dco/c++ User Guide

Version 3.1

Doc Version 1.1

Copyright Statement

Uwe Naumann

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

and

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

www: `http://www.nag.co.uk`
www: `http://www.stce.rwth-aachen.de`

contact email: `support@nag.co.uk`

© The Numerical Algorithms Group Limited, 2015 All rights reserved. Duplication of this User Guide in printed form or by electronic means for the private and sole use of the software licensee is permitted provided that the individual copying the document is not

- selling or reselling the documentation;
- distributing the documentation to others, who have not licensed the software for professional use;
- using it for the purpose of critical review, publication in printed form or any electronic publication including the Internet without the prior written permission of the copyright owner.

The copyright owner gives no warranties and makes no representations about the contents of this User Guide and specifically disclaims any implied warranties or merchantability or fitness for any purpose.

The copyright owner reserves the right to revise this User Guide and to make changes from time to time in its contents without notifying any person of such revisions or changes. Produced by NAG.

`dco/c++` Version 3.1 Doc Version 1.0 released March 2015

NAG is a registered trademark of:

The Numerical Algorithms Group Limited

The Numerical Algorithms Group Inc

Nihon Numerical Algorithms Group KK

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 run time 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 call back interface of dco/c++ facilitates use of accelerators (such as GPUs) as well as the coupling with established and highly optimized numerical libraries. A growing set of methods from the NAG Library are supported as intrinsics requiring linkage with the separate NAG AD Library.

Disclaimer: This user documentation 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 documentation targets readers with a very good understanding of AD. See [8] 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) **9**
1.1 First-order Tangent Mode . . . 9
1.1.1 Generic first-order scalar tangents . . . 9
1.1.2 Generic first-order vector tangents . . . 9
1.1.3 Preaccumulation through use of expression templates . . . 10
1.2 First-order Adjoint Mode . . . 10
1.2.1 Generic first-order scalar adjoints . . . 10
1.2.2 Generic first-order vector adjoints . . . 10
1.2.3 Global "blob" tape . . . 10
1.2.4 Global "chunk" tape . . . 10
1.2.5 (Thread-)local tapes . . . 10
1.2.6 Preaccumulation through use of expression templates . . . 10
1.2.7 Repeated evaluation of tape . . . 10
1.2.8 Adjoint callback interface . . . 10
1.2.9 User-defined local Jacobians interface . . . 11
1.3 Second- and Higher-order Tangent Mode . . . 11
1.3.1 Generic second-order scalar tangents . . . 11
1.3.2 Generic second-order vector tangents . . . 11
1.3.3 Other generic second-order tangents . . . 11
1.3.4 Generic third- and higher-order tangents . . . 11
1.4 Second- and Higher-order Adjoint Mode . . . 11
1.4.1 Generic second-order scalar adjoints . . . 11
1.4.2 Generic second-order vector adjoints . . . 12
1.4.3 Other generic second-order adjoints . . . 12
1.4.4 Generic third- and higher-order adjoints . . . 12

Chapter 2 First-order Tangent Mode **13**
2.1 Purpose . . . 13
2.2 Example . . . 13
2.2.1 Example Text . . . 13
2.2.2 Example Results . . . 14
2.3 New dco/c++ Features . . . 14
2.3.1 dco.hpp . . . 14
2.3.2 DCO_BASE_TYPE . . . 14
2.3.3 **gt1s**<DCO_BASE_TYPE> . . . 14
2.3.4 DCO_MODE . . . 14
2.3.5 **gt1s**<DCO_BASE_TYPE>**::type** . . . 15
2.3.6 DCO_TYPE . . . 15

2.3.7 value . . . 15
2.3.8 derivative . . . 15

Chapter 3 First-order Adjoint Mode **16**
3.1 Purpose . . . 16
3.2 Example . . . 16
3.2.1 Example Text . . . 16
3.2.2 Example Results . . . 17
3.3 New dco/c++ Features . . . 17
3.3.1 **ga1s**<DCO_BASE_TYPE> . . . 17
3.3.2 **ga1s**<DCO_BASE_TYPE>**::type** . . . 17
3.3.3 DCO_TAPE_TYPE . . . 18
3.3.4 DCO_MODE::global_tape . . . 18
3.3.5 DCO_TAPE_TYPE::create . . . 18
3.3.6 DCO_TAPE_TYPE::register_variable . . . 18
3.3.7 DCO_TAPE_TYPE::register_output_variable . . . 18
3.3.8 DCO_TAPE_TYPE::interpret_adjoint . . . 18
3.3.9 DCO_TAPE_TYPE::remove . . . 18

Chapter 4 Second-order Tangent Mode **19**
4.1 Purpose . . . 19
4.2 Example . . . 19
4.2.1 Example Text . . . 19
4.2.2 Example Results . . . 20
4.3 New dco/c++ Features . . . 21
4.3.1 passive_value . . . 21

Chapter 5 Second-order Adjoint Mode **22**
5.1 Purpose . . . 22
5.2 Example . . . 22
5.2.1 Example Text . . . 22
5.2.2 Example Results . . . 24
5.3 New dco/c++ Features . . . 24

Chapter 6 Approximate Second-order Adjoint Mode **25**
6.1 Purpose . . . 25
6.2 Example . . . 25
6.2.1 Example Text . . . 25
6.2.2 Example Results . . . 26
6.3 New dco/c++ Features . . . 26

Chapter 7 Third-order Tangent Mode **27**
7.1 Purpose . . . 27
7.2 Example . . . 27
7.2.1 Example Text . . . 27
7.2.2 Example Results . . . 29

7.3 New dco/c++ Features . . . 29

Chapter 8 Third-order Adjoint Mode **30**

8.1 Purpose . . . 30

8.2 Example . . . 30

8.2.1 Example Text . . . 30

8.2.2 Example Results . . . 33

8.3 New dco/c++ Features . . . 33

Chapter 9 First-order Tangent Mode: Vector Mode **34**

9.1 Purpose . . . 34

9.2 Example . . . 34

9.2.1 Example Text . . . 34

9.2.2 Example Results . . . 35

9.3 New dco/c++ Features . . . 35

9.3.1 derivative . . . 35

Chapter 10 First-order Adjoint Mode: Vector Mode **36**

10.1 Purpose . . . 36

10.2 Example . . . 36

10.2.1 Example Text . . . 36

10.2.2 Example Results . . . 37

10.3 New dco/c++ Features . . . 38

Chapter 11 Adjoint First-order Scalar Mode: Multiple Tapes **39**

11.1 Purpose . . . 39

11.2 Example . . . 39

11.2.1 Example Text . . . 39

11.2.2 Example Results . . . 40

11.3 New dco/c++ Features . . . 40

11.3.1 **ga1sm**<DCO_BASE_TYPE> . . . 40

11.3.2 **ga1sm**<DCO_BASE_TYPE>::**type** . . . 40

11.3.3 DCO_TAPE_TYPE::create . . . 40

11.3.4 DCO_TAPE_TYPE::remove . . . 40

Chapter 12 First-order Adjoint Mode: Basic Checkpointing **41**

12.1 Purpose . . . 41

12.2 Example . . . 41

12.2.1 Example Text . . . 41

12.2.2 Example Results . . . 43

12.3 New dco/c++ Features . . . 43

12.3.1 DCO_TAPE_TYPE::**position_t** . . . 43

12.3.2 DCO_TAPE_TYPE::get_position . . . 43

12.3.3 DCO_TAPE_TYPE::interpret_adjoint_and_reset_to 43
12.3.4 dco::mode<DCO_TYPE> 43
12.3.5 DCO_MODE::**value_t** 43
12.3.6 DCO_MODE::**external_adjoint_object_t** 43
12.3.7 DCO_TAPE_TYPE::create_callback_object 43
12.3.8 DCO_EAO_TYPE::register_input 43
12.3.9 DCO_EAO_TYPE::write_data 44
12.3.10 DCO_EAO_TYPE::register_output 44
12.3.11 DCO_TAPE_TYPE::insert_callback 44
12.3.12 DCO_EAO_TYPE::read_data 44
12.3.13 DCO_EAO_TYPE::get_output_adjoint 44
12.3.14 DCO_EAO_TYPE::increment_input_adjoint 44
12.3.15 DCO_TAPE_TYPE::reset_to 44
12.3.16 DCO_TAPE_TYPE::get_tape_memory_size() 44

Chapter 13 Second-order Adjoint Mode: Basic Checkpointing **45**
13.1 Purpose 45
13.2 Example 45
13.2.1 Example Text 45
13.2.2 Example Results 46
13.3 New dco/c++ Features 46

Chapter 14 First-order Adjoint Mode: Checkpointing Evolutions Equidistantly **47**
14.1 Purpose 47
14.2 Example 47
14.2.1 Example Text 47
14.2.2 Example Results 49
14.3 New dco/c++ Features 49

Chapter 15 Second-order Adjoint Mode: Checkpointing Evolutions Equidistantly **50**
15.1 Purpose 50
15.2 Example 50
15.2.1 Example Text 50
15.2.2 Example Results 51
15.3 New dco/c++ Features 51

Chapter 16 First-order Adjoint Mode: Checkpointing Evolutions Recursively **52**
16.1 Purpose 52
16.2 Example 52
16.2.1 Example Text 52
16.2.2 Example Results 56
16.3 New dco/c++ Features 57

16.3.1 my_external_adjoint_object_t 57

Chapter 17 Second-order Adjoint Mode: Checkpointing Evolutions Recursively 58

17.1 Purpose 58
17.2 Example 58
17.2.1 Example Text 58
17.2.2 Example Results 59
17.3 New dco/c++ Features 59

Chapter 18 First-order Adjoint Mode: Checkpointing Ensembles 60

18.1 Purpose 60
18.2 Example 60
18.2.1 Example Text 60
18.2.2 Example Results 62
18.3 New dco/c++ Features 62

Chapter 19 Second-order Adjoint Mode: Checkpointing Ensembles 63

19.1 Purpose 63
19.2 Example 63
19.2.1 Example Text 63
19.2.2 Example Results 64
19.3 New dco/c++ Features 64

Chapter 20 First-order Adjoint Mode: Embedding Adjoint Source Code 65

20.1 Purpose 65
20.2 Example 65
20.2.1 Example Text 65
20.2.2 Example Results 67
20.3 New dco/c++ Features 67

Chapter 21 Second-order Adjoint Mode: Embedding First-order Adjoint Code 68

21.1 Purpose 68
21.2 Example 68
21.2.1 Example Text 68
21.2.2 Example Results 69
21.3 New dco/c++ Features 70

Chapter 22 First-order Adjoint Mode: Local Preaccumulation Using First-order Tangent Code 71

22.1 Purpose 71
22.2 Example 71
22.2.1 Example Text 71
22.2.2 Example Results 73

22.3 New dco/c++ Features 73

Chapter 23 Second-order Adjoint Mode: Local Preaccumulation Using First-order Tangent Code 74

23.1 Purpose 74
23.2 Example 74
23.2.1 Example Text 74
23.2.2 Example Results 75
23.3 New dco/c++ Features 75

Chapter 24 First-order Adjoint Mode: Edge Insertion for Preaccumulation of Local Gradients 76

24.1 Purpose 76
24.2 Example 76
24.2.1 Example Text 76
24.2.2 Example Results 78
24.3 New dco/c++ Features 78
24.3.1 DCO_MODE::local_gradient_t 78
24.3.2 DCO_TAPE_TYPE::create_local_gradient_object 78
24.3.3 DCO_LOCAL_GRADIENT_TYPE::put 78
24.3.4 DCO_LOCAL_GRADIENT_TYPE::finalize 78

Chapter 25 Second-order Adjoint Mode: Edge Insertion for Preaccumulation of Local Gradients 79

25.1 Purpose 79
25.2 Example 79
25.2.1 Example Text 79
25.2.2 Example Results 80
25.3 New dco/c++ Features 80

Chapter 26 First-order Adjoint Mode: User-Defined Adjoints 81

26.1 Purpose 81
26.2 Example 81
26.2.1 Example Text 81
26.2.2 Example Results 83
26.3 New dco/c++ Features 83

Chapter 27 Second-order Adjoint Mode: User-Defined First-order Adjoints 84

27.1 Purpose 84
27.2 Example 84
27.2.1 Example Text 84
27.2.2 Example Results 85
27.3 New dco/c++ Features 85

Appendix A Status of User Interface 86

A.1 Gold . . . 86
A.2 Silver . . . 86
A.3 Bronze . . . 93
A.4 Undefined . . . 96

Appendix B Case Study: Diffusion **100**
B.1 Purpose . . . 100
B.1.1 Problem Description . . . 100
B.1.2 Numerical Method . . . 101
B.1.3 Algorithmic Differentiation . . . 105
B.1.4 Optimization . . . 107
B.1.5 Code . . . 110
B.2 Function . . . 112
B.3 Observations . . . 113
B.4 Gradient . . . 113
B.4.1 Finite Differences . . . 113
B.4.2 **gt1s<double>** . . . 114
B.4.3 **ga1s<double>** . . . 115
B.4.4 **ga1s<double>** + Equidistant Checkpointing . . . 116
B.4.5 **ga1s<double>** + Symbolic Derivative of Linear Solver . . . 119
B.4.6 **ga1s<double>** + Symbolic Derivative of Linear Solver and Equidistant Checkpointing . . . 121
B.5 Hessian . . . 126
B.5.1 Finite Differences . . . 126
B.5.2 **gt1s<gt1s<double>::type>** . . . 127
B.5.3 **ga1s<gt1s<double>::type>** . . . 128
B.5.4 **ga1s<gt1s<double>::type>** + Equidistant Checkpointing . . . 129
B.5.5 **ga1s<gt1s<double>::type>** + Symbolic Derivative of Linear Solver . . . 132
B.5.6 **ga1s<gt1s<double>::type>** + Symbolic Derivative of Linear Solver and Equidistant Checkpointing . . . 135

Appendix C Racing on $y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$ **141**
C.1 Purpose . . . 141
C.2 Gradient . . . 141
C.2.1 Central Finite Differences . . . 141
C.2.2 First-order Tangent Mode . . . 142
C.2.3 First-order Tangent Mode: Vector Mode . . . 143
C.2.4 First-order Adjoint Mode . . . 143
C.3 Hessian . . . 144
C.3.1 Second-order Central Finite Differences . . . 144
C.3.2 Second-order Tangent Mode . . . 145
C.3.3 Second-order Adjoint Mode (Tangent over Adjoint) . . . 146
C.3.4 Second-order Adjoint Mode (Adjoint over Tangent) . . . 147
C.3.5 Second-order Adjoint Mode (Adjoint over Adjoint) . . . 148

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

$$\nabla^k f = \nabla^k f(\mathbf{x}, \mathbf{x}') \equiv \frac{\partial^k \mathbf{y}}{\partial \mathbf{x}^k}(\mathbf{x}, \mathbf{x}') \in D^{m \times \bigtimes_{l=1}^{k} n}.$$

We denote

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

and we use $*$ to denote the entire range of an index. For example, $[\nabla f]_i^*$ denotes the ith row and $[\nabla f]_*^j$ the jth column of the Jacobian, respectively. Algorithmic differentiation is implemented by 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.

1.1.2 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] We assume that the arithmetic inside f is completely defined (through overloading of the *elemental functions*; see below) for variables from 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 (Thread-)local tapes

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

1.2.6 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.

1.2.7 Repeated evaluation of tape

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

1.2.8 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

1.2.9 User-defined local Jacobians interface

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

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

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

for $\mathbf{x}^{(1)}, \mathbf{x}^{(2)} \in D^n$ over arbitrary base data types D. The computational cost of accumulating the whole Hessian over D becomes $O(n^2)\cdot \mathrm{Cost}(f)$ 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

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

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

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

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 $O(m\cdot n)\cdot \mathrm{Cost}(f)$ 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

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

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

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

1.4.4 Generic third- and higher-order adjoints

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

Chapter 2

First-order Tangent Mode

2.1 Purpose

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

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

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

$$\begin{aligned}\mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{y}^{(1)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1)} \rangle \equiv \frac{\partial f}{\partial \mathbf{x}} \cdot \mathbf{x}^{(1)}.\end{aligned} \tag{2.1}$$

2.2 Example

2.2.1 Example Text

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

```
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 (2.1) 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 int n=xv.size(), m=yv.size();
  vector<DCO_TYPE> x(n), y(m);
  for (int i=0;i<n;i++) { value(x[i])=xv[i]; derivative(x[i])=xt[i]; }
  f(x,y);
  for (int 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 (2.1) 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;
}
```

2.2.2 Example Results

The following output is generated:

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

2.3 New dco/c++ Features

2.3.1 dco.hpp

dco/c++ header to be included.

2.3.2 DCO_BASE_TYPE

Variable instantiation type for generic AD modes.

2.3.3 gt1s<DCO_BASE_TYPE>

Generic first-order tangent mode.

2.3.4 DCO_MODE

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

2.3.5 `gt1s<DCO_BASE_TYPE>::type`

Generic first-order tangent type.

2.3.6 `DCO_TYPE`

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

2.3.7 `value`

Returns reference to value of argument.

2.3.8 `derivative`

Returns reference to derivative of argument.

Chapter 3

First-order Adjoint Mode

3.1 Purpose

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

The first-order adjoint version

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

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

$$\begin{aligned}\mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)} \cdot \frac{\partial f}{\partial \mathbf{x}} \rangle \equiv \mathbf{x}_{(1)} + \left(\frac{\partial f}{\partial \mathbf{x}}\right)^T \cdot \mathbf{y}_{(1)}.\end{aligned} \tag{3.1}$$

3.2 Example

3.2.1 Example Text

For the same function $f : \mathbb{R}^4 \to \mathbb{R}^2$ used in Chapter 2 the driver computes (3.1) 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 "../gt1s/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();
  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]); derivative(y[i])=ya[i];
  }
  for (int i=0;i<n;i++) derivative(x[i])=xa[i];
  DCO_MODE::global_tape->interpret_adjoint();
  for (int i=0;i<n;i++) xa[i]=derivative(x[i]);
  for (int 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 (3.1) 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;
}
```

3.2.2 Example Results

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
```

3.3 New dco/c++ Features

3.3.1 ga1s<DCO_BASE_TYPE>

Generic first-order adjoint scalar mode.

3.3.2 ga1s<DCO_BASE_TYPE>::type

Generic first-order adjoint scalar type.

3.3.3 DCO_TAPE_TYPE

Type of tape associated with DCO_MODE.

3.3.4 DCO_MODE::global_tape

Global tape pointer associated with DCO_MODE.

3.3.5 DCO_TAPE_TYPE::create

Tape creator returns pointer to tape as DCO_TAPE_TYPE*.

3.3.6 DCO_TAPE_TYPE::register_variable

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

3.3.7 DCO_TAPE_TYPE::register_output_variable

Marks argument as a dependent variable in associated tape.

3.3.8 DCO_TAPE_TYPE::interpret_adjoint

Adjoint interpretation of entire tape.

3.3.9 DCO_TAPE_TYPE::remove

Deallocates global tape.

Chapter 4

Second-order Tangent Mode

4.1 Purpose

The second-order tangent version

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

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

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{y}^{(2)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(2)} \rangle \\ \mathbf{y}^{(1)} &:= \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1)} \rangle \\ \mathbf{y}^{(1,2)} &:= \langle \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \rangle + \langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1,2)} \rangle. \end{aligned} \tag{4.1}$$

4.2 Example

4.2.1 Example Text

For the same function $f : \mathbb{R}^4 \to \mathbb{R}^2$ used in Chapter 2 the driver computes (4.1) 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 "../gt1s/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 int n=xv.size(), m=yv.size();
  vector<DCO_TYPE> x(n), y(m);
  for (int 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 (int i=0;i<m;i++) {
    yv[i]=passive_value(y[i]);
    yt1[i]=derivative(value(y[i]));
    yt2[i]=value(derivative(y[i]));
    yt1t2[i]=derivative(derivative(y[i]));
  }
}
```

The main program initializes all variables on the right-hand side of (4.1) 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;
}
```

4.2.2 Example Results

The following output is generated:

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

4.3 New dco/c++ Features

4.3.1 `passive_value`

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

Chapter 5

Second-order Adjoint Mode

5.1 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

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{y}^{(2)} &:= \langle f(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\ \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \frac{\partial f}{\partial \mathbf{x}} \rangle \\ \mathbf{x}_{(1)}^{(2)} &:= \mathbf{x}_{(1)}^{(2)} + \langle \mathbf{y}_{(1)}^{(2)}, \frac{\partial f}{\partial \mathbf{x}} \rangle + \langle \mathbf{y}_{(1)}, \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(2)} \rangle. \end{aligned} \tag{5.1}$$

5.2 Example

5.2.1 Example Text

For the same function $f : \mathbb{R}^4 \to \mathbb{R}^2$ used in Chapter 2 the driver computes (5.1) for $\mathbf{x} \hat{=}$ xv, $\mathbf{x}_{(1)} \hat{=}$ xa1, $\mathbf{x}^{(2)} \hat{=}$ xt2, $\mathbf{x}_{(1)}^{(2)} \hat{=}$ xa1t2, $\mathbf{y} \hat{=}$ yv, $\mathbf{y}_{(1)} \hat{=}$ ya1, $\mathbf{y}^{(2)} \hat{=}$ yt2, and $\mathbf{y}_{(1)}^{(2)} \hat{=}$ 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 "../gt1s/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 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]);
    derivative(value(x[i]))=xt2[i];
  }
  f(x,y);
  for (int i=0;i<n;i++) {
    value(derivative(x[i]))=xa1[i];
    derivative(derivative(x[i]))=xa1t2[i];
  }
  for (int 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 (int i=0;i<n;i++) {
    xa1t2[i]=derivative(derivative(x[i]));
    xa1[i]=value(derivative(x[i]));
  }
  for (int 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 (5.1) and calls the driver.

```
#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++)
```