& y) {
  for (std::size_t i = 0; i < x.size(); i++) {
    if (i == 0) {
      y[i] = 2 * x[i] - x[i + 1];
    } else if (0 < i && i < x.size() - 1) {
      y[i] = -x[i - 1] + 2 * x[i] - x[i + 1];
    } else {
      y[i] = -x[i - 1] + 2 * x[i];
    }
  }
}

int main() {

  using mode_t = dco::ga1s<double>;
  using type = mode_t::type;

  dco::smart_tape_ptr_t<mode_t> tape;

  size_t n = 4;
  std::vector<type> x(n, 1.0), y(n, 0.0);
  tape->register_variable(x);
  f(x, y);
  tape->register_output_variable(y);

  // Indices of inputs with nonzero adjoints:
  std::vector<dco::index_t> indices;
  // Store Jacobian matrix in coordinate sparse format:
  std::vector<dco::index_t> rows, cols, vals;
  // A reference to the internal vector of adjoints:
  auto& adjoints = tape->adjoints();

  for (size_t j = 0; j < n; j++) {
    // Extract the row, column and value components of the Jacobian matrix
    // stored in COO Sparse format and indices of inputs with nonzero adjoints.
    auto lambda = [&](dco::index_t idx, mode_t::derivative_t const& inc) {
      if (idx <= dco::tape_index(x.back()) && inc != 0.0) {
        // Fill COO matrix:
        rows.push_back(j);
        cols.push_back(idx - 1);
        vals.push_back(adjoints[idx]);
        // Get indices of inputs with nonzero adjoints:
        indices.push_back(idx);
      }
    };
    tape->zero_adjoints();
    dco::derivative(y[j]) = 1.0;
    tape->interpret_adjoint(lambda);

    std::cout << "Printing indices of nonzero adjoints for j=" << j << ":"
              << std::endl;
    for (size_t i = 0; i < indices.size(); i++) {
      std::cout << "Input variable stored in index " << indices[i];
      std::cout << " has corresponding derivative = " << adjoints[indices[i]]
                << "." << std::endl;
    }
    indices.clear();
    std::cout << std::endl << std::endl;
  }

  // Multiple occurences of the same (row,column) pair could appear in this
  // matrix structure. In such case, the value of the element stored in the
  // (row,column) position is the sum of the value components in the occurences.
  std::cout << "Printing elements of the sparse Jacobian matrix stored in "
               "coordinate sparse format:"
            << std::endl;
  for (std::size_t i = 0; i < rows.size(); i++) {
    std::cout << "(row = " << rows[i] << ", column = " << cols[i]
              << ", value = " << vals[i] << ")" << std::endl;
  }
}
```

The following output is generated for `n=10` and `m=2`:

```
Printing indices of nonzero adjoints for j=0:
Input variable stored in index 2 has corresponding derivative = -1.
Input variable stored in index 1 has corresponding derivative = 2.


Printing indices of nonzero adjoints for j=1:
Input variable stored in index 3 has corresponding derivative = -1.
Input variable stored in index 2 has corresponding derivative = 2.
Input variable stored in index 1 has corresponding derivative = -1.


Printing indices of nonzero adjoints for j=2:
Input variable stored in index 4 has corresponding derivative = -1.
Input variable stored in index 3 has corresponding derivative = 2.
Input variable stored in index 2 has corresponding derivative = -1.


Printing indices of nonzero adjoints for j=3:
Input variable stored in index 4 has corresponding derivative = 2.
Input variable stored in index 3 has corresponding derivative = -1.


Printing elements of the sparse Jacobian matrix stored in coordinate sparse format:
(row = 0, column = 1, value = -1)
(row = 0, column = 0, value = 2)
(row = 1, column = 2, value = -1)
(row = 1, column = 1, value = 2)
(row = 1, column = 0, value = -1)
(row = 2, column = 3, value = -1)
(row = 2, column = 2, value = 2)
(row = 2, column = 1, value = -1)
(row = 3, column = 3, value = 2)
(row = 3, column = 2, value = -1)
```

### 4.42.3 Concepts introduced in this Section

**DCO_TAPE_TYPE::adjoints**

Returns a reference to the internal vector of adjoints.

## 4.43 Logging

### 4.43.1 Purpose

Logging might be useful for information and debugging purposes.

Logging has different levels of verbosity. The maximum logging level is defined at compile time via preprocessor define `DCO_LOG_MAX_LEVEL`. This is required, since high logging levels have an impact on performance. In addition, the logging level can be set at runtime (up to the maximum logging level).

With `g++` the logging can be disabled by

```
g++ main.cpp -DDCO_LOG_MAX_LEVEL=-1
```

and fully enable by

```
g++ main.cpp -DDCO_LOG_MAX_LEVEL=7
```

By default, the logger writes to `stderr`.

### 4.43.2 Example

The user interface to logging at runtime is via a global `logger` object.

```
...
  dco::logger::level()=dco::logINFO;
  dco::logger::stream()=stdout;
...
```

In the example, the level is set to `logINFO` and the stream is changed to `stdout`. The default level is set to `DCO_LOG_MAX_LEVEL`.

### 4.43.3 Concepts introduced in this Section

**enum dco::log_level_enum**

Different verbosity levels, defined as

```
 enum log_level_enum {logERROR, logWARNING, logINFO, logDEBUG, logDEBUG1,
     logDEBUG2, logDEBUG3, logDEBUG4};
```

**dco::log_level_enum& dco::logger::level()**

Set/get the current logging level.

```
FILE*& dco::logger::stream()
```

Set/get the current logging stream. Default is `stderr`, but could also be `stdout` or a file, i.e. `std::fopen("dco.log", "w")`.

## 4.44 Explicit Type Cast to Specified Type

### 4.44.1 Purpose

As already written in *General Remarks / Essentials*, Ch. 2: Usually, it is not desired to cast a dco/c++ type to a non-dco type (e.g. **double**), since dco/c++ is then unable to compute derivatives of that data flow. Nonetheless, there are cases, where a cast is actually what you want to do (e.g. cast to **int** for an index calculation). We therefore allow explicit cast operations to specified data types (implicit casts are not allowed). Use carefully.

#### Remarks

This is only available in C++11, since the implementation uses the **explicit** keyword on conversion functions. Please see https://en.cppreference.com/w/cpp/language/cast_operator for description of **explicit** keyword. This enables cast operations for explicit conversion and *direct-initialization*. The latter might be a source for errors, see Example.

### 4.44.2 Example

```
#include "dco.hpp"
DCO_ENABLE_EXPLICIT_TYPE_CAST_TO(int)

int main() {
  dco::gt1s<double>::type x(1.0);
  // explicit conversion
  int i  = static_cast<int>(x);
  // direct-initialization
  int j(x);
}
```

### 4.44.3 Concepts introduced in this Section

```
DCO_ENABLE_EXPLICIT_TYPE_CAST_TO(TYPE)
```

This macro enables the explicit operator to type `TYPE`, i.e.

```
explicit operator TYPE() { return static_cast<TYPE>(dco::value(*this)); }
```

That means, it will automatically try recursively to cast to the specified type, if requested. The recursion is required for higher-order types.

## 4.45 Huge Pages

This is currently only supported under Linux. Please contact NAG (`support@nag.co.uk`) if you want this feature for other platforms.

### 4.45.1 Purpose

See for example https://wiki.debian.org/Hugepages for a description of huge pages in Debian. By default, most systems use a page size of 4kB. Since dco/c++ uses a lot of memory for the tape, a lot of pages are allocated. To get faster lookup, larger page sizes can be used (e.g. 2MB on usual Linux systems).

### 4.45.2 Usage

The `tape_options` object can be used to enable huge page allocation in dco/c++.

```
dco::tape_options o;
o.alloc_use_huge_pages() = true;
dco::smart_tape_ptr_t<DCO_MODE> tape(o);
```

Please make sure you've enabled huge pages support in your system. Under Linux, memory for huge pages needs to be created at system level first by running

```
sudo sysctl -w vm.nr_hugepages=NUMBER
```

where `NUMBER` is the number of huge pages available for allocation. This memory will be directly blocked by the system (see e.g. `htop`). You can see how many pages are blocked by running

```
cat /proc/meminfo | grep Huge
```

A sample output would look like the following:

```
AnonHugePages:          0 kB
ShmemHugePages:         0 kB
HugePages_Total:     1024
HugePages_Free:      1024
HugePages_Rsvd:         0
HugePages_Surp:         0
Hugepagesize:        2048 kB
```

## 4.46 Thread-Parallel Taping: Monte Carlo

Example program can be found here: `examples/thread_parallel_taping`

### 4.46.1 Purpose

The dco/c++ tape is by default thread-local. Per thread, with ga1s, the tape has global scope. When using a main thread to compute the first part of a function, and then starting thread-parallel execution, two things need to be taken into account when computing adjoints.

- A synchronization between the main and the thread-local tapes need to take place when the thread-parallel execution starts.
- A reduction of thread-local adjoints is required after having interpreted the thread-local tapes.

### 4.46.2 Example

We've implemented a mocked-up Monte Carlo pricer. The main thread does the calibration, which produces a result used by all threads in the thread-parallel Monte Carlo. We implemented pathwise adjoints with early propagation here, since we know the output adjoints already.

### 4.46.3 Concepts introduced in this Section

**`DCO_TAPE_TYPE::synchronize_with(DCO_TAPE_TYPE const * const)`**

Synchronizes the tape state (internal counters) with the passed tape. This is required to be able to continue recording on this tape with variables previously registered in the other tape.

**`DCO_TAPE_TYPE::adjoint(DCO_TYPE const&)`**

Returns a reference to the corresponding adjoint variable from the internal vector of adjoints.

# Appendix A

# Status of User Interface

Definitions:

- **Gold**: tested, signature fix, fully documented
- **Silver**: tested, signature fix
- **Bronze**: tested
- **Undefined**: experimental

We assume global use of namespace **dco**.

Disclaimer: The following code listings do not show the exact dco source. They are meant to give you an overview of the available features. Refer to the example programs for more precise information.

## A.1 Gold

We are in the process of discussing the level of detail required for features to be considered "fully documented."

## A.2 Silver

— The tangent first-order scalar mode over the given `DCO_BASE_TYPE`.

```
typedef gt1s<DCO_BASE_TYPE> DCO_GT1S_MODE;
```

— The tangent first-order scalar type for the given `DCO_GT1S_MODE`.

```
typedef DCO_GT1S_MODE::type DCO_GT1S_TYPE;
```

— The adjoint first-order scalar mode over the given `DCO_BASE_TYPE`.

```
typedef ga1s<DCO_BASE_TYPE> DCO_GA1S_MODE;
```

— The adjoint first-order scalar mode over the given `DCO_VALUE_TYPE`, `DCO_PARTIAL_TYPE`, and `DCO_ADJOINT_TYPE`.

```
typedef ga1s< DCO_VALUE_TYPE, DCO_PARTIAL_TYPE, DCO_ADJOINT_TYPE >
        DCO_GA1S_MODE;
```

— The chunk size of tape determined by the user.

```
dco::tape_options o;
o.set_chunk_size_in_byte(1024*1024*1024);
o.set_chunk_size_in_kbyte(1024*1024);
o.set_chunk_size_in_mbyte(1024);
o.set_chunk_size_in_gbyte(1);
std::cout << o.chunk_size_in_byte() << std::endl;
dco::smart_tape_ptr_t<DCO_M> tape(o);
```

— The blob size of tape determined by the user.

```
dco::tape_options o;
o.set_blob_size_in_byte(1024*1024*1024);
o.set_blob_size_in_kbyte(1024*1024);
o.set_blob_size_in_mbyte(1024);
o.set_blob_size_in_gbyte(1);
std::cout << o.blob_size_in_byte() << std::endl;
dco::smart_tape_ptr_t<DCO_M> tape(o);
```

— When using chunk tape, whether to deallocate chunks on tape reset. They will still be deallocated on remove.

```
dco::tape_options o;
o.deallocation_on_reset() = false;
std::cout << o.deallocation_on_reset() << std::endl;
dco::smart_tape_ptr_t<DCO_M> tape(o);
```

— The adjoint first-order scalar type for the given `DCO_GA1S_MODE`.

```
typedef DCO_GA1S_MODE::type DCO_GA1S_TYPE;
```

— The tangent first-order vector mode over the given `DCO_BASE_TYPE` with the vector length `v_size`.

```
typedef gt1v< DCO_BASE_TYPE, v_size=1> DCO_GT1V_MODE;
```

— The tangent first-order vector type for the given `DCO_GT1V_MODE`.

```
typedef DCO_GT1V_MODE::type DCO_GT1V_TYPE;
```

— The adjoint first-order vector mode over the given `DCO_BASE_TYPE` with the vector length `v_size`.

```
typedef ga1v< DCO_BASE_TYPE, v_size=1 > DCO_GA1V_MODE;
```

— The adjoint first-order vector type for the given `DCO_GA1V_MODE`.

```
typedef DCO_GA1V_MODE::type DCO_GA1V_TYPE;
```

— The adjoint first-order scalar mode with support for multiple tapes over the given `DCO_BASE_TYPE`.

```
typedef ga1sm< DCO_BASE_TYPE > DCO_GA1SM_MODE;
```

— The adjoint first-order scalar type with support for multiple tapes for the given DCO_GA1SM_MODE.

```
typedef DCO_GA1SM_MODE::type DCO_GA1SM_TYPE;
```

— The adjoint first-order scalar mode over the given DCO_BASE_TYPE using modulo adjoint propagation.

```
typedef ga1s_mod< DCO_BASE_TYPE > DCO_GA1S_MOD_MODE;
```

— The adjoint first-order vector mode over the given DCO_BASE_TYPE using modulo adjoint propagation.

```
typedef ga1v_mod< DCO_BASE_TYPE, v_size=1 > DCO_GA1SV_MOD_MODE;
```

— The adjoint first-order scalar mode with support for multiple tapes using modulo adjoint propagation over the given DCO_BASE_TYPE.

```
typedef ga1sm_mod< DCO_BASE_TYPE > DCO_GA1SM_MOD_MODE;
```

— The adjoint first-order vector mode with support for multiple tapes using modulo adjoint propagation over the given DCO_BASE_TYPE.

```
typedef ga1vm_mod< DCO_BASE_TYPE > DCO_GA1vM_MOD_MODE;
```

From now on:

- DCO_MODE ∈ DCO_x_MODE with x ∈ {GT1S, GA1S, GA1SM, GT1V, GA1V, GA1S_MOD, GA1SM_MOD , GA1V_MOD, GA1VM_MOD} over base type DCO_BASE_TYPE
- **typedef** DCO_MODE::type DCO_TYPE;
- [] denotes optional descriptor

— The type of value component of variables of type DCO_MODE::type (equal DCO_BASE_TYPE).

```
typedef DCO_MODE::value_t DCO_VALUE_TYPE;
```

— The type of derivative component of variables of type DCO_MODE::type.

```
typedef DCO_MODE::derivative_t DCO_DERIVATIVE_TYPE;
```

— The type of underlying passive value component of variables of type DCO_MODE::type.

```
typedef DCO_MODE::passive_t DCO_PASSIVE_TYPE;
```

— Returns [read-only] reference to passive value of x (usually **double**); supports std::vector.

```
[const] DCO_PASSIVE_TYPE& passive_value ( [const] DCO_TYPE &x );
```

— Returns [read-only] reference to value component of x; supports std::vector.

```
[const] DCO_VALUE_TYPE& value ( [const] DCO_TYPE &x );
```

— Returns [read-only] reference to derivative component (tangent or adjoint) of x; supports std::vector.

```
[const] DCO_DERIVATIVE_TYPE& derivative ( [const] DCO_TYPE &x );
```

— The tape type; is **void** for all tangent modes.

```
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
```

The following description is only valid for adjoint modes.

— Smart pointer type for the tape. It allocates and deallocates in constructor and destructor, correspondingly. It is a unified interface for all adjoint modes. If used with a mode based on a global tape, it uses the (thread-local) global tape pointer internally.

```
template <typename MODE_T> struct dco::smart_tape_ptr_t;
```

— The global tape. Raw global pointer should only be used if necessary. Use `dco::smart_tape_ptr_t` as first option.

```
DCO_TAPE_TYPE* DCO_MODE::global_tape;
```

— Creates tape and returns pointer to it. Raw global tape pointer and corresponding allocation / deallocation routines (create, remove) should only be used if necessary. Use `dco::smart_tape_ptr_t` as first option.

```
static DCO_TAPE_TYPE* DCO_TAPE_TYPE::create ();
```

— Deallocates tape pointed at by `t` and sets `t = 0`. Raw global tape pointer and corresponding allocation / deallocation routines (create, remove) should only be used if necessary. Use `dco ::smart_tape_ptr_t` as first option.

```
static void DCO_TAPE_TYPE::remove ( DCO_TAPE_TYPE* &t );
```

— A position in tape.

```
typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
```

— Returns current position in tape.

```
DCO_TAPE_POSITION_TYPE DCO_TAPE_TYPE::get_position ();
```

— Marks `x` as an independent variable.

```
void DCO_TAPE_TYPE::register_variable ( DCO_TYPE &x );
```

— Marks `x` as a dependent variable.

```
void DCO_TAPE_TYPE::register_output_variable ( DCO_TYPE &x );
```

— Runs tape interpreter. A callable can be provided as an argument to customize the interpreter, e.g., for extracting Jacobian sparsity information.

```
template <typename LAMBDA_T>
void DCO_TAPE_TYPE::interpret_adjoint (
  LAMBDA_T const & lambda = [&](dco::index_t, DCO_DERIVATIVE_TYPE){}
);
```

— Runs tape interpreter from tape position '`from`' to first entry. A callable can be provided as an argument to customize the interpreter, e.g., for extracting Jacobian sparsity information.

```
template <typename LAMBDA_T>
void DCO_TAPE_TYPE::interpret_adjoint_from (
  const DCO_TAPE_POSITION_TYPE &from,
  LAMBDA_T const & lambda = [&](dco::index_t, DCO_DERIVATIVE_TYPE){}
);
```

— Runs tape interpreter from last entry to tape position 'to'. A callable can be provided as an argument to customize the interpreter, e.g., for extracting Jacobian sparsity information.

```
template <typename LAMBDA_T>
void DCO_TAPE_TYPE::interpret_adjoint_to (
  const DCO_TAPE_POSITION_TYPE &to,
  LAMBDA_T const & lambda = [&](dco::index_t, DCO_DERIVATIVE_TYPE){}
);
```

— Runs tape interpreter from tape position 'from' to tape position 'to'. A callable can be provided as an argument to customize the interpreter, e.g., for extracting Jacobian sparsity information.

```
template <typename LAMBDA_T>
void DCO_TAPE_TYPE::interpret_adjoint_from_to (
  const DCO_TAPE_POSITION_TYPE &from,
  const DCO_TAPE_POSITION_TYPE &to,
  LAMBDA_T const & lambda = [&](dco::index_t, DCO_DERIVATIVE_TYPE){}
);
```

— Resets current tape position to first entry.

Remark: Make sure you are not using any active variables from the reset tape section afterwards.

```
void DCO_TAPE_TYPE::reset ();
```

— Runs tape interpreter from last entry to tape position 'to' and sets current tape position to 'to'. A callable can be provided as an argument to customize the interpreter, e.g., for extracting Jacobian sparsity information.

Remark: Make sure you are not using any active variables from the reset tape section afterwards.

```
template <typename LAMBDA_T>
void DCO_TAPE_TYPE::interpret_adjoint_and_reset_to (
  const DCO_TAPE_POSITION_TYPE &to,
  LAMBDA_T const & lambda = [&](dco::index_t, DCO_DERIVATIVE_TYPE){}
);
```

— Runs tape interpreter from tape position 'from' to tape position 'to' and sets corresponding adjoints to zero. A callable can be provided as an argument to customize the interpreter, e.g., for extracting Jacobian sparsity information.

```
template <typename LAMBDA_T>
void DCO_TAPE_TYPE::interpret_adjoint_and_zero_adjoints_from_to (
  const DCO_TAPE_POSITION_TYPE &from,
  const DCO_TAPE_POSITION_TYPE &to,
  LAMBDA_T const & lambda = [&](dco::index_t, DCO_DERIVATIVE_TYPE){}
);
```

— Runs tape interpreter from current position to tape position 'to' and sets corresponding adjoints to zero. A callable can be provided as an argument to customize the interpreter, e.g., for extracting Jacobian sparsity information.

```
template <typename LAMBDA_T>
void DCO_TAPE_TYPE::interpret_adjoint_and_zero_adjoints_to (
  const DCO_TAPE_POSITION_TYPE &to,
  LAMBDA_T const & lambda = [&](dco::index_t, DCO_DERIVATIVE_TYPE){}
);
```

— Set and unset sparse interetation.

```
bool& DCO_TAPE_TYPE::sparse_interpret();
```

— Resets current tape position to 'to'.

Remark: Make sure you are not using any active variables from the reset tape section afterwards.

```
void DCO_TAPE_TYPE::reset_to (
  const DCO_TAPE_POSITION_TYPE &to
);
```

— Sets all adjoints in tape to zero.

```
void DCO_TAPE_TYPE::zero_adjoints ();
```

— Sets all adjoints in tape to zero from tape position 'from' to first tape entry.

```
void DCO_TAPE_TYPE::zero_adjoints_from (
  const DCO_TAPE_POSITION_TYPE &from
);
```

— Sets adjoints in tape to zero from current tape position to tape position 'to'.

```
void DCO_TAPE_TYPE::zero_adjoints_to (
  const DCO_TAPE_POSITION_TYPE &to
);
```

— Sets adjoints in tape to zero from tape position 'from' to tape position 'to'.

```
void DCO_TAPE_TYPE::zero_adjoints_from_to (
  const DCO_TAPE_POSITION_TYPE &from,
  const DCO_TAPE_POSITION_TYPE &to
);
```

— Switches scalar callback mode on or off.

```
bool& DCO_TAPE_TYPE::scalar_callback_mode ();
```

— External adjoint object base class to be derived from for non-standard handling of *gaps*. Not recommended anymore: Use automatic capture functionality of lambdas. See `DCO_TAPE_TYPE ::insert_callback`.

```
typedef DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
```

— Creates external adjoint object and returns pointer to it.

```
DCO_EAO_TYPE* DCO_TAPE_TYPE::create_callback_object<DCO_EAO_TYPE> ();
```

— There are to variants for inserting a callback. The previously recommended and the currently recommended one using lambdas.

  – Currently recommended: Inserts callback into tape. The user has to ensure that the captured state is still alive / valid during tape interpretation. Easiest way of achieving that is by using capture by copy of the lambda `[=]`.

    ```
    template <typename LAMBDA_T>
    void DCO_TAPE_TYPE::insert_callback (
      LAMBDA_T && lambda
    );
    ```

  – Not recommended anymore: Inserts external adjoint object `eao` into tape alongside pointer to function `fill_gap`.

    ```
    void DCO_TAPE_TYPE::insert_callback (
      void (*fill_gap)(DCO_EAO_TYPE*),
      DCO_EAO_TYPE *eao
    );
    ```

— Synchronizes the tape state (internal counters) with the passed tape. This is required to be able to continue recording on this tape with variables previously registered in the other tape.

```
void DCO_TAPE_TYPE::synchronize_with(DCO_TAPE_TYPE const * const)
```

— Returns a reference to the internal vector of adjoints. `DCO_ADJOINT_VECTOR` is an internal data structure which holds the adjoint values. For accessing individual elements `operator[]` can be used.

```
DCO_ADJOINT_VECTOR& DCO_TAPE_TYPE::adjoints()
```

— Returns a reference to the corresponding adjoint variable from the internal vector of adjoints.

```
  DCO_DERIVATIVE_TYPE& DCO_TAPE_TYPE::adjoint(DCO_TYPE const&)
```

— Returns active variable with value `v` after registration with tape `t`. Not recommended anymore: Use automatic capture functionality of lambdas. See `DCO_TAPE_TYPE::insert_callback` .

```
DCO_TYPE DCO_EAO_TYPE::register_output (
  const DCO_BASE_TYPE &v,
  DCO_TAPE_TYPE *t=NULL
);
```

— $i$th call returns adjoint of variable registered as output of *gap* by $i$th call of `register_output`. Not recommended anymore: Use automatic capture functionality of lambdas. See `DCO_TAPE_TYPE ::insert_callback`.

```
DCO_BASE_TYPE DCO_EAO_TYPE::get_output_adjoint ();
```

— Registers `x` as an input to the *gap* and returns its value. Not recommended anymore: Use automatic capture functionality of lambdas. See `DCO_TAPE_TYPE::insert_callback`.

```
DCO_BASE_TYPE DCO_EAO_TYPE::register_input ( const DCO_TYPE &x );
```

— $i$th call adds v to adjoint of variable registered as input of *gap* by $i$th call of register_input. Not recommended anymore: Use automatic capture functionality of lambdas. See DCO_TAPE_TYPE ::insert_callback.

```
void DCO_EAO_TYPE::increment_input_adjoint ( const DCO_BASE_TYPE &v );
```

— Stores generic data required to fill the *gap*;
**important:** a copy is stored internally: copy constructor required. Not recommended anymore: Use automatic capture functionality of lambdas. See DCO_TAPE_TYPE::insert_callback .

```
template<typename TYPE>
void DCO_EAO_TYPE::write_data ( const TYPE &v );
```

— $i$th call returns read-only reference to internal data stored by $i$th call of write_data. Not recommended anymore: Use automatic capture functionality of lambdas. See DCO_TAPE_TYPE ::insert_callback.

```
template<typename TYPE>
const TYPE& DCO_EAO_TYPE::read_data ();
```

— Preaccumulation of local Jacobian.

```
DCO_MODE::jacobian_preaccumulator_t jp(dco::tape(x));
jp.start();
y=f(x); // to be preaccumulated
jp.register_output(y);
jp.finish();
```

— Multiple adjoint vectors for single tape.

```
dco::adjoint_vector<DCO_TAPE_TYPE,DCO_ADJOINT_TYPE> av(dco::tape(x));
dco::derivative(y,av)=1;
av.interpret_adjoint();
dydx=dco::derivative(x,av);
```

## A.3 Bronze

— DCO_MODE for the given DCO_TYPE.

```
typedef mode<DCO_TYPE> DCO_MODE;
```

— Proxy types to access passive value, value and derivative component of one entry in the passed vector.

```
typedef vector_reference <PASSIVE_VALUE, DCO_TYPE,CONTAINER_T> VEC_REF_PVAL;
typedef vector_reference <VALUE        , DCO_TYPE,CONTAINER_T> VEC_REF_VAL;
typedef vector_reference <DERIVATIVE   , DCO_TYPE,CONTAINER_T> VEC_REF_DER;
```

Individual elements can be accessed through member operators

```
double&              VEC_REF_PVAL::operator[](int);
DCO_BASE_TYPE&       VEC_REF_VAL::operator[](int);
DCO_DERIVATIVE_TYPE& VEC_REF_DER::operator[](int);
```

— Returns proxy object for passive value components of x.

```
VEC_REF_PVAL passive_value ( std::vector< DCO_TYPE > &x );
```

— Returns proxy object for value components of x.

```
VEC_REF_VAL value ( std::vector< DCO_TYPE > &x );
```

— Returns proxy object for derivative (tangent or adjoint) components of x.

```
VEC_REF_DER derivative ( std::vector<DCO_TYPE> &x );
```

The following description is only valid for adjoint modes.

— Local gradient for direct insertion into tape.

```
typedef DCO_TAPE_TYPE::local_gradient_t DCO_LOCAL_GRADIENT_TYPE;
```

— Creates local gradient of y for direct insertion into tape; gradient contains n elements.

```
DCO_LOCAL_GRADIENT_TYPE
DCO_MODE::create_local_gradient_object<DCO_LOCAL_GRADIENT_TYPE> (
  DCO_TYPE &y, size_t n
);
```

— Inserts value p of local partial derivative of y with respect to x into local gradient of y.

```
void DCO_LOCAL_GRADIENT_TYPE::put(
  const DCO_TYPE &x,
  const DCO_BASE_TYPE &p
);
```

— Marks elements of standard vector x as dependent.

```
void DCO_TAPE_TYPE::register_output_variable( std::vector<DCO_TYPE> &x );
```

— Marks elements of vector x of size n as dependent.

```
void DCO_TAPE_TYPE::register_output_variable( DCO_TYPE *x, size_t n );
```

— Marks elements of vector x of size n as independent.

```
void DCO_TAPE_TYPE::register_variable( DCO_TYPE *x, size_t n );
```

— Marks elements of standard vector x as dependent.

```
void DCO_TAPE_TYPE::register_variable( std::vector<DCO_TYPE> &x );
```

— Enables recording to tape.

```
void DCO_TAPE_TYPE::switch_to_active ();
```

— Disables recording to tape.

```
DCO_TAPE_TYPE::switch_to_passive ();
```

— Returns **true** if recording to tape is enabled, **false** otherwise.

```
bool DCO_TAPE_TYPE::is_active ();
```

## A.4 Undefined

See description of DCO_EAO_TYPE above.

— Writes adjoints of previously registered outputs of external adjoint object into standard vector v.

```
void DCO_EAO_TYPE::get_output_adjoint( std::vector<DCO_BASE_TYPE> &v );
```

— Writes adjoints of previously registered outputs of external adjoint object into vector v of size s.

```
void DCO_EAO_TYPE::get_output_adjoint( DCO_BASE_TYPE *v, size_t s );
```

— Increments adjoints of previously registered inputs to external adjoint object with values in vector v of size s.

```
void DCO_EAO_TYPE::increment_input_adjoint( const DCO_BASE_TYPE* const v,
                                                  size_t               s );
```

— Increments adjoints of previously registered inputs to external adjoint object with values in standard vector v.

```
void DCO_EAO_TYPE::increment_input_adjoint(
  const std::vector<DCO_BASE_TYPE> &v
);
```

— Registers inputs x to external adjoint object and returns their values as standard vector v.

```
void DCO_EAO_TYPE::register_input(
  const std::vector<DCO_TYPE>      &x,
        std::vector<DCO_BASE_TYPE> &v
);
```

— Registers inputs x to external adjoint object .

```
void DCO_EAO_TYPE::register_input( const std::vector<DCO_TYPE> &x );
```

— Registers s inputs x to external adjoint object and returns their values as vector v.

```
void  DCO_EAO_TYPE::register_input( const DCO_TYPE      *const x,
                                          DCO_BASE_TYPE *      v,
                                          size_t               s );
```

— Registers s outputs x of external adjoint object.

```
void DCO_EAO_TYPE::register_output( DCO_TYPE *x, size_t s );
```

— Registers s outputs x of external adjoint object with values v.

```
void DCO_EAO_TYPE::register_output( const DCO_BASE_TYPE *const v,
                                          DCO_TYPE      *      x,
                                          size_t               s );
```

— Registers values in v as outputs of external adjoint object.

```
void DCO_EAO_TYPE::register_output( const std::vector<DCO_BASE_TYPE> &v );
```

— Registers values in v as outputs of external adjoint object and returns them as vector x .

```
void DCO_EAO_TYPE::register_output( const std::vector<DCO_BASE_TYPE> &v,
                                    std::vector<DCO_TYPE>      &x );
```

— Registers elements of x as outputs of external adjoint object.

```
void DCO_EAO_TYPE::register_output( std::vector<DCO_TYPE> &x );
```

— Stores s items of type T in callback object

```
void DCO_EAO_TYPE::write_data( const T *const d, size_t s );
```

— Stores every $i$th out of s items of type T in callback object.

```
void DCO_EAO_TYPE::write_data( const T *const &d, size_t i, size_t s );
```

— Creates external adjoint object via constructor `DCO_CBO_TYPE(const PARS &p)` and returns pointer to it.

```
template <typename DCO_CBO_TYPE, typename PARS>
DCO_CBO_TYPE* DCO_TAPE_TYPE::create_callback_object( const PARS &p );
```

— Returns chunk size in byte (for adjoint modes only).

```
size_t dco::tape_options::chunk_size_in_byte() const;
```

— Returns blob size in byte (for adjoint modes only).

```
size_t dco::tape_options::blob_size_in_byte() const;
```

— Returns the pointer to the tape x has been registered in.

```
DCO_TAPE_TYPE* tape(const DCO_TYPE &x);
```

— Returns tape index of x

```
size_t tape_index(const DCO_TYPE &x);
```

— Returns the logging level.

```
dco::log_level_enum& dco::logger::level();
```

— Returns the stream the logger writes to.

```
std::ostringstream& dco::logger::stream();
```

# Appendix B

# Case Study: Diffusion

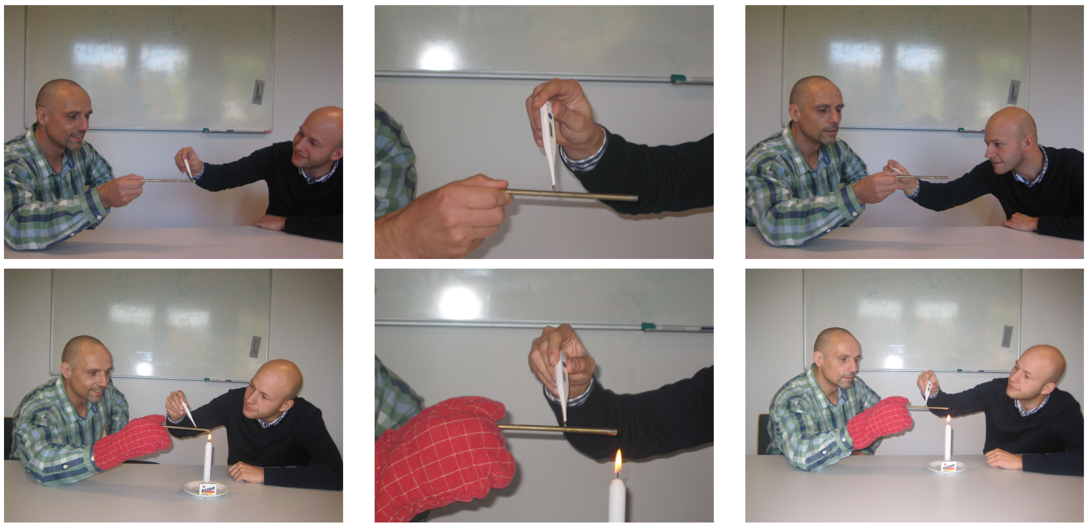

Figure B.1: Illustration of the Real-World Problem

## B.1 Purpose

We consider the estimation of the *thermal diffusivity* $c \in \mathbb{R}$ of a given very thin stick. Thermal diffusivity describes the speed at which heat "spreads" within the material. High thermal diffusivity implies quick heat conduction. Our mathematical model depends on the unknown/uncertain parameter $c$, which is to be calibrated using experimentally obtained real-world measurements. The results of a numerical simulation of the temperature distribution within the stick after heating one of its two ends for some time are compared with given measurements. Refer to Figure B.1 for a graphical illustration of the problem. The pictures in the upper row show the measuring of the initial temperature distribution in the stick. This process is continued while heating one end of the stick as shown in the pictures in the lower row and where the temperature at the other end is kept constant.

### B.1.1 Problem Description

The distribution of heat within the stick over time is modelled by a parabolic partial differential equation (PDE). We look for $T = T(t, x, c) : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ as the solution of an initial and boundary value problem for the one-dimensional heat equation

$$\frac{\partial T}{\partial t} = c \cdot \frac{\partial^2 T}{\partial x^2} \tag{B.1}$$

over the bounded domain (interval) $\Omega = (0, 1)$ with initial values $T(t = 0) = i(x)$ for $x \in \Omega$ and boundary values $T(x = 0) = b_0$ and $T(x = 1) = b_1$. Time is denoted by $t \in \mathbb{R}$. The position with the stick is represented by $x \in \mathbb{R}$. Figure B.9 shows the development of the temperature distribution within the stick for $T(t = 0) = T(x = 0) = 300K$, $T(x = 1) = 1700K$,[1] and $c = 0.01$. While the effect of the flame on the temperature of various sections of the stick is not dramatic at the final time $t = 1$ a longer lasting exposure will eventually yield an increase of the temperature in the neighbourhood of the left end of the stick beyond the comfort level. Linearity of temperature $T$ as a function of the position $x$ within the stick lets the plot of the temperature distribution converge for $t \to \infty$ to a straight line that connects the two points (0,300) and (1,1700). The dependence on $c$ vanishes asymptotically. Hence, we consider the calibration of the parameter $c$ at time $t = 1$.

For the given value of $c$ and given observations $O(x)$ at time $t = 1$ we aim to solve the unconstrained least squares problem

$$\min_{c\in\mathbb{R}} \int_\Omega (T(1, x, c) - O(x))^2 \, dx. \tag{B.2}$$

The observations are obtained through the measurement procedure illustrated in the lower row of pictures in Figure B.1. We aim to estimate the unknown/uncertain material property, $c$, of the stick such that the real-world measurements are reproduced as closely as possible by the implementation of the mathematical model.

### B.1.2 Numerical Method

The continuous mathematical model needs to be translated into a discrete formulation in order to be able to solve it on today's computers. We use *finite difference quotients* to replace both the spatial (see Sec. B.1.2) and the temporal (see Sec. B.1.2) differential terms in Equation (B.1). The integral in Equation (B.2) is discretized using a simple *quadrature rule* (see Sec. B.1.2). See, for example, [5] for a gentle introduction to finite difference discretization methods.

#### Spatial Discretization

The given stick of unit length $[0, 1]$ is divided into $n - 1$ sub-intervals of equal length

$$\Delta x = \frac{1}{n - 1} \tag{B.3}$$

yielding a spatial discretization with $n - 2$ inner points $x_1, \ldots, x_{n-2}$ in addition to the two end (boundary) points $x_0 = 0$ and $x_{n-1} = 0$. See Figure B.2 for illustration.

Second spatial derivatives are approximated by second-order finite differences based on central finite difference approximation of the first derivatives at the centre points

$$a_j \equiv \frac{x_{j-1} + x_j}{2} \tag{B.4}$$

[1]approximate temperature of a candle flame

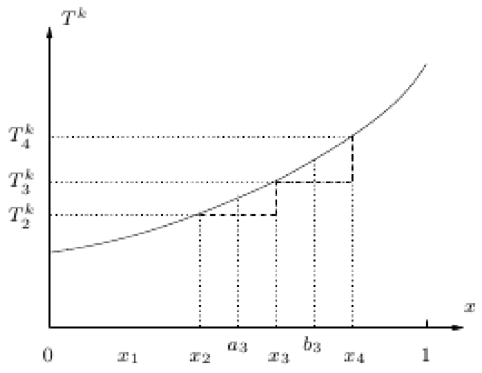

Figure B.2: Spatial Discretization: Central Finite Differences

and

$$b_j \equiv \frac{x_j + x_{j+1}}{2} \tag{B.5}$$

of the corresponding intervals. Figure B.2 illustrates the approximation of $\frac{\partial^2 T^k}{\partial x^2}$ at grid point $x_3$ for $n = 6$ and, hence, $\Delta x = 0.2$. The first derivatives of $T^k$ at $a_3$ and $b_3$ are approximated by backward finite differences. From

$$\frac{\partial T}{\partial x}(t, a_j, c) \approx \frac{T(t, x_j, c) - T(t, x_{j-1}, c)}{\Delta x} = \frac{T_j - T_{j-1}}{\Delta x} \tag{B.6}$$

for $j = 1, \ldots, n-2$ and

$$\frac{\partial T}{\partial x}(t, b_j, c) \approx \frac{T(t, x_{j+1}, c) - T(t, x_j, c)}{\Delta x} = \frac{T_{j+1} - T_j}{\Delta x} \tag{B.7}$$

for $j = 1, \ldots, n-2$ follows

$$\frac{\partial^2 T}{\partial x^2}(t, x_j, c) \approx \frac{\frac{\partial T}{\partial x}, c(t, b_j) - \frac{\partial T}{\partial x}(t, a_j, c)}{\Delta x} = \frac{T_{j+1} - 2 \cdot T_j + T_{j-1}}{(\Delta x)^2} \tag{B.8}$$

for $j = 1, \ldots, n-2$ and hence the system of ordinary differential equations (ODE)

$$\frac{\partial T_j}{\partial t} = r_j(c, \Delta x, T) \ , \quad j = 0, \ldots, n-1 \ , \tag{B.9}$$

where

$$r_j = 0 \qquad j \in \{0, n-1\} \tag{B.10}$$

$$r_j = \frac{c}{(\Delta x)^2} \cdot (T_{j+1} - 2 \cdot T_j + T_{j-1}) \qquad j \in \{1, \ldots, n-2\}, \tag{B.11}$$

and where $r$ denotes the right-hand side (also: residual) of the ODE resulting from the discretization of the second spatial derivative in Equation (B.1). The residual vanishes at both end points due to constant Dirichlet-type boundary conditions.

The ODE in Equation (B.9) is linear in $T$. Hence, it can be expanded into a first-order Taylor series at point $T \equiv 0$ ($T_j = 0$ for $j = 1, \ldots, n-2$) as follows:

$$\frac{\partial T}{\partial t} = \underbrace{r(c, \Delta x, 0)}_{=0} + \frac{\partial r}{\partial T}(c, \Delta x) \cdot (T - 0). \tag{B.12}$$

The residual at $T \equiv 0$ vanishes identically as a result of Equations (B.10) and (B.11). Linearity of the residual in $T$ implies the independence of its first derivative from $T$, that is, the Jacobian $\frac{\partial r}{\partial T} = \frac{\partial r}{\partial T}(c, \Delta x)$ is constant with respect to temperature (and time). For given $c$ and $\Delta x$ a directional derivative in direction $T = (T_j)_{j=0,\ldots,n-1}$ is computed by the second term.

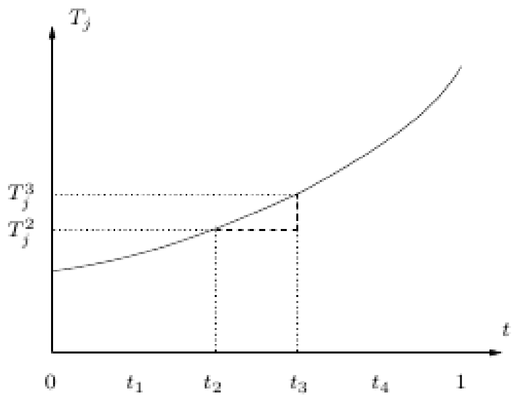

Figure B.3: Temporal Discretization: Backward Finite Difference

**Example** For $n = 6$ the Jacobian of the residual becomes

$$\frac{\partial r}{\partial T}(c, \Delta x) = \frac{c}{(\Delta x)^2} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}. \tag{B.13}$$

The first and the last rows are identically equal to zero due to the constant Dirichlet boundary conditions (see Equation (B.10)).

The residual $r(c, \Delta x, T)$ is evaluated by the function

```
template <typename TYPE>
inline void residual(const vector<TYPE>& c, vector<TYPE>& T) {
  size_t n=T.size();
  vector<TYPE> T_new(n);
  for (size_t i=1;i<n-1;++i)
    T_new[i]=c[i]*(n-1)*(n-1)*(T[i-1]-2*T[i]+T[i+1]);
  T[0]=0;
  for (size_t i=0;i<n;++i) T[i]=T_new[i];
  T[n-1]=0;
}
```

and using Equation (B.3).

## Temporal Discretization

The differential left-hand side of the ODE in Equation (B.9) is approximated by a backward finite difference approximation yielding the *backward Euler method* (see, for example, [5]). Similar to Sec. B.1.2, the unit time interval $[0, 1]$ is decomposed into $m$ sub-intervals of equal length

$$\Delta t = \frac{1}{m} \tag{B.14}$$

yielding a temporal discretization with $m$ states $T_j^1, \ldots, T_j^m$, and a given initial state $T_j^0$ for $j = 0, \ldots, n-1$. Figure B.3 illustrates the approximation of $\frac{\partial T_j}{\partial t}$ at time $t_3$ for $m = 5$ and, hence, $\Delta t = 0.2$. From

$$\frac{\partial T_j}{\partial t}(t_{k+1}, x, c) \approx \frac{T_j^{k+1} - T_j^k}{\Delta t}, \tag{B.15}$$

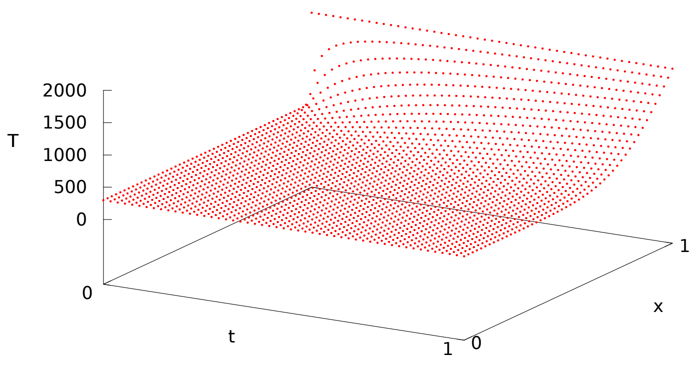

Figure B.4: Discrete Simulation of Temperature $T$ as a Function of Time $t$ and Position $x$.

follows

$$\frac{T^{k+1}-T^k}{\Delta t} = r(c,\Delta x,T^{k+1}) = \frac{\partial r}{\partial T}(c,\Delta x)\cdot T^{k+1} \tag{B.16}$$

yielding the recurrence

$$\left(\Delta t\cdot \frac{\partial r}{\partial T}(c,\Delta x) - I\right)\cdot T^{k+1} = -T^k \tag{B.17}$$

with a constant system matrix on the left-hand side for given $c$, $\Delta x$, and $\Delta t$ and where $I$ denotes the identity in $\mathbb{R}^n$.

Methods for computing the Jacobian $\frac{\partial r}{\partial T}(c,\Delta x)$ of the residual by the function

```
template <typename TYPE>
inline void residual_jacobian(const vector<TYPE>& c, vector<TYPE>& J);
```

are discussed in Sec. B.1.3. Moreover, we require an $LU$ decomposition

```
template <class TYPE>
inline void LUDecomp(vector<TYPE>& A);
```

and a linear solver for a given $LU$ decomposition of `A`

```
template <class TYPE>
inline void Solve(const vector<TYPE>& LU, vector<TYPE>& b);
```

for the solution of the system of linear equations

$$\underbrace{(\Delta t\cdot \frac{\partial r}{\partial T}(c,\Delta x) - I)}_{=:\mathtt{A}}\cdot T^{k+1} = \underbrace{-T^k}_{:=\mathtt{b}} \tag{B.18}$$

to be called during the simulation of the heat distribution by the following function:

```
template <typename TYPE>
inline void sim(const vector<TYPE>& c, const size_t m, vector<TYPE>& T) {
  size_t n=T.size();
  vector<TYPE> A(n*n,0);
  residual_jacobian(c,T,A);
  for (size_t i=0;i<n;++i) {
```

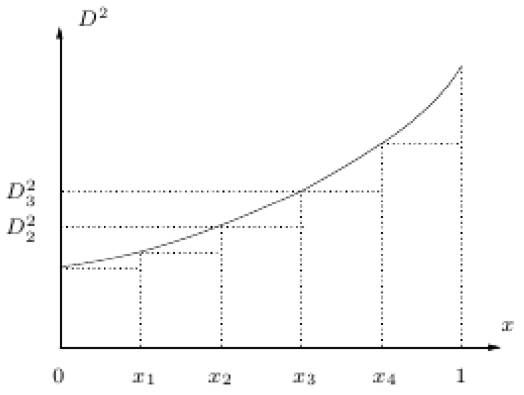

Figure B.5: Discretization of Objective: Quadrature

```
    for (size_t j=0;j<n;++j)
      A[i*n+j]=A[i*n+j]/m;
    A[i+i*n]=A[i+i*n]-1;
  }
  LUDecomp(A);
  for (size_t j=0;j<m;++j) {
    for (size_t i=0;i<n;++i)
      T[i]=-T[i];
    Solve(A,T);
  }
}
```

See Figure B.4 for a spherical plot of the simulated function $T = T(x, t)$.

### Discretization of the Objective and Optimization

Discretization of the least squares objective in Equation (B.2) using a simple quadrature rule over the given spatial discretization yields the objective function

$$y = f(c, n, m, T, O) = \frac{1}{n-1} \cdot \sum_{j=0}^{n-2} (T_j^m(c) - O_j)^2. \tag{B.19}$$

Figure B.5 illustrates the quadrature formula. For example, the contribution due to the space interval $[x_2, x_3]$ is equal to

$$\Delta x \cdot D_2^2 = \frac{D_2^2}{n-1}$$

where $D_j = T_j^m(c) - O_j$.

The evaluation of the objective as a function of the initial temperature distribution includes the simulation of the temperature distribution at the target time. It can be implemented as follows:

```
template <typename TYPE, typename OTYPE>
inline void f(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
    vector<OTYPE>& O, TYPE& v) {
  size_t n=T.size();
  sim(c,m,T);
  v=0;
  for (size_t i=0;i<n-1;++i)
```

```
    v=v+(T[i]-O[i])*(T[i]-O[i]);
  v=v/(n-1);
}
```

The objective function $f$ is optimized by a simple steepest gradient descent method with embedded local line search. Termination is defined by $\|\nabla f\| \leq \epsilon$ for $\epsilon \ll 1$ and it is based on the necessary first-order optimality condition $\nabla f = 0$. At each iteration $i$ we take a step into negative gradient direction the size ($\alpha$) of which is defined by recursive bisection such that a decrease in the value of the objective is ensured. This very basic approach turns out to be sufficient for the given simple problem. Formally, the steepest gradient descent method is described by the iteration

$$c_{i+1} = c_i - \alpha \cdot \nabla f(c_i) \quad \text{while } \|\nabla f\| > \epsilon.$$

### B.1.3 Algorithmic Differentiation

Figure B.6 visualizes the structure of the given simulation. Local partial derivatives to be combined in forward or reverse order in tangent or adjoint mode AD, respectively, are given in square brackets as edge labels. Linearity of the residual in $T$ yields the independence of its Jacobian of $T$ (node 6). The corresponding matrix $A = \Delta t \cdot \frac{\partial r}{\partial T}(c, \Delta x) - I$ (nodes 6,7,8) is decomposed into lower and upper triangular factors $L$ and $U$ (node 9) that enter the subsequent backward Euler steps (nodes 10,11 and 12,13). The first two are shown in Figure B.6. For given right-hand sides ($-T_0$ and $-T_1$) the next step is computed by simple forward and backward substitution based on the given $LU$ decomposition of $A$.

#### Tangent Residual

The tangent residual returns the product of the Jacobian of the residual with a given vector $T^{(1)} \hat{=}$ `t1_T`$\in \mathbb{R}^n$ without prior accumulation of the Jacobian itself. It computes a *matrix-free* directional derivative.

#### Jacobian of the Residual

The Jacobian $\frac{\partial r}{\partial T} \hat{=}$ `J=J[:][:]` $\in \mathbb{R}^{n \times n}$ of the residual is computed column-wise by letting the input vector $T^{(1)} \hat{=}$ `t1_T` $\in \mathbb{R}^n$ range over the Cartesian basis vectors in $\mathbb{R}^n$.

```
template <typename TYPE>
inline void residual_jacobian(const vector<TYPE>& c, vector<TYPE>& J) {
  size_t n=c.size();
  vector<TYPE> t1_T(n);
  for (size_t i=0;i<n;++i) t1_T[i]=0;
  for (size_t i=0;i<n;++i) {
    t1_T[i]=1;
    residual(c,t1_T);
    for (size_t j=0;j<n;++j) {
      J[j*n+i]=t1_T[j];
      t1_T[j]=0;
    }
  }
}
```

Linearity of the residual enables the use of the primal function `residual(...)` for the computation of the directional derivative. Sparsity of the (in this case tridiagonal) Jacobian can and should be exploited. Refer to [4] or [9] for details on corresponding compression techniques. Associated graph colouring problems are discussed in [2].

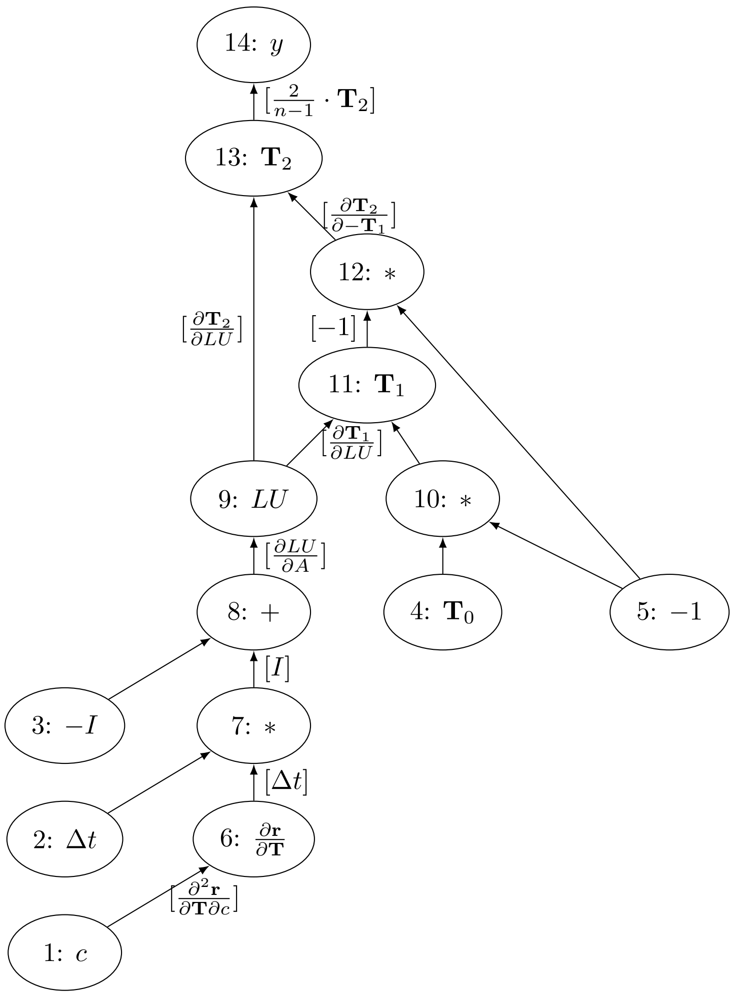
14: $y$
$[\frac{2}{n-1}\cdot\mathbf{T}_2]$
13: $\mathbf{T}_2$
$[\frac{\partial\mathbf{T}_2}{\partial-\mathbf{T}_1}]$
12: $*$
$[\frac{\partial\mathbf{T}_2}{\partial LU}]$
$[-1]$
11: $\mathbf{T}_1$
$[\frac{\partial\mathbf{T}_1}{\partial LU}]$
9: $LU$
10: $*$
$[\frac{\partial LU}{\partial A}]$
8: $+$
4: $\mathbf{T}_0$
5: $-1$
$[I]$
3: $-I$
7: $*$
$[\Delta t]$
2: $\Delta t$
6: $\frac{\partial\mathbf{r}}{\partial\mathbf{T}}$
$[\frac{\partial^2\mathbf{r}}{\partial\mathbf{T}\partial c}]$
1: $c$

Figure B.6: Linearized DAG of Simulation Method

### Gradient of the Objective

The gradient of the objective with respect to the free parameters that enter the simulation ($c \in \mathbb{R}$ in our case; see Sec. B.1.4 for comments on the non-scalar case $c = c(x)$) is required by the steepest descent algorithm. It can be obtained by overloading the entire simulation and the following evaluation of the objective for either a tangent or an adjoint AD type. Adjoint mode should be preferred for large numbers of parameters where the actual number depends on the performance of the AD implementation provided by the given AD tool.

AD of `f` implies AD of the function `sim` and thus the differentiation of the direct linear solver called during the simulation. While naive application of AD to the direct linear solvers produces correct results, its computational cost, which is of order $O(n^3)$, can be reduced to $O(n^2)$ through the exploitation of further mathematical insight as shown in [3].

AD by overloading of `f` in tangent mode implies the evaluation of a tangent model of the Jacobian computation (see Sec. B.1.3). Implicitly, second-order derivatives are computed. The computation of

$$A = \Delta t \cdot \frac{\partial r}{\partial T}(c, \Delta x) - I$$

yields the tangent projection

$$A^{(2)} = \langle \frac{\partial A}{\partial c}, c^{(2)} \rangle + \langle \frac{\partial A}{\partial T}, T^{(2)} \rangle = \Delta t \cdot \langle \frac{\partial^2 r}{\partial T \partial c}(c, \Delta x), c^{(2)} \rangle,$$

where $\frac{\partial^2 r}{\partial T^2} = 0$ due to the linearity of $r$ in $T$.

Similarly, overloading in adjoint mode yields a second-order adjoint projection of the Hessian of the residual as

$$\begin{aligned}
\begin{pmatrix} T_{(2)} \\ c_{(2)} \end{pmatrix} &= \langle A_{(2)}, \frac{\partial A}{\partial (T, c)} \rangle && \text{(B.20)} \\
&= \Delta t \cdot \langle A_{(2)}, \frac{\partial^2 r}{\partial T \partial (T, c)}(c, \Delta x) \rangle && \text{(B.21)} \\
&= \begin{pmatrix} 0 \\ \langle A_{(2)}, \frac{\partial^2 r}{\partial T \partial c}(c, \Delta x) \rangle \end{pmatrix} . && \text{(B.22)}
\end{aligned}$$

Linearity of $r$ in $T$ yields $T_{(2)} = 0 \in \mathbb{R}^n$.

## B.1.4 Optimization

Figure B.7 illustrates the convergence behaviour of the steepest descent algorithm. Both the value of the objective and the norm of the gradient (first-order local optimality condition) decrease monotonously with the growing number of steepest descent steps.

For further motivation consider Equation (B.1) with spatially varying thermal diffusivity $c(x)$ (after discretization $c_i$). The gradient of the objective with respect to the discrete $c = (c_i)_{i=0,\ldots,n-1}$ becomes a vector of size $n$. In tangent mode it is computed as $n$ inner products with the Cartesian basis vectors in $\mathbb{R}^n$. A single run of the adjoint code performs the same task at considerably lower computational cost for $n \gg 1$. Figure B.8 shows the relative computational cost

$$R = \frac{\text{runtime for gradient computation}}{\text{run time for function evaluation}}$$

of the gradient computation in tangent versus adjoint mode. The relative cost of gradient computation in tangent mode grows linearly with $n$ while in adjoint mode it remains constant. Availability of an adjoint simulation may turn out to be crucial for the applicability of gradient-based methods to large-scale problems in Computational Science, Engineering, and Finance.

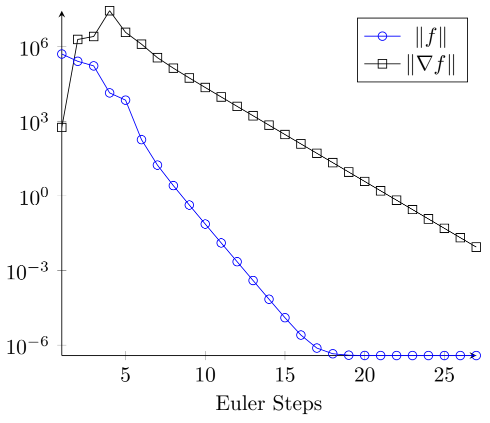

Figure B.7: Convergence Behaviour for $n = 160$.

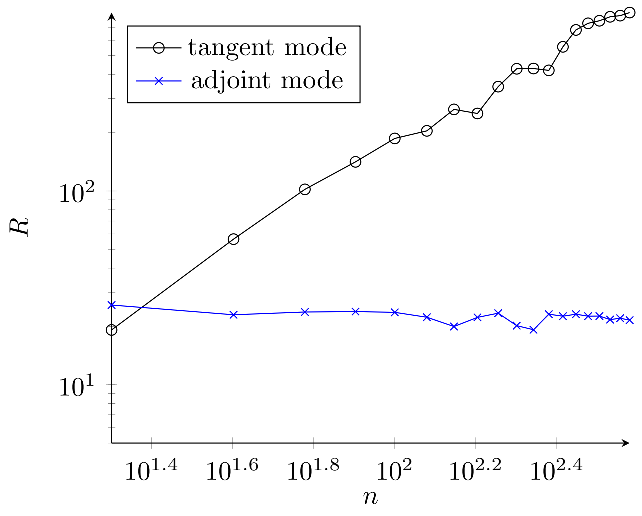

Figure B.8: Relative Run Time $R$ for Growing Problem Sizes $n$.

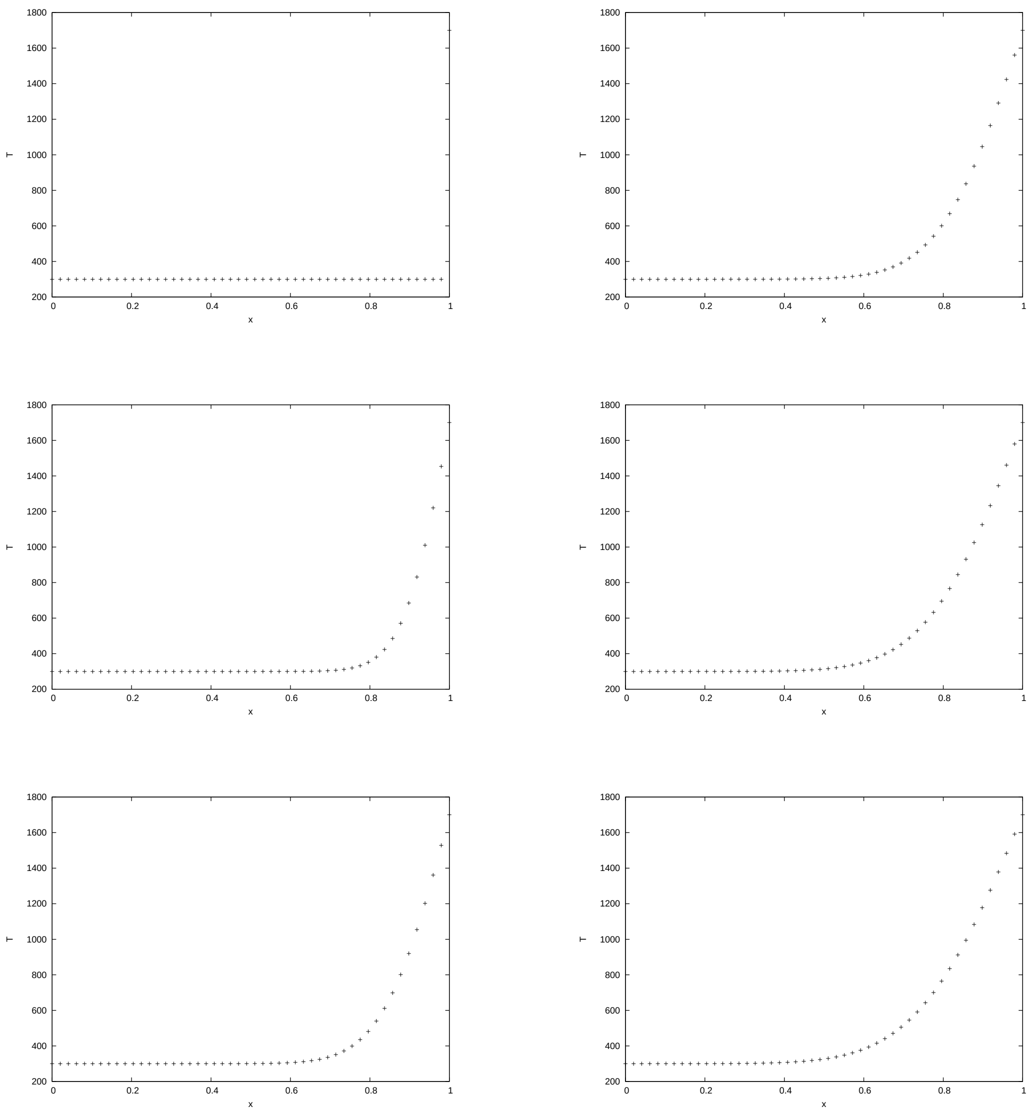

Figure B.9: Simulated Temperature Distribution for $t = 0, 0.2, 0.4$ (Left Column) and $t = 0.6, 0.8, 1$ (Right Column)

### B.1.5 Code

This section shows the source code for the given implementation of the diffusion problem.

```
#pragma once
#include "dco.hpp"

//** required to be compatible with C -- only use two-digit exponent
//** in output of scientific numbers.
#if defined(WIN32) && _MSC_VER < 1900
struct outputformatter {
  outputformatter() {
    _set_output_format(_TWO_DIGIT_EXPONENT);
  }
};
static outputformatter of;
#endif

template <typename cTYPE, typename TTYPE>
  inline void residual(const std::vector<cTYPE>& c, std::vector<TTYPE>& T) {
  size_t n=T.size();
  std::vector<TTYPE> T_new(n);
  for (size_t i=1;i<n-1;++i)
    T_new[i]=c[i]*static_cast<double>(n-1)*static_cast<double>(n-1)*(T[i-1]-2*T[
        i]+T[i+1]);
  T[0]=0;
  for (size_t i=0;i<n;++i) T[i]=T_new[i];
  T[n-1]=0;
}

/*
template <typename TYPE>
inline void residual_jacobian(const vector<TYPE>& c, vector<TYPE>& J) {
  size_t n=c.size();
  vector<TYPE> t1_T(n);
  for (size_t i=0;i<n;++i) t1_T[i]=0;
  for (size_t i=0;i<n;++i) {
    t1_T[i]=1;
    residual(c,t1_T);
    for (size_t j=0;j<n;++j) {
      J[j*n+i]=t1_T[j];
      t1_T[j]=0;
    }
  }
}

template <typename TYPE>
inline void residual_jacobian(const vector<TYPE>& c, const vector<TYPE>& T,
    vector<TYPE>& J) {
  typedef typename dco::gt1s<TYPE>::type DCO_T1S_TYPE;
  size_t n=T.size();
  vector<DCO_T1S_TYPE> t1_T(n);
  for (size_t i=0;i<n;i++) t1_T[i]=T[i];
  for (size_t i=0;i<n;i++) {
    derivative(t1_T[i])=1;
    residual(c,t1_T);
    for (size_t j=0;j<n;++j) {
      J[j*n+i]=derivative(t1_T[j]);
      derivative(t1_T[j])=0;
    }
  }
}
*/

template <typename TYPE>
inline void residual_jacobian(const std::vector<TYPE>& c, const std::vector<TYPE
    >& T, std::vector<TYPE>& J) {
  typedef typename dco::gt1s<TYPE>::type DCO_T1S_TYPE;
  size_t n=T.size();
  std::vector<DCO_T1S_TYPE> t1T(n);
  const size_t bw=3;
  for (size_t i=0;i<n;i++) t1T[i]=T[i];
  for (size_t i=0;i<bw;i++) {
    for (size_t j=i;j<n;j+=bw) derivative(t1T[j])=1;
    residual(c,t1T);
    for (size_t j=i;j<n;j+=bw) {
      for (size_t k=(j<(bw-1)/2)? 0 : j-(bw-1)/2;k<= std::min(n-1,j+(bw-1)/2);k
          ++)
        J[k*n+j]=derivative(t1T[k]);
      derivative(t1T[j])=0;
    }
    for (size_t j=0;j<n;j++) derivative(t1T[j])=0;
  }
}

template <class TYPE>
inline void LUDecomp(std::vector<TYPE>& A) {
  size_t n = static_cast<size_t>(sqrt(double(A.size())));
  for (size_t k=0;k<n;k++) {
    for (size_t i=k+1;i<n;i++)
      A[i*n+k]=A[i*n+k]/A[k*n+k];
    for (size_t j=k+1;j<n;j++)
      for (size_t i=k+1;i<n;i++)
        A[i*n+j]=A[i*n+j]-A[i*n+k]*A[k*n+j];
  }
}

// L*y=b
template <class TYPE>
inline void FSubst(const std::vector<TYPE>& LU, std::vector<TYPE>& b) {
  size_t n=b.size();
  for (size_t i=0;i<n;i++)
    for (size_t j=0;j<i;j++)
      b[i]=b[i]-LU[i*n+j]*b[j];
}

// U*x=y
template <class TYPE>
```

```
inline void BSubst(const std::vector<TYPE>& LU, std::vector<TYPE>& y) {
  size_t n=y.size();
  for (size_t k=n,i=n-1;k>0;k--,i--) {
    for (size_t j=n-1;j>i;j--)
      y[i]=y[i]-LU[i*n+j]*y[j];
    y[i]=y[i]/LU[i*n+i];
  }
}

template <class TYPE>
inline void Solve(const std::vector<TYPE>& LU, std::vector<TYPE>& b) {
  FSubst(LU,b);
  BSubst(LU,b);
}

template <typename TYPE>
inline void sim(const std::vector<TYPE>& c, const size_t m, std::vector<TYPE>& T
    ) {
  size_t n=T.size();
  std::vector<TYPE> A(n*n,0);

  residual_jacobian(c,T,A);

  for (size_t i=0;i<n;++i) {
    for (size_t j=0;j<n;++j)
      A[i*n+j]=A[i*n+j]/static_cast<double>(m);
    A[i+i*n]=A[i+i*n]-1;
  }

  LUDecomp(A);

  for (size_t j=0;j<m;++j) {
    for (size_t i=0;i<n;++i)
      T[i]=-T[i];
    Solve(A,T);
  }
}

template <typename TYPE, typename OTYPE>
inline void f(const std::vector<TYPE>& c, const size_t m, std::vector<TYPE>& T,
    const std::vector<OTYPE>& O, TYPE& v) {
  size_t n=T.size();
  sim(c,m,T);

  v=0;
  for (size_t i=0;i<n-1;++i)
    v=v+(T[i]-O[i])*(T[i]-O[i]);
  v=v/(static_cast<double>(n)-1);
}
```

## B.2 Function

This section shows the evaluation of the objective.

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>

//#include "../include/f_residual_Jacobian_by_hand.hpp"
//#include "../include/f_residual_Jacobian_by_dco_dense.hpp"
#include "../include/f.hpp"

using namespace std;

int main(int argc, char* argv[]){
  cout.precision(5);
  if (argc!=3) {
    cerr << "2 parameters expected:" << endl
   << "  1. number of spatial finite difference grid points" << endl
   << "  2. number of implicit Euler steps" << endl;
    return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  double v;
  f(c, m, T, O, v);
  cout << "v=" << v << endl;
}
```

## B.3 Observations

This section shows the generation of the "observations."

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>
#include "../include/f.hpp"
using namespace std;
#include "../include/mrg32k3a.h"

int main(int argc, char* argv[]){
  cout.precision(5);
  if (argc!=3) {
    cerr << "2 parameters expected:" << endl
```

```cpp
     << "  1. number of spatial finite difference grid points" << endl
     << "  2. number of implicit Euler steps" << endl;
      return EXIT_FAILURE;
    }
    size_t  n=atoi(argv[1]), m=atoi(argv[2]);
    vector<double> c(n);
    for (size_t i=0;i<n;i++) c[i]=0.01;
    vector<double> T(n);
    for (size_t i=0;i<n-1;i++) T[i]=300.;
    T[n-1]=1700.;
    vector<double> O(n,0);
    double v;
    ofstream ofs("O_generated.txt");
    f(c,m,T,O,v);

    static std::vector<double> Z(n), rngseed(6);
    for(int i=0; i<6; i++) rngseed[i] = i+1;
    randNormal(n, rngseed.data(), Z);

    for (size_t i=0;i<n;i++) ofs << T[i] + Z[i] << endl;
  }
```

## B.4 Gradient

### B.4.1 Finite Differences

This section shows the approximation of the gradient by finite differences.

```cpp
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>
#include <cfloat>

#include "../include/f.hpp"
using namespace std;

int main(int argc, char* argv[]){
  cout.precision(5);
  if (argc<3) {
    cerr << "2 parameters expected:" << endl
     << "  1. number of spatial finite difference grid points" << endl
     << "  2. number of implicit Euler steps" << endl
     << "  3. if third parameter given, print output" << endl;
      return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  vector<double> dvdc(n,0),cph(n,0),cmh(n,0);
  double vmh;
  double vph;
  for(size_t j=0;j<n;j++) {
    double h=(c[j]==0) ? sqrt(DBL_EPSILON) : sqrt(DBL_EPSILON)*std::abs(c[j]);
    for (size_t i=0;i<n;i++) cmh[i]=c[i];
    for (size_t i=0;i<n-1;i++) T[i]=300.;
    T[n-1]=1700.;
    cmh[j]-=h;
    f(cmh,m,T,O,vmh);
    for (size_t i=0;i<n;i++) cph[i]=c[i];
    for (size_t i=0;i<n-1;i++) T[i]=300.;
    T[n-1]=1700.;
    cph[j]+=h;
    f(cph,m,T,O,vph);
    dvdc[j]=(vph-vmh)/(2*h);
  }
  cout.precision(5);
  if (argc == 4) {
    for(size_t i=0;i<n;i++)
      cout << "dvdc[" << i << "]=" << dvdc[i] << endl;
  } else {
    cout << "Finite differences too unstable to give reproducible numbers on
        various platforms." << endl
         << "Run with third parameter to print derivative results." << endl;
  }
}
```

### B.4.2 gt1s< double >

This section shows the computation of the gradient in tangent mode.

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include "dco.hpp"
using namespace std;
using namespace dco;

typedef gt1s<double>::type DCO_TYPE;

#include "../include/f.hpp"

int main(int argc, char* argv[]){
  if (argc!=3) {
    cerr << "2 parameters expected:" << endl
   << "  1. number of spatial finite difference grid points" << endl
   << "  2. number of implicit Euler steps" << endl;
    return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  vector<double> dvdc(n);
  vector<DCO_TYPE> ca(n);
  vector<DCO_TYPE> Ta(n);
  DCO_TYPE va;
  for(size_t i=0;i<n;i++) {
    for(size_t j=0;j<n;j++) { ca[j]=c[j]; Ta[j]=T[j]; }
    derivative(ca[i])=1.0;
    f(ca,m,Ta,O,va);
    dvdc[i]=derivative(va);
  }
  cout.precision(5);
  for(size_t i=0;i<n;i++)
    cout << "dvdc[" << i << "]=" << dvdc[i] << endl;
}
```

### B.4.3 ga1s< double >

This section shows the computation of the gradient in adjoint mode.

```
#include <cstdlib>
#include <iostream>
#include "dco.hpp"
using namespace std;
using namespace dco;
typedef ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "../include/f.hpp"

int main(int argc, char* argv[]){
  if (argc!=3) {
    cerr << "2 parameters expected:" << endl
   << "  1. number of spatial finite difference grid points" << endl
   << "  2. number of implicit Euler steps" << endl;
    return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  vector<double> dvdc(n);

  vector<DCO_TYPE> ca(n);
  vector<DCO_TYPE> Ta(n);
  DCO_TYPE va;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    tape->register_variable(ca[i]);
    tape->register_variable(Ta[i]);
  }
  f(ca,m,Ta,O,va);
  cerr << "record (0," << m << ")=" << dco::size_of(tape) << "B" << endl;
  derivative(va)=1.0;
  tape->register_output_variable(va);
  tape->interpret_adjoint();
  for (size_t i=0;i<n;i++) dvdc[i]=derivative(ca[i]);
  cout.precision(5);
  for(size_t i=0;i<n;i++)
    cout << "dvdc[" << i << "]=" << dvdc[i] << endl;
}
```

### B.4.4 ga1s< double > + Equidistant Checkpointing

This section shows the computation of the gradient in adjoint mode with equidistant checkpointing.

```
#include <cstdlib>
#include <iostream>
#include <cassert>
#include "dco.hpp"
using namespace std;
using namespace dco;
typedef ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "../include/f.hpp"

template<typename TYPE>
void EulerSteps(const size_t m, const vector<TYPE>& A, vector<TYPE>& T){
  size_t n=T.size();
  for (size_t j=0;j<m;j++) {
    for (size_t i=0;i<n;++i) T[i]=-T[i];
    Solve(A,T);
  }
}

template<typename DCO_TYPE>
void EulerSteps_make_gap(const size_t m, const vector<DCO_TYPE>& A, vector<
    DCO_TYPE>& T){
```

```
  size_t n=T.size();

  auto T_in(T);

  // forward run
  dco::tape(A)->switch_to_passive();
  EulerSteps(m,A,T);
  dco::tape(A)->switch_to_active();

  // register output
  for (size_t i=0;i<n;i++) dco::tape(A)->register_variable(T[i]);

  auto fill_gap = [=,&A]() {
    auto p = dco::tape(A)->get_position();
    vector<DCO_TYPE> T1(T_in);

    //forward run
    EulerSteps(m,A,T1);
    cerr << "record =" << dco::size_of(dco::tape(A)) << "B" << endl;

    //get output adjoints (seeds for this section)
    for (size_t i=0;i<n;i++) {
      dco::tape(A)->register_output_variable(T1[i]);
      dco::derivative(T1[i]) = dco::derivative(T[i]);
    }

    //reverse run and reset to position p (so only reverse run of Solve)
    dco::tape(A)->interpret_adjoint_and_reset_to(p);
  };

  dco::tape(A)->insert_callback(std::move(fill_gap));
}

template <typename TYPE>
inline void sim(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
    size_t cs) {
  size_t n=T.size();
  static vector<TYPE> A(n*n); // static avoids copy
  residual_jacobian(c,T,A);
  for (size_t i=0;i<n;++i) {
    for (size_t j=0;j<n;++j)
      A[i*n+j]=A[i*n+j]/m;
    A[i+i*n]=A[i+i*n]-1;
  }
  LUDecomp(A);
  for (size_t j=0;j<m;j+=cs) {
    size_t s=(j+cs<m) ? cs : m-j;
    EulerSteps_make_gap<DCO_TYPE>(s,A,T);
  }
}

template <typename TYPE, typename OTYPE>
inline void f(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
    vector<OTYPE>& O, TYPE& v, const size_t cs) {
```

```
  sim(c,m,T,cs);
  v=0;
  size_t n=T.size();
  for (size_t i=0;i<n-1;++i)
    v=v+(T[i]-O[i])*(T[i]-O[i]);
  v=v/(n-1);
}

int main(int argc, char* argv[]){
  if (argc!=4) {
    cerr << "2 parameters expected:" << endl
   << "  1. number of spatial finite difference grid points" << endl
   << "  2. number of implicit Euler steps" << endl
   << "  3. distance between checkpoints" << endl;
    return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]), cs=atoi(argv[3]);
  assert(cs<=m);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  vector<double> dvdc(n);

  vector<DCO_TYPE> ca(n);
  vector<DCO_TYPE> Ta(n);
  DCO_TYPE va;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    tape->register_variable(ca[i]);
    tape->register_variable(Ta[i]);
  }
  f(ca,m,Ta,O,va,cs);
  cerr << "record=" << dco::size_of(tape) << "B" << endl;
  derivative(va)=1.0;
  tape->register_output_variable(va);
  tape->interpret_adjoint();
  for (size_t i=0;i<n;i++) dvdc[i]=derivative(ca[i]);
  cout.precision(5);
  for(size_t i=0;i<n;i++)
    cout << "dvdc[" << i << "]=" << dvdc[i] << endl;
}
```

### B.4.5 `ga1s< double >` + Symbolic Derivative of Linear Solver

This section shows the computation of the gradient in adjoint mode with symbolically differentiated linear solver.

```
#include <cstdlib>
#include <iostream>
#include <cassert>
#include "dco.hpp"
using namespace std;
using namespace dco;
typedef ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "../include/f.hpp"

template<typename DCO_MODE, typename TYPE>
void LUDecomp_make_gap(vector<TYPE>& A){
  size_t n=static_cast<size_t>(sqrt(double(A.size())));
  vector<double> Ap(n*n);
  for(size_t i=0;i<n*n;i++) Ap[i]=value(A[i]);
  LUDecomp(Ap);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      value(A[i*n+j])=Ap[i*n+j];
}

// U^T*y=b
template <class TYPE>
inline void FSubstT(const vector<TYPE>& LU, vector<TYPE>& y) {
  size_t n=y.size();
  for (size_t i=0;i<n;i++){
    for (size_t j=0;j<i;j++)
      y[i]=y[i]-LU[j*n+i]*y[j];
    y[i]=y[i]/LU[i*n+i];
  }
}

// L^T*x=y
template <class TYPE>
inline void BSubstT(const vector<TYPE>& LU, vector<TYPE>& b) {
  size_t n=b.size();
  for (size_t k=n,i=n-1;k>0;k--,i--)
    for (size_t j=n-1;j>i;j--)
      b[i]=b[i]-LU[j*n+i]*b[j];
}

// LU^T*x=y
template <class TYPE>
inline void SolveT(const vector<TYPE>& LU, vector<TYPE>& b) {
  FSubstT(LU,b);
  BSubstT(LU,b);
}

template<typename DCO_MODE, typename TYPE>
void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b){
  const size_t n = b.size();
  auto rhs(b);
  dco::tape(LU)->switch_to_passive();
  Solve(LU,b);
  dco::tape(LU)->switch_to_active();
  dco::tape(LU)->register_variable(b);

  auto fill_gap = [=,&LU]() {
    vector<DCO_TYPE> a1b(n);
    for (size_t i = 0; i < n; ++i) a1b[i] = dco::derivative(b[i]);
    dco::tape(LU)->switch_to_passive();
    SolveT(LU,a1b);

    for (size_t i=0;i<n;i++)
      for (size_t j=0;j<n;j++)
        dco::derivative(LU[i*n+j]) += dco::value(-a1b[i]*b[j]);
    dco::derivative(rhs) += dco::value(a1b);
    dco::tape(LU)->switch_to_active();
  };

  dco::tape(LU)->insert_callback(std::move(fill_gap));
}

template <>
inline void sim(const vector<DCO_TYPE>& c, const size_t m, vector<DCO_TYPE>& T)
    {
  const size_t n=c.size();
  static vector<DCO_TYPE> A(n*n);
  residual_jacobian(c,T,A);
  for (size_t i=0;i<n;++i) {
    for (size_t j=0;j<n;++j)
      A[i*n+j]=A[i*n+j] / static_cast<double>(m);
    A[i+i*n]=A[i+i*n]-1;
  }
  LUDecomp_make_gap<DCO_MODE>(A);
  for (size_t j=0;j<m;++j) {
    for (size_t i=0;i<n;++i) T[i]=-T[i];
    Solve_make_gap<DCO_MODE>(A,T);
  }
}

int main(int argc, char* argv[]){
  if (argc!=3) {
    cerr << "2 parameters expected:" << endl
   << "  1. number of spatial finite difference grid points" << endl
   << "  2. number of implicit Euler steps" << endl;
    return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  vector<double> dvdc(n);

  vector<DCO_TYPE> ca(n);
  vector<DCO_TYPE> Ta(n);
  DCO_TYPE va;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    tape->register_variable(ca[i]);
  }
  f(ca,m,Ta,O,va);
  cerr << "record (0," << m << ")=" << dco::size_of(tape) << "B" << endl;
  derivative(va)=1.0;
  tape->register_output_variable(va);
  tape->interpret_adjoint();
  tape->write_to_dot();

  for (size_t i=0;i<n;i++) dvdc[i]=derivative(ca[i]);
  cout.precision(5);
  for(size_t i=0;i<n;i++)
    cout << "dvdc[" << i << "]=" << dvdc[i] << endl;
}
```

### B.4.6 ga1s< double > + Symbolic Derivative of Linear Solver and Equidistant Checkpointing

This section shows the computation of the gradient in adjoint mode with equidistant checkpointing and symbolically differentiated linear solver.

```
#include <cstdlib>
#include <iostream>
#include <cassert>
#include "dco.hpp"
using namespace std;
using namespace dco;
typedef ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "../include/f.hpp"

template<typename DCO_MODE, typename TYPE>
void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b);

void EulerSteps(const size_t m, const vector<DCO_TYPE>& A, vector<DCO_TYPE>& T){
  size_t n=T.size();
  for (size_t j=0;j<m;j++) {
    for (size_t i=0;i<n;++i) T[i]=-T[i];
    if (dco::tape(A)->is_active())
        Solve_make_gap<DCO_MODE>(A,T);
      else
```

```
        Solve(A,T);
    }
}

template<typename DCO_TYPE>
void EulerSteps_make_gap(const size_t m, const vector<DCO_TYPE>& A, vector<
    DCO_TYPE>& T){
  size_t n=T.size();

  auto T_in(T);

  // forward run
  dco::tape(A)->switch_to_passive();
  EulerSteps(m,A,T);
  dco::tape(A)->switch_to_active();

  // register output
  for (size_t i=0;i<n;i++) dco::tape(A)->register_variable(T[i]);

  auto fill_gap = [=,&A]() {
    auto p = dco::tape(A)->get_position();
    vector<DCO_TYPE> T1(T_in);

    //forward run
    EulerSteps(m,A,T1);
    cerr << "record =" << dco::size_of(dco::tape(A)) << "B" << endl;

    //get output adjoints (seeds for this section)
    for (size_t i=0;i<n;i++) {
      dco::tape(A)->register_output_variable(T1[i]);
      dco::derivative(T1[i]) = dco::derivative(T[i]);
    }

    //reverse run and reset to position p (so only reverse run of Solve)
    dco::tape(A)->interpret_adjoint_and_reset_to(p);
  };

  dco::tape(A)->insert_callback(std::move(fill_gap));
}

template<typename DCO_MODE, typename TYPE>
void LUDecomp_make_gap(vector<TYPE>& A){
  size_t n=sqrt(double(A.size()));
  vector<double> Ap(n*n);
  for(size_t i=0;i<n*n;i++) Ap[i]=value(A[i]);
  LUDecomp(Ap);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      value(A[i*n+j])=Ap[i*n+j];
}

// U^T*y=b
template <class TYPE>
inline void FSubstT(const vector<TYPE>& LU, vector<TYPE>& y) {
  size_t n=y.size();
  for (size_t i=0;i<n;i++){
    for (size_t j=0;j<i;j++)
      y[i]=y[i]-LU[j*n+i]*y[j];
    y[i]=y[i]/LU[i*n+i];
  }
}

// L^T*x=y
template <class TYPE>
inline void BSubstT(const vector<TYPE>& LU, vector<TYPE>& b) {
  size_t n=b.size();
  for (size_t k=n,i=n-1;k>0;k--,i--)
    for (size_t j=n-1;j>i;j--)
      b[i]=b[i]-LU[j*n+i]*b[j];
}

// LU^T*x=y
template <class TYPE>
inline void SolveT(const vector<TYPE>& LU, vector<TYPE>& b) {
  FSubstT(LU,b);
  BSubstT(LU,b);
}

template<typename DCO_MODE, typename TYPE>
void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b){

  const size_t n = b.size();
  auto rhs(b);
  dco::tape(LU)->switch_to_passive();
  Solve(LU,b);
  dco::tape(LU)->switch_to_active();
  dco::tape(LU)->register_variable(b);

  auto fill_gap = [=,&LU]() {
    vector<DCO_TYPE> a1b(n);
    for (size_t i = 0; i < n; ++i) a1b[i] = dco::derivative(b[i]);
    dco::tape(LU)->switch_to_passive();
    SolveT(LU,a1b);

    for (size_t i=0;i<n;i++)
      for (size_t j=0;j<n;j++)
        dco::derivative(LU[i*n+j]) += dco::value(-a1b[i]*b[j]);
    dco::derivative(rhs) += dco::value(a1b);
    dco::tape(LU)->switch_to_active();
  };

  dco::tape(LU)->insert_callback(std::move(fill_gap));
}

inline void sim(const vector<DCO_TYPE>& c, const size_t m, vector<DCO_TYPE>& T,
    const size_t cs) {
  const size_t n=c.size();
  static vector<DCO_TYPE> A(n*n);
  residual_jacobian(c,T,A);
  for (size_t i=0;i<n;++i) {
    for (size_t j=0;j<n;++j)
      A[i*n+j]=A[i*n+j]/m;
    A[i+i*n]=A[i+i*n]-1;
  }
  LUDecomp_make_gap<DCO_MODE>(A);
  for (size_t j=0;j<m;j+=cs) {
    size_t s=(j+cs<m) ? cs : m-j;
    EulerSteps_make_gap<DCO_TYPE>(s,A,T);
  }
}

template <typename TYPE, typename OTYPE>
inline void f(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
    vector<OTYPE>& O, TYPE& v, const size_t cs) {
  sim(c,m,T,cs);
  v=0;
  size_t n=T.size();
  for (size_t i=0;i<n-1;++i)
    v=v+(T[i]-O[i])*(T[i]-O[i]);
  v=v/(n-1);
}

int main(int argc, char* argv[]){
  if (argc!=4) {
    cerr << "3 parameters expected:" << endl
   << "  1. number of spatial finite difference grid points" << endl
   << "  2. number of implicit Euler steps" << endl
   << "  3. distance between checkpoints " << endl;
    return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]), cs=atoi(argv[3]);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  vector<double> dvdc(n);

  vector<DCO_TYPE> ca(n);
  vector<DCO_TYPE> Ta(n);
  DCO_TYPE va;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    tape->register_variable(ca[i]);
  }
  f(ca,m,Ta,O,va,cs);
  cerr << "record (0," << m << ")=" << dco::size_of(tape) << "B" << endl;
  derivative(va)=1.0;
  tape->register_output_variable(va);
  tape->interpret_adjoint();
  for (size_t i=0;i<n;i++) dvdc[i]=derivative(ca[i]);
  cout.precision(5);
  for(size_t i=0;i<n;i++)
    cout << "dvdc[" << i << "]=" << dvdc[i] << endl;
}
```

## B.5 Hessian

### B.5.1 Finite Differences

This section shows the approximation of the Hessian by finite difference.

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>
#include <cfloat>
#include "../include/f.hpp"
using namespace std;


int main(int argc, char* argv[]){
  cout.precision(5);
  if (argc<3) {
    cerr << "2 parameters expected:" << endl
   << "  1. number of spatial finite difference grid points" << endl
   << "  2. number of implicit Euler steps" << endl
   << "  3. if third parameter given, print output" << endl;
    return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  vector<double> cp(n,0),Tp(n,0);
  vector<vector<double> > d2vdc2(n,vector<double>(n,0));
  double vp1, vp2;
  for (size_t i=0;i<n;i++) {
    for (size_t j=0;j<n;j++) {
      double h=(c[j]==0) ? sqrt(sqrt(DBL_EPSILON)) : sqrt(sqrt(DBL_EPSILON))*std
          ::abs(c[j]);
      for (size_t k=0;k<n;k++) { cp[k]=c[k]; Tp[k]=T[k]; }
      cp[i]+=h; cp[j]+=h;
      f(cp,m,Tp,O,vp2);
      for (size_t k=0;k<n;k++) { cp[k]=c[k]; Tp[k]=T[k]; }
      cp[i]-=h; cp[j]+=h;
      f(cp,m,Tp,0,vp1);
      vp2-=vp1;
      for (size_t k=0;k<n;k++) { cp[k]=c[k]; Tp[k]=T[k]; }
      cp[i]+=h; cp[j]-=h;
      f(cp,m,Tp,0,vp1);
      vp2-=vp1;
      for (size_t k=0;k<n;k++) { cp[k]=c[k]; Tp[k]=T[k]; }
      cp[i]-=h; cp[j]-=h;
      f(cp,m,Tp,0,vp1);
      vp2+=vp1;
      d2vdc2[i][j]=vp2/(4*h*h);
    }
  }
  cout.precision(5);
  if (argc == 4) {
    for (size_t i=0;i<n;i++)
      for (size_t j=0;j<n;j++)
        cout << "d2vdc2[" << i << "][" << j << "]="
             << d2vdc2[i][j] << endl;
  } else {
    cout << "2nd-order finite differences too unstable to give reproducible
        numbers on various platforms." << endl
         << "Run with third parameter to print derivative results." << endl;
  }
}
```

### B.5.2 gt1s<gt1s< double >::type>

This section shows the computation of the Hessian in second-order tangent mode.

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include "dco.hpp"
using namespace std;
using namespace dco;

typedef gt1s<gt1s<double>::type>::type DCO_TYPE;

#include "../include/f.hpp"

int main(int argc, char* argv[]){
  if (argc!=3) {
    cerr << "2 parameters expected:" << endl
   << "  1. number of spatial finite difference grid points" << endl
   << "  2. number of implicit Euler steps" << endl;
    return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  vector<vector<double> > d2vdc2(n,vector<double>(n,0));
  vector<DCO_TYPE> ca(n);
  vector<DCO_TYPE> Ta(n);
  DCO_TYPE va;
  for(size_t i=0;i<n;i++) {
    for(size_t j=0;j<n;j++) {
      for(size_t k=0;k<n;k++) { ca[k]=c[k]; Ta[k]=T[k]; }
      value(derivative(ca[i]))=1.0; derivative(value(ca[j]))=1.0;
      f(ca,m,Ta,O,va);
      d2vdc2[i][j]=derivative(derivative(va));
    }
  }
  cout.precision(5);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      cout << "dvdc[" << i << "][" << j << "]=" << d2vdc2[i][j] << endl;
}
```

### B.5.3 ga1s<gt1s< double >::type>

This section shows the approximation of the Hessian in second-order adjoint mode.

```
#include <cstdlib>
#include <iostream>
#include "dco.hpp"
using namespace std;
using namespace dco;
typedef ga1s<gt1s<double>::type> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

#include "../include/f.hpp"

int main(int argc, char* argv[]){
  if (argc!=3) {
    cerr << "2 parameters expected:" << endl
   << "  1. number of spatial finite difference grid points" << endl
   << "  2. number of implicit Euler steps" << endl;
    return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  vector<vector<double> > d2vdc2(n,vector<double>(n,0));

  vector<DCO_TYPE> ca(n);
  vector<DCO_TYPE> Ta(n);
  DCO_TYPE va;
  dco::smart_tape_ptr_t<DCO_MODE> tape;

  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    tape->register_variable(ca[i]);
  }
  DCO_TAPE_POSITION_TYPE p=tape->get_position();
  for(size_t i=0;i<n;i++) {
    for(size_t j=0;j<n;j++) Ta[j]=T[j];
    if (i>0) tape->zero_adjoints();
    derivative(value(ca[i]))=1;
    f(ca,m,Ta,O,va);
    value(derivative(va))=1;
    tape->interpret_adjoint();
    for(size_t j=0;j<n;j++) derivative(value(ca[j]))=0;
    for(size_t j=0;j<n;j++) d2vdc2[j][i]=derivative(derivative(ca[j]));
    tape->reset_to(p);
  }
  cout.precision(5);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      cout << "d2vdc2[" << i << "][" << j << "]=" << d2vdc2[i][j] << endl;

}
```

### B.5.4 `ga1s<gt1s< double >::type>` + Equidistant Checkpointing

This section shows the computation of the Hessian in second-adjoint mode with equidistant checkpointing.

```
#include <cstdlib>
#include <iostream>
#include <cassert>
#include "dco.hpp"
using namespace std;
using namespace dco;
typedef ga1s<gt1s<double>::type> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

#include "../include/f.hpp"

template<typename TYPE>
```

```
void EulerSteps(const size_t m, const vector<TYPE>& A, vector<TYPE>& T) {
  size_t n=T.size();
  for (size_t j=0;j<m;j++) {
    for (size_t i=0;i<n;++i) T[i]=-T[i];
    Solve(A,T);
  }
}

template<typename DCO_TYPE>
void EulerSteps_make_gap(const size_t m, const vector<DCO_TYPE>& A, vector<
    DCO_TYPE>& T){
  size_t n=T.size();

  auto T_in(T);

  // forward run
  dco::tape(A)->switch_to_passive();
  EulerSteps(m,A,T);
  dco::tape(A)->switch_to_active();

  // register output
  for (size_t i=0;i<n;i++) dco::tape(A)->register_variable(T[i]);

  auto fill_gap = [=,&A]() {
    auto p = dco::tape(A)->get_position();
    vector<DCO_TYPE> T1(T_in);

    //forward run
    EulerSteps(m,A,T1);
    cerr << "record =" << dco::size_of(dco::tape(A)) << "B" << endl;

    //get output adjoints (seeds for this section)
    for (size_t i=0;i<n;i++) {
      dco::tape(A)->register_output_variable(T1[i]);
      dco::derivative(T1[i]) = dco::derivative(T[i]);
    }

    //reverse run and reset to position p (so only reverse run of Solve)
    dco::tape(A)->interpret_adjoint_and_reset_to(p);
  };

  dco::tape(A)->insert_callback(std::move(fill_gap));
}

template <typename TYPE>
inline void sim(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
    size_t cs) {
  size_t n=T.size();
  static vector<TYPE> A(n*n); // static avoids copy

  residual_jacobian(c,T,A);
  for (size_t i=0;i<n;++i) {
    for (size_t j=0;j<n;++j)
      A[i*n+j]=A[i*n+j]/m;
    A[i+i*n]=A[i+i*n]-1;
  }
  LUDecomp(A);
  for (size_t j=0;j<m;j+=cs) {
    size_t s=(j+cs<m) ? cs : m-j;
    EulerSteps_make_gap<DCO_TYPE>(s,A,T);
  }
}

template <typename TYPE, typename OTYPE>
inline void f(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
    vector<OTYPE>& O, TYPE& v, const size_t cs) {
  sim(c,m,T,cs);
  v=0;
  size_t n=T.size();
  for (size_t i=0;i<n-1;++i)
    v=v+(T[i]-O[i])*(T[i]-O[i]);
  v=v/(n-1);
}

int main(int argc, char* argv[]){
  if (argc!=4) {
    cerr << "3 parameters expected:" << endl
   << "  1. number of spatial finite difference grid points" << endl
   << "  2. number of implicit Euler steps" << endl
   << "  3. distance between checkpoints" << endl;
    return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]), cs=atoi(argv[3]);
  assert(cs<=m);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  vector<vector<double> > d2vdc2(n,vector<double>(n,0));
  vector<DCO_TYPE> ca(n);
  vector<DCO_TYPE> Ta(n);
  DCO_TYPE va;

  dco::smart_tape_ptr_t<DCO_MODE> tape;
  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    tape->register_variable(ca[i]);
  }
  DCO_TAPE_POSITION_TYPE p=tape->get_position();
  for(size_t i=0;i<n;i++) {
    for(size_t j=0;j<n;j++) Ta[j]=T[j];
    if (i>0) tape->zero_adjoints();
    derivative(value(ca[i]))=1;
    f(ca,m,Ta,0,va,cs);
    value(derivative(va))=1;
    tape->interpret_adjoint();
    for(size_t j=0;j<n;j++) derivative(value(ca[j]))=0;
    for(size_t j=0;j<n;j++) d2vdc2[j][i]=derivative(derivative(ca[j]));
    tape->reset_to(p);
  }
  cout.precision(5);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      cout << "d2vdc2[" << i << "][" << j << "]=" << d2vdc2[i][j] << endl;
}
```

### B.5.5 ga1s<gt1s< double >::type> + Symbolic Derivative of Linear Solver

This section shows the computation of the Hessian in second-adjoint mode with symbolically differentiated linear solver.

```
#include <cstdlib>
#include <iostream>
#include <cassert>
#include "dco.hpp"
using namespace std;
using namespace dco;
typedef gt1s<double>::type DCO_BASE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

#include "../include/f.hpp"

template<typename DCO_MODE, typename TYPE>
void LUDecomp_make_gap(vector<TYPE>& A){
  size_t n=sqrt(double(A.size()));
  vector<DCO_BASE_TYPE> Ap(n*n);
  for(size_t i=0;i<n*n;i++) Ap[i]=value(A[i]);
  LUDecomp(Ap);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      value(A[i*n+j])=Ap[i*n+j];
}

// U^T*y=b
template <class TYPE>
inline void FSubstT(const vector<TYPE>& LU, vector<TYPE>& y) {
  size_t n=y.size();
  for (size_t i=0;i<n;i++){
    for (size_t j=0;j<i;j++)
      y[i]=y[i]-LU[j*n+i]*y[j];
    y[i]=y[i]/LU[i*n+i];
  }
}
```

```

// L^T*x=y
template <class TYPE>
inline void BSubstT(const vector<TYPE>& LU, vector<TYPE>& b) {
  size_t n=b.size();
  for (size_t k=n,i=n-1;k>0;k--,i--)
    for (size_t j=n-1;j>i;j--)
      b[i]=b[i]-LU[j*n+i]*b[j];
}

// LU^T*x=y
template <class TYPE>
inline void SolveT(const vector<TYPE>& LU, vector<TYPE>& b) {
  FSubstT(LU,b);
  BSubstT(LU,b);
}

template<typename DCO_MODE, typename TYPE>
void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b){

  const size_t n = b.size();
  auto rhs(b);
  dco::tape(LU)->switch_to_passive();
  Solve(LU,b);
  dco::tape(LU)->switch_to_active();
  dco::tape(LU)->register_variable(b);

  auto fill_gap = [=,&LU]() {
    vector<DCO_TYPE> a1b(n);
    for (size_t i = 0; i < n; ++i) a1b[i] = dco::derivative(b[i]);
    dco::tape(LU)->switch_to_passive();
    SolveT(LU,a1b);

    for (size_t i=0;i<n;i++)
      for (size_t j=0;j<n;j++)
        dco::derivative(LU[i*n+j]) += dco::value(-a1b[i]*b[j]);
    dco::derivative(rhs) += dco::value(a1b);
    dco::tape(LU)->switch_to_active();
  };

  dco::tape(LU)->insert_callback(std::move(fill_gap));
}

template <>
inline void sim(const vector<DCO_TYPE>& c, const size_t m, vector<DCO_TYPE>& T)
    {
  const size_t n=c.size();
  static vector<DCO_TYPE> A(n*n);
  residual_jacobian(c,T,A);
  for (size_t i=0;i<n;++i) {
    for (size_t j=0;j<n;++j)
      A[i*n+j]=A[i*n+j]/m;
    A[i+i*n]=A[i+i*n]-1;
  }
  LUDecomp_make_gap<DCO_MODE>(A);
  for (size_t j=0;j<m;++j) {
    for (size_t i=0;i<n;++i) T[i]=-T[i];
    Solve_make_gap<DCO_MODE>(A,T);
  }
}

int main(int argc, char* argv[]){
  if (argc!=3) {
    cerr << "2 parameters expected:" << endl
   << "  1. number of spatial finite difference grid points" << endl
   << "  2. number of implicit Euler steps" << endl;
    return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  vector<vector<double> > d2vdc2(n,vector<double>(n,0));

  vector<DCO_TYPE> ca(n);
  vector<DCO_TYPE> Ta(n);
  DCO_TYPE va;
  dco::smart_tape_ptr_t<DCO_MODE> tape;

  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    tape->register_variable(ca[i]);
  }
  DCO_TAPE_POSITION_TYPE p=tape->get_position();
  for(size_t i=0;i<n;i++) {
    for(size_t j=0;j<n;j++) Ta[j]=T[j];
    if (i>0) tape->zero_adjoints();
    derivative(value(ca[i]))=1;
    f(ca,m,Ta,O,va);
    value(derivative(va))=1;
    tape->interpret_adjoint();
    for(size_t j=0;j<n;j++) derivative(value(ca[j]))=0;
    for(size_t j=0;j<n;j++) d2vdc2[j][i]=derivative(derivative(ca[j]));
    tape->reset_to(p);
  }
  cout.precision(5);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      cout << "d2vdc2[" << i << "][" << j << "]=" << d2vdc2[i][j] << endl;
}
```

### B.5.6 ga1s<gt1s< double >::type> + Symbolic Derivative of Linear Solver and Equidistant Checkpointing

This section shows the computation of the Hessian in second-adjoint mode with equidistant checkpointing and symbolically differentiated linear solver.

```
#include <cstdlib>
#include <iostream>
#include <cassert>
#include "dco.hpp"
using namespace std;
using namespace dco;
typedef gt1s<double>::type DCO_BASE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

#include "../include/f.hpp"

template<typename DCO_MODE, typename TYPE>
void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b);

void EulerSteps(const size_t m, const vector<DCO_TYPE>& A, vector<DCO_TYPE>& T){
  size_t n=T.size();
  for (size_t j=0;j<m;j++) {
    for (size_t i=0;i<n;++i) T[i]=-T[i];
    if (dco::tape(A)->is_active())
        Solve_make_gap<DCO_MODE>(A,T);
      else
        Solve(A,T);
    }
}

template<typename DCO_TYPE>
void EulerSteps_make_gap(const size_t m, const vector<DCO_TYPE>& A, vector<
    DCO_TYPE>& T){
  size_t n=T.size();

  auto T_in(T);

  // forward run
  dco::tape(A)->switch_to_passive();
  EulerSteps(m,A,T);
  dco::tape(A)->switch_to_active();

  // register output
  for (size_t i=0;i<n;i++) dco::tape(A)->register_variable(T[i]);

  auto fill_gap = [=,&A]() {
    auto p = dco::tape(A)->get_position();
    vector<DCO_TYPE> T1(T_in);

    //forward run
    EulerSteps(m,A,T1);
    cerr << "record =" << dco::size_of(dco::tape(A)) << "B" << endl;

    //get output adjoints (seeds for this section)
    for (size_t i=0;i<n;i++) {
      dco::tape(A)->register_output_variable(T1[i]);
      dco::derivative(T1[i]) = dco::derivative(T[i]);
    }

    //reverse run and reset to position p (so only reverse run of Solve)
    dco::tape(A)->interpret_adjoint_and_reset_to(p);
  };

  dco::tape(A)->insert_callback(std::move(fill_gap));
}

template<typename DCO_MODE, typename TYPE>
void LUDecomp_make_gap(vector<TYPE>& A){
  size_t n=sqrt(double(A.size()));
  vector<DCO_BASE_TYPE> Ap(n*n);
  for(size_t i=0;i<n*n;i++) Ap[i]=value(A[i]);
  LUDecomp(Ap);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      value(A[i*n+j])=Ap[i*n+j];
}

// U^T*y=b
template <class TYPE>
inline void FSubstT(const vector<TYPE>& LU, vector<TYPE>& y) {
  size_t n=y.size();
  for (size_t i=0;i<n;i++){
    for (size_t j=0;j<i;j++)
      y[i]=y[i]-LU[j*n+i]*y[j];
    y[i]=y[i]/LU[i*n+i];
  }
}

// L^T*x=y
template <class TYPE>
inline void BSubstT(const vector<TYPE>& LU, vector<TYPE>& b) {
  size_t n=b.size();
  for (size_t k=n,i=n-1;k>0;k--,i--)
    for (size_t j=n-1;j>i;j--)
      b[i]=b[i]-LU[j*n+i]*b[j];
}

// LU^T*x=y
template <class TYPE>
inline void SolveT(const vector<TYPE>& LU, vector<TYPE>& b) {
  FSubstT(LU,b);
  BSubstT(LU,b);
}

```

```
template<typename DCO_MODE, typename TYPE>
void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b){
  const size_t n = b.size();
  auto rhs(b);
  dco::tape(LU)->switch_to_passive();
  Solve(LU,b);
  dco::tape(LU)->switch_to_active();
  dco::tape(LU)->register_variable(b);

  auto fill_gap = [=,&LU]() {
    vector<DCO_TYPE> a1b(n);
    for (size_t i = 0; i < n; ++i) a1b[i] = dco::derivative(b[i]);
    dco::tape(LU)->switch_to_passive();
    SolveT(LU,a1b);

    for (size_t i=0;i<n;i++)
      for (size_t j=0;j<n;j++)
        dco::derivative(LU[i*n+j]) += dco::value(-a1b[i]*b[j]);
    dco::derivative(rhs) += dco::value(a1b);
    dco::tape(LU)->switch_to_active();
  };

  dco::tape(LU)->insert_callback(std::move(fill_gap));
}

inline void sim(const vector<DCO_TYPE>& c, const size_t m, vector<DCO_TYPE>& T,
    const size_t cs) {
  const size_t n=c.size();
  static vector<DCO_TYPE> A(n*n);
  residual_jacobian(c,T,A);
  for (size_t i=0;i<n;++i) {
    for (size_t j=0;j<n;++j)
      A[i*n+j]=A[i*n+j]/m;
    A[i+i*n]=A[i+i*n]-1;
  }
  LUDecomp_make_gap<DCO_MODE>(A);
  for (size_t j=0;j<m;j+=cs) {
    size_t s=(j+cs<m) ? cs : m-j;
    EulerSteps_make_gap<DCO_TYPE>(s,A,T);
  }
}

template <typename TYPE, typename OTYPE>
inline void f(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
    vector<OTYPE>& O, TYPE& v, const size_t cs) {
  sim(c,m,T,cs);
  v=0;
  size_t n=T.size();
  for (size_t i=0;i<n-1;++i)
    v=v+(T[i]-O[i])*(T[i]-O[i]);
  v=v/(n-1);
}

int main(int argc, char* argv[]){
  if (argc!=4) {
    cerr << "3 parameters expected:" << endl
   << "  1. number of spatial finite difference grid points" << endl
   << "  2. number of implicit Euler steps" << endl
   << "  3. distance between checkpoints" << endl;
    return EXIT_FAILURE;
  }
  size_t n=atoi(argv[1]), m=atoi(argv[2]), cs=atoi(argv[3]);
  assert(cs<=m);
  vector<double> c(n);
  for (size_t i=0;i<n;i++) c[i]=0.01;
  vector<double> T(n);
  for (size_t i=0;i<n-1;i++) T[i]=300.;
  T[n-1]=1700.;
  ifstream ifs("O.txt");
  vector<double> O(n);
  for (size_t i=0;i<n;i++) ifs >> O[i] ;
  vector<vector<double> > d2vdc2(n,vector<double>(n,0));
  vector<DCO_TYPE> ca(n);
  vector<DCO_TYPE> Ta(n);
  DCO_TYPE va;

  dco::smart_tape_ptr_t<DCO_MODE> tape;
  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    tape->register_variable(ca[i]);
  }
  DCO_TAPE_POSITION_TYPE p=tape->get_position();
  for(size_t i=0;i<n;i++) {
    for(size_t j=0;j<n;j++) Ta[j]=T[j];
    if (i>0) tape->zero_adjoints();
    derivative(value(ca[i]))=1;
    f(ca,m,Ta,O,va,cs);
    value(derivative(va))=1;
    tape->interpret_adjoint();
    for(size_t j=0;j<n;j++) derivative(value(ca[j]))=0;
    for(size_t j=0;j<n;j++) d2vdc2[j][i]=derivative(derivative(ca[j]));
    tape->reset_to(p);
  }
  cout.precision(5);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      cout << "d2vdc2[" << i << "][" << j << "]=" << d2vdc2[i][j] << endl;
}
```

# Appendix C

# Case Study: Race

## C.1 Purpose

Let the function

$$y = f(\mathbf{x}) = \left( \sum_{i=0}^{n-1} x_i^2 \right)^2$$

be implemented as follows:

```
#pragma once

template<class T>
void f(const vector<T> x, T& y) {
  y=0;
  for (size_t i=0;i<x.size();i++) y=y+x[i]*x[i];
  y=y*y;
}
```

We compare various methods for computing the gradient and the Hessian of $f$.

## C.2 Gradient

| $n$ | cfd | gt1s | gt1v[1] | ga1s |
|---|---|---|---|---|
| $10^5$ | 89 | 78 | 56 | 0.3 |

Table C.1: Run times for identical results (up to machine accuracy) by AD modes; poor approximation by central finite differences.

### C.2.1 Central Finite Differences

```
#include<iostream>
#include<cfloat>
#include<cmath>
#include<cassert>
#include<cstdlib>
#include<vector>
using namespace std;

#include "../x22.hpp"

template<typename T>
void cfd_driver(const vector<T> &x, T &y, vector<T> &g) {
  size_t n=x.size();
  vector<T> x_ph(n), x_mh(n);
  T y_ph, y_mh;
  f(x,y);
  for (size_t i=0;i<n;i++) {
    for (size_t j=0;j<n;j++)  x_ph[j]=x_mh[j]=x[j];
    T h=(x[i]==0) ? sqrt(DBL_EPSILON) : sqrt(DBL_EPSILON)*std::abs(x[i]);
    x_ph[i]+=h;
    f(x_ph,y_ph);
    x_mh[i]-=h;
    f(x_mh,y_mh);
    g[i]=(y_ph-y_mh)/(2*h);
  }
}

int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(5);
  size_t n=atoi(v[1]);
  vector<double> x(n), g(n); double y;
  for (size_t i=0;i<n;i++) x[i]=cos(double(i));
  cfd_driver(x,y,g);
  cout << y << endl;
  for (size_t i=0;i<n;i++)
    cout << g[i] << endl;
}
```

### C.2.2 First-order Tangent Mode

```
#include<vector>
```

```
#include<iostream>
#include "dco.hpp"
using namespace std;
using namespace dco;
#include "../x22.hpp"

template<typename T>
void gt1s_driver(const vector<T>& xv, T& yv, vector<T>& g) {
  typedef typename gt1s<T>::type DCO_TYPE;
  size_t n=xv.size();
  vector<DCO_TYPE> x(n); DCO_TYPE y;
  for (size_t i=0;i<n;i++) {
    for (size_t j=0;j<n;j++) x[j]=xv[j];
    derivative(x[i])=1.;
    f(x,y);
    g[i]=derivative(y);
  }
  yv=value(y);
}

int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(5);
  size_t n=atoi(v[1]);
  vector<double> x(n), g(n); double y;
  for (size_t i=0;i<n;i++) x[i]=cos(double(i));
  gt1s_driver(x,y,g);
  cout << y << endl;
  for (size_t i=0;i<n;i++)
    cout << g[i] << endl;
}
```

### C.2.3 First-order Tangent Vector Mode

```
#include<vector>
#include<iostream>
#include "dco.hpp"
using namespace std;
using namespace dco;
#include "../x22.hpp"

template<typename T>
void gt1v_driver(const vector<T>& xv, T& yv, vector<T>& g) {
  size_t n=xv.size(); const size_t l=100; assert(!(n%l));
  typedef typename gt1v<T,l>::type DCO_TYPE;
  vector<DCO_TYPE> x(n); DCO_TYPE y;
  for (size_t j=0;j<n/l;j++) {
    for (size_t i=0;i<n;i++) x[i]=xv[i];
    for (size_t i=0;i<l;i++) derivative(x[j*l+i])[i]=1.;
    f(x,y);
    for (size_t i=0;i<l;i++) g[j*l+i]=derivative(y)[i];
  }
  yv=value(y);
}
```

```

int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(5);
  size_t n=atoi(v[1]);
  vector<double> x(n), g(n); double y;
  for (size_t i=0;i<n;i++) x[i]=cos(double(i));
  gt1v_driver(x,y,g);
  cout << y << endl;
  for (size_t i=0;i<n;i++)
    cout << g[i] << endl;
}
```

### C.2.4 First-order Adjoint Mode

```
#include<vector>
#include<iostream>
#include "dco.hpp"
using namespace std;
using namespace dco;
#include "../x22.hpp"

template<typename T>
void ga1s_driver(const vector<T>& xv, T& yv, vector<T>& g) {
  typedef ga1s<T> DCO_MODE;
  typedef typename DCO_MODE::type DCO_TYPE;
  size_t n=xv.size();
  vector<DCO_TYPE> x(n); DCO_TYPE y;

  typename dco::smart_tape_ptr_t<DCO_MODE> tape;
  for (size_t j=0;j<n;j++) {
    tape->register_variable(x[j]);
    value(x[j])=xv[j];
  }
  f(x,y);
  tape->register_output_variable(y);
  yv=value(y);
  derivative(y)=1;
  tape->interpret_adjoint();
  for (size_t j=0;j<n;j++) g[j]=derivative(x[j]);
}

int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(5);
  size_t n=atoi(v[1]);
  vector<double> x(n), g(n); double y;
  for (size_t i=0;i<n;i++) x[i]=cos(double(i));
  ga1s_driver(x,y,g);
  cout << y << endl;
  for (size_t i=0;i<n;i++)
    cout << g[i] << endl;
}
```

## C.3 Hessian

| $n$ | socfd | gt2s_gt1s | gt2s_ga1s | ga2s_gt1s | ga2s_ga1s |
|---|---|---|---|---|---|
| $10^3$ | 11 | 11 | 2.5 | 2.6 | 2.5 |
| $2 \cdot 10^3$ | 106 | 91 | 9.6 | 10.8 | 9.6 |

Table C.2: Run times for identical results (up to machine accuracy) by AD modes; poor approximation by central finite differences.

### C.3.1 Second-order Central Finite Differences

```
#include<iostream>
#include<cfloat>
#include<cmath>
#include<cassert>
#include<cstdlib>
#include<vector>
using namespace std;

#include "../x22.hpp"

template<typename T>
void cfd_driver(const vector<T> &x, T &y, vector<T> &g) {
  size_t n=x.size();
  vector<T> x_ph(n), x_mh(n);
  T y_ph, y_mh;
  f(x,y);
  for (size_t i=0;i<n;i++) {
    for (size_t j=0;j<n;j++)  x_ph[j]=x_mh[j]=x[j];
    T h=(x[i]==0) ? sqrt(sqrt(DBL_EPSILON)) : sqrt(sqrt(DBL_EPSILON))*std::abs(x
        [i]);
    x_ph[i]+=h;
    f(x_ph,y_ph);
    x_mh[i]-=h;
    f(x_mh,y_mh);
    g[i]=(y_ph-y_mh)/(2*h);
  }
}

template<typename T>
void socfd_driver(const vector<T> &x, T &y, vector<T> &g, vector<vector<T> >& H)
     {
  size_t n=x.size();
  vector<T> x_ph(n), x_mh(n), g_ph(n), g_mh(n);
  T y_ph, y_mh;
  cfd_driver(x,y,g);
  for (size_t i=0;i<n;i++) {
    for (size_t j=0;j<n;j++)  x_ph[j]=x_mh[j]=x[j];
    T h=(x[i]==0) ? sqrt(sqrt(DBL_EPSILON)) : sqrt(sqrt(DBL_EPSILON))*std::abs(x
        [i]);
    x_ph[i]+=h;
    f(x_ph,y_ph);
    cfd_driver(x_ph,y_ph,g_ph);
    x_mh[i]-=h;
    cfd_driver(x_mh,y_mh,g_mh);
    for (size_t j=0;j<n;j++) H[i][j]=(g_ph[j]-g_mh[j])/(2*h);
  }
}

int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(5);
  size_t n=atoi(v[1]);
  vector<vector<double> > H(n, vector<double>(n));
  vector<double> x(n), g(n); double y;
  for (size_t i=0;i<n;i++) x[i]=cos(double(i));
  socfd_driver(x,y,g,H);
  cout << y << endl;
  for (size_t i=0;i<n;i++)
    cout << g[i] << endl;
  for (size_t i=0;i<n;i++)
    for (size_t j=0;j<n;j++)
      cout << H[i][j] << endl;
}
```

### C.3.2 Second-order Tangent Mode

```
#include<iostream>
#include<vector>
#include "dco.hpp"

using namespace std;
using namespace dco;

typedef gt1s<double> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef gt1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

#include "../x22.hpp"

template<typename T>
void driver(const vector<T>& xv, T& yv, vector<T>& g, vector<vector<T> >& h)  {
  size_t n=xv.size();
  vector<DCO_TYPE> x(n); DCO_TYPE y;
  for (size_t i=0;i<n;i++) {
    for (size_t j=0;j<n;j++) {
      for (size_t k=0;k<n;k++) x[k]=xv[k];
      derivative(value(x[i]))=1;
      value(derivative(x[j]))=1;
      f(x,y);
      h[i][j]=derivative(derivative(y));
    }
    g[i]=derivative(value(y));
  }
  yv=passive_value(y);
}

int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(5);
  size_t n=atoi(v[1]);
  vector<vector<double> > H(n, vector<double>(n));
  vector<double> x(n), g(n); double y;
  for (size_t i=0;i<n;i++) x[i]=cos(double(i));
  driver(x,y,g,H);
  cout << y << endl;
  for (size_t i=0;i<n;i++)
    cout << g[i] << endl;
  for (size_t i=0;i<n;i++)
    for (size_t j=0;j<n;j++)
      cout << H[i][j] << endl;
}
```

### C.3.3 Second-order Adjoint Mode (Tangent over Adjoint)

```
#include<vector>
#include<iostream>
#include "dco.hpp"
using namespace std;
using namespace dco;
#include "../x22.hpp"

template<typename T>
void driver(const vector<T>& xv, T& yv, vector<T>& g, vector<vector<T> >&  H) {
  typedef typename gt1s<T>::type DCO_BASE_TYPE;
  typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
  typedef typename DCO_MODE::type DCO_TYPE;
  size_t n=xv.size();
  vector<DCO_TYPE> x(n); DCO_TYPE y;

  dco::smart_tape_ptr_t<DCO_MODE> tape;
  for (size_t i=0;i<n;i++) {
    for (size_t j=0;j<n;j++) {
      tape->register_variable(x[j]);
      passive_value(x[j])=xv[j];
      derivative(value(x[j]))=0;
    }
    derivative(value(x[i]))=1;
    f(x,y);
    tape->register_output_variable(y);
    yv=passive_value(y);
    g[i]=derivative(value(y));
    value(derivative(y))=1;
    tape->interpret_adjoint();
    for (size_t j=0;j<n;j++) H[i][j]=derivative(derivative(x[j]));
    tape->reset();
  }
}

```

```
int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(5);
  size_t n=atoi(v[1]);
  vector<vector<double> > H(n, vector<double>(n));
  vector<double> x(n), g(n); double y;
  for (size_t i=0;i<n;i++) x[i]=cos(double(i));
  driver(x,y,g,H);
  cout << y << endl;
  for (size_t i=0;i<n;i++)
    cout << g[i] << endl;
  for (size_t i=0;i<n;i++)
    for (size_t j=0;j<n;j++)
      cout << H[i][j] << endl;
}
```

### C.3.4 Second-order Adjoint Mode (Adjoint over Tangent)

```
#include<iostream>
#include<vector>
#include "dco.hpp"

using namespace std;
using namespace dco;

typedef ga1s<double> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef gt1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

#include "../x22.hpp"

template<typename T>
void driver(const vector<T>& xv, T& yv, vector<T>& g, vector<vector<T> >& h) {
  dco::smart_tape_ptr_t<DCO_BASE_MODE> tape;
  size_t n=xv.size();
  vector<DCO_TYPE> x(n),x_in(n);
  DCO_TYPE y;
  for (size_t i=0;i<n;i++) {
    for (size_t j=0;j<n;j++) {
      x[j]=xv[j];
      tape->register_variable(value(x[j]));
      tape->register_variable(derivative(x[j]));
      x_in[j]=x[j];
    }
    value(derivative(x[i]))=1;
    f(x,y);
    tape->register_output_variable(value(y));
    tape->register_output_variable(derivative(y));
    derivative(derivative(y))=1;
    tape->interpret_adjoint();
    for (size_t j=0;j<n;j++)
      h[j][i] = derivative(value(x_in[j]));
    tape->reset();
    g[i]=value(derivative(y));
  }
  yv=passive_value(y);
}

int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(5);
  size_t n=atoi(v[1]);
  vector<vector<double> > H(n, vector<double>(n));
  vector<double> x(n), g(n); double y;
  for (size_t i=0;i<n;i++) x[i]=cos(double(i));
  driver(x,y,g,H);
  cout << y << endl;
  for (size_t i=0;i<n;i++)
    cout << g[i] << endl;
  for (size_t i=0;i<n;i++)
    for (size_t j=0;j<n;j++)
      cout << H[i][j] << endl;
}
```

### C.3.5 Second-order Adjoint Mode (Adjoint over Adjoint)

```
#include<iostream>
#include<vector>
#include "dco.hpp"

using namespace std;
using namespace dco;

typedef ga1s<double> DCO_BASE_MODE;
typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
typedef DCO_BASE_MODE::tape_t DCO_BASE_TAPE_TYPE;
typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;

#include "../x22.hpp"

template<typename T>
void driver(const vector<T>& xv, T& yv, vector<T>& g, vector<vector<T> >& h) {
  size_t n=xv.size();
  vector<DCO_TYPE> x(n),x_in(n);
  DCO_TYPE y;
  dco::smart_tape_ptr_t<DCO_BASE_MODE> tape_base;
  dco::smart_tape_ptr_t<DCO_MODE> tape;
  for (size_t j=0;j<n;j++) {
    x[j]=xv[j];
    tape_base->register_variable(value(x[j]));
    tape->register_variable(x[j]);
    x_in[j]=x[j];
  }
  f(x,y);
  derivative(y)=1.0;
  tape_base->register_variable(derivative(y));
  tape->interpret_adjoint();
  for (size_t j=0;j<n;j++)
    g[j]=value(derivative(x_in[j]));
  for (size_t i=0;i<n;i++) {
    derivative(derivative(x_in[i]))=1;
    tape_base->interpret_adjoint();
    for (size_t j=0;j<n;j++)
      h[i][j]=derivative(value(x_in[j]));
    tape_base->zero_adjoints();
  }
  yv=passive_value(y);
}

int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(5);
  size_t n=atoi(v[1]);
  vector<vector<double> > H(n, vector<double>(n));
  vector<double> x(n), g(n); double y;
  for (size_t i=0;i<n;i++) x[i]=cos(double(i));
  driver(x,y,g,H);
  cout << y << endl;
  for (size_t i=0;i<n;i++)
    cout << g[i] << endl;
  for (size_t i=0;i<n;i++)
    for (size_t j=0;j<n;j++)
      cout << H[i][j] << endl;
}
```

# Index

DCO_ADJOINT_TYPE, 116
DCO_BASE_TYPE, 22
DCO_EAO_TYPE::get_output_adjoint, 121
DCO_EAO_TYPE::increment_input_adjoint, 122
DCO_EAO_TYPE::read_data, 122
DCO_EAO_TYPE::register_input, 121
DCO_EAO_TYPE::register_output, 121
DCO_EAO_TYPE::write_data, 122
DCO_LOCAL_GRADIENT_TYPE::put, 95
DCO_MODE::derivative_t, 117
DCO_MODE::external_adjoint_object_t, 120
DCO_MODE::global_tape, 118
DCO_MODE::jacobian_preaccumulator_t::finish, 122
DCO_MODE::jacobian_preaccumulator_t::register_output, 122
DCO_MODE::jacobian_preaccumulator_t::start, 122
DCO_MODE::jacobian_preaccumulator_t, 122
DCO_MODE::local_gradient_t, 95
DCO_MODE::passive_t, 117
DCO_MODE::value_t, 63, 117
DCO_MODE, 23, 115–117
DCO_PARTIAL_TYPE, 116
DCO_TAPE_TYPE::adjoints, 111
DCO_TAPE_TYPE::adjoint, 114
DCO_TAPE_TYPE::create_callback_object, 121
DCO_TAPE_TYPE::create, 25, 35, 118
DCO_TAPE_TYPE::get_position, 62
DCO_TAPE_TYPE::insert_callback, 63, 121
DCO_TAPE_TYPE::interpret_adjoint_and_reset_to, 63
DCO_TAPE_TYPE::interpret_adjoint, 25, 118
DCO_TAPE_TYPE::iterator_t, 62
DCO_TAPE_TYPE::register_output_variable, 25, 118
DCO_TAPE_TYPE::register_variable, 25, 118
DCO_TAPE_TYPE::remove, 25, 118
DCO_TAPE_TYPE::reset_to, 63
DCO_TAPE_TYPE::scalar_callback_mode, 90, 120
DCO_TAPE_TYPE::synchronize_with, 114
DCO_TAPE_TYPE, 25, 118
DCO_TYPE, 23, 115–117
DCO_VALUE_TYPE, 116
adjoint, 27
dco::adjoint_vector::interpret_adjoint, 122
dco::adjoint_vector, 122

dco::derivative(DCO_TYPE,dco::adjoint_vector), 122
dco::log_level_enum, 111
dco::logger::level, 111
dco::logger::stream, 112
dco::mode<DCO_TYPE>, 63
dco::size_of(DCO_TAPE_TYPE*), 63
derivative, 23, 53, 117
ga1s<DCO_BASE_TYPE>::type, 25
ga1sm<DCO_BASE_TYPE>::type, 55
gt1s<DCO_BASE_TYPE>::type, 23
gtas<DCO_BASE_TYPE>::type, 27
mode<DCO_TYPE>, 122
passive_value, 39, 117
smart_tape_ptr_t, 25, 118
tangent, 27
value, 23, 117

adjoint
- checkpointing
  - ensemble, 78
  - joint call reversal, 60
- embedding adjoint source code, 82, 87
- first-order
  - DCO_ADJOINT_TYPE, 36
  - DCO_ADJOINT_VECTOR_TYPE::interpret_adjoint(), 104
  - DCO_ADJOINT_VECTOR_TYPE, 104
  - DCO_MODE::gbcp_order, 60
  - DCO_MODE::order, 60
  - DCO_PARTIAL_TYPE, 36
  - DCO_VALUE_TYPE, 36
  - adjoint_vector<DCO_TAPE_TYPE,DCO_ADJOINT_TYPE>, 104
  - create_gbcp<GBCP_TYPE>(ACTIVE_TYPE), 60
  - derivative(DCO_T&,DCO_ADJOINT_VECTOR_TYPE&), 104
  - gbcp<DCO_ACTIVE_TYPE>::type, 60
  - jacobian_preaccumulator_t::finish(), 101
  - jacobian_preaccumulator_t::register_output(DCO_TYPE&), 101
  - jacobian_preaccumulator_t::start(), 101
  - jacobian_preaccumulator_t, 101
  - blob tape, 34
  - chunk tape, 34
  - file tape, 35
  - preaccumulation, 99
  - scalar, 23, 25, 55
  - vector, 53, 55
- second-order
  - checkpointing ensemble, 80
  - joint call reversal, 63
  - preaccumulation, 101
  - scalar, 39, 41
- sparsity
  - interpreter, 108
- third-order
  - scalar, 48

casting, 112

dco.hpp, 22
`dco::tape_options`, 34, 116

logging, 111

**`static_cast`**, 112

tangent
- first-order
  - scalar, 21
  - vector, 51
- second-order
  - scalar, 37
- third-order
  - scalar, 44

type cast, 112

vector type, 28
vectorization, 32
vectorized
- tangent
  - type, 30

# Bibliography

[1] C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, number 64 in LNCSE, Berlin, 2008. Springer.

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

[3] Mike B Giles. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. *Advances in Automatic Differentiation*, pages 35–44, 2008.

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

[5] M. Heath. *Scientific Computing. An Introductory Survey.* McGraw-Hill, New York, 1998.

[6] U. Naumann. DAG reversal is NP-complete. *J. Discr. Alg.*, 7:402–410, 2009.

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

[8] U. Naumann and O. Schenk, editors. *Combinatorial Scientific Computing*, Computational Science Series. Chapman & Hall / CRC Press, Taylor and Francis Group, 2012.

[9] U. Naumann and A. Walther. Combinatorial problems in Algorithmic Differentiation. In [8], pages 129–162, 2011.