# `dco/c++` User Guide

Uwe Naumann
Klaus Leppkes
Johannes Lotz

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2014-03
RWTH Aachen University | Department of Computer Science | May 2014

# dco/c++ User Guide

version 3.3.0

The Numerical Algorithms Group Ltd. (NAG)
Wilkinson House
Jordan Hill Road
Oxford OX2 8DR, United Kingdom

www: http://www.nag.co.uk
contact email: Uwe.Naumann@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: http://www.stce.rwth-aachen.de
contact email: naumann@stce.rwth-aachen.de

# Copyright Statement



Produced by NAG and STCE.

Version 3.3 released September 2018
ISBN 978-1-85206-219-4

# 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.

**Version 3.2** of `dco/c++` features a redesigned internal representation and an extended application programming interface resulting in smaller tapes and improved computational performance. Memory occupied by the tape can now be extended to disk making brute-force evaluation of adjoints for larger problems feasible. A special tape compression feature enhances and simplifies the user's control over the size of the tape. A single tape can now be combined with several adjoint vectors enabling parallel interpretation using potentially distinct adjoint data types.

**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

# Chapter 1

# Features (Summary)

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

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

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

### 1.1.1 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 $D$.

### 1.1.2 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 $D$.

### 1.1.3 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

### 1.2.1 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 $D$.

### 1.2.2 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 $D$.

### 1.2.3 Global “blob” tape

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

### 1.2.4 Global “chunk” tape

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

### 1.2.5 Global “file” tape (v3.2)

Chunks are written to and read from disk.

### 1.2.6 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.

### 1.2.7 (Thread-)local tapes

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

### 1.2.8 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.

### 1.2.9 Repeated evaluation of tape

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

### 1.2.10 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

### 1.2.11 User-defined local Jacobians interface

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

### 1.2.12 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.

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

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

### 1.2.14 Modulo Adjoint Propagation (v3.3)

The vector of adjoints is compressed by analysing the maximum number of required distinct adjoint memory locations.

## 1.3 Second- and Higher-Order Tangent Mode

### 1.3.1 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

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

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 $O(n^2) \cdot \mathrm{Cost}(f)$. with both $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ ranging independently over the Cartesian basis vectors in $D^n$.

### 1.3.2 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

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

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 $O(n^2) \cdot \mathrm{Cost}(f)$ with both $X^{(1)}$ and $X^{(2)}$ set equal to the identities in $D^n$.

### 1.3.3 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.

### 1.3.4 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

### 1.4.1 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

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

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.

### 1.4.2 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

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

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.

### 1.4.3 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

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

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

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

### 1.4.4 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.

# Chapter 2

# General Remarks

## 2.1 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.

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

- The environment variable `DCO_MEM_RATIO` defines the ratio of available physical memory that should be allocated (default: 0.5).
- The environment variable `DCO_MAX_ALLOCATION` defines the maximal allocation size to be used in kilobytes.

## 2.2 Feature Selection

Definition of the following preprocessor variables select the respective feature.

- `DCO_CHUNK_TAPE` (default: undefined): Switches to *chunk tape*. The default chunk size is 128MB.
- `DCO_LOG_MAX_LEVEL` (default: $-1$): Defines the maximal and default logging level. If $-1$, logging is optimized out completely.
- `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_USE_LONG_INT` (default: undefined): Switches internal counter from 'int' to 'long int'. 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_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. 37 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, the user must define the environment variable DCO_MAX_ALLOCATION (see Section 2.1) or use the chunk tape.

The following preprocessor variables can only be used with the base source version of dco/c++.

- DCO_TAPE_ACTIVITY (default: ON): Runtime varied analysis is performed.
- DCO_DEBUG (default: OFF): Improved safety through enhanced checking, e.g., integer overflow is checked.
- DCO_TAPE_MERGE_PARALLEL_EDGES (default: OFF): Parallel edges are merged in the tape.
- DCO_TAPE_BOUNDS_CHECK (default: ON): Tape bounds are checked if using a blob tape.
- DCO_ZERO_EDGE_CHECK (default: ON): Vanishing partial derivatives are detected and not recorded.

## 2.3 Parallelization

- All tangent types are thread safe.
- dco::ga1s<T>, dco::ga1v<T>, etc. are not thread safe due to use of a global tape.
- For thread safety, use respective multiple tape versions: dco::ga1sm<T>, dco::ga1vm<T>, etc.

# 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::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.

### 3.1.4 `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.

### 3.1.5 `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.

### 3.1.6 `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 {
  size_of_stack = 1,
  size_of_allocated_stack = 2,
  size_of_internal_adjoint_vector = 4,
  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);
```

## 3.2 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;
```

#### 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

# First-Order Tangent Mode

## 4.1 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 $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, $\mathbf{y} = f(\mathbf{x})$, computes

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

## 4.2 Example

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

```
template<typename T>
void f(const vector<T>& x, vector<T>& 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];
}
```

The driver computes (4.1) for $\mathbf{x} \hat{=}$ xv, $\mathbf{x}^{(1)} \hat{=}$ xt, $\mathbf{y} \hat{=}$ yv, and $\mathbf{y}^{(1)} \hat{=}$ yt.

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

#include "dco.hpp"
using namespace dco;

typedef double 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>& xt,
    vector<double>& yv, vector<double>& yt
) {
  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(x[i])=xv[i]; derivative(x[i])=xt[i]; }
  f(x,y);
  for (size_t i=0;i<m;i++) { yv[i]=value(y[i]); yt[i]=derivative(y[i]); }
}
```

The main program initializes all variables on the right-hand side of (4.1) followed by calling the driver and printing the results.

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

#include "driver.hpp"

int main() {
  const int m=2, n=4; cout.precision(15);
  vector<double> xv(n), xt(n), yv(m), yt(m);
  for (int i=0;i<n;i++) { xv[i]=1; xt[i]=1; }
  driver(xv,xt,yv,yt);
  for (int i=0;i<m;i++)
    cout << "y[" << i << "]=" << yv[i] << endl;
  for (int i=0;i<m;i++)
    cout << "y^{(1)}[" << i << "]=" << yt[i] << endl;
  return 0;
}
```

The following output is generated:

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

## 4.3 New dco/c++ Features

### 4.3.1 dco.hpp

dco/c++ header to be included.

### 4.3.2 DCO_BASE_TYPE

Variable instantiation type for generic AD modes.

### 4.3.3 gt1s<DCO_BASE_TYPE>

Generic first-order tangent mode.

### 4.3.4 DCO_MODE

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

### 4.3.5 gt1s<DCO_BASE_TYPE>::type

Generic first-order tangent type.

### 4.3.6 DCO_TYPE

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

### 4.3.7 value

Returns reference to value of argument.

### 4.3.8 derivative

Returns reference to derivative of argument.

# Chapter 5

# First-Order Adjoint Mode

## 5.1 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 $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, $\mathbf{y} = f(\mathbf{x})$, computes

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)} \cdot \frac{\partial f}{\partial \mathbf{x}} \rangle \equiv \mathbf{x}_{(1)} + \left(\frac{\partial f}{\partial \mathbf{x}}\right)^T \cdot \mathbf{y}_{(1)}. \end{aligned} \tag{5.1}$$

## 5.2 Example

For the same function $f : \mathbb{R}^4 \rightarrow \mathbb{R}^2$ used in Chapter 4 the driver computes (5.1) for $\mathbf{x} \hat{=}$ xv, $\mathbf{x}_{(1)} \hat{=}$ xa, $\mathbf{y} \hat{=}$ yv, and $\mathbf{y}_{(1)} \hat{=}$ ya.

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

#include "dco.hpp"
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_MODE::global_tape=DCO_TAPE_TYPE::create();
  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];
    DCO_MODE::global_tape->register_variable(x[i]);
  }
  f(x,y);
  for (size_t i=0;i<m;i++) {
    DCO_MODE::global_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];

  DCO_MODE::global_tape->write_to_dot();
  DCO_MODE::global_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]);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

The main program initializes all variables on the right-hand side of (5.1) 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(15);
  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);
  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;
  return 0;
}
```

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
```

## 5.3 New dco/c++ Features

### 5.3.1 ga1s<DCO_BASE_TYPE>

Generic first-order adjoint scalar mode.

### 5.3.2 ga1s<DCO_BASE_TYPE>::type

Generic first-order adjoint scalar type.

### 5.3.3 DCO_TAPE_TYPE

Type of tape associated with DCO_MODE.

### 5.3.4 DCO_MODE::global_tape

Global tape pointer associated with DCO_MODE.

### 5.3.5 DCO_TAPE_TYPE::create

Tape creator returns pointer to tape as DCO_TAPE_TYPE*.

### 5.3.6 DCO_TAPE_TYPE::register_variable

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

### 5.3.7 DCO_TAPE_TYPE::register_output_variable

Marks argument as a dependent variable in associated tape.

### 5.3.8 DCO_TAPE_TYPE::interpret_adjoint

Adjoint interpretation of entire tape.

### 5.3.9 DCO_TAPE_TYPE::remove

Deallocates global tape.

# Chapter 6

# First-Order Adjoint Mode: Blob/Chunk Tapes

## 6.1 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.

## 6.2 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
  DCO_M::global_tape=DCO_TAPE_T::create(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 here.

## 6.3 New dco/c++ Features

### 6.3.1 dco::tape_options

Tape configuration object.

### 6.3.2 size_t dco::tape_options::chunk_size_in_byte()

Get chunk size in bytes.

### 6.3.3 void dco::tape_options::set_chunk_size_in_kbyte( double )

Set chunk size in kilobytes.

### 6.3.4 void dco::tape_options::set_chunk_size_in_mbyte( double )

Set chunk size in megabytes.

### 6.3.5 void dco::tape_options::set_chunk_size_in_gbyte( double )

Set chunk size in gigabytes.

### 6.3.6 size_t dco::tape_options::blob_size_in_byte()

Get blob size in bytes.

### 6.3.7 void dco::tape_options::set_blob_size_in_kbyte( double )

Set blob size in kilobytes.

### 6.3.8 void dco::tape_options::set_blob_size_in_mbyte( double )

Set blob size in megabytes.

### 6.3.9 void dco::tape_options::set_blob_size_in_gbyte( double )

Set blob size in gigabytes.

### 6.3.10 DCO_TAPE_TYPE::create

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

# Chapter 7

# First-Order Adjoint Mode: File Tape

## 7.1 Purpose

*File tapes* are chunk tapes. To enable it, 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.

## 7.2 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_M::global_tape=DCO_TAPE_T::create(o);
  std::cout << o.chunk_size_in_byte() << std::endl;
...
  DCO_TAPE_T::remove(DCO_M::global_tape);
...
```

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.

## 7.3 New dco/c++ Features

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

# Chapter 8

# First-Order Adjoint Mode: Tape Data Types

## 8.1 Purpose

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

See Chapter 5 for general information on first-order adjoint mode.

## 8.2 Example

## 8.3 Example Text

Replace lines 7–10 in the driver in Section 5.2 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;
}
```

## 8.4 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
```

## 8.5 New **dco/c++** Features

### 8.5.1 **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).

# Chapter 9

# Second-Order Tangent Mode

## 9.1 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

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{y}^{(2)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(2)} \rangle \\ \mathbf{y}^{(1)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1)} \rangle \\ \mathbf{y}^{(1,2)} &:= \langle \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \rangle + \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1,2)} \rangle. \end{aligned} \tag{9.1}$$

## 9.2 Example

For the same function $f : \mathbb{R}^4 \to \mathbb{R}^2$ used in Chapter 4 the driver computes (9.1) 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>
using namespace std;

#include "dco.hpp"
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 (9.1) 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(15);
  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;
  return 0;
}
```

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
```

## 9.3 New dco/c++ Features

### 9.3.1 passive_value

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

# Chapter 10

# Second-Order Adjoint Mode

## 10.1 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

$$\begin{aligned}
\mathbf{y} &:= f(\mathbf{x}) \\
\mathbf{y}^{(2)} &:= \langle f(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \frac{\partial f}{\partial \mathbf{x}} \rangle \\
\mathbf{x}_{(1)}^{(2)} &:= \mathbf{x}_{(1)}^{(2)} + \langle \mathbf{y}_{(1)}^{(2)}, \frac{\partial f}{\partial \mathbf{x}} \rangle + \langle \mathbf{y}_{(1)}, \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(2)} \rangle.
\end{aligned} \tag{10.1}$$

## 10.2 Example

For the same function $f : \mathbb{R}^4 \to \mathbb{R}^2$ used in Chapter 4 the driver computes (10.1) for $\mathbf{x} \widehat{=}$ `xv`, $\mathbf{x}_{(1)} \widehat{=}$ `xa1`, $\mathbf{x}^{(2)} \widehat{=}$ `xt2`, $\mathbf{x}_{(1)}^{(2)} \widehat{=}$ `xa1t2`, $\mathbf{y} \widehat{=}$ `yv`, $\mathbf{y}_{(1)} \widehat{=}$ `ya1`, $\mathbf{y}^{(2)} \widehat{=}$ `yt2`, and $\mathbf{y}_{(1)}^{(2)} \widehat{=}$ `ya1t2`,

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

#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;

```

```
#include "f.hpp"

void driver(
    const vector<double>& xv,
    const vector<double>& xt2,
    vector<double>& xa1,
    vector<double>& xa1t2,
    vector<double>& yv,
    vector<double>& yt2,
    vector<double>& ya1,
    vector<double>& ya1t2
) {
  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  const 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];
    DCO_MODE::global_tape->register_variable(x[i]);
    derivative(value(x[i]))=xt2[i];
  }
  f(x,y);
  for (size_t i=0;i<n;i++) {
    value(derivative(x[i]))=xa1[i];
    derivative(derivative(x[i]))=xa1t2[i];
  }
  for (size_t i=0;i<m;i++) {
    yv[i]=passive_value(y[i]);
    yt2[i]=derivative(value(y[i]));
    DCO_MODE::global_tape->register_output_variable(y[i]);
    value(derivative(y[i]))=ya1[i];
    derivative(derivative(y[i]))=ya1t2[i];
  }
  DCO_MODE::global_tape->interpret_adjoint();
  for (size_t i=0;i<n;i++) {
    xa1t2[i]=derivative(derivative(x[i]));
    xa1[i]=value(derivative(x[i]));
  }
  for (size_t i=0;i<m;i++) {
    ya1t2[i]=derivative(derivative(y[i]));
    ya1[i]=value(derivative(y[i]));
  }
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

The main program initializes all variables on the right-hand side of (10.1) 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(15);
  vector<double> xv(n), xa1(n), xt2(n), xa1t2(n);
  vector<double> yv(m), ya1(m), yt2(m), ya1t2(m);
  for (int i=0;i<n;i++) { xv[i]=1; xt2[i]=1; xa1[i]=1; xa1t2[i]=0; }
  for (int i=0;i<m;i++) { ya1[i]=1; ya1t2[i]=0; }
  driver(xv,xt2,xa1,xa1t2,yv,yt2,ya1,ya1t2);
  for (int i=0;i<m;i++)
    cout << "y[" << i << "]=" << yv[i] << endl;
  for (int i=0;i<n;i++)
    cout << "x_{(1)}[" << i << "]=" << xa1[i] << endl;
  for (int i=0;i<m;i++)
    cout << "y^{(2)}[" << i << "]=" << yt2[i] << endl;
  for (int i=0;i<n;i++)
    cout << "x_{(1)}^{(2)}[" << i << "]=" << xa1t2[i] << endl;
  return 0;
}
```

```
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]=14.2435494001203
y^{(2)}[1]=11.4495304876283
x_{(1)}^{(2)}[0]=26.6930798877486
x_{(1)}^{(2)}[1]=153.74997790304
x_{(1)}^{(2)}[2]=-225.32463701389
x_{(1)}^{(2)}[3]=-225.32463701389
```

## 10.3 New dco/c++ Features

None.

# Chapter 11

# Approximate Second-Order Adjoint Mode

## 11.1 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

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{y}^{(2)} &:\approx \langle f(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\ \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \frac{\partial f}{\partial \mathbf{x}} \rangle \\ \mathbf{x}_{(1)}^{(2)} &:\approx \mathbf{x}_{(1)}^{(2)} + \langle \mathbf{y}_{(1)}^{(2)}, \frac{\partial f}{\partial \mathbf{x}} \rangle + \langle \mathbf{y}_{(1)}, \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(2)} \rangle. \end{aligned} \tag{11.1}$$

## 11.2 Example

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

#include "dco.hpp"
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_MODE::global_tape=DCO_TAPE_TYPE::create();
  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];
    DCO_MODE::global_tape->register_variable(x[i]);
  }
  f(x,y);
  for (size_t i=0;i<m;i++) {
    DCO_MODE::global_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];
  DCO_MODE::global_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]);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}


#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(15);
  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;
  return 0;
}
```

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
```

## 11.3 New dco/c++ Features

None.

# Chapter 12

# Third-Order Tangent Mode

## 12.1 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

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

## 12.2 Example

```
#include<iostream>
```

```
using namespace std;

#include "dco.hpp"
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(15);
  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;
  return 0;
}
```

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
```

## 12.3 New dco/c++ Features

None.

# Chapter 13

# Third-Order Adjoint Mode

## 13.1 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

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

## 13.2 Example

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

#include "dco.hpp"
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_MODE::global_tape=DCO_TAPE_TYPE::create();
  const size_t n=xv.size(), m=yv.size();
  vector<DCO_TYPE> x(n), y(m);
  for (size_t i=0;i<n;i++) {
    DCO_MODE::global_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])));
    DCO_MODE::global_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];
  }
  DCO_MODE::global_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])));
  }
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

```
#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;
  return 0;
}
```

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
```

## 13.3 New dco/c++ Features

None.

# Chapter 14

# First-Order Tangent Mode: Vector Mode

## 14.1 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

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

## 14.2 Example

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

```
#include<iostream>
#include<vector>
using namespace std;
#include "dco.hpp"
using namespace dco;
#include "f.hpp"

const DCO_INTEGRAL_TAPE_INT l=4;

typedef double DCO_BASE_MODE;
typedef gt1v<DCO_BASE_MODE, l> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

void driver(
    const vector<double>& x,
    const vector<vector<double> >& xt1,
    vector<double>& y,
    vector<vector<double> >& yt1
```

```
) {
  const size_t n=x.size(), m=y.size();
  vector<DCO_TYPE> t1v_x(n), t1v_y(m);
  for (size_t i=0;i<n;i++) {
    value(t1v_x[i])=x[i];
    for (DCO_TAPE_INT j=0;j<l;j++)
      derivative(t1v_x[i])[j]=xt1[i][j];
  }
  f(t1v_x,t1v_y);
  for (size_t i=0;i<m;i++) {
    y[i]=value(t1v_y[i]);
    for (DCO_TAPE_INT j=0;j<l;j++)
      yt1[i][j]=derivative(t1v_y[i])[j];
  }
}
```

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

#include "driver.hpp"

int main() {
  cout.precision(15);
  const int m=2, n=4;
  vector<double> x(n),y(m);
  vector<vector<double> > xt1(n, vector<double>(n)), yt1(m, vector<double>(n));
  for (int i=0;i<n;i++) {
    x[i]=1;
    for (int j=0;j<n;j++) xt1[i][j]=0;
    xt1[i][i]=1;
  }
  driver(x,xt1,y,yt1);
  for (int j=0;j<m;j++)
    cout << "y[" << j << "]=" << y[j] << endl;
  for (int j=0;j<m;j++)
    for (int i=0;i<n;i++)
      cout << "y^{(1)}[" << j << "][" << i << "]="
     << yt1[j][i] << endl;
  return 0;
}
```

The following output is generated:

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

## 14.3 New dco/c++ Features

### 14.3.1 derivative

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

# Chapter 15

# First-Order Adjoint Mode: Vector Mode

## 15.1 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 $f : \mathbb{R}^n \to \mathbb{R}^m$, $\mathbf{y} = f(\mathbf{x})$, computes

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ X_{(1)} &:= X_{(1)} + \langle Y_{(1)} \cdot \frac{\partial f}{\partial \mathbf{x}} \rangle \equiv X_{(1)} + \left( \frac{\partial f}{\partial \mathbf{x}} \right)^T \cdot Y_{(1)}. \end{aligned} \tag{15.1}$$

## 15.2 Example

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

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

#include "dco.hpp"
using namespace dco;

const int l=5;

typedef double DCO_BASE_TYPE;
typedef ga1v<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_MODE::global_tape=DCO_TAPE_TYPE::create();
  int n=xv.size(), m=yv.size();
  vector<DCO_TYPE> x(n), y(m);
  for (int i=0;i<n;i++) {
    x[i]=xv[i];
    DCO_MODE::global_tape->register_variable(x[i]);
  }
  f(x,y);
  for (int i=0;i<m;i++) {
    DCO_MODE::global_tape->register_output_variable(y[i]);
    yv[i]=value(y[i]);
    for (int j=0;j<l;j++) derivative(y[i])[j] = ya[i][j];
  }
  for (int i=0;i<n;i++) {
    for (int j=0;j<l;j++) derivative(x[i])[j] = xa[i][j];
  }
  DCO_MODE::global_tape->interpret_adjoint();
  for (int i=0;i<n;i++) {
    for (int j=0;j<l;j++) xa[i][j]=derivative(x[i])[j];
  }
  for (int i=0;i<m;i++) {
    for (int j=0;j<l;j++) ya[i][j]=derivative(y[i])[j];
  }
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
  const int n=4, m=5; cout.precision(15);
  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;
  return 0;
}
```

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
```

## 15.3 New dco/c++ Features

None.

# Chapter 16

# Adjoint First-Order Scalar Mode: Multiple Tapes

## 16.1 Purpose

The `dco/c++` type `ga1sm<DCO_BASE_TYPE>::type` enables the use of multiple tapes in adjoint first-order scalar mode.

## 16.2 Example

For the same function $f : \mathbb{R}^4 \rightarrow \mathbb{R}^2$ used in Chapter 4 the driver computes (5.1) for $\mathbf{x} \hat{=}$ `xv`, $\mathbf{x}_{(1)} \hat{=}$ `xa`, $\mathbf{y} \hat{=}$ `yv`, and $\mathbf{y}_{(1)} \hat{=}$ `ya`.

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

#include "dco.hpp"
using namespace dco;

typedef double DCO_BASE_TYPE;
typedef ga1sm<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_TAPE_TYPE *tape=DCO_TAPE_TYPE::create();
  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]);
  DCO_TAPE_TYPE::remove(tape);
}
```

The main program initializes all variables on the right-hand side of (5.1) 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(15);
  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);
  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;
  return 0;
}
```

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
```

## 16.3 New dco/c++ Features

### 16.3.1 ga1sm<DCO_BASE_TYPE>

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

### 16.3.2 ga1sm<DCO_BASE_TYPE>::type

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

### 16.3.3 DCO_TAPE_TYPE::create

Tape creator returns pointer to local tape as DCO_TAPE_TYPE*.

### 16.3.4 DCO_TAPE_TYPE::remove

Deallocates local tape.

# Chapter 17

# Adjoint First-Order Vector Mode: Multiple Tapes

## 17.1 Purpose

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

## 17.2 Example

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

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

#include "dco.hpp"
using namespace dco;

const DCO_INTEGRAL_TAPE_INT 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_MODE::tape_t *tape=DCO_TAPE_TYPE::create();
  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]);
    for (int j=0;j<l;j++) derivative(y[i])[j] = ya[i][j];
  }
  for (size_t i=0;i<n;i++) {
    for (DCO_TAPE_INT j=0;j<l;j++) derivative(x[i])[j] = xa[i][j];
  }
  tape->interpret_adjoint();
  for (size_t i=0;i<n;i++) {
    for (DCO_TAPE_INT j=0;j<l;j++) xa[i][j]=derivative(x[i])[j];
  }
  for (size_t i=0;i<m;i++) {
    for (DCO_TAPE_INT j=0;j<l;j++) ya[i][j]=derivative(y[i])[j];
  }
  DCO_TAPE_TYPE::remove(tape);
}
```

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

#include "driver.hpp"

int main() {
  const int n=4, m=5; cout.precision(15);
  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;
  return 0;
}
```

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
```

## 17.3 New dco/c++ Features

None.

# Chapter 18

# First-Order Adjoint Mode: Basic Checkpointing

## 18.1 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 Chapter 19.

## 18.2 Example

```
#include "g_gap.hpp"

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

```
#include "g.hpp"

template<typename DCO_MODE>
void g_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
  typedef typename DCO_MODE::type DCO_TYPE;
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
  typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

  DCO_TAPE_POSITION_TYPE p0=DCO_MODE::global_tape->get_position();
  const int &n = D->template read_data<int>();
  const DCO_VALUE_TYPE &xv = D->template read_data<DCO_VALUE_TYPE>();
  DCO_TYPE x=xv;
  DCO_MODE::global_tape->register_variable(x);
  DCO_TYPE x_in=x;
  DCO_TAPE_POSITION_TYPE p1=DCO_MODE::global_tape->get_position();
  g(n,x);
```

```
  cerr << "ts1=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  derivative(x)=D->get_output_adjoint();
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p1);
  D->increment_input_adjoint(derivative(x_in));
  DCO_MODE::global_tape->reset_to(p0);
}

template<typename DCO_TYPE>
void g_make_gap(int n, DCO_TYPE &x) {
  typedef dco::mode<DCO_TYPE> DCO_MODE;
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;

  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();
  DCO_VALUE_TYPE xv=D->register_input(x);
  D->write_data(n); D->write_data(xv);
  g(n,xv);
  x=D->register_output(xv);
  DCO_MODE::global_tape->insert_callback(g_fill_gap<DCO_MODE>,D);
}
```

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

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

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

#include "dco.hpp"
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 int n, double& xv, double& xa) {
  DCO_TYPE x=xv;
  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(x);
  DCO_TYPE x_in=x;
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  f(n,x);
  derivative(x)=xa;
  cerr << "ts0=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
  xv=value(x);
```

```
  xa=derivative(x_in);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
    int n=10; cout.precision(15);
    double x=2.1,xa1=1.0;
    driver(n,x,xa1);
    cout << "x=" << x << endl;
    cout << "x_{(1)}=" << xa1 << endl;
    return 0;
}
```

The following output is generated for `n=10`:

```
ts0=0.000335693359375MB
ts1=0.0003204345703125MB

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

## 18.3 New dco/c++ Features

### 18.3.1 DCO_TAPE_TYPE::iterator_t

Position in tape.

### 18.3.2 DCO_TAPE_TYPE::get_position

Returns current position in associated tape.

### 18.3.3 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.

### 18.3.4 dco::mode<DCO_TYPE>

`DCO_MODE` for given `DCO_TYPE`.

### 18.3.5 DCO_MODE::value_t

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

### 18.3.6 DCO_MODE::external_adjoint_object_t

External adjoint object base class to be specialized for custom handling of *gaps*.

### 18.3.7 DCO_TAPE_TYPE::create_callback_object

Creates external adjoint object and returns pointer to it.

### 18.3.8 DCO_EAO_TYPE::register_input

Registers argument as an input to the *gap* and returns its value.

### 18.3.9 DCO_EAO_TYPE::write_data

Stores data of generic type `TYPE` required to fill the *gap*.

### 18.3.10 DCO_EAO_TYPE::register_output

Returns active variable with argument's value after its registration with the associated tape.

### 18.3.11 DCO_TAPE_TYPE::insert_callback

Inserts external adjoint object into tape alongside pointer to function for filling the *gap*.

Remark: This function needs to be called after having registered all in- and outputs.

### 18.3.12 DCO_EAO_TYPE::read_data

$i$th call returns generic data stored by $i$th call of `write_data`.

### 18.3.13 DCO_EAO_TYPE::get_output_adjoint

$i$th call returns adjoint of variable registered as output of *gap* by $i$th call of `register_output`.

### 18.3.14 DCO_EAO_TYPE::increment_input_adjoint

$i$th call adds value of argument to adjoint of variable registered as input of *gap* by $i$th call of `register_input`.

### 18.3.15 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.

### 18.3.16 DCO_TAPE_TYPE::get_tape_memory_size()

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

# Chapter 19

# Second-Order Adjoint Mode: Basic Checkpointing

## 19.1 Purpose

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

## 19.2 Example

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

#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, double& xv, double& xt2, double& xa1, double& xt2a1) {
  DCO_TYPE x;
  passive_value(x)=xv;
  derivative(value(x))=xt2;
  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(x);
  DCO_TYPE x_in=x;
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  f(n,x);
```

```
  xv=passive_value(x);
  xt2=derivative(value(x));
  value(derivative(x))=xa1;
  derivative(derivative(x))=xt2a1;
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
  xa1=value(derivative(x_in));
  xt2a1=derivative(derivative(x_in));
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
  cout.precision(15);
  int n=10;
  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;
  return 0;
}
```

The following output is generated:

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

## 19.3 New dco/c++ Features

None.

# Chapter 20

# First-Order Adjoint Mode: Checkpointing Evolutions Equidistantly

## 20.1 Purpose

This section illustrates the equidistant checkpointing of evolutions.

## 20.2 Example

```
#include "g_gap.hpp"

template<typename DCO_TYPE>
void f(int n, int m, DCO_TYPE& x) {
  for (int i=0;i<n;i+=m)
    g_make_gap(std::min(m,n-i),x);
}
```

```
#include "g.hpp"

template<typename DCO_MODE>
void g_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
  typedef typename DCO_MODE::type DCO_TYPE;
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
  typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

  DCO_TAPE_POSITION_TYPE p0=DCO_MODE::global_tape->get_position();
  const int &n=D->template read_data<int>();
  DCO_TYPE x=D->template read_data<DCO_VALUE_TYPE>();
  const int &c=D->template read_data<int>();
  DCO_MODE::global_tape->register_variable(x);
  DCO_TYPE x_in=x;
  DCO_TAPE_POSITION_TYPE p1=DCO_MODE::global_tape->get_position();
  g(n,x);
```

```
  cerr << "ts" << c << "="
       << dco::size_of(DCO_MODE::global_tape) << "MB" << endl;
  derivative(x)=D->get_output_adjoint();
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p1);
  D->increment_input_adjoint(derivative(x_in));
  DCO_MODE::global_tape->reset_to(p0);
}

template<typename DCO_TYPE>
void g_make_gap(int n, DCO_TYPE &x) {
  typedef dco::mode<DCO_TYPE> DCO_MODE;
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
  static int c=1;

  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();
  DCO_VALUE_TYPE xp=D->register_input(x);
  D->write_data(n); D->write_data(xp); D->write_data(c++);
  g(n,xp);
  x=D->register_output(xp);
  DCO_MODE::global_tape->insert_callback(g_fill_gap<DCO_MODE>,D);
}
```

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

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

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

#include "dco.hpp"
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 int n, const int m, double& xv, double& xa) {
  DCO_TYPE x=xv;
  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(x);
  DCO_TYPE x_in=x;
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  f(n,m,x);
  cerr << "ts0=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
```

```
  derivative(x)=xa;
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
  xv=value(x);
  xa=derivative(x_in);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
    cout.precision(15);
    int n=10; int m=2;
    double xv=2.1, xa=1.0;
    driver(n,m,xv,xa);
    cout << "x=" << xv << endl;
    cout << "x_{(1)}=" << xa << endl;
    return 0;
}
```

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
```

## 20.3 New dco/c++ Features

None.

# Chapter 21

# Second-Order Adjoint Mode: Checkpointing Evolutions Equidistantly

## 21.1 Purpose

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

## 21.2 Example

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

#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;
// instantiate global tape
DCO_TAPE_TYPE*& DCO_TAPE_POINTER=DCO_MODE::global_tape;

#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;
  v = value(t2s_a1s_x);
  value(v) = x;
```

```
  derivative(v) = xt2;
  value(t2s_a1s_x) = v;
  DCO_TAPE_POINTER=DCO_TAPE_TYPE::create();
  DCO_TAPE_POINTER->register_variable(t2s_a1s_x);
  DCO_TYPE x_indep=t2s_a1s_x;
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  f(n,m,t2s_a1s_x);
  cerr << "ts0=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  v = derivative(t2s_a1s_x);
  value(v) = xa1;
  derivative(v) = xt2a1;
  derivative(t2s_a1s_x) = v;
  DCO_TAPE_POINTER->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);
  DCO_TAPE_TYPE::remove(DCO_TAPE_POINTER);
}
```

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

#include "driver.hpp"

int main() {
    int n=10; int m=2; cout.precision(15);
    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;
    return 0;
}
```

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
```

## 21.3 New dco/c++ Features

None.

# Chapter 22

# First-Order Adjoint Mode: Checkpointing Evolutions Recursively

## 22.1 Purpose

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

## 22.2 Example

```
#include "g_gap.hpp"

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

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

enum RUN_MODE {
  CHECKPOINT_ARGUMENTS_AND_RUN_PASSIVELY,
  GENERATE_TAPE
};

template<typename DCO_MODE>
class my_external_adjoint_object_t : public DCO_MODE::external_adjoint_object_t{
```

```
  public:
    typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
    static stack<pair<int,DCO_VALUE_TYPE> > state;
    static int stride;
    my_external_adjoint_object_t(pair<int,int> p) : DCO_MODE::external_adjoint_object_t(p) {}
};

template<typename DCO_MODE>
stack<pair<int,typename my_external_adjoint_object_t<DCO_MODE>::DCO_VALUE_TYPE> > my_external_adjoint_object_t<DCO_MODE>::state;
template<typename DCO_MODE>
int my_external_adjoint_object_t<DCO_MODE>::stride;

template<typename DCO_MODE>
void g_fill_gap(my_external_adjoint_object_t<DCO_MODE> *D);

template<typename DCO_MODE>
void g_make_gap(int from, int to, int stride,
    typename DCO_MODE::type &x, RUN_MODE m=GENERATE_TAPE) {
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  typedef my_external_adjoint_object_t<DCO_MODE> DCO_EAO_TYPE;

  if (m==CHECKPOINT_ARGUMENTS_AND_RUN_PASSIVELY) {
    cout << "STORE CHECKPOINT FOR SECTION "
   << from << " ... " << to-1 << endl;
    DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<DCO_EAO_TYPE>(make_pair(1,1));
    DCO_VALUE_TYPE xv=D->register_input(x);

    // write argument checkpoint (FIFO)
    D->write_data(from);
    D->write_data(to);
    if (D->state.empty()||from!=D->state.top().first) {
      cout << "PUSHING (" << from << ", " << xv << ")" << endl;
      D->state.push(make_pair(from,xv));
    }

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

    // register output x with tape and store its
    // position for retrieval of incoming adjoint required
    // during interpretation
    x=D->register_output(xv);
    DCO_MODE::global_tape->insert_callback(g_fill_gap<DCO_MODE>,D);

  } else if (m==GENERATE_TAPE) {
    cout << "GENERATE TAPE FOR SECTION "
   << from << " ... " << to-1 << endl;
    stringstream s;
    s << from << "GENERATE_TAPE_" << to-1 << ".dot";
```

```
    DCO_MODE::global_tape->write_to_dot(s.str());
    my_external_adjoint_object_t<DCO_MODE>::stride=stride;
    // in taping mode, the interval is subdivided further if
    // its length exceeds the desired stride ...
    if (to-from>stride) {
      g_make_gap<DCO_MODE>(from,from+(to-from)/2,stride,x,
         CHECKPOINT_ARGUMENTS_AND_RUN_PASSIVELY);
      g_make_gap<DCO_MODE>(from+(to-from)/2,to,stride,x,
         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++) x=sin(x);
  }

  if (dco::size_of(DCO_MODE::global_tape) > max_tape_size)
    max_tape_size = dco::size_of(DCO_MODE::global_tape);
}

template<typename DCO_MODE>
void g_fill_gap(
   my_external_adjoint_object_t<DCO_MODE> *D
) {
  typedef typename DCO_MODE::type DCO_TYPE;
  typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
  typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

  const int &from=D->template read_data<int>();
  const int &to=D->template read_data<int>();
  cout << "top=" << D->state.top().second << endl;
  cout << "RESTORE CHECKPOINT FOR SECTION "
       << from << " ... " << to-1 << endl;
  DCO_TYPE x=D->state.top().second;
  DCO_MODE::global_tape->register_variable(x);
  DCO_TYPE x_in=x;
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  g_make_gap<DCO_MODE>(from,to,D->stride,x,GENERATE_TAPE);
  derivative(x)=D->get_output_adjoint();
  cout << "INTERPRET SECTION "
       << from << " ... " << to-1 << endl;
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
  D->increment_input_adjoint(derivative(x_in));
  if (to-from<=D->stride) {
    cout << "poping " << D->state.top().first << ", " << D->state.top().second
        << endl;
    D->state.pop();
  }
}
```

```
#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>
using namespace std;

#include "dco.hpp"
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_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(x);
  DCO_TYPE x_in=x;

  f<DCO_MODE>(0,n,stride,x);

  DCO_MODE::global_tape->register_output_variable(x);
  derivative(x)=1;

#ifdef VERBOSE
  cout << "INTERPRET SECTION: 0 ... " << n-1 << endl;
```

```
#endif
  DCO_MODE::global_tape->write_to_dot("driver.dot");
  DCO_MODE::global_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
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
  cout.precision(15);
  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;
  return 0;
}
```

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
```

## 22.3 New dco/c++ Features

### 22.3.1 my_external_adjoint_object_t

Specialization of DCO_MODE::external_adjoint_object_t.

# Chapter 23

# Second-Order Adjoint Mode: Checkpointing Evolutions Recursively

## 23.1 Purpose

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

## 23.2 Example

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

#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_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_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(t2s_a1s_x);
  DCO_TYPE x_indep=t2s_a1s_x;
  DCO_TAPE_ITERATOR_TYPE to_be_reset_to=DCO_MODE::global_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;
  DCO_MODE::global_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);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
    int n = 10; int stride=2; cout.precision(15);
    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;
    return 0;
}
```

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
```

## 23.3 New dco/c++ Features

None.

# Chapter 24

# First-Order Adjoint Mode: Checkpointing Ensembles

## 24.1 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 Chapter 25.

## 24.2 Example

```
#include "g_gap.hpp"

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=rand();
    g_make_gap(m,x,r,y);
    sum+=y;
  }
  y=sum/n;
}
```

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

template<typename DCO_MODE>
void g_fill_gap(typename DCO_MODE::external_adjoint_object_t*D) {
  typedef typename DCO_MODE::type DCO_TYPE;
  typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
  typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

  const int &m=D->template read_data<int>();
```

```
  const DCO_TYPE* const &x_p=
    D->template read_data<const DCO_TYPE*>();
  const double &r=D->template read_data<const double>();
  const int &c=D->template read_data<const int>();
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  DCO_TYPE y;
  g(m,*x_p,r,y);
  cerr << "ts" << c << "="
       << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  derivative(y)=D->get_output_adjoint();
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
}

template<typename DCO_TYPE>
void g_make_gap(int m, const DCO_TYPE& x, const double& r, DCO_TYPE& y) {
  typedef dco::mode<DCO_TYPE> DCO_MODE;
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
  static int c=1;

  DCO_EAO_TYPE *D=DCO_MODE::global_tape->
    template create_callback_object<DCO_EAO_TYPE>();
  D->write_data(m); D->write_data(&x); D->write_data(r); D->write_data(c++);
  DCO_VALUE_TYPE yp;
  g(m,value(x),r,yp);
  y=D->register_output(yp,DCO_MODE::global_tape);
  DCO_MODE::global_tape->
    insert_callback(g_fill_gap<DCO_MODE>,D);
}
```

```
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_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(x);
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  f(n,m,x,y);
```

```
  cerr << "ts0=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  yv=value(y);
  DCO_MODE::global_tape->register_output_variable(y);
  derivative(y)=ya; derivative(x)=xa;
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
  //DCO_MODE::global_tape->interpret_adjoint();
  xa=derivative(x);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
  cout.precision(15);
  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;
  return 0;
}
```

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
```

## 24.3 New dco/c++ Features

None.

# Chapter 25

# Second-Order Adjoint Mode: Checkpointing Ensembles

## 25.1 Purpose

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

## 25.2 Example

```
#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_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(x);
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  f(n,m,x,y);
  cerr << "ts0=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  yv=passive_value(y);
  yt2=derivative(value(y));
  DCO_MODE::global_tape->register_output_variable(y);
  value(derivative(y))=ya1;
  derivative(derivative(y))=ya1t2;
```

```
  value(derivative(x))=xa1;
  derivative(derivative(x))=xa1t2;
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
  xa1=value(derivative(x));
  xa1t2=derivative(derivative(x));
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
  cout.precision(15);
  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;
  return 0;
}
```

The following output is generated:

```
ts0=0.001678466796875MB
ts0=0.001678466796875MB
ts10=0.002349853515625MB
ts9=0.0021896362304687 5MB
ts8=0.0020294189453125MB
ts7=0.0018692016601562 5MB
ts6=0.001708984375MB
ts5=0.0015487670898437 5MB
ts4=0.0013885498046875MB
ts3=0.0012283325195312 5MB
ts2=0.001068115234375MB
ts1=0.0009307861328125MB

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

## 25.3 New dco/c++ Features

None.

# Chapter 26

# First-Order Adjoint Mode: Embedding Adjoint Source Code

## 26.1 Purpose

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

## 26.2 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"

template<typename DCO_MODE>
void g_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  const int &n=D->template read_data<int>();
  vector<DCO_VALUE_TYPE> xa1(n);
  DCO_VALUE_TYPE ya1=D->get_output_adjoint();
  g_a1s<DCO_VALUE_TYPE>(xa1,ya1);
  for (int i=0;i<n;i++) D->increment_input_adjoint(xa1[i]);
}

template<typename DCO_TYPE>
```

```
void g_make_gap(std::vector<DCO_TYPE>& x, DCO_TYPE &y) {
    typedef dco::mode<DCO_TYPE> DCO_MODE;
    typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
    typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
    DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();

    size_t n=x.size();
    vector<DCO_VALUE_TYPE> xv(n);
    for (size_t i=0;i<n;i++) xv[i]=D->register_input(x[i]);
    D->write_data(n);
    DCO_VALUE_TYPE yv;
    g<DCO_VALUE_TYPE>(xv,yv);
    y=D->register_output(yv);
    DCO_MODE::global_tape->insert_callback(g_fill_gap<DCO_MODE>,D);
}
```

```
#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>
using namespace std;

#include "dco.hpp"
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 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_MODE::global_tape=DCO_TAPE_TYPE::create();
  for (size_t i=0;i<n;i++) {
    DCO_MODE::global_tape->register_variable(x[i]);
    value(x[i])=xv[i];
    derivative(x[i])=xa1[i];
    x_in[i]=x[i];
  }
  DCO_TYPE y;
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  f(x,y);
  cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  yv=value(y);
  derivative(y)=ya1;
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
  for (size_t i=0;i<n;i++)
    xa1[i]=derivative(x_in[i]);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
  cout.precision(15);
  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;

  return 0;
}
```

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
...
```

## 26.3 New dco/c++ Features

None.

# Chapter 27

# Second-Order Adjoint Mode: Embedding First-Order Adjoint Code

## 27.1 Purpose

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

## 27.2 Example

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

#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 "../ga1s_external_manual/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
```

```
) {
  ga1s<DCO_BASE_TYPE>::global_tape=ga1s<DCO_BASE_TYPE>::tape_t::create();
  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++) {
    ga1s<DCO_BASE_TYPE>::global_tape->register_variable(x[i]);
    value(value(x[i]))=xv[i];
    derivative(value(x[i]))=xt2[i];
    x_in[i]=x[i];
  }
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  f(x,y);
  cerr << "ts=" << dco::size_of(DCO_MODE::global_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));
  ga1s<DCO_BASE_TYPE>::global_tape->register_output_variable(y);
  value(derivative(y))=ya1;
  derivative(derivative(y))=ya1t2;
  ga1s<DCO_BASE_TYPE>::global_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]));
  }
  ga1s<DCO_BASE_TYPE>::tape_t::remove(ga1s<DCO_BASE_TYPE>::global_tape);
}
```

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

#include "driver.hpp"

int main() {
    cout.precision(15);
    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;
```

```
    return 0;
}
```

The following output is generated:

```
ts=0.000770568847656250MB

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
...
```

## 27.3 New dco/c++ Features

None.

# Chapter 28

# First-Order Adjoint Mode: Local Preaccumulation Using First-Order Tangent Code

## 28.1 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.

## 28.2 Example

```
#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);
}
```

```
#include "g.hpp"

template<typename DCO_BASE_TYPE>
void gt1(int n, DCO_BASE_TYPE& x, DCO_BASE_TYPE& xt1) {
  for (int i=0;i<n;i++) {
    xt1=cos(x)*xt1;
    x=sin(x);
  }
}

template<typename DCO_MODE>
void g_fill_gap(
   typename DCO_MODE::external_adjoint_object_t *D
) {
```

```
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;

  const DCO_VALUE_TYPE &p=D->template read_data<DCO_VALUE_TYPE>();
  DCO_VALUE_TYPE r=D->get_output_adjoint();
  D->increment_input_adjoint(r*p);
}

template<typename DCO_TYPE>
void g_make_gap(int n, DCO_TYPE &x) {
    typedef typename dco::mode<DCO_TYPE> DCO_MODE;
    typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
    typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
    DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
    DCO_VALUE_TYPE xv=D->register_input(x);
    DCO_VALUE_TYPE xt1=1.;
    gt1(n,xv,xt1);
    D->write_data(xt1);
    x=D->register_output(xv);
    DCO_MODE::global_tape->insert_callback(g_fill_gap<DCO_MODE>,D);
}
```

```
#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>
using namespace std;

#include "dco.hpp"
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 int n, double& xv, double& xa) {
  DCO_TYPE x=xv;
  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(x);
  DCO_TYPE x_in=x;
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  f(n,x);
  cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
```

```
  derivative(x)=xa;
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
  xv=value(x);
  xa=derivative(x_in);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
  cout.precision(15);
  int n=10;
  double x=2.1,xa1=1.0;
  driver(n,x,xa1);
  cout << "x=" << x << endl;
  cout << "x_{(1)}=" << xa1 << endl;
  return 0;
}
```

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
```

## 28.3 New dco/c++ Features

None.

# Chapter 29

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

## 29.1 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.

## 29.2 Example

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

#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, double& xv, double& xt2, double& xa1, double& xt2a1) {
  DCO_TYPE x;
  passive_value(x)=xv;
  derivative(value(x))=xt2;
  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(x);
  DCO_TYPE x_in=x;
```

```
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  f(n,x);
  cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  xv=passive_value(x);
  xt2=derivative(value(x));
  value(derivative(x))=xa1;
  derivative(derivative(x))=xt2a1;
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
  xa1=value(derivative(x_in));
  xt2a1=derivative(derivative(x_in));
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
    int n=10; cout.precision(15);
    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;
    return 0;
}
```

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
```

## 29.3 New dco/c++ Features

None.

# Chapter 30

# First-Order Adjoint Mode: Edge Insertion for Preaccumulation of Local Gradients

## 30.1 Purpose

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

## 30.2 Example

```
#include "g_gap.hpp"

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

```
#include "g.hpp"

template<typename DCO_BASE_TYPE>
void gt1(int n, DCO_BASE_TYPE& x, DCO_BASE_TYPE& xt1) {
  for (int i=0;i<n;i++) {
    xt1=cos(x)*xt1;
    x=sin(x);
  }
}

template<typename DCO_TYPE>
void g_make_gap(int n, DCO_TYPE &x) {
    typedef typename dco::mode<DCO_TYPE> DCO_MODE;
    typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
    typedef typename DCO_MODE::local_gradient_t local_gradient_t;
```

```

    DCO_TYPE xc = x; // copy required because output=input
    local_gradient_t y1o(x);
    // new tape index for output x

    DCO_VALUE_TYPE xt1=1.;
    gt1(n,value(x),xt1); // value of x changed

    y1o.put(xc, xt1); // add edge from new output to input
}
```

```
#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>
using namespace std;

#include "dco.hpp"
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 int n, double& xv, double& xa) {
  DCO_TYPE x=xv;
  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(x);
  DCO_TYPE x_in=x;
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  f(n,x);
  cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  derivative(x)=xa;
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
  xv=value(x);
  xa=derivative(x_in);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

```
#include "driver.hpp"

int main() {
  cout.precision(15);
  int n=10;
  double x=2.1,xa1=1.0;
  driver(n,x,xa1);
  cout << "x=" << x << endl;
  cout << "x_{(1)}=" << xa1 << endl;
  return 0;
}
```

The following output is generated:

```
ts=0.000328063964843 75MB

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

Calling g instead of g_make_gap in f yields

```
ts=0.000404357910156 25MB

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

## 30.3 New dco/c++ Features

### 30.3.1 DCO_MODE::local_gradient_t

Local gradient type associated with DCO_MODE.

### 30.3.2 DCO_TAPE_TYPE::create_local_gradient_object

Returns local gradient of size passed as second argument of variable passed as first argument.

### 30.3.3 DCO_LOCAL_GRADIENT_TYPE::put

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

### 30.3.4 DCO_LOCAL_GRADIENT_TYPE::finalize

Adds local gradient to tape.

# Chapter 31

# Second-Order Adjoint Mode: Edge Insertion for Preaccumulation of Local Gradients

## 31.1 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.

## 31.2 Example

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

#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, double& xv, double& xt2, double& xa1, double& xt2a1) {
  DCO_TYPE x;
  passive_value(x)=xv;
  derivative(value(x))=xt2;
```

```
  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(x);
  DCO_TYPE x_in=x;
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
  f(n,x);
  cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  xv=passive_value(x);
  xt2=derivative(value(x));
  value(derivative(x))=xa1;
  derivative(derivative(x))=xt2a1;
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
  xa1=value(derivative(x_in));
  xt2a1=derivative(derivative(x_in));
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
    int n=10; cout.precision(15);
    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;
    return 0;
}
```

The following output is generated:

```
ts=0.00052642822265625MB
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
```

## 31.3 New dco/c++ Features

None.

# Chapter 32

# First-Order Adjoint Mode: User-Defined Adjoints

## 32.1 Purpose

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

## 32.2 Example

```
#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;
}
```

```
#include "g.hpp"

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

template<typename DCO_MODE>
void g_fill_gap(typename DCO_MODE::external_adjoint_object_t* D) {
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  const DCO_VALUE_TYPE &xv = D->template read_data<DCO_VALUE_TYPE>();
  DCO_VALUE_TYPE xa,pa;
  xa=D->get_output_adjoint();
  a1s_g(pa,xv,xa);
  D->increment_input_adjoint(pa);
}
```

```
template<typename DCO_TYPE>
void g_make_gap(const DCO_TYPE& p, DCO_TYPE& x) {
  typedef mode<DCO_TYPE> DCO_MODE;
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();
  DCO_VALUE_TYPE xv=value(x),pv;
  pv=D->register_input(p);
  g(pv,xv);
  x=D->register_output(xv);
  D->write_data(xv);
  DCO_MODE::global_tape->insert_callback(g_fill_gap<DCO_MODE>,D);
}
```

```
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);
  }
}
```

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

#include "dco.hpp"
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_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_TYPE x=xv,p=pv;
  DCO_MODE::global_tape->register_variable(p);
  f(p,x);
  cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  xv=value(x);
  DCO_MODE::global_tape->register_output_variable(x);
  derivative(x)=xa;
  DCO_MODE::global_tape->interpret_adjoint();
  pa=derivative(p);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
```

```
}
```

```
#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(15);
  cout << "x=" << xv << endl;
  cout << "p_{(1)}=" << pa << endl;
  return 0;
}
```

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
```

## 32.3 New dco/c++ Features

None.

# Chapter 33

# Second-Order Adjoint Mode: User-Defined First-Order Adjoints

## 33.1 Purpose

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

## 33.2 Example

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

#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;

#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_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(p);
  f(p,x);
  cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  xv=passive_value(x);
  xt2=derivative(value(x));
```

```
  DCO_MODE::global_tape->register_output_variable(x);
  value(derivative(x))=xa1;
  derivative(derivative(x))=xa1t2;
  value(derivative(p))=pa1;
  derivative(derivative(p))=pa1t2;
  DCO_MODE::global_tape->interpret_adjoint();
  pa1=value(derivative(p));
  pa1t2=derivative(derivative(p));
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

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

#include "driver.hpp"

int main() {
  cout.precision(15);
  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;
  return 0;
}
```

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
```

## 33.3 New dco/c++ Features

None.

# Chapter 34

# First-Order Adjoint Mode: Local Jacobian Preaccumulation

## 34.1 Purpose

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

## 34.2 Example

### 34.2.1 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.

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

#include "dco.hpp"

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=1000, s=100;
16   const double x0=1;
17   vector<DCO_TYPE> p(n,1);
18   DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
19   for (int i=0;i<n;i++)
20     DCO_MODE::global_tape->register_variable(p[i]);
21   DCO_TYPE x=x0;
22   DCO_MODE::jacobian_preaccumulator_t jp(DCO_MODE::global_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   DCO_MODE::global_tape->register_output_variable(x);
30   dco::derivative(x)=1;
31   DCO_MODE::global_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   DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
35 }
```

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.

### 34.2.2 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
```

## 34.3 New dco/c++ Features

### 34.3.1 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.

### 34.3.2 jacobian_preaccumulator_t::start()

Start recording of tape used for preaccumulation.

### 34.3.3 jacobian_preaccumulator_t::register_output(DCO_TYPE&)

Register active outputs of computation subject to preaccumulation.

### 34.3.4 jacobian_preaccumulator_t::finish()

Preaccumulate local Jacobian and integrate into associated tape.

# Chapter 35

# Second-Order Adjoint Mode: Local Jacobian Preaccumulation

## 35.1 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.

## 35.2 Example

The example from Section 34.2 is extended towards second-order adjoints.

### 35.2.1 Example Text

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

#include "dco.hpp"

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=100, s=10;
  const double x0=1;
  vector<DCO_TYPE> p(n,1);
  vector<vector<double> > H(n,vector<double>(n,0));
  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  for (int i=0;i<n;i++)
```

```
    DCO_MODE::global_tape->register_variable(p[i]);
  DCO_TAPE_TYPE::position_t tpos=DCO_MODE::global_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(DCO_MODE::global_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();
    }
    DCO_MODE::global_tape->register_output_variable(x);
    dco::derivative(x)=1;
    DCO_MODE::global_tape->interpret_adjoint();
    for (int i=0;i<n;i++) {
      H[i][j]=dco::derivative(dco::derivative(p[i]));
      dco::derivative(p[i])=0;
    }
    DCO_MODE::global_tape->reset_to(tpos);
    dco::derivative(dco::value(p[j]))=0;
  }
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
  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;
}
```

### 35.2.2 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
```

## 35.3 New dco/c++ Features

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

# Chapter 36

# First-Order Adjoint Mode: Multiple Adjoint Vectors

## 36.1 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 Chapter 8.

## 36.2 Example

The implicit Euler case study from Section 34.2 is considered.

## 36.3 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>
using namespace std;
#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"

void driver() {
  const int n=1000;
  double x0=1;
  vector<DCO_TYPE> p(n,1);
  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  for (int i=0;i<n;i++)
```

```
19     DCO_MODE::global_tape->register_variable(p[i]);
20   DCO_TYPE x=x0;
21   euler(0,n,x,p);
22   DCO_MODE::global_tape->register_output_variable(x);
23 
24   omp_set_num_threads(2);
25   float adjoint_float;
26   double adjoint_double;
27 #pragma omp parallel
28   {
29     switch (omp_get_thread_num()) {
30     case 0: {
31       dco::adjoint_vector<DCO_TAPE_TYPE,float>
32         av_float(DCO_MODE::global_tape);
33       dco::derivative(x, av_float)=1;
34       av_float.interpret_adjoint();
35       adjoint_float = dco::derivative(p[n/2], av_float);
36       break;
37     }
38     case 1: {
39       dco::adjoint_vector<DCO_TAPE_TYPE,double>
40         av_double(DCO_MODE::global_tape);
41       dco::derivative(x, av_double)=1;
42       av_double.interpret_adjoint();
43       adjoint_double = dco::derivative(p[n/2], av_double);
44     }
45     }
46   }
47 
48   cout << "float " << "dx/dp=" << adjoint_float << endl;
49   cout << "double " << "dx/dp=" << adjoint_double << endl;
50 
51   DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
52 }
```

A single tape is recorded in lines 17-22 prior to entering the parallel regions in line 27. Thread 0 allocates an adjoint vector over `DCO_ADJOINT_TYPE=float` in lines 31-32. Similarly, an adjoint vector over `DCO_ADJOINT_TYPE=double` is allocated by thread 1 in lines 39-40. Access to derivative components requires specification of the associated adjoint vector (e.g., lines 35 and 43). Interpretation is called for the individual adjoint vectors in lines 34 und 42 followed by printing the respective results.

## 36.4 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.

## 36.5 New dco/c++ Features

### 36.5.1 `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 argument.

### 36.5.2 `derivative(DCO_T&,DCO_ADJOINT_VECTOR_TYPE&)`

Derivative access requires specification of the associated adjoint vector.

### 36.5.3 `DCO_ADJOINT_VECTOR_TYPE::interpret_adjoint()`

Interpreters are called directly on the adjoint vector.

# Chapter 37

# Modulo Adjoint Propagation

## 37.1 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`.

## 37.2 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 above optimization. (Thanks to M. Towara)

Please check memory decrease with

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

see Sec. 3.1.6.

## 37.3 Example

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

$$\frac{dx}{dt} = f(x(t,p),t,p) \tag{37.1}$$

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_MODE::global_tape=DCO_TAPE_TYPE::create();
  DCO_MODE::global_tape->register_variable(p);

  f(n,x,p);

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

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

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

  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
  return 0;
}
```

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.

## 37.4 New dco/c++ Features

### 37.4.1 ga1s_mod<DCO_BASE_TYPE>

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

# Chapter 38

# Sparse Tape Interpretation

## 38.1 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_M::global_tape=DCO_TAPE_T::create();
  // record
  DCO_M::global_tape->sparse_interpret() = true;
  DCO_M::global_tape->interpret_adjoint();

  DCO_M::global_tape->zero_adjoints();
  DCO_M::global_tape->sparse_interpret() = false;
  DCO_M::global_tape->interpret_adjoint();
...
```

## 38.2 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_MODE::global_tape = DCO_TAPE_TYPE::create();
  DCO_TYPE x(0.0);
```

```

  DCO_MODE::global_tape->register_variable(x);

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

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

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

  DCO_MODE::global_tape->zero_adjoints();
  DCO_MODE::global_tape->sparse_interpret() = true;

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

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

  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}
```

## 38.3 New dco/c++ Features

### 38.3.1 TAPE_T::sparse_interpret()

Enable and disable sparse interpretation.

# Chapter 39

# Logging

## 39.1 Purpose

The logging might be useful for information and debugging purposes.

Logging has different levels of verbosity. The maximal 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 maximal 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`.

## 39.2 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`.

## 39.3 New dco/c++ Features

### 39.3.1 enum dco::log_level_enum

Different verbosity levels, defined as

```
enum log_level_enum {logERROR, logWARNING, logINFO, logDEBUG, logDEBUG1,
    logDEBUG2, logDEBUG3, logDEBUG4};
```

### 39.3.2 dco::log_level_enum& dco::logger::level()

Set/get the current logging level.

### 39.3.3 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")`.

# 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 the user a survey 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 given DCO_BASE_TYPE.

```
typedef gt1s<DCO_BASE_TYPE> DCO_GT1S_MODE;
```

— The tangent first-order scalar type for given DCO_GT1S_MODE.

```
typedef DCO_GT1S_MODE::type DCO_GT1S_TYPE;
```

— The adjoint first-order scalar mode over given DCO_BASE_TYPE.

```
typedef ga1s<DCO_BASE_TYPE> DCO_GA1S_MODE;
```

— The adjoint first-order scalar mode over 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_M::global_tape=DCO_TAPE_T::create(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_M::global_tape=DCO_TAPE_T::create(o);
```

— The adjoint first-order scalar type for given `DCO_GA1S_MODE`.

```
typedef DCO_GA1S_MODE::type DCO_GA1S_TYPE;
```

— The tangent first-order vector mode over given `DCO_BASE_TYPE` with vector length `v_size` and activity analysis `with_activity` switched on or off.

```
typedef gt1v< DCO_BASE_TYPE, v_size=1, with_activity=true > DCO_GT1V_MODE;
```

— The tangent first-order vector type for given `DCO_GT1V_MODE`.

```
typedef DCO_GT1V_MODE::type DCO_GT1V_TYPE;
```

— The adjoint first-order vector mode over given `DCO_BASE_TYPE` with vector length `v_size`.

```
typedef ga1v< DCO_BASE_TYPE, v_size=1 > DCO_GA1V_MODE;
```

— The adjoint first-order vector type for given `DCO_GA1V_MODE`.

```
typedef DCO_GA1V_MODE::type DCO_GA1V_TYPE;
```

— The adjoint first-order scalar mode with support for multiple tapes over given `DCO_BASE_TYPE`.

```
typedef ga1sm< DCO_BASE_TYPE > DCO_GA1SM_MODE;
```

— The adjoint first-order scalar type with support for multiple tapes for given `DCO_GA1SM_MODE`.

```
typedef DCO_GA1SM_MODE::type DCO_GA1SM_TYPE;
```

— The adjoint first-order scalar mode over 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 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 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 given DCO_BASE_TYPE.

```
typedef ga1vm_mod< DCO_BASE_TYPE > DCO_GA1vM_MOD_MODE;
```

From now on:

- DCO_MODE $\in$ DCO_x_MODE with x $\in$ {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;
```

— Returns [read-only] reference to passive value of x (usually **double**); supports std::vector.

```
[const] double& 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 following access routines only work for

— scalar modes, where DCO_VALUE_TYPE = DCO_DERIVATIVE_TYPE, and

— vector modes, where DCO_VALUE_TYPE = the base type of DCO_DERIVATIVE_TYPE.

They are present for backward compatibility. Users are encouraged to use the passive_value, value, and derivative routines to set and get individual components from dco variables.

```
void  DCO_GT1S_MODE::get ( const DCO_GT1S_TYPE          &x,
                                 DCO_GT1S_MODE::value_t &v, int w=0);
void  DCO_GT1S_MODE::set (       DCO_GT1S_TYPE          &x,
                           const DCO_GT1S_MODE::value_t &v, int w=0);
void  DCO_GA1S_MODE::get ( const DCO_GA1S_TYPE          &x,
                                 DCO_GA1S_MODE::value_t &v, int w=0);
void  DCO_GA1S_MODE::set (       DCO_GA1S_TYPE          &x,
                           const DCO_GA1S_MODE::value_t &v, int w=0);
void DCO_GA1SM_MODE::get ( const DCO_GA1SM_TYPE         &x,
                                 DCO_GA1SM_MODE::value_t &v, int w=0);
void DCO_GA1SM_MODE::set (       DCO_GA1SM_TYPE         &x,
                           const DCO_GA1SM_MODE::value_t &v, int w=0);
void  DCO_GT1V_MODE::get ( const DCO_GT1V_TYPE          &x,
                                 DCO_GT1V_MODE::value_t &v, int w=0, int i=0);
void  DCO_GT1V_MODE::set (       DCO_GT1V_TYPE          &x,
                           const DCO_GT1V_MODE::value_t &v, int w=0, int i=0);
void  DCO_GA1V_MODE::get ( const DCO_TYPE               &x,
                                 DCO_BASE_TYPE          &v, int w=0, int i=0);
void  DCO_GA1V_MODE::set (       DCO_TYPE               &x,
                           const DCO_BASE_TYPE          &v, int w=0, int i=0);
```

— 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.

— The global tape.

```
DCO_TAPE_TYPE* DCO_MODE::global_tape;
```

— Creates tape and returns pointer to it.

```
static DCO_TAPE_TYPE* DCO_TAPE_TYPE::create ();
```

— Deallocates tape pointed at by `t` and sets `t = 0`.

```
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.

```
void DCO_TAPE_TYPE::interpret_adjoint ();
```

— Runs tape interpreter from tape position 'from' to first entry.

```
void DCO_TAPE_TYPE::interpret_adjoint_from (
  const DCO_TAPE_POSITION_TYPE &from
);
```

— Runs tape interpreter from last entry to tape position 'to'.

```
void DCO_TAPE_TYPE::interpret_adjoint_to (
  const DCO_TAPE_POSITION_TYPE &to
);
```

— Runs tape interpreter from tape position 'from' to tape position 'to'.

```
void DCO_TAPE_TYPE::interpret_adjoint_from_to (
  const DCO_TAPE_POSITION_TYPE &from,
  const DCO_TAPE_POSITION_TYPE &to
);
```

— 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'.

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

```
void DCO_TAPE_TYPE::interpret_adjoint_and_reset_to (
  const DCO_TAPE_POSITION_TYPE &to
);
```

— Runs tape interpreter from tape position 'from' to tape position 'to' and sets corresponding adjoints to zero.

```
void DCO_TAPE_TYPE::interpret_adjoint_and_zero_adjoints_from_to (
  const DCO_TAPE_POSITION_TYPE &from,
  const DCO_TAPE_POSITION_TYPE &to
);
```

— Runs tape interpreter from current position to tape position 'to' and sets corresponding adjoints to zero.

```
void DCO_TAPE_TYPE::interpret_adjoint_and_zero_adjoints_to (
  const DCO_TAPE_POSITION_TYPE &to
);
```

— 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
);
```

— External adjoint object base class to be derived from for non-standard handling of *gaps*.

```
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> ();
```

— Inserts external adjoint object `eao` into tape alongside pointer to function `fill_gap`.

Remark: This function needs to be called after having registered all in- and outputs.

```
void DCO_TAPE_TYPE::insert_callback (
  void (*fill_gap)(DCO_EAO_TYPE*),
  DCO_EAO_TYPE *eao
);
```

— Returns active variable with value `v` after registration with tape `t`.

```
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`.

```
DCO_BASE_TYPE DCO_EAO_TYPE::get_output_adjoint ();
```

— Registers x as an input to the *gap* and returns its value.

```
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.

```
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.

```
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.

```
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 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 local gradient into tape.

```
void DCO_LOCAL_GRADIENT_TYPE::finalize();
```

— 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 );
```

— Object carrying options for tape creation; is **void** for tangent modes.

```
typedef DCO_MODE::tape_options_t DCO_TAPE_OPTIONS_TYPE;
```

— Returns chunk size in byte (for adjoint modes only).

```
size_t DCO_TAPE_OPTIONS_TYPE::chunk_size_in_byte() const;
```

— Returns blob size in byte (for adjoint modes only).

```
size_t DCO_TAPE_OPTIONS_TYPE::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

$$\min_{c \in \mathbb{R}} \int_{\Omega} \left(T(1, x, c) - O(x)\right)^2 dx. \tag{B.2}$$

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 Section B.1.2) and the temporal (see Section B.1.2) differential terms in Equation (B.1). The integral in Equation (B.2) is discretized using a simple *quadrature rule* (see Section 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

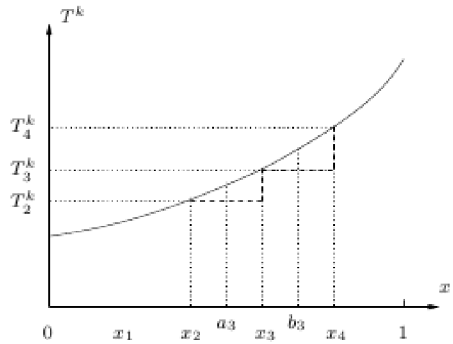


Figure B.2: Spatial Discretization: Central Finite Differences

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

$$r_j = 0 \qquad\qquad j \in \{0, n-1\} \tag{B.10}$$

$$r_j = \frac{c}{(\Delta x)^2} \cdot (T_{j+1} - 2 \cdot T_j + T_{j-1}) \qquad\qquad j \in \{1, \ldots, n-2\}, \tag{B.11}$$

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.

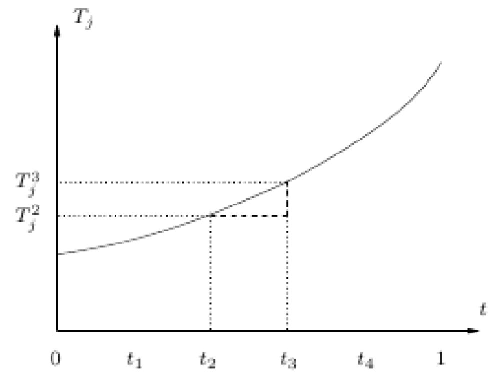


Figure B.3: Temporal Discretization: Backward Finite Difference

**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 Section 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

$$\frac{\partial T_j}{\partial t}(t_{k+1}, x, c) \approx \frac{T_j^{k+1} - T_j^k}{\Delta t}, \tag{B.15}$$

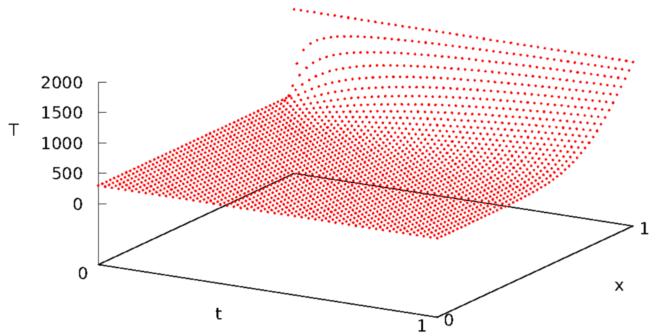


Figure B.4: Discrete Simulation of Temperature $T$ as a Function of Time $t$ and Position $x$.

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 Section 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) {
```

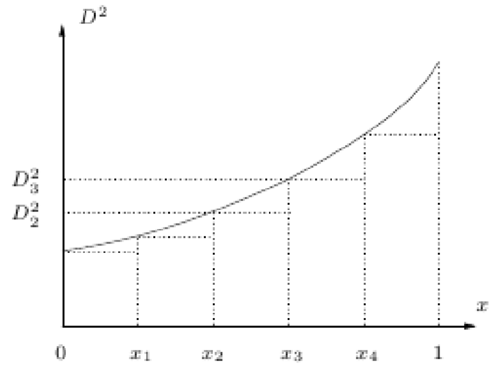


Figure B.5: Discretization of Objective: Quadrature

```
    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 coloring problems are discussed in [2].

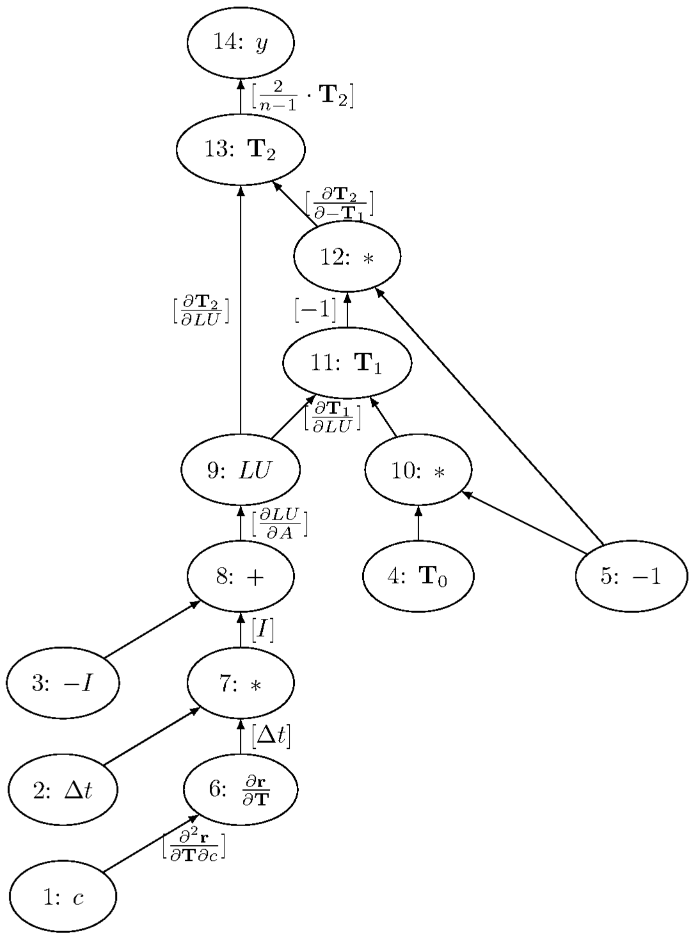


Figure B.6: Linearized DAG of Simulation Method

**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 Section 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 Section 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.

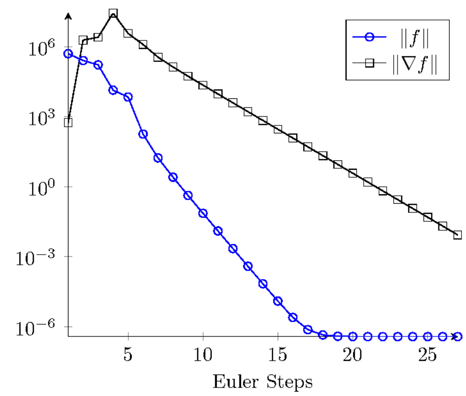


Figure B.7: Convergence Behaviour for $n = 160$.

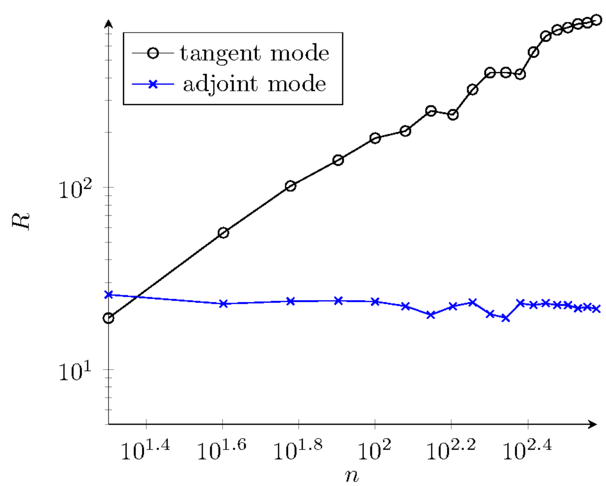


Figure B.8: Relative Run Time $R$ for Growing Problem Sizes $n$.

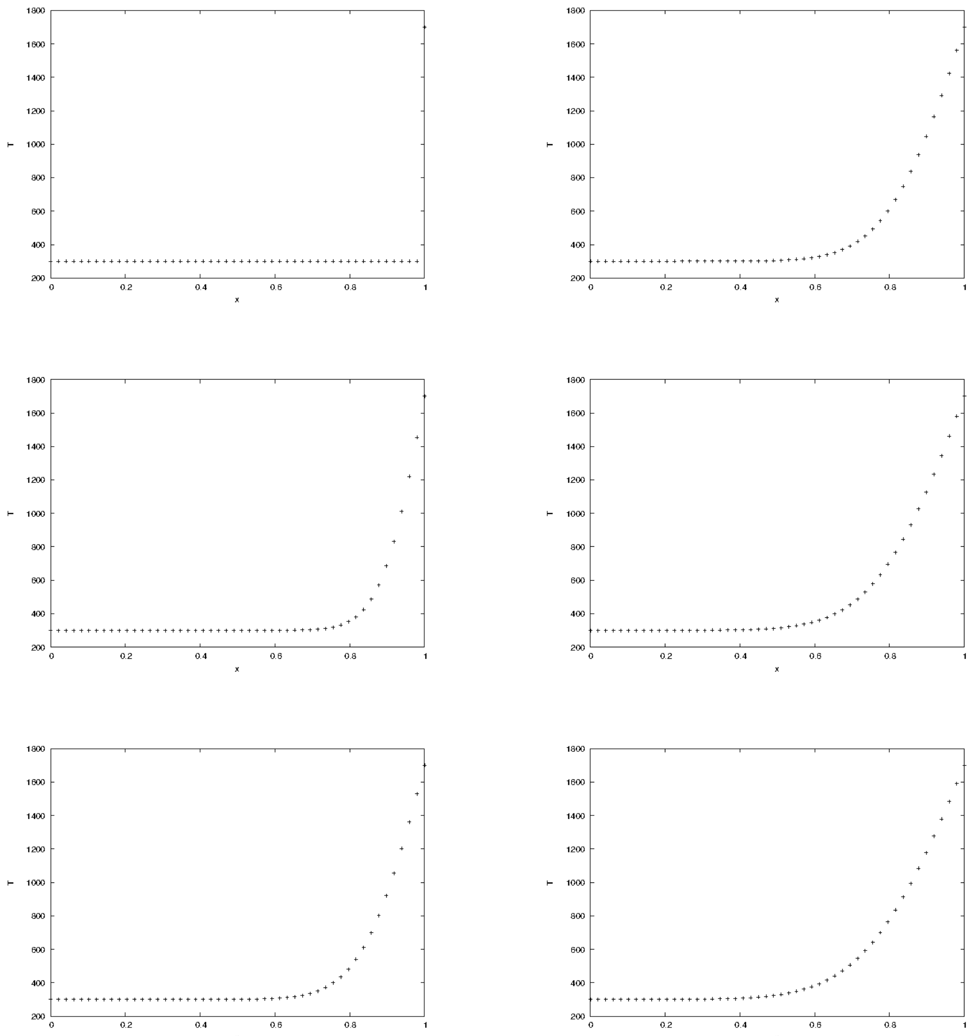


Figure B.9: Simulated Temperature Distribution for $t = 0, 0.2, 0.4$ (Left Column) and $t = 0.6, 0.8, 1$ (Right Column)

### B.1.5 Code

This section shows the source code for the given implementation of the diffusion problem.

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

template <typename cTYPE, typename TTYPE>
inline void residual(const vector<cTYPE>& c, vector<TTYPE>& T) {
  size_t n=T.size();
  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 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> 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<= 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(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 vector<TYPE>& LU, 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 vector<TYPE>& LU, 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 vector<TYPE>& LU, vector<TYPE>& b) {
  FSubst(LU,b);
```

```
  BSubst(LU,b);
}

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) {
    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 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/(static_cast<double>(n)-1);
}
```

## B.2 Function

This section shows the evaluation of the objective.

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

//#include "../include/f_residual_Jacobian_by_hand.hpp"
//#include "../include/f_residual_Jacobian_by_dco_dense.hpp"
#include "../include/f.hpp"

int main(int argc, char* argv[]){
  cout.precision(15);
```

```
  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 -1;
  }
  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;
  return 0;
}
```

## B.3 Observations

This section shows the generation of the "observations."

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

#include "../include/f.hpp"

int main(int argc, char* argv[]){
  cout.precision(15);
  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 -1;
  }
  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.txt");
  f(c,m,T,O,v);
  for (size_t i=0;i<n;i++) ofs << T[i]+(double) rand()/RAND_MAX << endl;
  return 0;
```

```
}
```

# B.4 Gradient

## B.4.1 Finite Differences

This section shows the approximation of the gradient by finite differences.

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>
#include <cfloat>
using namespace std;

#include "../include/f.hpp"

int main(int argc, char* argv[]){
  cout.precision(15);
  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 -1;
  }
  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)*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(15);
```

```
  for(size_t i=0;i<n;i++)
    cout << "dvdc[" << i << "]=" << dvdc[i] << endl;
  return 0;
}
```

### B.4.2 gt1s< double >

This section shows the computation of the gradient in tangent mode.

```
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace std;
#include "dco.hpp"
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 -1;
  }
  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(15);
  for(size_t i=0;i<n;i++)
    cout << "dvdc[" << i << "]=" << dvdc[i] << endl;
  return 0;
}
```

### B.4.3 ga1s< double >

This section shows the computation of the gradient in adjoint mode.

```
#include <cstdlib>
#include <iostream>
using namespace std;
#include "dco.hpp"
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 -1;
  }
  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_MODE::global_tape=DCO_TAPE_TYPE::create();
  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    DCO_MODE::global_tape->register_variable(ca[i]);
    DCO_MODE::global_tape->register_variable(Ta[i]);
  }
  f(ca,m,Ta,O,va);
  cerr << "record (0," << m << ")=" << dco::size_of(DCO_MODE::global_tape) << "B
      " << endl;
  derivative(va)=1.0;
  DCO_MODE::global_tape->register_output_variable(va);
  DCO_MODE::global_tape->interpret_adjoint();
  for (size_t i=0;i<n;i++) dvdc[i]=derivative(ca[i]);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
  cout.precision(15);
  for(size_t i=0;i<n;i++)
    cout << "dvdc[" << i << "]=" << dvdc[i] << endl;
```

```
  return 0;
}
```

### 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>
using namespace std;
#include "dco.hpp"
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_MODE>
void EulerSteps_fill_gap(typename DCO_MODE::external_adjoint_object_t *D){
  typedef typename DCO_MODE::type DCO_TYPE;
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
  typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

  const DCO_TAPE_POSITION_TYPE p0 = DCO_MODE::global_tape->get_position();
  const size_t& m=D->template read_data<size_t>();
  const size_t& n = D->template read_data<size_t>();
  vector<DCO_TYPE>* A_p=D->template read_data<vector<DCO_TYPE>*>();
  vector<DCO_TYPE> T(n);
  for (size_t i=0;i<n;i++) {
    T[i]=D->template read_data<DCO_VALUE_TYPE>();
    DCO_MODE::global_tape->register_variable(T[i]);
  }
  vector<DCO_TYPE> T_in(n);
  for (size_t i=0;i<n;i++) T_in[i]=T[i];
  DCO_TAPE_POSITION_TYPE p1=DCO_MODE::global_tape->get_position();
  EulerSteps(m,*A_p,T);
  cerr << "record =" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  for (size_t i=0;i<n;i++) {
    DCO_MODE::global_tape->register_output_variable(T[i]);
    derivative(T[i])=D->get_output_adjoint();
  }
```

```
  DCO_MODE::global_tape->interpret_adjoint_to(p1);
  for (size_t i=0;i<n;i++)
    D->increment_input_adjoint(derivative(T_in[i]));
  DCO_MODE::global_tape->reset_to(p0);
}

template<typename DCO_TYPE>
void EulerSteps_make_gap(const size_t m, const vector<DCO_TYPE>& A, vector<
    DCO_TYPE>& T){
  typedef mode<DCO_TYPE> DCO_MODE;
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
  size_t n=T.size();

  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();
  D->write_data(m); D->write_data(n); D->write_data(&A);
  for (size_t i=0;i<n;i++) {
    D->register_input(T[i]);
    D->write_data(value(T[i]));
  }
  DCO_MODE::global_tape->switch_to_passive();
  EulerSteps(m,A,T);
  DCO_MODE::global_tape->switch_to_active();
  for (size_t i=0;i<n;i++) T[i]=D->register_output(value(T[i]));
  DCO_MODE::global_tape->insert_callback(EulerSteps_fill_gap<DCO_MODE>,D);
}

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 -1;
  }
  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_MODE::global_tape=DCO_TAPE_TYPE::create();
  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    DCO_MODE::global_tape->register_variable(ca[i]);
    DCO_MODE::global_tape->register_variable(Ta[i]);
  }
  f(ca,m,Ta,O,va,cs);
  cerr << "record=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  derivative(va)=1.0;
  DCO_MODE::global_tape->register_output_variable(va);
  DCO_MODE::global_tape->interpret_adjoint();
  for (size_t i=0;i<n;i++) dvdc[i]=derivative(ca[i]);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
  cout.precision(15);
  for(size_t i=0;i<n;i++)
    cout << "dvdc[" << i << "]=" << dvdc[i] << endl;
  return 0;
}
```

### 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>
```

```
using namespace std;
#include "dco.hpp"
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>
void LUDecomp_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
  (void) D;
}

template<typename DCO_MODE, typename TYPE>
void LUDecomp_make_gap(vector<TYPE>& A){
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;

  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();

  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];

  DCO_MODE::global_tape->insert_callback(LUDecomp_fill_gap<DCO_MODE>,D);
}

// 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>
void Solve_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
  typedef typename DCO_MODE::type DCO_TYPE;
  vector<DCO_TYPE>* LU_p = D->template read_data<vector<DCO_TYPE>*>();
  size_t n=static_cast<size_t>(sqrt(double(LU_p->size())));
  vector<DCO_TYPE> T(n), a1T(n);

  for (size_t i=0;i<n;i++) a1T[i]=D->get_output_adjoint();
  DCO_MODE::global_tape->switch_to_passive();
  SolveT(*LU_p,a1T);
  for (size_t i=0;i<n;i++) T[i]=D->template read_data<double>();
  for (size_t i=0;i<n;i++)
    for (size_t j=0;j<n;j++)
      D->increment_input_adjoint(value(-a1T[i]*T[j]));
  for (size_t i=0;i<n;i++)
    D->increment_input_adjoint(value(a1T[i]));
  DCO_MODE::global_tape->switch_to_active();
}

template<typename DCO_MODE, typename TYPE>
void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b){
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;

  const size_t n=b.size();
  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();

  D->write_data(&LU);
  for (size_t i=0;i<n*n;i++) D->register_input(LU[i]);
  for (size_t i=0;i<n;i++) D->register_input(b[i]);
  DCO_MODE::global_tape->switch_to_passive();
  Solve(LU,b);
  DCO_MODE::global_tape->switch_to_active();
  for(size_t i=0;i<n;i++) {
    D->write_data(value(b[i]));
    b[i]=D->register_output(value(b[i]));
  }
  DCO_MODE::global_tape->insert_callback(Solve_fill_gap<DCO_MODE>,D);
}

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 -1;
  }
  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_MODE::global_tape=DCO_TAPE_TYPE::create();
  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    DCO_MODE::global_tape->register_variable(ca[i]);
  }
  f(ca,m,Ta,O,va);
  cerr << "record (0," << m << ")=" << dco::size_of(DCO_MODE::global_tape) << "B
      " << endl;
  derivative(va)=1.0;
  DCO_MODE::global_tape->register_output_variable(va);
  DCO_MODE::global_tape->interpret_adjoint();
  DCO_MODE::global_tape->write_to_dot();

  for (size_t i=0;i<n;i++) dvdc[i]=derivative(ca[i]);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
  cout.precision(15);
  for(size_t i=0;i<n;i++)
    cout << "dvdc[" << i << "]=" << dvdc[i] << endl;
  return 0;
}
```

### 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>
using namespace std;
#include "dco.hpp"
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_MODE::global_tape->is_active())
        Solve_make_gap<DCO_MODE>(A,T);
      else
        Solve(A,T);
    }
}

template<typename DCO_MODE>
void EulerSteps_fill_gap(typename DCO_MODE::external_adjoint_object_t *D){
  typedef typename DCO_MODE::type DCO_TYPE;
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
  typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

  const DCO_TAPE_POSITION_TYPE p0 = DCO_MODE::global_tape->get_position();
  const size_t& m=D->template read_data<size_t>();
  const size_t& n = D->template read_data<size_t>();

  vector<DCO_TYPE>* A_p=D->template read_data<vector<DCO_TYPE>*>();
  vector<DCO_TYPE> T(n);
  for (size_t i=0;i<n;i++) {
    T[i]=D->template read_data<DCO_VALUE_TYPE>();
    DCO_MODE::global_tape->register_variable(T[i]);
  }

  //save position of overwritten inputs
  vector<DCO_TYPE> T_in(n);
  for (size_t i=0;i<n;i++) T_in[i]=T[i];

```

```
  //save this position
  DCO_TAPE_POSITION_TYPE p1=DCO_MODE::global_tape->get_position();

  //forward run
  EulerSteps(m,*A_p,T);
  cerr << "record =" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;

  //get output adjoints (seeds for this section)
  for (size_t i=0;i<n;i++) {
    DCO_MODE::global_tape->register_output_variable(T[i]);
    derivative(T[i])=D->get_output_adjoint();
  }

  //reverse run and reset to position p1 (so only reverse run of Solve)
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p1);

  //increment input adjoint
  for (size_t i=0;i<n;i++) {
    D->increment_input_adjoint(derivative(T_in[i]));
  }

   //reset to position p0
   DCO_MODE::global_tape->reset_to(p0);
}
template<typename DCO_TYPE>
void EulerSteps_make_gap(const size_t m, const vector<DCO_TYPE>& A, vector<
    DCO_TYPE>& T){
  typedef mode<DCO_TYPE> DCO_MODE;
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
  size_t n=T.size();

  // create call back object
  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();

  D->write_data(m);
  D->write_data(n);
  D->write_data(&A);
  for (size_t i=0;i<n;i++) {
    D->register_input(T[i]);
    D->write_data(value(T[i]));
  }

  // forward run
  DCO_MODE::global_tape->switch_to_passive();
  EulerSteps(m,A,T);
  DCO_MODE::global_tape->switch_to_active();

  // register output
  for (size_t i=0;i<n;i++) T[i]=D->register_output(value(T[i]));

  DCO_MODE::global_tape->insert_callback(EulerSteps_fill_gap<DCO_MODE>,D);
}

```

```
template<typename DCO_MODE>
void LUDecomp_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
  (void) D;
}

template<typename DCO_MODE, typename TYPE>
void LUDecomp_make_gap(vector<TYPE>& A){
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;

  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();

  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];

  DCO_MODE::global_tape->insert_callback(LUDecomp_fill_gap<DCO_MODE>,D);
}

// 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>
void Solve_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
  typedef typename DCO_MODE::type DCO_TYPE;
```

```
  vector<DCO_TYPE>* LU_p = D->template read_data<vector<DCO_TYPE>*>();
  size_t n=sqrt(double(LU_p->size()));
  vector<DCO_TYPE> T(n), a1T(n);

  for (size_t i=0;i<n;i++) a1T[i]=D->get_output_adjoint();
  DCO_MODE::global_tape->switch_to_passive();
  SolveT(*LU_p,a1T);
  for (size_t i=0;i<n;i++) T[i]=D->template read_data<double>();
  for (size_t i=0;i<n;i++)
    for (size_t j=0;j<n;j++)
      D->increment_input_adjoint(value(-a1T[i]*T[j]));
  for (size_t i=0;i<n;i++)
    D->increment_input_adjoint(value(a1T[i]));
  DCO_MODE::global_tape->switch_to_active();
}

template<typename DCO_MODE, typename TYPE>
void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b){
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;

  const size_t n=b.size();
  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();

  D->write_data(&LU);
  for (size_t i=0;i<n*n;i++) D->register_input(LU[i]);
  for (size_t i=0;i<n;i++) D->register_input(b[i]);
  DCO_MODE::global_tape->switch_to_passive();
  Solve(LU,b);
  DCO_MODE::global_tape->switch_to_active();
  for(size_t i=0;i<n;i++) {
    D->write_data(value(b[i]));
    b[i]=D->register_output(value(b[i]));
  }
  DCO_MODE::global_tape->insert_callback(Solve_fill_gap<DCO_MODE>,D);
}

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 -1;
  }
  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_MODE::global_tape=DCO_TAPE_TYPE::create();
  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    DCO_MODE::global_tape->register_variable(ca[i]);
  }
  f(ca,m,Ta,O,va,cs);
  cerr << "record (0," << m << ")=" << dco::size_of(DCO_MODE::global_tape) << "B
    " << endl;
  derivative(va)=1.0;
  DCO_MODE::global_tape->register_output_variable(va);
  DCO_MODE::global_tape->interpret_adjoint();
  for (size_t i=0;i<n;i++) dvdc[i]=derivative(ca[i]);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
  cout.precision(15);
  for(size_t i=0;i<n;i++)
    cout << "dvdc[" << i << "]=" << dvdc[i] << endl;
  return 0;
}
```

## 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>
using namespace std;

#include "../include/f.hpp"

int main(int argc, char* argv[]){
  cout.precision(15);
  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 -1;
  }
  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))*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,O,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,O,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,O,vp1);
      vp2+=vp1;
```

```
      d2vdc2[i][j]=vp2/(4*h*h);
    }
  }
  cout.precision(15);
  for (size_t i=0;i<n;i++)
    for (size_t j=0;j<n;j++)
       cout << "d2vdc2[" << i << "][" << j << "]="
               << d2vdc2[i][j] << endl;
  return 0;
}
```

### 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>
using namespace std;
#include "dco.hpp"
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 -1;
  }
  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(15);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      cout << "dvdc[" << i << "][" << j << "]=" << d2vdc2[i][j] << endl;
  return 0;
}
```

### B.5.3 ga1s<gt1s< double >::type>

This section shows the approximation of the Hessian in second-order adjoint mode.

```
#include <cstdlib>
#include <iostream>
using namespace std;
#include "dco.hpp"
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 -1;
  }
  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_MODE::global_tape=DCO_TAPE_TYPE::create();

  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    DCO_MODE::global_tape->register_variable(ca[i]);
  }
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_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) DCO_MODE::global_tape->zero_adjoints();
    derivative(value(ca[i]))=1;
    f(ca,m,Ta,0,va);
    value(derivative(va))=1;
    DCO_MODE::global_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]));
    DCO_MODE::global_tape->reset_to(p);
  }
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
  cout.precision(15);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      cout << "d2vdc2[" << i << "][" << j << "]=" << d2vdc2[i][j] << endl;

  return 0;
}
```

### 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>
using namespace std;
#include "dco.hpp"
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_MODE>
void EulerSteps_fill_gap(typename DCO_MODE::external_adjoint_object_t *D){
  typedef typename DCO_MODE::type DCO_TYPE;
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
  typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
```

```

  const DCO_TAPE_POSITION_TYPE p0 = DCO_MODE::global_tape->get_position();
  const size_t& m=D->template read_data<size_t>();
  const size_t& n = D->template read_data<size_t>();
  vector<DCO_TYPE>* A_p=D->template read_data<vector<DCO_TYPE>*>();
  vector<DCO_TYPE> T(n);
  for (size_t i=0;i<n;i++) {
    T[i]=D->template read_data<DCO_VALUE_TYPE>();
    DCO_MODE::global_tape->register_variable(T[i]);
  }
  vector<DCO_TYPE> T_in(n);
  for (size_t i=0;i<n;i++) T_in[i]=T[i];
  DCO_TAPE_POSITION_TYPE p1=DCO_MODE::global_tape->get_position();
  EulerSteps(m,*A_p,T);
  cerr << "record =" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
  for (size_t i=0;i<n;i++) {
    DCO_MODE::global_tape->register_output_variable(T[i]);
    derivative(T[i])=D->get_output_adjoint();
  }
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p1);
  for (size_t i=0;i<n;i++)
    D->increment_input_adjoint(derivative(T_in[i]));
  DCO_MODE::global_tape->reset_to(p0);
}

template<typename DCO_TYPE>
void EulerSteps_make_gap(const size_t m, const vector<DCO_TYPE>& A, vector<
    DCO_TYPE>& T){
  typedef mode<DCO_TYPE> DCO_MODE;
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
  size_t n=T.size();

  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();
  D->write_data(m);
  D->write_data(n);
  D->write_data(&A);
  for (size_t i=0;i<n;i++) {
    D->register_input(T[i]);
    D->write_data(value(T[i]));
  }
  DCO_MODE::global_tape->switch_to_passive();
  EulerSteps(m,A,T);
  DCO_MODE::global_tape->switch_to_active();
  for (size_t i=0;i<n;i++) T[i]=D->register_output(value(T[i]));
  DCO_MODE::global_tape->insert_callback(EulerSteps_fill_gap<DCO_MODE>,D);
}

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 -1;
  }
  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_MODE::global_tape=DCO_TAPE_TYPE::create();
  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    DCO_MODE::global_tape->register_variable(ca[i]);
  }
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_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) DCO_MODE::global_tape->zero_adjoints();
    derivative(value(ca[i]))=1;
    f(ca,m,Ta,0,va,cs);
    value(derivative(va))=1;
    DCO_MODE::global_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]));
    DCO_MODE::global_tape->reset_to(p);
  }
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
  cout.precision(15);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      cout << "d2vdc2[" << i << "][" << j << "]=" << d2vdc2[i][j] << endl;
  return 0;
}
```

### 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>
using namespace std;
#include "dco.hpp"
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>
void LUDecomp_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
  (void) D;
}

template<typename DCO_MODE, typename TYPE>
void LUDecomp_make_gap(vector<TYPE>& A){
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;

  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();

  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];

  DCO_MODE::global_tape->insert_callback(LUDecomp_fill_gap<DCO_MODE>,D);
}

// 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>
void Solve_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
  typedef typename DCO_MODE::type DCO_TYPE;
  vector<DCO_TYPE>* LU_p = D->template read_data<vector<DCO_TYPE>*>();
  size_t n=sqrt(double(LU_p->size()));
  vector<DCO_TYPE> T(n), a1T(n);

  for (size_t i=0;i<n;i++) a1T[i]=D->get_output_adjoint();
  DCO_MODE::global_tape->switch_to_passive();
  SolveT(*LU_p,a1T);
  for (size_t i=0;i<n;i++) T[i]=D->template read_data<DCO_BASE_TYPE>();
  for (size_t i=0;i<n;i++)
    for (size_t j=0;j<n;j++)
      D->increment_input_adjoint(value(-a1T[i]*T[j]));
  for (size_t i=0;i<n;i++)
    D->increment_input_adjoint(value(a1T[i]));
  DCO_MODE::global_tape->switch_to_active();
}

```

```
template<typename DCO_MODE, typename TYPE>
void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b){
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;

  const size_t n=b.size();
  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();

  D->write_data(&LU);
  for (size_t i=0;i<n*n;i++) D->register_input(LU[i]);
  for (size_t i=0;i<n;i++) D->register_input(b[i]);
  DCO_MODE::global_tape->switch_to_passive();
  Solve(LU,b);
  DCO_MODE::global_tape->switch_to_active();
  for(size_t i=0;i<n;i++) {
    D->write_data(value(b[i]));
    b[i]=D->register_output(value(b[i]));
  }
  DCO_MODE::global_tape->insert_callback(Solve_fill_gap<DCO_MODE>,D);
}

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 -1;
  }
  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_MODE::global_tape=DCO_TAPE_TYPE::create();

  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    DCO_MODE::global_tape->register_variable(ca[i]);
  }
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_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) DCO_MODE::global_tape->zero_adjoints();
    derivative(value(ca[i]))=1;
    f(ca,m,Ta,O,va);
    value(derivative(va))=1;
    DCO_MODE::global_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]));
    DCO_MODE::global_tape->reset_to(p);
  }
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
  cout.precision(15);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      cout << "d2vdc2[" << i << "][" << j << "]=" << d2vdc2[i][j] << endl;

  return 0;
}
```

### 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>
using namespace std;
#include "dco.hpp"
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_MODE::global_tape->is_active())
        Solve_make_gap<DCO_MODE>(A,T);
      else
        Solve(A,T);
    }
}

template<typename DCO_MODE>
void EulerSteps_fill_gap(typename DCO_MODE::external_adjoint_object_t *D){
  typedef typename DCO_MODE::type DCO_TYPE;
  typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
  typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
  typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

  const DCO_TAPE_POSITION_TYPE p0 = DCO_MODE::global_tape->get_position();
  const size_t& m=D->template read_data<size_t>();
  const size_t& n = D->template read_data<size_t>();

  vector<DCO_TYPE>* A_p=D->template read_data<vector<DCO_TYPE>*>();
  vector<DCO_TYPE> T(n);
  for (size_t i=0;i<n;i++) {
    T[i]=D->template read_data<DCO_VALUE_TYPE>();
    DCO_MODE::global_tape->register_variable(T[i]);
  }

  //save position of overwritten inputs
  vector<DCO_TYPE> T_in(n);
  for (size_t i=0;i<n;i++) T_in[i]=T[i];

  //save this position
  DCO_TAPE_POSITION_TYPE p1=DCO_MODE::global_tape->get_position();

  //forward run
 EulerSteps(m,*A_p,T);
  cerr << "record =" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;

  //get output adjoints (seeds for this section)
  for (size_t i=0;i<n;i++) {
    DCO_MODE::global_tape->register_output_variable(T[i]);
    derivative(T[i])=D->get_output_adjoint();
  }

  //reverse run and reset to position p1 (so only reverse run of Solve)
  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p1);

  //increment input adjoint
```

```
  for (size_t i=0;i<n;i++) {
    D->increment_input_adjoint(derivative(T_in[i]));
  }

   //reset to position p0
   DCO_MODE::global_tape->reset_to(p0);
}
template<typename DCO_TYPE>
void EulerSteps_make_gap(const size_t m, const vector<DCO_TYPE>& A, vector<
    DCO_TYPE>& T){
  typedef mode<DCO_TYPE> DCO_MODE;
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
  size_t n=T.size();

  // create call back object
  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();

  D->write_data(m);
  D->write_data(n);
  D->write_data(&A);
  for (size_t i=0;i<n;i++) {
    D->register_input(T[i]);
    D->write_data(value(T[i]));
  }

  // forward run
  DCO_MODE::global_tape->switch_to_passive();
  EulerSteps(m,A,T);
  DCO_MODE::global_tape->switch_to_active();

  // register output
  for (size_t i=0;i<n;i++) T[i]=D->register_output(value(T[i]));

  DCO_MODE::global_tape->insert_callback(EulerSteps_fill_gap<DCO_MODE>,D);
}


template<typename DCO_MODE>
void LUDecomp_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
  (void) D;
}

template<typename DCO_MODE, typename TYPE>
void LUDecomp_make_gap(vector<TYPE>& A){
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;

  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();

  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];

  DCO_MODE::global_tape->insert_callback(LUDecomp_fill_gap<DCO_MODE>,D);
}

// 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>
void Solve_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
  typedef typename DCO_MODE::type DCO_TYPE;
  vector<DCO_TYPE>* LU_p = D->template read_data<vector<DCO_TYPE>*>();
  size_t n=sqrt(double(LU_p->size()));
  vector<DCO_TYPE> T(n), a1T(n);

  for (size_t i=0;i<n;i++) a1T[i]=D->get_output_adjoint();
  DCO_MODE::global_tape->switch_to_passive();
  SolveT(*LU_p,a1T);
  for (size_t i=0;i<n;i++) T[i]=D->template read_data<DCO_BASE_TYPE>();
  for (size_t i=0;i<n;i++)
    for (size_t j=0;j<n;j++)
      D->increment_input_adjoint(value(-a1T[i]*T[j]));
  for (size_t i=0;i<n;i++)
    D->increment_input_adjoint(value(a1T[i]));
  DCO_MODE::global_tape->switch_to_active();
}

template<typename DCO_MODE, typename TYPE>
```

```
void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b){
  typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;

  const size_t n=b.size();
  DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
      DCO_EAO_TYPE>();

  D->write_data(&LU);
  for (size_t i=0;i<n*n;i++) D->register_input(LU[i]);
  for (size_t i=0;i<n;i++) D->register_input(b[i]);
  DCO_MODE::global_tape->switch_to_passive();
  Solve(LU,b);
  DCO_MODE::global_tape->switch_to_active();
  for(size_t i=0;i<n;i++) {
    D->write_data(value(b[i]));
    b[i]=D->register_output(value(b[i]));
  }
  DCO_MODE::global_tape->insert_callback(Solve_fill_gap<DCO_MODE>,D);
}

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 -1;
  }
  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_MODE::global_tape=DCO_TAPE_TYPE::create();
  for(size_t i=0;i<n;i++) {
    ca[i]=c[i];
    Ta[i]=T[i];
    DCO_MODE::global_tape->register_variable(ca[i]);
  }
  DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_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) DCO_MODE::global_tape->zero_adjoints();
    derivative(value(ca[i]))=1;
    f(ca,m,Ta,O,va,cs);
    value(derivative(va))=1;
    DCO_MODE::global_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]));
    DCO_MODE::global_tape->reset_to(p);
  }
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
  cout.precision(15);
  for(size_t i=0;i<n;i++)
    for(size_t j=0;j<n;j++)
      cout << "d2vdc2[" << i << "][" << j << "]=" << d2vdc2[i][j] << endl;
  return 0;
}
```

# Appendix C

# 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:

```
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)*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(15);
  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;
  return 0;
}
```

### C.2.2 First-Order Tangent Mode

```
#include<vector>
#include<iostream>
using namespace std;
#include "dco.hpp"
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(15);
  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;
  return 0;
}
```

### C.2.3 First-Order Tangent Mode: Vector Mode

```
#include<vector>
#include<iostream>
using namespace std;
#include "dco.hpp"
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(15);
  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;
  return 0;
}
```

### C.2.4 First-Order Adjoint Mode

```
#include<vector>
#include<iostream>
using namespace std;
#include "dco.hpp"
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;
  typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
  size_t n=xv.size();
  vector<DCO_TYPE> x(n); DCO_TYPE y;

  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  for (size_t j=0;j<n;j++) {
    DCO_MODE::global_tape->register_variable(x[j]);
    value(x[j])=xv[j];
  }
  f(x,y);
  DCO_MODE::global_tape->register_output_variable(y);
  yv=value(y);
  derivative(y)=1;
  DCO_MODE::global_tape->interpret_adjoint();
  for (size_t j=0;j<n;j++) g[j]=derivative(x[j]);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}

int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(15);
  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;
  return 0;
}
```

## 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))*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))*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(15);
  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;
  return 0;
}
```

### C.3.2 Second-Order Tangent Mode

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

#include "dco.hpp"
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(15);
  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;
  return 0;
}
```

### C.3.3 Second-Order Adjoint Mode (Tangent over Adjoint)

```
#include<vector>
#include<iostream>
using namespace std;
#include "dco.hpp"
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;
  typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
  size_t n=xv.size();
  vector<DCO_TYPE> x(n); DCO_TYPE y;

  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  for (size_t i=0;i<n;i++) {
    for (size_t j=0;j<n;j++) {
      DCO_MODE::global_tape->register_variable(x[j]);
      passive_value(x[j])=xv[j];
      derivative(value(x[j]))=0;
    }
    derivative(value(x[i]))=1;
    f(x,y);
    DCO_MODE::global_tape->register_output_variable(y);
    yv=passive_value(y);
    g[i]=derivative(value(y));
    value(derivative(y))=1;
    DCO_MODE::global_tape->interpret_adjoint();
    for (size_t j=0;j<n;j++) H[i][j]=derivative(derivative(x[j]));
    DCO_MODE::global_tape->reset();
  }
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
```

```
}

int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(15);
  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;
  return 0;
}
```

### C.3.4 Second-Order Adjoint Mode (Adjoint over Tangent)

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

#include "dco.hpp"
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;
typedef DCO_BASE_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) {
  DCO_BASE_MODE::global_tape=DCO_TAPE_TYPE::create();
  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];
      DCO_BASE_MODE::global_tape->register_variable(value(x[j]));
      DCO_BASE_MODE::global_tape->register_variable(derivative(x[j]));
      x_in[j]=x[j];
    }
    value(derivative(x[i]))=1;
    f(x,y);
    DCO_BASE_MODE::global_tape->register_output_variable(value(y));
    DCO_BASE_MODE::global_tape->register_output_variable(derivative(y));
    derivative(derivative(y))=1;
```

```
    DCO_BASE_MODE::global_tape->interpret_adjoint();
    for (size_t j=0;j<n;j++)
      h[j][i] = derivative(value(x_in[j]));
    DCO_BASE_MODE::global_tape->reset();
    g[i]=value(derivative(y));
  }
  yv=passive_value(y);
  DCO_TAPE_TYPE::remove(DCO_BASE_MODE::global_tape);
}

int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(15);
  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;
  return 0;
}
```

### C.3.5 Second-Order Adjoint Mode (Adjoint over Adjoint)

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

#include "dco.hpp"
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_BASE_MODE::global_tape=DCO_BASE_TAPE_TYPE::create();
  DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
  for (size_t j=0;j<n;j++) {
    x[j]=xv[j];
```

```
    DCO_BASE_MODE::global_tape->register_variable(value(x[j]));
    DCO_MODE::global_tape->register_variable(x[j]);
    x_in[j]=x[j];
  }
  f(x,y);
  derivative(y)=1.0;
  DCO_BASE_MODE::global_tape->register_variable(derivative(y));
  DCO_MODE::global_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;
    DCO_BASE_MODE::global_tape->interpret_adjoint();
    for (size_t j=0;j<n;j++)
      h[i][j]=derivative(value(x_in[j]));
    DCO_BASE_MODE::global_tape->zero_adjoints();
  }
  yv=passive_value(y);
  DCO_BASE_TAPE_TYPE::remove(DCO_BASE_MODE::global_tape);
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
}

int main(int c, char* v[]) {
  assert(c==2); (void)c;
  cout.precision(15);
  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;
  return 0;
}
```

# Index

# 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.

## Aachener Informatik-Berichte

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

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

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

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

2015-01 * Fachgruppe Informatik: Annual Report 2015

2015-02 Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications

2015-05 Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity

2015-06 Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"

2015-07 Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon

2015-08 Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization

2015-09 Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models

2015-11 Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus

2015-12 Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic

2015-13 Andreas Polzer: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

2015-14 Niloofar Safiran and Uwe Naumann: Symbolic vs. Algorithmic Differentiation of GSL Integration Routines

2016-01 * Fachgruppe Informatik: Annual Report 2016

2016-02 Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems

2016-03 Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jrgen Giesl: Lower Runtime Bounds for Integer Programs

2016-04 Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution

2016-05 Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization

2016-06 Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud

2016-07 Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis

2016-08 Klaus Leppkes, Johannes Lotz, and Uwe Naumann: Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features

2016-09 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic

2016-10 Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering

2017-01 * Fachgruppe Informatik: Annual Report 2017

2017-02 Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity

2017-04 Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE

2017-05 Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems

2017-06 Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy

2017-07 Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++

2017-08 Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts

2017-09 Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing

2018-01 * Fachgruppe Informatik: Annual Report 2018

| | |
|---|---|
| 2018-02 | Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen |
| 2018-03 | Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices |
| 2018-04 | Andreas Ganser: Operation-Based Model Recommenders |

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.