

dco/c++ User Guide

version 3.4.1

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

nag[®]



Copyright Statement

©The Numerical Algorithms Group Limited and
Software and Tools for Computational Engineering @ RWTH Aachen University, 2019

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 User Guide;
- distributing the User Guide 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 and STCE.

Version 3.4 released March 2019

ISBN 978-1-85206-219-4

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

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

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

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

Contents

1	Summary of Features: v3.4.1	14
1.1	First-order Tangent Mode	14
1.1.1	Generic first-order scalar tangents	14
1.1.2	Generic first-order vector tangents	15
1.1.3	Preaccumulation through use of expression templates	15
1.2	First-order Adjoint Mode	15
1.2.1	Generic first-order scalar adjoints	15
1.2.2	Generic first-order vector adjoints	15
1.2.3	Global “blob” tape	15
1.2.4	Global “chunk” tape	15
1.2.5	Global “file” tape (v3.2)	15
1.2.6	Thread-local tape (v3.4)	15
1.2.7	Distinct data types for values, partial derivatives and adjoints with all kinds of tapes (v3.2)	16
1.2.8	(Thread-)local tapes	16
1.2.9	Preaccumulation through use of expression templates	16
1.2.10	Repeated evaluation of tape	16
1.2.11	Adjoint callback interface	16
1.2.12	User-defined local Jacobians interface	16
1.2.13	User-driven preaccumulation of local Jacobians (v3.2)	16
1.2.14	Multiple adjoint vectors for a single tape (v3.2)	16
1.2.15	Modulo Adjoint Propagation (v3.3)	17
1.2.16	Sparse Tape Interpretation (v3.3)	17
1.3	Second- and Higher-order Tangent Mode	17
1.3.1	Generic second-order scalar tangents	17
1.3.2	Generic second-order vector tangents	17
1.3.3	Other generic second-order tangents	17
1.3.4	Generic third- and higher-order tangents	18
1.4	Second- and Higher-order Adjoint Mode	18
1.4.1	Generic second-order scalar adjoints	18
1.4.2	Generic second-order vector adjoints	18

1.4.3	Other generic second-order adjoints	18
1.4.4	Generic third- and higher-order adjoints	19
1.5	Additional Functionality	19
1.5.1	Vectorized (AVX) tangent and adjoint modes (v3.4)	19
1.5.2	Debugging capabilities (v3.4)	19
2	General Remarks / Essentials	20
2.1	C++ Standard	20
2.2	Memory Allocation	20
2.3	Parallelization	20
2.4	Performance	21
2.5	Type Casting	21
2.6	File I/O	21
2.7	Feature Selection	21
2.8	Coupling with Other Template Libraries	22
2.8.1	Standard Template Library (STL)	22
2.8.2	Boost Interval Arithmetic Library	23
2.8.3	Eigen Library	23
2.8.4	Stan Math Library	23
3	General Functions and Traits	24
3.1	Functions	24
3.1.1	<code>dco::value</code>	24
3.1.2	<code>dco::derivative</code>	24
3.1.3	<code>dco::passive_value</code>	24
3.1.4	<code>dco::tape</code>	25
3.1.5	<code>dco::tape_index</code>	25
3.1.6	<code>dco::size_of</code>	25
3.1.7	Allocation / Deallocation Functions	26
3.2	Traits	26
3.2.1	<code>dco::mode</code>	26
3.2.2	<code>dco::trait_size_of</code>	26
4	First-order Tangent Mode	28
4.1	Purpose	28
4.2	Example	28
4.3	New dco/c++ Features	29
4.3.1	<code>dco.hpp</code>	29
4.3.2	<code>DCO_BASE_TYPE</code>	29
4.3.3	<code>gt1s<DCO_BASE_TYPE></code>	29
4.3.4	<code>DCO_MODE</code>	30

4.3.5	<code>gtls<DCO_BASE_TYPE>::type</code>	30
4.3.6	<code>DCO_TYPE</code>	30
4.3.7	<code>value</code>	30
4.3.8	<code>derivative</code>	30
5	First-order Adjoint Mode	31
5.1	Purpose	31
5.2	Example	31
5.3	New dco/c++ Features	33
5.3.1	<code>ga1s<DCO_BASE_TYPE></code>	33
5.3.2	<code>ga1s<DCO_BASE_TYPE>::type</code>	33
5.3.3	<code>DCO_TAPE_TYPE</code>	33
5.3.4	<code>DCO_MODE::global_tape</code>	33
5.3.5	<code>DCO_TAPE_TYPE::create</code>	33
5.3.6	<code>DCO_TAPE_TYPE::register_variable</code>	33
5.3.7	<code>DCO_TAPE_TYPE::register_output_variable</code>	33
5.3.8	<code>DCO_TAPE_TYPE::interpret_adjoint</code>	33
5.3.9	<code>DCO_TAPE_TYPE::remove</code>	33
6	Vector Type	34
6.1	Purpose	34
6.1.1	Remarks	34
6.2	Example	34
6.2.1	Without Diverging Control Flow	34
6.2.2	With Diverging Control Flow	35
6.3	New dco/c++ Features	36
6.3.1	<code>gv<DCO_BASE_TYPE, VECTOR_SIZE></code>	36
6.3.2	<code>operator[]</code>	36
6.3.3	<code>template <size_t m> subrange_t<m> subranges<m>(n)</code>	36
6.3.4	Comparison Operator	36
6.3.5	<code>vmask(...)</code>	36
7	Vectorized Tangent Type	37
7.1	Purpose	37
7.1.1	Remarks	37
7.2	Example	37
7.2.1	Without Diverging Control Flow	38
7.2.2	With Diverging Control Flow	38
8	Vectorized Adjoint Type	39
8.1	Purpose	39

8.1.1	Remarks	39
8.2	Example	39
8.2.1	Without Diverging Control Flow	40
8.2.2	With Diverging Control Flow	40
9	First-order Adjoint Mode: Blob/Chunk Tapes	42
9.1	Purpose	42
9.2	Example	42
9.3	New dco/c++ Features	43
9.3.1	<code>dco::tape_options</code>	43
9.3.2	<code>size_t dco::tape_options::chunk_size_in_byte()</code>	43
9.3.3	<code>void dco::tape_options::set_chunk_size_in_kbyte(double)</code>	43
9.3.4	<code>void dco::tape_options::set_chunk_size_in_mbyte(double)</code>	43
9.3.5	<code>void dco::tape_options::set_chunk_size_in_gbyte(double)</code>	43
9.3.6	<code>size_t dco::tape_options::blob_size_in_byte()</code>	43
9.3.7	<code>void dco::tape_options::set_blob_size_in_kbyte(double)</code>	43
9.3.8	<code>void dco::tape_options::set_blob_size_in_mbyte(double)</code>	43
9.3.9	<code>void dco::tape_options::set_blob_size_in_gbyte(double)</code>	43
9.3.10	<code>DCO_TAPE_TYPE::create</code>	43
10	First-order Adjoint Mode: File Tape	44
10.1	Purpose	44
10.2	Example	44
10.3	New dco/c++ Features	45
11	First-order Adjoint Mode: Tape Data Types	46
11.1	Purpose	46
11.2	Example	46
11.3	Example Text	46
11.4	Example Results	47
11.5	New dco/c++ Features	47
11.5.1	<code>ga1s<DCO_VALUE_TYPE,DCO_PARTIAL_TYPE,DCO_ADJOINT_TYPE></code>	47
12	Second-order Tangent Mode	48
12.1	Purpose	48
12.2	Example	48
12.3	New dco/c++ Features	50
12.3.1	<code>passive_value</code>	50
13	Second-order Adjoint Mode	51
13.1	Purpose	51

13.2 Example	51
13.3 New dco/c++ Features	53
14 Approximate Second-order Adjoint Mode	54
14.1 Purpose	54
14.2 Example	54
14.3 New dco/c++ Features	56
15 Third-order Tangent Mode	57
15.1 Purpose	57
15.2 Example	58
15.3 New dco/c++ Features	60
16 Third-order Adjoint Mode	61
16.1 Purpose	61
16.2 Example	62
16.3 New dco/c++ Features	65
17 First-order Tangent Mode: Vector Mode	66
17.1 Purpose	66
17.2 Example	66
17.3 New dco/c++ Features	68
17.3.1 derivative	68
18 First-order Adjoint Mode: Vector Mode	69
18.1 Purpose	69
18.2 Example	69
18.3 New dco/c++ Features	71
19 Adjoint First-order Scalar Mode: Multiple Tapes	72
19.1 Purpose	72
19.2 Example	72
19.3 New dco/c++ Features	73
19.3.1 <code>gaism<DCO_BASE_TYPE></code>	73
19.3.2 <code>gaism<DCO_BASE_TYPE>::type</code>	73
19.3.3 <code>DCO_TAPE_TYPE::create</code>	74
19.3.4 <code>DCO_TAPE_TYPE::remove</code>	74
20 Adjoint First-order Vector Mode: Multiple Tapes	75
20.1 Purpose	75
20.2 Example	75
20.3 New dco/c++ Features	77

21 Binary Compatible Passive Type: gbc	78
21.1 Purpose	78
21.2 Example	78
21.3 Example Text	80
21.4 Example Results	80
21.5 New dco/c++ Features	80
21.5.1 gbc<DCO_ACTIVE_TYPE>::type	80
21.5.2 create_gbc<GBCP_TYPE>(ACTIVE_TYPE)	80
21.5.3 DCO_MODE::order	80
21.5.4 DCO_MODE::gbc_order	81
22 First-order Adjoint Mode: Basic Checkpointing	82
22.1 Purpose	82
22.2 Example	82
22.3 New dco/c++ Features	84
22.3.1 DCO_TAPE_TYPE::iterator_t	84
22.3.2 DCO_TAPE_TYPE::get_position	84
22.3.3 DCO_TAPE_TYPE::interpret_adjoint_and_reset_to	84
22.3.4 dco::mode<DCO_TYPE>	84
22.3.5 DCO_MODE::value_t	84
22.3.6 DCO_MODE::external_adjoint_object_t	85
22.3.7 DCO_TAPE_TYPE::create_callback_object	85
22.3.8 DCO_EAO_TYPE::register_input	85
22.3.9 DCO_EAO_TYPE::write_data	85
22.3.10 DCO_EAO_TYPE::register_output	85
22.3.11 DCO_TAPE_TYPE::insert_callback	85
22.3.12 DCO_EAO_TYPE::read_data	85
22.3.13 DCO_EAO_TYPE::get_output_adjoint	85
22.3.14 DCO_EAO_TYPE::increment_input_adjoint	85
22.3.15 DCO_TAPE_TYPE::reset_to	85
22.3.16 DCO_TAPE_TYPE::get_tape_memory_size()	85
23 Second-order Adjoint Mode: Basic Checkpointing	86
23.1 Purpose	86
23.2 Example	86
23.3 New dco/c++ Features	87
24 First-order Adjoint Mode: Equidistant Checkpointing of Evolutions	88
24.1 Purpose	88
24.2 Example	88
24.3 New dco/c++ Features	90

25 Second-order Adjoint Mode: Equidistant Checkpointing of Evolutions	91
25.1 Purpose	91
25.2 Example	91
25.3 New dco/c++ Features	93
26 First-order Adjoint Mode: Recursive Checkpointing of Evolutions	94
26.1 Purpose	94
26.2 Example	94
26.3 New dco/c++ Features	100
26.3.1 my_external_adjoint_object_t	100
27 Second-order Adjoint Mode: Recursive Checkpointing of Evolutions	101
27.1 Purpose	101
27.2 Example	101
27.3 New dco/c++ Features	104
28 First-order Adjoint Mode: Checkpointing Ensembles	105
28.1 Purpose	105
28.2 Example	105
28.3 New dco/c++ Features	107
29 Second-order Adjoint Mode: Checkpointing Ensembles	108
29.1 Purpose	108
29.2 Example	108
29.3 New dco/c++ Features	109
30 First-order Adjoint Mode: Embedding Adjoint Source Code	110
30.1 Purpose	110
30.2 Example	110
30.3 New dco/c++ Features	113
31 Second-order Adjoint Mode: Embedding First-order Adjoint Code	114
31.1 Purpose	114
31.2 Example	114
31.3 New dco/c++ Features	116
32 First-order Adjoint Mode: Local Preaccumulation Using First-order Tangent Code	117
32.1 Purpose	117
32.2 Example	117
32.3 New dco/c++ Features	119
33 Second-order Adjoint Mode: Local Preaccumulation Using First-order Tangent	

Code	120
33.1 Purpose	120
33.2 Example	120
33.3 New dco/c++ Features	121
34 First-order Adjoint Mode: Edge Insertion for Preaccumulation of Local Gradients	122
34.1 Purpose	122
34.2 Example	122
34.3 New dco/c++ Features	124
34.3.1 DCO_MODE::local_gradient_t	124
34.3.2 DCO_TAPE_TYPE::create_local_gradient_object	124
34.3.3 DCO_LOCAL_GRADIENT_TYPE::put	124
34.3.4 DCO_LOCAL_GRADIENT_TYPE::finalize	124
35 Second-order Adjoint Mode: Edge Insertion for Preaccumulation of Local Gradients	125
35.1 Purpose	125
35.2 Example	125
35.3 New dco/c++ Features	126
36 First-order Adjoint Mode: User-Defined Adjoints	127
36.1 Purpose	127
36.2 Example	127
36.3 New dco/c++ Features	129
37 Second-order Adjoint Mode: User-Defined First-order Adjoints	130
37.1 Purpose	130
37.2 Example	130
37.3 New dco/c++ Features	131
38 First-order Adjoint Mode: Local Jacobian Preaccumulation	132
38.1 Purpose	132
38.2 Remarks	132
38.3 Example	132
38.3.1 Example Text	132
38.3.2 Example Results	133
38.4 New dco/c++ Features	134
38.4.1 jacobian_preaccumulator_t	134
38.4.2 jacobian_preaccumulator_t::start()	134
38.4.3 jacobian_preaccumulator_t::register_output(DCO_TYPE&)	134
38.4.4 jacobian_preaccumulator_t::finish()	134

39 Second-order Adjoint Mode: Local Jacobian Preaccumulation	135
39.1 Purpose	135
39.2 Example	135
39.2.1 Example Text	135
39.2.2 Example Results	136
39.3 New dco/c++ Features	136
40 First-order Adjoint Mode: Multiple Adjoint Vectors	137
40.1 Purpose	137
40.2 Example	137
40.3 Example Text	137
40.4 Example Results	138
40.5 New dco/c++ Features	139
40.5.1 <code>adjoint_vector<DCO_TAPE_TYPE,DCO_ADJOINT_TYPE></code>	139
40.5.2 <code>derivative(DCO_T&,DCO_ADJOINT_VECTOR_TYPE&)</code>	139
40.5.3 <code>DCO_ADJOINT_VECTOR_TYPE::interpret_adjoint()</code>	139
41 Modulo Adjoint Propagation	140
41.1 Purpose	140
41.2 Optimization	140
41.3 Example	140
41.4 New dco/c++ Features	142
41.4.1 <code>gaiss_mod<DCO_BASE_TYPE></code>	142
42 Sparse Tape Interpretation	143
42.1 Purpose	143
42.2 Example	143
42.3 New dco/c++ Features	144
42.3.1 <code>TAPE_T::sparse_interpret()</code>	144
43 Logging	145
43.1 Purpose	145
43.2 Example	145
43.3 New dco/c++ Features	145
43.3.1 <code>enum dco::log_level_enum</code>	145
43.3.2 <code>dco::log_level_enum& dco::logger::level()</code>	146
43.3.3 <code>FILE*& dco::logger::stream()</code>	146
44 Explicit Type Cast to Specified Type	147
44.1 Purpose	147
44.1.1 Remarks	147

44.2 Example	147
44.3 New dco/c++ Features	148
44.3.1 DCO_ENABLE_EXPLICIT_TYPE_CAST_TO(TYPE)	148
45 Huge Pages	149
45.1 Purpose	149
45.2 Usage	149
A Status of User Interface	150
A.1 Gold	150
A.2 Silver	150
A.3 Bronze	156
A.4 Undefined	157
B Case Study: Diffusion	160
B.1 Purpose	160
B.1.1 Problem Description	161
B.1.2 Numerical Method	161
B.1.3 Algorithmic Differentiation	166
B.1.4 Optimization	168
B.1.5 Code	171
B.2 Function	173
B.3 Observations	174
B.4 Gradient	175
B.4.1 Finite Differences	175
B.4.2 <code>gt1s<double></code>	176
B.4.3 <code>ga1s<double></code>	177
B.4.4 <code>ga1s<double></code> + Equidistant Checkpointing	178
B.4.5 <code>ga1s<double></code> + Symbolic Derivative of Linear Solver	180
B.4.6 <code>ga1s<double></code> + Symbolic Derivative of Linear Solver and Equidistant Checkpointing	184
B.5 Hessian	189
B.5.1 Finite Differences	189
B.5.2 <code>gt1s<gt1s<double>::type></code>	190
B.5.3 <code>ga1s<gt1s<double>::type></code>	191
B.5.4 <code>ga1s<gt1s<double>::type></code> + Equidistant Checkpointing	192
B.5.5 <code>ga1s<gt1s<double>::type></code> + Symbolic Derivative of Linear Solver	195
B.5.6 <code>ga1s<gt1s<double>::type></code> + Symbolic Derivative of Linear Solver and Equidistant Checkpointing	198
C Race	204
C.1 Purpose	204

C.2	Gradient	205
C.2.1	Central Finite Differences	205
C.2.2	First-order Tangent Mode	205
C.2.3	First-order Tangent Mode: Vector Mode	206
C.2.4	First-order Adjoint Mode	207
C.3	Hessian	209
C.3.1	Second-order Central Finite Differences	209
C.3.2	Second-order Tangent Mode	210
C.3.3	Second-order Adjoint Mode (Tangent over Adjoint)	211
C.3.4	Second-order Adjoint Mode (Adjoint over Tangent)	212
C.3.5	Second-order Adjoint Mode (Adjoint over Adjoint)	213

Chapter 1

Summary of Features: v3.4.1

We consider implementations of multivariate vector functions

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

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

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

We denote

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

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 .

¹We 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 $y_{(1)} \in D^m$

$$x_{(1)} = \nabla f^T \cdot y_{(1)} \in D^n$$

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 Thread-local tape (v3.4)

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

1.2.7 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.

1.2.8 (Thread-)local tapes

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

1.2.9 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.

1.2.10 Repeated evaluation of tape

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

1.2.11 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

1.2.12 User-defined local Jacobians interface

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

1.2.13 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.

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

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

1.2.15 Modulo Adjoint Propagation (v3.3)

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

1.2.16 Sparse Tape Interpretation (v3.3)

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

1.3 Second- and Higher-order Tangent Mode

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

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

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 $O(n^2) \cdot \text{Cost}(f)$. 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

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

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 $O(n^2) \cdot \text{Cost}(f)$ 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

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

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

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

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

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

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

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

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

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

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.

1.5 Additional Functionality

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

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

1.5.2 Debugging capabilities (v3.4)

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

Chapter 2

General Remarks / Essentials

2.1 C++ Standard

`dco/c++` is fully compliant with C++98. Some features, however, might require C++11. For example, the following features require C++11:

- the vector data type (`dco::gv<...>`); see Ch. 6, 7, and 8.
- the binary compatible passive type (`dco::gbcp<...>`); see Ch. 21.
- explicit type casting (`static_cast<double>(...)`); see Sec. 2.5.

See Sec. 2.7 for more details on how to enable these features.

2.2 Memory Allocation

- Default configuration uses a *blob tape*, i.e. no dynamic growth of memory during recording. Selection of a *chunk tape* is done at precompile time by the preprocessor (see next section).
- Allocated memory is not initialized.
- Under Linux, huge pages are supported. Those might enhance performance immensely (we've seen up to 30%). It can be switched on/off via the `tape_options` object. See Sec. 45.

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

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

2.3 Parallelization

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

- For explicit thread safety, you can use multiple tape versions: `dco::ga1sm<T>`, `dco::ga1vm<T>`, etc.

2.4 Performance

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

2.5 Type Casting

Usually, it is not desired to cast a `dco/c++` type to a non-dco type (e.g. `double`), since `dco/c++` is then unable to compute derivatives of that data flow. Nonetheless, there are cases, where a cast is actually what you want to do (e.g. cast to `int` for an index calculation). We therefore allow explicit cast operations to specified data type (implicit casts are not allowed). Use carefully. See Sec. 44.

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

2.6 File I/O

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

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

2.7 Feature Selection

Definition of the following preprocessor variables select the respective feature.

- `DCO_DEBUG` (default: OFF): Improved safety through enhanced checking, e.g. overflow checks (has performance impact).
- `DCO_CHUNK_TAPE` (default: undefined): Switches to *chunk tape*. The default chunk size is 128MB.
- `DCO_NO_THREADLOCAL` (default: undefined): Switches thread-local global tapes off (i.e. no use of `thread_local` keyword).
- `DCO_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. 41 for details.
- `DCO_SKIP_WINDOWS_H_INCLUDE` (default: undefined): Under Windows, `windows.h` is included by default. This is required for getting the total physical memory size (for automatic blob tape size calculation). If define is set, `windows.h` is not included and when using the blob tape, you must define the environment variable `DCO_MAX_ALLOCATION` (see Sec. 2.2) or use the chunk tape.
- `DCO_USE_STD_COMPLEX` (default: undefined): Removes specialization of `std::complex` for dco/c++ types. See Sec. 2.8.
- `DCO_CPP11` (default: compiler dependent): This variable is used throughout the source to distinguish whether the compiler has C++11 support. It is automatically set, if the preprocessor statement `__cplusplus > 199711L` evaluates to `true` or if you are using VS2015 or newer. If your compiler supports C++11, but doesn't set `__cplusplus` accordingly, `DCO_CPP11` can be set manually.
- `DCO_AGRESSIVE_INLINE` (default: platform dependent): See Sec. 2.4.
- `DCO_NO_AGRESSIVE_INLINE` (default: undefined): See Sec. 2.4.

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_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.

2.8 Coupling with Other Template Libraries

2.8.1 Standard Template Library (STL)

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

- `std::complex`
Unfortunately, there are some issues with the implementation of `std::complex` under Windows. Therefore, a specialized implementation of `std::complex` for dco/c++ types is used by default. This can be switched off by using the define `DCO_USE_STD_COMPLEX`.

2.8.2 Boost Interval Arithmetic Library

To obtain semi-local derivative information with dco/c++ the `interval<T,Policies>` type from the Boost Interval Arithmetic Library¹ can be used as a base type of dco/c++ types. Instead of using the default policies, you should specify the rounding and checking policies to enable transcendental and trigonometric functions. As an example the following type can be used.

```
1 boost::numeric::interval<double, boost::numeric::interval_lib::policies<
2   boost::numeric::interval_lib::save_state<
3     boost::numeric::interval_lib::rounded_transc_std<double> >,
4   boost::numeric::interval_lib::checking_base<double> > >
```

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

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

2.8.3 Eigen Library

As Eigen states on its webpage², for supporting custom data types as template argument, `Eigen::NumTraits` needs to be specialized. In addition, some compilation issues might occur, since Eigen doesn't seem to fully support template expression types as template arguments. If you run into this issue, please contact uwe.naumann@nag.co.uk; we have been working with Eigen for some time and have a fork resolving some of those problems.

2.8.4 Stan Math Library

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

¹https://www.boost.org/doc/libs/1_62_0/libs/numeric/interval/doc/interval.htm

²https://eigen.tuxfamily.org/dox/TopicCustomizing_CustomScalar.html

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<...>`, `gals<...>`, ...) 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`:

```
1  enum size_of_mode {
2      size_of_stack = 1,
3      size_of_allocated_stack = 2,
4      size_of_internal_adjoint_vector = 4,
5      size_of_checkpoints = 8,
6      size_of_default = size_of_stack | size_of_internal_adjoint_vector
7  };
```

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

```

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

```

3.1.7 Allocation / Deallocation Functions

Definition

```

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

```

Description

Allocate `size` bytes of uninitialized storage. `use_huge_pages` is currently only available under Linux and uses `mmap` to allocate huge pages (see Ch. 45). The memory allocated with `dco::alloc` needs to be deallocated using `dco::dealloc` function. Returns 0 if unable to allocate requested memory.

3.2 Traits

3.2.1 `dco::mode`

Definition

```

1      template <typename T> struct mode;
2      typename mode::value_t;
3      typename mode::passive_t;
4      typename mode::derivative_t;
5      typename mode::tape_t;
6      typename mode::local_gradient_t;
7      typename mode::external_adjoint_object_t;
8      typename mode::jacobian_preaccumulator_t;
9      bool mode::is_dco_type;
10     bool mode::is_adjoint_type;
11     bool mode::is_tangent_type;
12     int mode::order;

```

Description

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

3.2.2 `dco::trait_size_of`

Definition

```

1      template <typename T> struct trait_size_of {
2          size_t get(const T&);
3      };

```

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

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

4.2 Example

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

```
1 template<typename T>
2 void f(const vector<T>& x, vector<T>& y) {
3     T v=tan(x[2]*x[3]); T w=x[1]-v;
4     y[0]=x[0]*v/w;
5     y[1]=y[0]*x[1];
6 }
```

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

```
1 #include <vector>
2 using namespace std;
3
4 #include "dco.hpp"
5 using namespace dco;
6
7 typedef double DCO_BASE_TYPE;
8 typedef gt1s<DCO_BASE_TYPE> DCO_MODE;
9 typedef DCO_MODE::type DCO_TYPE;
10
11 #include "f.hpp"
```

```

12
13 void driver(
14     const vector<double>& xv, const vector<double>& xt,
15     vector<double>& yv, vector<double>& yt
16 ) {
17     const size_t n=xv.size(), m=yv.size();
18     vector<DCO_TYPE> x(n), y(m);
19     for (size_t i=0;i<n;i++) { value(x[i])=xv[i]; derivative(x[i])=xt[i]; }
20     f(x,y);
21     for (size_t i=0;i<m;i++) { yv[i]=value(y[i]); yt[i]=derivative(y[i]); }
22 }

```

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

```

1 #include<iostream>
2 #include<vector>
3 using namespace std;
4
5 #include "driver.hpp"
6
7 int main() {
8     const int m=2, n=4; cout.precision(15);
9     vector<double> xv(n), xt(n), yv(m), yt(m);
10    for (int i=0;i<n;i++) { xv[i]=1; xt[i]=1; }
11    driver(xv,xt,yv,yt);
12    for (int i=0;i<m;i++)
13        cout << "y[" << i << "]= " << yv[i] << endl;
14    for (int i=0;i<m;i++)
15        cout << "y^{(1)}[" << i << "]= " << yt[i] << endl;
16    return 0;
17 }

```

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

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

5.2 Example

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

```
1  #include <vector>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  typedef double DCO_BASE_TYPE;
8  typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
9  typedef DCO_MODE::type DCO_TYPE;
10 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
11
12 #include "f.hpp"
13
14 void driver(
15     const vector<double>& xv, vector<double>& xa,
16     vector<double>& yv, vector<double>& ya
```

```

17 ) {
18     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
19     size_t n=xv.size(), m=yv.size();
20     vector<DCO_TYPE> x(n), y(m);
21     for (size_t i=0;i<n;i++) {
22         x[i]=xv[i];
23         DCO_MODE::global_tape->register_variable(x[i]);
24     }
25     f(x,y);
26     for (size_t i=0;i<m;i++) {
27         DCO_MODE::global_tape->register_output_variable(y[i]);
28         yv[i]=value(y[i]); derivative(y[i])=ya[i];
29     }
30     for (size_t i=0;i<n;i++) derivative(x[i])=xa[i];
31
32     DCO_MODE::global_tape->write_to_dot();
33     DCO_MODE::global_tape->interpret_adjoint();
34     for (size_t i=0;i<n;i++) xa[i]=derivative(x[i]);
35     for (size_t i=0;i<m;i++) ya[i]=derivative(y[i]);
36     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
37 }

```

The main program initializes all variables on the right-hand side of (5.1) followed by calling the driver and printing the results.

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  #include "driver.hpp"
6
7  int main() {
8      const int n=4, m=2; cout.precision(15);
9      vector<double> xv(n), xa(n), yv(m), ya(m);
10     for (int i=0;i<n;i++) { xv[i]=1; xa[i]=1; }
11     for (int i=0;i<m;i++) ya[i]=1;
12     driver(xv,xa,yv,ya);
13     for (int i=0;i<m;i++)
14         cout << "y[" << i << "]= " << yv[i] << endl;
15     for (int i=0;i<n;i++)
16         cout << "x_{(1)}[" << i << "]= " << xa[i] << endl;
17     return 0;
18 }

```

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

Vector Type

6.1 Purpose

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

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

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

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

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

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

6.1.1 Remarks

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

6.2 Example

6.2.1 Without Diverging Control Flow

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

```
1 template<typename T>
2 void f(const T& x, T& y) {
3     T v=sin(x*x);
```

```

4   T w=-cos(v);
5   y=v*w;
6 }

```

The driver computes the elements of y by running a vectorized version of f .

```

1  typedef double DCO_BASE_TYPE;
2  static const unsigned long VECTOR_SIZE = 8;
3  typedef gv<DCO_BASE_TYPE,VECTOR_SIZE> DCO_MODE;
4  typedef DCO_MODE::type DCO_TYPE;
5
6  void driver(
7      const vector<double>& x,
8      vector<double>& y
9  ) {
10     const size_t n=x.size();
11     DCO_TYPE vec_x, vec_y;
12     for (auto range : dco::subranges<VECTOR_SIZE>(n)) {
13         for (auto i : range) { vec_x[i.sub_index]=x[i.global_index]; }
14         f(vec_x,vec_y);
15         for (auto i : range) { y[i.global_index]=vec_y[i.sub_index]; }
16     }
17 }

```

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

```

1
2  int main() {
3      const int m=12, n=12; cout.precision(15);
4      vector<double> x(n), y(m);
5      for (int i=0;i<n;i++) { x[i]=sin(static_cast<double>(i+1)); }
6      driver(x,y);
7      for (int i=0;i<m;i++)
8          cout << "y[" << i << "]= " << y[i] << endl;
9      return 0;

```

The following output is generated:

```

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

```

6.2.2 With Diverging Control Flow

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

```

1  template<typename T>
2  void f(const T& x, T& y) {
3      T v=sin(x*x);
4      T w=-cos(v);
5      y=v*w;
6      if (y < -0.5) y += sin(exp(x));
7  }

```

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

```

1  template<typename T>
2  void f(const T& x, T& y) {
3      T v=sin(x*x);
4      T w=-cos(v);
5      y  = v*w;
6      y += dco::vmask(y < -0.5, sin(exp(x)));
7  }
```

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

The following output is generated:

```

y[0]=0.21746389190323
y[1]=-0.153056484768117
[...]
y[10]=-0.22019755696259
y[11]=-0.357012857533558
```

6.3 New dco/c++ Features

6.3.1 `gv<DCO_BASE_TYPE, VECTOR_SIZE>`

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

6.3.2 `operator []`

```

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

Returns reference to *i*th element of vector.

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

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

6.3.4 Comparison Operator

```

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

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

6.3.5 `vmask(...)`

```

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

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

Chapter 7

Vectorized Tangent Type

7.1 Purpose

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

7.1.1 Remarks

Please see remark in Sec. [6.1.1](#).

7.2 Example

We consider the same examples as in Sec. [6.2](#).

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

```
1
2 int main() {
3     const int m=12, n=12; cout.precision(15);
4     vector<double> xv(n), xt(n), yv(m), yt(m);
5     for (int i=0;i<n;i++) { xv[i]=sin(static_cast<double>(i+1)); xt[i]=1; }
6     driver(xv,xt,yv,yt);
7     for (int i=0;i<m;i++)
8         cout << "y[" << i << "]= " << yv[i] << endl;
9     for (int i=0;i<n;i++)
10        cout << "y^{(1)}[" << i << "]= " << yt[i] << endl;
11    return 0;
```

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

```
1 static const unsigned long VECTOR_SIZE = 8;
2 typedef gv<double,VECTOR_SIZE> DCO_BASE_MODE;
3 typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
4 typedef gt1s<DCO_BASE_TYPE> DCO_MODE;
5 typedef DCO_MODE::type DCO_TYPE;
6
7 void driver(
8     const vector<double>& xv, vector<double>& xt,
```

```

9         vector<double>& yv, vector<double>& yt
10     ) {
11         const size_t n=xv.size();
12         DCO_TYPE vec_x, vec_y;
13         for (auto range : dco::subranges<VECTOR_SIZE>(n)) {
14             for (auto i : range) {
15                 value(vec_x)[i.sub_index]=xv[i.global_index];
16                 derivative(vec_x)[i.sub_index]=xt[i.global_index];
17             }
18             f(vec_x,vec_y);
19             for (auto i : range) {
20                 yv[i.global_index]=value(vec_y)[i.sub_index];
21                 yt[i.global_index]=derivative(vec_y)[i.sub_index];
22             }
23         }
24     }

```

7.2.1 Without Diverging Control Flow

The following output is generated:

```

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

```

7.2.2 With Diverging Control Flow

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

The following output is generated:

```

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

```

Chapter 8

Vectorized Adjoint Type

8.1 Purpose

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

8.1.1 Remarks

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

Please see remark in Sec. 6.1.1.

8.2 Example

We consider the same examples as in Sec. 6.2.

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

```
1
2 int main() {
3     const int m=12, n=12; cout.precision(15);
4     vector<double> xv(n), xa(n), yv(m), ya(m);
5     for (int i=0;i<n;i++) { xv[i]=sin(static_cast<double>(i+1)); xa[i]=1; }
6     for (int i=0;i<m;i++) ya[i]=1;
7     driver(xv,xa,yv,ya);
8     for (int i=0;i<m;i++)
9         cout << "y[" << i << "]= " << yv[i] << endl;
10    for (int i=0;i<n;i++)
11        cout << "x_{(1)}[" << i << "]= " << xa[i] << endl;
12    return 0;
```

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

```
1 static const unsigned long VECTOR_SIZE = 8;
2 typedef gv<double,VECTOR_SIZE> DCO_BASE_MODE;
3 typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
4 typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
```

```

5  typedef DCO_MODE::type DCO_TYPE;
6  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
7
8  void driver(
9      const vector<double>& xv, vector<double>& xa,
10     vector<double>& yv, vector<double>& ya
11 ) {
12     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
13
14     const size_t n=xv.size();
15     DCO_TYPE vec_x, vec_y;
16     for (auto range : dco::subranges<VECTOR_SIZE>(n)) {
17         for (auto i : range) {
18             value(vec_x)[i.sub_index]=xv[i.global_index];
19         }
20         DCO_MODE::global_tape->register_variable(vec_x);
21         f(vec_x,vec_y);
22         DCO_MODE::global_tape->register_output_variable(vec_y);
23         for (auto i : range) {
24             yv[i.global_index]=value(vec_y)[i.sub_index];
25             derivative(vec_y)[i.sub_index]=ya[i.global_index];
26             derivative(vec_x)[i.sub_index]=xa[i.global_index];
27         }
28         DCO_MODE::global_tape->interpret_adjoint();
29         for (auto i : range) {
30             ya[i.global_index]=derivative(vec_y)[i.sub_index];
31             xa[i.global_index]=derivative(vec_x)[i.sub_index];
32         }
33         DCO_MODE::global_tape->reset();
34     }
35     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
36 }

```

8.2.1 Without Diverging Control Flow

The following output is generated:

```

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

```

8.2.2 With Diverging Control Flow

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

The following output is generated:

```

y[0]=0.21746389190323
[...]
y[11]==-0.357012857533553

```

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

Chapter 9

First-order Adjoint Mode: Blob/Chunk Tapes

9.1 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.

9.2 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.

```
1  ...
2  dco::tape_options o;
3  o.set_chunk_size_in_byte(1024); // for chunk tape
4  o.set_blob_size_in_mbyte(1); // for blob tape
5  DCO_M::global_tape=DCO_TAPE_T::create(o);
6  std::cout << o.chunk_size_in_byte() << std::endl;
7  ...
```

A `tape_options` object allows for the chunk size to be specified, for example, one kilobyte. It is passed as an argument to the tape creation routine. If blob tape is used, the blob tape size will be one megabyte in this example.

9.3 New dco/c++ Features

9.3.1 `dco::tape_options`

Tape configuration object.

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

Get chunk size in bytes.

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

Set chunk size in kilobytes.

9.3.4 `void dco::tape_options::set_chunk_size_in_mbyte(double)`

Set chunk size in megabytes.

9.3.5 `void dco::tape_options::set_chunk_size_in_gbyte(double)`

Set chunk size in gigabytes.

9.3.6 `size_t dco::tape_options::blob_size_in_byte()`

Get blob size in bytes.

9.3.7 `void dco::tape_options::set_blob_size_in_kbyte(double)`

Set blob size in kilobytes.

9.3.8 `void dco::tape_options::set_blob_size_in_mbyte(double)`

Set blob size in megabytes.

9.3.9 `void dco::tape_options::set_blob_size_in_gbyte(double)`

Set blob size in gigabytes.

9.3.10 `DCO_TAPE_TYPE::create`

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

Chapter 10

First-order Adjoint Mode: File Tape

10.1 Purpose

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

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

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

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

Second- and higher-order adjoint work correspondingly.

10.2 Example

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

```
1 ...  
2   dco::tape_options o;  
3   o.write_to_file()=true;  
4   o.set_chunk_size_in_byte(1024);  
5   DCO_M::global_tape=DCO_TAPE_T::create(o);  
6   std::cout << o.chunk_size_in_byte() << std::endl;  
7   ...  
8   DCO_TAPE_T::remove(DCO_M::global_tape);  
9   ...
```

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.

10.3 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 11

First-order Adjoint Mode: Tape Data Types

11.1 Purpose

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

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

11.2 Example

11.3 Example Text

Replace lines 7–10 in the driver in Sec. 5.2 with

```
1 typedef long double DCO_VALUE_TYPE;
2 typedef double DCO_PARTIAL_TYPE;
3 typedef float DCO_ADJOINT_TYPE;
4 typedef gals<DCO_VALUE_TYPE,DCO_PARTIAL_TYPE,DCO_ADJOINT_TYPE> DCO_MODE;
5 typedef DCO_MODE::type DCO_TYPE;
6 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.

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 #include "driver.hpp"
6
7 int main() {
8     const int n=4, m=2;
9     vector<double> xv(n), xa(n), yv(m), ya(m);
10    for (int i=0;i<n;i++) { xv[i]=1; xa[i]=1; }
```

```

11     for (int i=0;i<m;i++) ya[i]=1;
12     driver(xv,xa,yv,ya);
13     cout.precision(18);
14     for (int i=0;i<m;i++)
15         cout << "y[" << i << "]= " << yv[i] << endl;
16     cout.precision(6);
17     for (int i=0;i<n;i++)
18         cout << "x_{(1)}[" << i << "]= " << xa[i] << endl;
19     return 0;
20 }

```

11.4 Example Results

The following output is generated:

```

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

```

11.5 New dco/c++ Features

11.5.1 `gals<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 12

Second-order Tangent Mode

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

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{y}^{(2)} &:= \left\langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(2)} \right\rangle \\ \mathbf{y}^{(1)} &:= \left\langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1)} \right\rangle \\ \mathbf{y}^{(1,2)} &:= \left\langle \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \right\rangle + \left\langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(1,2)} \right\rangle. \end{aligned} \tag{12.1}$$

12.2 Example

For the same function $f : \mathbb{R}^4 \rightarrow \mathbb{R}^2$ used in Ch. 4 the driver computes (12.1) for $\mathbf{x} \hat{=} \mathbf{xv}$, $\mathbf{x}^{(1)} \hat{=} \mathbf{xt1}$, $\mathbf{x}^{(2)} \hat{=} \mathbf{xt2}$, $\mathbf{x}^{(1,2)} \hat{=} \mathbf{xt1t2}$, $\mathbf{y} \hat{=} \mathbf{yv}$, $\mathbf{y}^{(1)} \hat{=} \mathbf{yt1}$, $\mathbf{y}^{(2)} \hat{=} \mathbf{yt2}$, and $\mathbf{y}^{(1,2)} \hat{=} \mathbf{yt1t2}$.

```
1 #include<iostream>
2 using namespace std;
3
4 #include "dco.hpp"
5 using namespace dco;
6 typedef gtls<double> DCO_BASE_MODE;
7 typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
8 typedef gtls<DCO_BASE_TYPE> DCO_MODE;
9 typedef DCO_MODE::type DCO_TYPE;
10
11 #include "f.hpp"
12
```

```

13 void driver(
14     const vector<double>& xv,
15     const vector<double>& xt1,
16     const vector<double>& xt2,
17     const vector<double>& xt1t2,
18     vector<double>& yv,
19     vector<double>& yt1,
20     vector<double>& yt2,
21     vector<double>& yt1t2
22 ) {
23     const size_t n=xv.size(), m=yv.size();
24     vector<DCO_TYPE> x(n), y(m);
25     for (size_t i=0;i<n;i++) {
26         value(value(x[i]))=xv[i];
27         derivative(value(x[i]))=xt1[i];
28         value(derivative(x[i]))=xt2[i];
29         derivative(derivative(x[i]))=xt1t2[i];
30     }
31     f(x,y);
32     for (size_t i=0;i<m;i++) {
33         yv[i]=passive_value(y[i]);
34         yt1[i]=derivative(value(y[i]));
35         yt2[i]=value(derivative(y[i]));
36         yt1t2[i]=derivative(derivative(y[i]));
37     }
38 }

```

The main program initializes all variables on the right-hand side of (12.1) followed by calling the driver and printing the results.

```

1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  #include "driver.hpp"
6
7  int main() {
8      const int n=4, m=2; cout.precision(15);
9      vector<double> xv(n), xt1(n), xt2(n), xt1t2(n);
10     vector<double> yv(m), yt1(m), yt2(m), yt1t2(m);
11     for (int i=0;i<n;i++) { xv[i]=1; xt1[i]=1; xt2[i]=1; xt1t2[i]=1;}
12     driver(xv,xt1,xt2,xt1t2,yv,yt1,yt2,yt1t2);
13     for (int i=0;i<m;i++)
14         cout << "y[" << i << "]= " << yv[i] << endl;
15     for (int i=0;i<m;i++)
16         cout << "y^{(1)}[" << i << "]= " << yt1[i] << endl;
17     for (int i=0;i<m;i++)
18         cout << "y^{(2)}[" << i << "]= " << yt2[i] << endl;
19     for (int i=0;i<m;i++)
20         cout << "y^{(1,2)}[" << i << "]= " << yt1t2[i] << endl;
21     return 0;
22 }

```

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
```

12.3 New dco/c++ Features

12.3.1 `passive_value`

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

Chapter 13

Second-order Adjoint Mode

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

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{y}^{(2)} &:= \left\langle \frac{\partial f}{\partial \mathbf{x}}, \mathbf{x}^{(2)} \right\rangle \\ \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \left\langle \mathbf{y}_{(1)}, \frac{\partial f}{\partial \mathbf{x}} \right\rangle \\ \mathbf{x}_{(1)}^{(2)} &:= \mathbf{x}_{(1)}^{(2)} + \left\langle \mathbf{y}_{(1)}^{(2)}, \frac{\partial f}{\partial \mathbf{x}} \right\rangle + \left\langle \mathbf{y}_{(1)}, \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(2)} \right\rangle. \end{aligned} \tag{13.1}$$

13.2 Example

For the same function $f : \mathbb{R}^4 \rightarrow \mathbb{R}^2$ used in Ch. 4 the driver computes (13.1) for $\mathbf{x} \hat{=} \mathbf{xv}$, $\mathbf{x}_{(1)} \hat{=} \mathbf{xa1}$, $\mathbf{x}^{(2)} \hat{=} \mathbf{xt2}$, $\mathbf{x}_{(1)}^{(2)} \hat{=} \mathbf{xa1t2}$, $\mathbf{y} \hat{=} \mathbf{yv}$, $\mathbf{y}_{(1)} \hat{=} \mathbf{ya1}$, $\mathbf{y}^{(2)} \hat{=} \mathbf{yt2}$, and $\mathbf{y}_{(1)}^{(2)} \hat{=} \mathbf{ya1t2}$,

```
1 #include<vector>
2 using namespace std;
3
4 #include "dco.hpp"
5 using namespace dco;
6
7 typedef gt1s<double> DCO_BASE_MODE;
8 typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
9 typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
10 typedef DCO_MODE::type DCO_TYPE;
11 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
12
```

```

13 #include "f.hpp"
14
15 void driver(
16     const vector<double>& xv,
17     const vector<double>& xt2,
18     vector<double>& xa1,
19     vector<double>& xa1t2,
20     vector<double>& yv,
21     vector<double>& yt2,
22     vector<double>& ya1,
23     vector<double>& ya1t2
24 ) {
25     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
26     const size_t n=xv.size(), m=yv.size();
27     vector<DCO_TYPE> x(n), y(m);
28     for (size_t i=0;i<n;i++) {
29         x[i]=xv[i];
30         DCO_MODE::global_tape->register_variable(x[i]);
31         derivative(value(x[i]))=xt2[i];
32     }
33     f(x,y);
34     for (size_t i=0;i<n;i++) {
35         value(derivative(x[i]))=xa1[i];
36         derivative(derivative(x[i]))=xa1t2[i];
37     }
38     for (size_t i=0;i<m;i++) {
39         yv[i]=passive_value(y[i]);
40         yt2[i]=derivative(value(y[i]));
41         DCO_MODE::global_tape->register_output_variable(y[i]);
42         value(derivative(y[i]))=ya1[i];
43         derivative(derivative(y[i]))=ya1t2[i];
44     }
45     DCO_MODE::global_tape->interpret_adjoint();
46     for (size_t i=0;i<n;i++) {
47         xa1t2[i]=derivative(derivative(x[i]));
48         xa1[i]=value(derivative(x[i]));
49     }
50     for (size_t i=0;i<m;i++) {
51         ya1t2[i]=derivative(derivative(y[i]));
52         ya1[i]=value(derivative(y[i]));
53     }
54     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
55 }

```

The main program initializes all variables on the right-hand side of (13.1) followed by calling the driver and printing the results.

```

1 #include<iostream>
2 #include<vector>
3 using namespace std;
4
5 #include "driver.hpp"
6
7 int main() {

```

```

8   const int n=4,m=2; cout.precision(15);
9   vector<double> xv(n), xa1(n), xt2(n), xa1t2(n);
10  vector<double> yv(m), ya1(m), yt2(m), ya1t2(m);
11  for (int i=0;i<n;i++) { xv[i]=1; xt2[i]=1; xa1[i]=1; xa1t2[i]=0; }
12  for (int i=0;i<m;i++) { ya1[i]=1; ya1t2[i]=0; }
13  driver(xv,xt2,xa1,xa1t2,yv,yt2,ya1,ya1t2);
14  for (int i=0;i<m;i++)
15      cout << "y[" << i << "]= " << yv[i] << endl;
16  for (int i=0;i<n;i++)
17      cout << "x_{(1)}[" << i << "]= " << xa1[i] << endl;
18  for (int i=0;i<m;i++)
19      cout << "y^{(2)}[" << i << "]= " << yt2[i] << endl;
20  for (int i=0;i<n;i++)
21      cout << "x_{(1)}^{(2)}[" << i << "]= " << xa1t2[i] << endl;
22  return 0;
23 }

```

```

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

```

13.3 New dco/c++ Features

None.

Chapter 14

Approximate Second-order Adjoint Mode

14.1 Purpose

The approximate second-order adjoint version

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

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

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ \mathbf{y}^{(2)} &\approx \langle f(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\ \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \frac{\partial f}{\partial \mathbf{x}} \rangle \\ \mathbf{x}_{(1)}^{(2)} &\approx \mathbf{x}_{(1)}^{(2)} + \langle \mathbf{y}_{(1)}^{(2)}, \frac{\partial f}{\partial \mathbf{x}} \rangle + \langle \mathbf{y}_{(1)}, \frac{\partial^2 f}{\partial \mathbf{x}^2}, \mathbf{x}^{(2)} \rangle. \end{aligned} \tag{14.1}$$

14.2 Example

```
1  #include <vector>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  typedef double DCO_BASE_TYPE;
8  typedef gals<DCO_BASE_TYPE> DCO_MODE;
9  typedef DCO_MODE::type DCO_TYPE;
10 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
11
12 #include "f.hpp"
13
```

```

14 void driver(
15     const vector<double>& xv, vector<double>& xa,
16     vector<double>& yv, vector<double>& ya
17 ) {
18     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
19     size_t n=xv.size(), m=yv.size();
20     vector<DCO_TYPE> x(n), y(m);
21     for (size_t i=0;i<n;i++) {
22         x[i]=xv[i];
23         DCO_MODE::global_tape->register_variable(x[i]);
24     }
25     f(x,y);
26     for (size_t i=0;i<m;i++) {
27         DCO_MODE::global_tape->register_output_variable(y[i]);
28         yv[i]=value(y[i]); derivative(y[i])=ya[i];
29     }
30     for (size_t i=0;i<n;i++) derivative(x[i])=xa[i];
31     DCO_MODE::global_tape->interpret_adjoint();
32     for (size_t i=0;i<n;i++) xa[i]=derivative(x[i]);
33     for (size_t i=0;i<m;i++) ya[i]=derivative(y[i]);
34     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
35 }
36
37
38 #include <cfloat>
39
40 void fd_driver (
41     const vector<double>& xv, vector<double>& xa, vector<double>& xa1t2x,
42     vector<double>& yv, vector<double>& yt2x, vector<double>& ya) {
43     size_t n=xv.size(), m=yv.size();
44     vector<double> h(n), xap(n), xvp(n), yvp(m), yap(m);
45     for (size_t i=0;i<m;i++) yap[i]=ya[i];
46     for (size_t i=0;i<n;i++) {
47         xap[i]=xa[i];
48         h[i]=(xv[i]==0) ? sqrt(DBL_EPSILON) : sqrt(DBL_EPSILON)*abs(xv[i]);
49         xvp[i]=xv[i]+h[i];
50     }
51     driver(xv,xa,yv,ya);
52     driver(xvp,xap,yvp,yap);
53     for (size_t i=0;i<m;i++)
54         yt2x[i]=(yvp[i]-yv[i])/h[i];
55     for (size_t i=0;i<n;i++)
56         xa1t2x[i]=(xap[i]-xa[i])/h[i];
57 }
58
59
60 #include <iostream>
61 #include <vector>
62 using namespace std;
63
64 #include "driver.hpp"
65
66
67 int main() {
68     const int n=4, m=2; cout.precision(15);
69     vector<double> xv(n), xa(n), xa1t2x(n), yv(m), yt2x(m), ya(m);

```

```

10   for (int i=0;i<n;i++) { xv[i]=1; xa[i]=1; }
11   for (int i=0;i<m;i++) ya[i]=1;
12   fd_driver(xv,xa,xa1t2x,yv,yt2x,ya);
13   for (int i=0;i<m;i++)
14     cout << "y[" << i << "]= " << yv[i] << endl;
15   for (int i=0;i<n;i++)
16     cout << "x_{(1)}[" << i << "]= " << xa[i] << endl;
17   for (int i=0;i<m;i++)
18     cout << "y^{(2)}[" << i << "]~=" << yt2x[i] << endl;
19   for (int i=0;i<n;i++)
20     cout << "x_{(1)}^{(2)}[" << i << "]~=" << xa1t2x[i] << endl;
21   return 0;
22 }

```

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

```

14.3 New dco/c++ Features

None.

Chapter 15

Third-order Tangent Mode

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

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

15.2 Example

```

1  #include<iostream>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  typedef gt1s<double> DCO_BASE_BASE_MODE;
8  typedef DCO_BASE_BASE_MODE::type DCO_BASE_BASE_TYPE;
9  typedef gt1s<DCO_BASE_BASE_TYPE> DCO_BASE_MODE;
10 typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
11 typedef gt1s<DCO_BASE_TYPE> DCO_MODE;
12 typedef DCO_MODE::type DCO_TYPE;
13
14 #include "f.hpp"
15
16 void driver(
17     const vector<double>& xv,
18     const vector<double>& xt3,
19     const vector<double>& xt1,
20     const vector<double>& xt1t3,
21     const vector<double>& xt2,
22     const vector<double>& xt2t3,
23     const vector<double>& xt1t2,
24     const vector<double>& xt1t2t3,
25     vector<double>& yv,
26     vector<double>& yt3,
27     vector<double>& yt1,
28     vector<double>& yt1t3,
29     vector<double>& yt2,
30     vector<double>& yt2t3,
31     vector<double>& yt1t2,
32     vector<double>& yt1t2t3
33 ) {
34     const size_t n=xv.size(), m=yv.size();
35     vector<DCO_TYPE> x(n), y(m);
36     for (size_t i=0;i<n;i++) {
37         value(value(value(x[i])))=xv[i];
38         value(value(derivative(x[i])))=xt1[i];
39         value(derivative(value(x[i])))=xt2[i];
40         derivative(value(value(x[i])))=xt3[i];
41         value(derivative(derivative(x[i])))=xt1t2[i];
42         derivative(derivative(value(x[i])))=xt2t3[i];
43         derivative(value(derivative(x[i])))=xt1t3[i];
44         derivative(derivative(derivative(x[i])),xt1t2t3[i];
45     }
46     f(x,y);
47     for (size_t j=0;j<m;j++) {
48         yt1t2t3[j]=derivative(derivative(derivative(y[j])));
49         yt2t3[j]=derivative(derivative(value(y[j])));
50         yt1t3[j]=derivative(value(derivative(y[j])));
51         yt1t2[j]=value(derivative(derivative(y[j])));

```

```

52     yt1[j]=value(value(derivative(y[j])));
53     yt2[j]=value(derivative(value(y[j])));
54     yt3[j]=derivative(value(value(y[j])));
55     yv[j]=value(value(value(y[j])));
56 }
57 }

1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  #include "driver.hpp"
6
7  int main() {
8      const int n=4, m=2; cout.precision(15);
9      vector<double> x(n), xt3(n), xt1(n), xt1t3(n), xt2(n), xt2t3(n), xt1t2(n),
10         xt1t2t3(n);
11      vector<double> y(m), yt3(m), yt1(m), yt1t3(m), yt2(m), yt2t3(m), yt1t2(m),
12         yt1t2t3(m);
13      for (int i=0;i<n;i++) x[i]=xt3[i]=xt1[i]=xt1t3[i]=xt2[i]=xt2t3[i]=xt1t2[i]=
14         xt1t2t3[i]=1;
15      driver(x,xt3,xt1,xt1t3,xt2,xt2t3,xt1t2,xt1t2t3,
16         y,yt3,yt1,yt1t3,yt2,yt2t3,yt1t2,yt1t2t3);
17      for (int j=0;j<m;j++)
18         cout << "y[" << j << "]= " << y[j] << endl;
19      for (int j=0;j<m;j++)
20         cout << "y^{(3)}[" << j << "]= " << yt3[j] << endl;
21      for (int j=0;j<m;j++)
22         cout << "y^{(1)}[" << j << "]= " << yt1[j] << endl;
23      for (int j=0;j<m;j++)
24         cout << "y^{(1,3)}[" << j << "]= " << yt1t3[j] << endl;
25      for (int j=0;j<m;j++)
26         cout << "y^{(2)}[" << j << "]= " << yt2[j] << endl;
27      for (int j=0;j<m;j++)
28         cout << "y^{(2,3)}[" << j << "]= " << yt2t3[j] << endl;
29      for (int j=0;j<m;j++)
30         cout << "y^{(1,2)}[" << j << "]= " << yt1t2[j] << endl;
31      for (int j=0;j<m;j++)
32         cout << "y^{(1,2,3)}[" << j << "]= " << yt1t2t3[j] << endl;
33      return 0;
34 }

```

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
```

15.3 New dco/c++ Features

None.

Chapter 16

Third-order Adjoint Mode

16.1 Purpose

The third-order adjoint version

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

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

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

16.2 Example

```

1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  #include "dco.hpp"
6  using namespace dco;
7
8  typedef gt1s<double> DCO_BASE_BASE_MODE;
9  typedef DCO_BASE_BASE_MODE::type DCO_BASE_BASE_TYPE;
10 typedef gt1s<DCO_BASE_BASE_TYPE> DCO_BASE_MODE;
11 typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
12 typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
13 typedef DCO_MODE::type DCO_TYPE;
14 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
15
16 #include "f.hpp"
17
18 void driver(
19     const vector<double>& xv,
20     const vector<double>& xt3,
21     const vector<double>& xt2,
22     const vector<double>& xt2t3,
23     vector<double>& xa1,
24     vector<double>& xa1t3,
25     vector<double>& xa1t2,
26     vector<double>& xa1t2t3,
27     vector<double>& yv,
28     vector<double>& yt3,
29     vector<double>& yt2,
30     vector<double>& yt2t3,
31     vector<double>& ya1,
32     vector<double>& ya1t3,
33     vector<double>& ya1t2,
34     vector<double>& ya1t2t3
35 ) {
36     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
37     const size_t n=xv.size(), m=yv.size();
38     vector<DCO_TYPE> x(n), y(m);
39     for (size_t i=0;i<n;i++) {
40         DCO_MODE::global_tape->register_variable(x[i]);
41         value(value(value(x[i])))=xv[i];
42         derivative(value(value(x[i])))=xt3[i];
43         value(derivative(value(x[i])))=xt2[i];
44         derivative(derivative(value(x[i])))=xt2t3[i];
45     }
46     f(x,y);
47     for (size_t i=0;i<n;i++) {
48         value(value(derivative(x[i])))=xa1t3[i];
49         derivative(value(derivative(x[i])))=xa1t3[i];
50         value(derivative(derivative(x[i])))=xa1t2[i];
51         derivative(derivative(derivative(x[i])))=xa1t2t3[i];

```

```

52     }
53     for (size_t i=0;i<m;i++) {
54         yv[i]=value(value(value(y[i])));
55         yt3[i]=derivative(value(value(y[i])));
56         yt2[i]=value(derivative(value(y[i])));
57         yt2t3[i]=derivative(derivative(value(y[i])));
58         DCO_MODE::global_tape->register_output_variable(y[i]);
59         derivative(derivative(derivative(y[i])))=ya1t2t3[i];
60         value(derivative(derivative(y[i])))=ya1t2[i];
61         derivative(value(derivative(y[i])))=ya1t3[i];
62         value(value(derivative(y[i])))=ya1[i];
63     }
64     DCO_MODE::global_tape->interpret_adjoint();
65     for (size_t i=0;i<n;i++) {
66         xa1t2t3[i]=derivative(derivative(derivative(x[i])));
67         xa1t2[i]=value(derivative(derivative(x[i])));
68         xa1t3[i]=derivative(value(derivative(x[i])));
69         xa1[i]=value(value(derivative(x[i])));
70     }
71     for (size_t i=0;i<m;i++) {
72         ya1t2t3[i]=derivative(derivative(derivative(y[i])));
73         ya1t2[i]=value(derivative(derivative(y[i])));
74         ya1t3[i]=derivative(value(derivative(y[i])));
75         ya1[i]=value(value(derivative(y[i])));
76     }
77     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
78 }

1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  #include "driver.hpp"
6
7  int main() {
8      const int n=4,m=2;
9      vector<double> x(n), xt3(n), xa1(n), xa1t3(n), xt2(n), xt2t3(n), xa1t2(n),
10         xa1t2t3(n);
11      vector<double> y(m), yt3(m), ya1(m), ya1t3(m), yt2(m), yt2t3(m), ya1t2(m),
12         ya1t2t3(m);
13      // initialization of inputs
14      for (int i=0;i<n;i++) x[i]=xt3[i]=xt2[i]=xt2t3[i]=xa1[i]=xa1t3[i]=xa1t2[i]=
15         xa1t2t3[i]=1;
16      for (int j=0;j<m;j++) ya1[j]=ya1t3[j]=ya1t2[j]=ya1t2t3[j]=1;
17      // driver
18      driver(x,xt3,xt2,xt2t3,xa1,xa1t3,xa1t2,xa1t2t3,
19         y,yt3,yt2,yt2t3,ya1,ya1t3,ya1t2,ya1t2t3);
20      // results
21      for (int j=0;j<m;j++)
22         cout << "y[" << j << "]= " << y[j] << endl;
23      for (int j=0;j<m;j++)
24         cout << "y~{(3)}[" << j << "]= " << yt3[j] << endl;
25      for (int i=0;i<n;i++)
26         cout << "x_{(1)}[" << i << "]= " << xa1[i] << endl;

```

```

24     for (int i=0;i<n;i++)
25         cout << "x_{(1)}^{(3)}[" << i << "]= " << xa1t3[i] << endl;
26     for (int j=0;j<m;j++)
27         cout << "y_{(2)}[" << j << "]= " << yt2[j] << endl;
28     for (int j=0;j<m;j++)
29         cout << "y_{(2,3)}[" << j << "]= " << yt2t3[j] << endl;
30     for (int i=0;i<n;i++)
31         cout << "x_{(1)}^{(2)}[" << i << "]= " << xa1t2[i] << endl;
32     for (int i=0;i<n;i++)
33         cout << "x_{(1)}^{(2,3)}[" << i << "]= " << xa1t2t3[i] << endl;
34     for (int j=0;j<m;j++)
35         cout << "y_{(1)}[" << j << "]= " << ya1[j] << endl;
36     for (int j=0;j<m;j++)
37         cout << "y_{(1)}^{(3)}[" << j << "]= " << ya1t3[j] << endl;
38     for (int j=0;j<m;j++)
39         cout << "y_{(1)}^{(2)}[" << j << "]= " << ya1t2[j] << endl;
40     for (int j=0;j<m;j++)
41         cout << "y_{(1)}^{(2,3)}[" << j << "]= " << ya1t2t3[j] << endl;
42     return 0;
43 }

```

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

```

$y_{\{(1)\}^{\{(2,3)\}}[1]=1$

16.3 New dco/c++ Features

None.

Chapter 17

First-order Tangent Mode: Vector Mode

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

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

17.2 Example

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

```
1  #include<iostream>
2  #include<vector>
3  using namespace std;
4  #include "dco.hpp"
5  using namespace dco;
6  #include "f.hpp"
7
8  const DCO_INTEGRAL_TAPE_INT l=4;
9
10 typedef double DCO_BASE_MODE;
11 typedef gt1v<DCO_BASE_MODE, 1> DCO_MODE;
12 typedef DCO_MODE::type DCO_TYPE;
13
14 void driver(
15     const vector<double>& x,
16     const vector<vector<double>> >& xt1,
17     vector<double>& y,
18     vector<vector<double>> >& yt1
```

```

19 ) {
20     const size_t n=x.size(), m=y.size();
21     vector<DCO_TYPE> t1v_x(n), t1v_y(m);
22     for (size_t i=0;i<n;i++) {
23         value(t1v_x[i])=x[i];
24         for (DCO_TAPE_INT j=0;j<1;j++)
25             derivative(t1v_x[i])[j]=xt1[i][j];
26     }
27     f(t1v_x,t1v_y);
28     for (size_t i=0;i<m;i++) {
29         y[i]=value(t1v_y[i]);
30         for (DCO_TAPE_INT j=0;j<1;j++)
31             yt1[i][j]=derivative(t1v_y[i])[j];
32     }
33 }

1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  #include "driver.hpp"
6
7  int main() {
8      cout.precision(15);
9      const int m=2, n=4;
10     vector<double> x(n),y(m);
11     vector<vector<double>> > xt1(n, vector<double>(n)), yt1(m, vector<double>(n));
12     for (int i=0;i<n;i++) {
13         x[i]=1;
14         for (int j=0;j<n;j++) xt1[i][j]=0;
15         xt1[i][i]=1;
16     }
17     driver(x,xt1,y,yt1);
18     for (int j=0;j<m;j++)
19         cout << "y[" << j << "]=" << y[j] << endl;
20     for (int j=0;j<m;j++)
21         for (int i=0;i<n;i++)
22             cout << "y^{(1)}[" << j << "]" << i << "]="
23             << yt1[j][i] << endl;
24     return 0;
25 }

```

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

```

17.3 New dco/c++ Features

17.3.1 `derivative`

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

Chapter 18

First-order Adjoint Mode: Vector Mode

18.1 Purpose

The first-order vector adjoint version

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

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

$$\begin{aligned} \mathbf{y} &:= f(\mathbf{x}) \\ X_{(1)} &:= X_{(1)} + \langle Y_{(1)} \cdot \frac{\partial f}{\partial \mathbf{x}} \rangle \equiv X_{(1)} + \left(\frac{\partial f}{\partial \mathbf{x}} \right)^T \cdot Y_{(1)}. \end{aligned} \quad (18.1)$$

18.2 Example

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

```
1  #include <vector>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  const int l=5;
8
9  typedef double DCO_BASE_TYPE;
10 typedef ga1v<DCO_BASE_TYPE,l> DCO_MODE;
11 typedef DCO_MODE::type DCO_TYPE;
12 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
13
14 #include "f.hpp"
15
16 void driver(
17     const vector<double>& xv, vector<vector<double>> & xa,
```

```

18     vector<double>& yv, vector<vector<double> >& ya
19 ) {
20     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
21     int n=xv.size(), m=yv.size();
22     vector<DCO_TYPE> x(n), y(m);
23     for (int i=0;i<n;i++) {
24         x[i]=xv[i];
25         DCO_MODE::global_tape->register_variable(x[i]);
26     }
27     f(x,y);
28     for (int i=0;i<m;i++) {
29         DCO_MODE::global_tape->register_output_variable(y[i]);
30         yv[i]=value(y[i]);
31         for (int j=0;j<l;j++) derivative(y[i])[j] = ya[i][j];
32     }
33     for (int i=0;i<n;i++) {
34         for (int j=0;j<l;j++) derivative(x[i])[j] = xa[i][j];
35     }
36     DCO_MODE::global_tape->interpret_adjoint();
37     for (int i=0;i<n;i++) {
38         for (int j=0;j<l;j++) xa[i][j]=derivative(x[i])[j];
39     }
40     for (int i=0;i<m;i++) {
41         for (int j=0;j<l;j++) ya[i][j]=derivative(y[i])[j];
42     }
43     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
44 }

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  #include "driver.hpp"
6
7  int main() {
8      const int n=4, m=5; cout.precision(15);
9      vector<double> xv(n), yv(m);
10     vector<vector<double> > xa(n, vector<double>(m)), ya(m, vector<double>(m));
11     for (int i=0;i<n;i++) {
12         xv[i]=1;
13         for (int j=0;j<m;j++) xa[i][j]=0;
14     }
15     for (int i=0;i<m;i++) {
16         for (int j=0;j<m;j++) ya[i][j]=0;
17         ya[i][i]=1;
18     }
19     driver(xv,xa,yv,ya);
20     for (int i=0;i<m;i++)
21         cout << "y[" << i << "]= " << yv[i] << endl;
22     for (int i=0;i<n;i++)
23         for (int j=0;j<m;j++)
24             cout << "x_{(1)}[" << i << "][" << j << "]= " << xa[i][j] << endl;
25     return 0;
26 }

```

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
```

18.3 New dco/c++ Features

None.

Chapter 19

Adjoint First-order Scalar Mode: Multiple Tapes

19.1 Purpose

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

19.2 Example

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

```
1  #include <vector>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  typedef double DCO_BASE_TYPE;
8  typedef ga1sm<DCO_BASE_TYPE> DCO_MODE;
9  typedef DCO_MODE::type DCO_TYPE;
10 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
11
12 #include "f.hpp"
13
14 void driver(
15     const vector<double>& xv, vector<double>& xa,
16     vector<double>& yv, vector<double>& ya
17 ) {
18     DCO_TAPE_TYPE *tape=DCO_TAPE_TYPE::create();
19     size_t n=xv.size(), m=yv.size();
20     vector<DCO_TYPE> x(n), y(m);
21     for (size_t i=0; i<n; i++) {
22         x[i]=xv[i];
23         tape->register_variable(x[i]);
24     }
```

```

25     f(x,y);
26     for (size_t i=0;i<m;i++) {
27         tape->register_output_variable(y[i]);
28         yv[i]=value(y[i]); derivative(y[i])=ya[i];
29     }
30     for (size_t i=0;i<n;i++) derivative(x[i])=xa[i];
31     tape->interpret_adjoint();
32     for (size_t i=0;i<n;i++) xa[i]=derivative(x[i]);
33     for (size_t i=0;i<m;i++) ya[i]=derivative(y[i]);
34     DCO_TAPE_TYPE::remove(tape);
35 }

```

The main program initializes all variables on the right-hand side of (5.1) followed by calling the driver and printing the results.

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  #include "driver.hpp"
6
7  int main() {
8      const int n=4, m=2; cout.precision(15);
9      vector<double> xv(n), xa(n), yv(m), ya(m);
10     for (int i=0;i<n;i++) { xv[i]=1; xa[i]=1; }
11     for (int i=0;i<m;i++) ya[i]=1;
12     driver(xv,xa,yv,ya);
13     for (int i=0;i<m;i++)
14         cout << "y[" << i << "]= " << yv[i] << endl;
15     for (int i=0;i<n;i++)
16         cout << "x_{(1)}[" << i << "]= " << xa[i] << endl;
17     return 0;
18 }

```

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

```

19.3 New dco/c++ Features

19.3.1 ga1sm<DCO_BASE_TYPE>

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

19.3.2 ga1sm<DCO_BASE_TYPE>::type

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

19.3.3 DCO_TAPE_TYPE::create

Tape creator returns pointer to local tape as DCO_TAPE_TYPE*.

19.3.4 DCO_TAPE_TYPE::remove

Deallocates local tape.

Chapter 20

Adjoint First-order Vector Mode: Multiple Tapes

20.1 Purpose

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

20.2 Example

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

```
1  #include <vector>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  const DCO_INTEGRAL_TAPE_INT l=5;
8
9  typedef double DCO_BASE_TYPE;
10 typedef galvm<DCO_BASE_TYPE,l> DCO_MODE;
11 typedef DCO_MODE::type DCO_TYPE;
12 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
13
14 #include "f.hpp"
15
16 void driver(
17     const vector<double>& xv, vector<vector<double> >& xa,
18     vector<double>& yv, vector<vector<double> >& ya
19 ) {
20     DCO_MODE::tape_t *tape=DCO_TAPE_TYPE::create();
21     size_t n=xv.size(), m=yv.size();
22     vector<DCO_TYPE> x(n), y(m);
23     for (size_t i=0;i<n;i++) {
24         x[i]=xv[i];
25         tape->register_variable(x[i]);
```

```

26     }
27     f(x,y);
28     for (size_t i=0;i<m;i++) {
29         tape->register_output_variable(y[i]);
30         yv[i]=value(y[i]);
31         for (int j=0;j<l;j++) derivative(y[i])[j] = ya[i][j];
32     }
33     for (size_t i=0;i<n;i++) {
34         for (DCO_TAPE_INT j=0;j<l;j++) derivative(x[i])[j] = xa[i][j];
35     }
36     tape->interpret_adjoint();
37     for (size_t i=0;i<n;i++) {
38         for (DCO_TAPE_INT j=0;j<l;j++) xa[i][j]=derivative(x[i])[j];
39     }
40     for (size_t i=0;i<m;i++) {
41         for (DCO_TAPE_INT j=0;j<l;j++) ya[i][j]=derivative(y[i])[j];
42     }
43     DCO_TAPE_TYPE::remove(tape);
44 }

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  #include "driver.hpp"
6
7  int main() {
8      const int n=4, m=5; cout.precision(15);
9      vector<double> xv(n), yv(m);
10     vector<vector<double>> > xa(n, vector<double>(m)), ya(m, vector<double>(m));
11     for (int i=0;i<n;i++) {
12         xv[i]=1;
13         for (int j=0;j<m;j++) xa[i][j]=0;
14     }
15     for (int i=0;i<m;i++) {
16         for (int j=0;j<m;j++) ya[i][j]=0;
17         ya[i][i]=1;
18     }
19     driver(xv,xa,yv,ya);
20     for (int i=0;i<m;i++)
21         cout << "y[" << i << "]= " << yv[i] << endl;
22     for (int i=0;i<n;i++)
23         for (int j=0;j<m;j++)
24             cout << "x_{(1)}[" << i << "][" << j << "]= " << xa[i][j] << endl;
25     return 0;
26 }

```

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
```

20.3 New dco/c++ Features

None.

Chapter 21

Binary Compatible Passive Type: gbcp

21.1 Purpose

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

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

21.2 Example

```
1  template<typename T, typename U>
2  void f(const vector<T>& x, vector<U>& y) {
3      T v=tan(x[2]*x[3]); T w=x[1]-v;
4      y[0]=x[0]*v/w;
5      y[1]=y[0]*x[1];
6  }

1  #include <vector>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  typedef double DCO_BASE_TYPE;
8  typedef gals<DCO_BASE_TYPE> DCO_MODE;
9  typedef DCO_MODE::type DCO_TYPE;
10 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
11
12 #include "f.hpp"
13
14 void driver(
```

```

15     const vector<double>& xv, vector<double>& xa,
16     vector<double>& yv, vector<double>& ya, vector<double>& yp
17 ) {
18     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
19     size_t n=xv.size(), m=yv.size();
20     vector<DCO_TYPE> x(n), y(m);
21     for (size_t i=0;i<n;i++) {
22         x[i]=xv[i];
23         DCO_MODE::global_tape->register_variable(x[i]);
24     }
25     f(x,y);
26
27     std::vector<double> py(n);
28 #ifdef DCO_CPP11
29     typedef gbc<DCO_TYPE>::type DCO_GBCP_TYPE;
30     auto const& px = create_gbc<vector<DCO_GBCP_TYPE>>(x);
31     f(px,py);
32 #else
33     for (size_t i=0;i<m;i++) py[i] = dco::value(y[i]);
34 #endif
35     for (size_t i=0;i<m;i++) {
36         DCO_MODE::global_tape->register_output_variable(y[i]);
37         yv[i]=value(y[i]); derivative(y[i])=ya[i];
38     }
39     for (size_t i=0;i<n;i++) derivative(x[i])=xa[i];
40
41     DCO_MODE::global_tape->write_to_dot();
42     DCO_MODE::global_tape->interpret_adjoint();
43     for (size_t i=0;i<n;i++) xa[i]=derivative(x[i]);
44     for (size_t i=0;i<m;i++) ya[i]=derivative(y[i]);
45     for (size_t i=0;i<m;i++) yp[i]=py[i];
46     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
47 }

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  #include "driver.hpp"
6
7  int main() {
8      const int n=4, m=2; cout.precision(15);
9      vector<double> xv(n), xa(n), yv(m), ya(m), yp(m);
10     for (int i=0;i<n;i++) { xv[i]=1; xa[i]=1; }
11     for (int i=0;i<m;i++) ya[i]=1;
12     driver(xv,xa,yv,ya, yp);
13     for (int i=0;i<m;i++)
14         cout << "y[" << i << "]= " << yv[i] << endl;
15     for (int i=0;i<m;i++)
16         cout << "yp[" << i << "]= " << yp[i] << endl;
17     for (int i=0;i<n;i++)
18         cout << "x_{(1)}[" << i << "]= " << xa[i] << endl;
19     return 0;
20 }

```

21.3 Example Text

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

21.4 Example Results

The following output is generated.

```

1 y[0]=-2.79401891249195
2 y[1]=-2.79401891249195
3 yp[0]=-2.79401891249195
4 yp[1]=-2.79401891249195
5 x_{(1)}[0]=-4.5880378249839
6 x_{(1)}[1]=-11.8190644542334
7 x_{(1)}[2]=23.050091083483
8 x_{(1)}[3]=23.050091083483

```

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

21.5 New dco/c++ Features

21.5.1 `gbcp<DCO_ACTIVE_TYPE>::type`

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

21.5.2 `create_gbcpc<GBCP_TYPE>(ACTIVE_TYPE)`

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

```

1 ACTIVE_TYPE x;
2
3 auto& ref_px    = dco::create_gbcpc<GBCP_TYPE>(x);
4 auto pointer_px = dco::create_gbcpc<GBCP_TYPE*>(x);
5
6 auto& ref_px    = dco::create_gbcpc<const GBCP_TYPE>(x);
7 auto pointer_px = dco::create_gbcpc<const GBCP_TYPE*>(x);

```

21.5.3 `DCO_MODE::order`

Integer containing the order N of the corresponding active type.

21.5.4 DCO_MODE::gbc_order

Integer containing the `gbc_order` M of the corresponding active type. Only available for `gbc` types.

Chapter 22

First-order Adjoint Mode: Basic Checkpointing

22.1 Purpose

This section illustrates the *joint reversal* [4] of a subprogram call. The given solution allows for second and higher derivatives to be computed with minimal implementation effort; see Ch. 23.

22.2 Example

```
1  #include "g_gap.hpp"
2
3  template<typename DCO_TYPE>
4  void f(int n, DCO_TYPE& x) {
5      g(n/3,x);
6      g_make_gap(n/3,x);
7      g(n-n/3*2,x);
8  }
9
10 #include "g.hpp"
11
12 template<typename DCO_MODE>
13 void g_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
14     typedef typename DCO_MODE::type DCO_TYPE;
15     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
16     typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
17     typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
18
19     DCO_TAPE_POSITION_TYPE p0=DCO_MODE::global_tape->get_position();
20     const int &n = D->template read_data<int>();
21     const DCO_VALUE_TYPE &xv = D->template read_data<DCO_VALUE_TYPE>();
22     DCO_TYPE x=xv;
23     DCO_MODE::global_tape->register_variable(x);
24     DCO_TYPE x_in=x;
25     DCO_TAPE_POSITION_TYPE p1=DCO_MODE::global_tape->get_position();
26     g(n,x);
```

```

18     cerr << "ts1=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
19     derivative(x)=D->get_output_adjoint();
20     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p1);
21     D->increment_input_adjoint(derivative(x_in));
22     DCO_MODE::global_tape->reset_to(p0);
23 }
24
25 template<typename DCO_TYPE>
26 void g_make_gap(int n, DCO_TYPE &x) {
27     typedef dco::mode<DCO_TYPE> DCO_MODE;
28     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
29     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
30
31     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
32     DCO_VALUE_TYPE xv=D->register_input(x);
33     D->write_data(n); D->write_data(xv);
34     g(n,xv);
35     x=D->register_output(xv);
36     DCO_MODE::global_tape->insert_callback(g_fill_gap<DCO_MODE>,D);
37 }

1  #include<cmath>
2  using namespace std;
3
4  template<typename DCO_TYPE>
5  void g(int n, DCO_TYPE& x) {
6      for (int i=0;i<n;i++) x=sin(x);
7  }

1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  #include "dco.hpp"
6  using namespace dco;
7  typedef gals<double> DCO_MODE;
8  typedef DCO_MODE::type DCO_TYPE;
9  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
10 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
11
12 #include "f.hpp"
13
14 void driver(const int n, double& xv, double& xa) {
15     DCO_TYPE x=xv;
16     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
17     DCO_MODE::global_tape->register_variable(x);
18     DCO_TYPE x_in=x;
19     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
20     f(n,x);
21     derivative(x)=xa;
22     cerr << "ts0=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
23     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
24     xv=value(x);

```

```

25     xa=derivative(x_in);
26     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
27 }

1  #include <cassert>
2  #include <cstdlib>
3  #include <iostream>
4  using namespace std;
5
6  #include "driver.hpp"
7
8  int main() {
9      int n=10; cout.precision(15);
10     double x=2.1,xa1=1.0;
11     driver(n,x,xa1);
12     cout << "x=" << x << endl;
13     cout << "x_{(1)}=" << xa1 << endl;
14     return 0;
15 }

```

The following output is generated for n=10:

```

ts0=0.000335693359375MB
ts1=0.0003204345703125MB

```

```

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

```

22.3 New dco/c++ Features

22.3.1 DCO_TAPE_TYPE::iterator_t

Position in tape.

22.3.2 DCO_TAPE_TYPE::get_position

Returns current position in associated tape.

22.3.3 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.

22.3.4 dco::mode<DCO_TYPE>

DCO_MODE for given DCO_TYPE.

22.3.5 DCO_MODE::value_t

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

22.3.6 DCO_MODE::external_adjoint_object_t

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

22.3.7 DCO_TAPE_TYPE::create_callback_object

Creates external adjoint object and returns pointer to it.

22.3.8 DCO_EAO_TYPE::register_input

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

22.3.9 DCO_EAO_TYPE::write_data

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

22.3.10 DCO_EAO_TYPE::register_output

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

22.3.11 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 inputs and outputs.

22.3.12 DCO_EAO_TYPE::read_data

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

22.3.13 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`.

22.3.14 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`.

22.3.15 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.

22.3.16 DCO_TAPE_TYPE::get_tape_memory_size()

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

Chapter 23

Second-order Adjoint Mode: Basic Checkpointing

23.1 Purpose

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

23.2 Example

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  #include "dco.hpp"
6
7  using namespace dco;
8
9  typedef gt1s<double> DCO_BASE_MODE;
10 typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
11 typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
12 typedef DCO_MODE::type DCO_TYPE;
13 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
14 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
15
16 #include "f.hpp"
17
18 void driver(const int n, double& xv, double& xt2, double& xa1, double& xt2a1) {
19     DCO_TYPE x;
20     passive_value(x)=xv;
21     derivative(value(x))=xt2;
22     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
23     DCO_MODE::global_tape->register_variable(x);
24     DCO_TYPE x_in=x;
25     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
26     f(n,x);
```

```

27     xv=passive_value(x);
28     xt2=derivative(value(x));
29     value(derivative(x))=xa1;
30     derivative(derivative(x))=xt2a1;
31     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
32     xa1=value(derivative(x_in));
33     xt2a1=derivative(derivative(x_in));
34     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
35 }

1  #include <cassert>
2  #include <cstdlib>
3  #include <iostream>
4  using namespace std;
5
6  #include "driver.hpp"
7
8  int main() {
9      cout.precision(15);
10     int n=10;
11     double x=2.1,xa1=1.0,xt2=1.0,xt2a1=0.0;
12     driver(n,x,xt2,xa1,xt2a1);
13     cout << "x=" << x << endl;
14     cout << "x^{(2)}=" << xt2 << endl;
15     cout << "x_{(1)}=" << xa1 << endl;
16     cout << "x_{(1)}^{(2)}=" << xt2a1 << endl;
17     return 0;
18 }

```

The following output is generated:

```

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: Equidistant Checkpointing of Evolutions

24.1 Purpose

This section illustrates the equidistant checkpointing of evolutions.

24.2 Example

```
1  #include "g_gap.hpp"
2
3  template<typename DCO_TYPE>
4  void f(int n, int m, DCO_TYPE& x) {
5      for (int i=0; i<n; i+=m)
6          g_make_gap(std::min(m, n-i), x);
7  }
8
9
10 #include "g.hpp"
11
12 template<typename DCO_MODE>
13 void g_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
14     typedef typename DCO_MODE::type DCO_TYPE;
15     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
16     typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
17     typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
18
19     DCO_TAPE_POSITION_TYPE p0=DCO_MODE::global_tape->get_position();
20     const int &n=D->template read_data<int>();
21     DCO_TYPE x=D->template read_data<DCO_VALUE_TYPE>();
22     const int &c=D->template read_data<int>();
23     DCO_MODE::global_tape->register_variable(x);
24     DCO_TYPE x_in=x;
25     DCO_TAPE_POSITION_TYPE p1=DCO_MODE::global_tape->get_position();
26     g(n, x);
27 }
```

```

18     cerr << "ts" << c << "="
19         << dco::size_of(DCO_MODE::global_tape) << "MB" << endl;
20     derivative(x)=D->get_output_adjoint();
21     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p1);
22     D->increment_input_adjoint(derivative(x_in));
23     DCO_MODE::global_tape->reset_to(p0);
24 }
25
26 template<typename DCO_TYPE>
27 void g_make_gap(int n, DCO_TYPE &x) {
28     typedef dco::mode<DCO_TYPE> DCO_MODE;
29     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
30     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
31     static int c=1;
32
33     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
34     DCO_VALUE_TYPE xp=D->register_input(x);
35     D->write_data(n); D->write_data(xp); D->write_data(c++);
36     g(n,xp);
37     x=D->register_output(xp);
38     DCO_MODE::global_tape->insert_callback(g_fill_gap<DCO_MODE>,D);
39 }

1  #include <cmath>
2  using namespace std;
3
4  template<typename DCO_TYPE>
5  void g(int n, DCO_TYPE& x) {
6      for (int i=0;i<n;i++) x=sin(x);
7  }

1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  #include "dco.hpp"
6  using namespace dco;
7
8  typedef gals<double> DCO_MODE;
9  typedef DCO_MODE::type DCO_TYPE;
10 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
11 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
12
13 #include "f.hpp"
14
15 void driver(const int n, const int m, double& xv, double& xa) {
16     DCO_TYPE x=xv;
17     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
18     DCO_MODE::global_tape->register_variable(x);
19     DCO_TYPE x_in=x;
20     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
21     f(n,m,x);
22     cerr << "ts0=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;

```

```

23     derivative(x)=xa;
24     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
25     xv=value(x);
26     xa=derivative(x_in);
27     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
28 }

1  #include <iostream>
2  using namespace std;
3
4  #include "driver.hpp"
5
6  int main() {
7      cout.precision(15);
8      int n=10; int m=2;
9      double xv=2.1, xa=1.0;
10     driver(n,m,xv,xa);
11     cout << "x=" << xv << endl;
12     cout << "x_{(1)}=" << xa << endl;
13     return 0;
14 }

```

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

```

24.3 New dco/c++ Features

None.

Chapter 25

Second-order Adjoint Mode: Equidistant Checkpointing of Evolutions

25.1 Purpose

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

25.2 Example

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  #include "dco.hpp"
6
7  using namespace dco;
8
9  typedef gt1s<double> DCO_BASE_MODE;
10 typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
11 typedef gals<DCO_BASE_TYPE> DCO_MODE;
12 typedef DCO_MODE::type DCO_TYPE;
13 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
14 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
15 // instantiate global tape
16 DCO_TAPE_TYPE*& DCO_TAPE_POINTER=DCO_MODE::global_tape;
17
18 #include "f.hpp"
19
20 void driver(const int n, const int m, double& x, double& xt2, double& xa1,
21            double& xt2a1) {
22     DCO_TYPE t2s_a1s_x;
23     DCO_BASE_TYPE v;
24     v = value(t2s_a1s_x);
25     value(v) = x;
```

```

25     derivative(v) = xt2;
26     value(t2s_a1s_x) = v;
27     DCO_TAPE_POINTER=DCO_TAPE_TYPE::create();
28     DCO_TAPE_POINTER->register_variable(t2s_a1s_x);
29     DCO_TYPE x_indep=t2s_a1s_x;
30     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
31     f(n,m,t2s_a1s_x);
32     cerr << "ts0=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
33     v = derivative(t2s_a1s_x);
34     value(v) = xa1;
35     derivative(v) = xt2a1;
36     derivative(t2s_a1s_x) = v;
37     DCO_TAPE_POINTER->interpret_adjoint_and_reset_to(p);
38     v = value(t2s_a1s_x);
39     x = value(v);
40     xt2 = derivative(v);
41     v = derivative(x_indep);
42     xa1 = value(v);
43     xt2a1 = derivative(v);
44     DCO_TAPE_TYPE::remove(DCO_TAPE_POINTER);
45 }

1  #include <cassert>
2  #include <cstdlib>
3  #include <iostream>
4  using namespace std;
5
6  #include "driver.hpp"
7
8  int main() {
9      int n=10; int m=2; cout.precision(15);
10     double x=2.1,xa1=1.0,xt2=1.0,xt2a1=0.0;
11     driver(n,m,x,xt2,xa1,xt2a1);
12     cout << "x=" << x << endl;
13     cout << "x^{(2)}=" << xt2 << endl;
14     cout << "x_{(1)}=" << xa1 << endl;
15     cout << "x_{(1)}^{(2)}=" << xt2a1 << endl;
16     return 0;
17 }

```

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

```

25.3 New dco/c++ Features

None.

Chapter 26

First-order Adjoint Mode: Recursive Checkpointing of Evolutions

26.1 Purpose

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

26.2 Example

```
1  #include "g_gap.hpp"
2
3  template<typename AD_MODE>
4  void f(
5      int from,
6      int to,
7      int stride, // max number of consecutive tapings
8      typename AD_MODE::type& x
9  ) {
10     //g(from,to,stride,x); // for split reversal
11     g_make_gap<AD_MODE>(from,to,stride,x);
12 }

1  #include <stack>
2  #include "g.hpp"
3  using namespace std;
4
5  enum RUN_MODE {
6      CHECKPOINT_ARGUMENTS_AND_RUN_PASSIVELY,
7      GENERATE_TAPE
8  };
9
10 template<typename DCO_MODE>
11 class my_external_adjoint_object_t : public DCO_MODE::external_adjoint_object_t{
```

```

12 public:
13     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
14     static stack<pair<int,DCO_VALUE_TYPE> > state;
15     static int stride;
16     my_external_adjoint_object_t(pair<int,int> p) : DCO_MODE::
        external_adjoint_object_t(p) {}
17 };
18
19 template<typename DCO_MODE>
20 stack<pair<int,typename my_external_adjoint_object_t<DCO_MODE>::DCO_VALUE_TYPE>
    > my_external_adjoint_object_t<DCO_MODE>::state;
21 template<typename DCO_MODE>
22 int my_external_adjoint_object_t<DCO_MODE>::stride;
23
24 template<typename DCO_MODE>
25 void g_fill_gap(my_external_adjoint_object_t<DCO_MODE> *D);
26
27 template<typename DCO_MODE>
28 void g_make_gap(int from, int to, int stride,
29     typename DCO_MODE::type &x, RUN_MODE m=GENERATE_TAPE) {
30     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
31     typedef my_external_adjoint_object_t<DCO_MODE> DCO_EAO_TYPE;
32
33     if (m==CHECKPOINT_ARGUMENTS_AND_RUN_PASSIVELY) {
34         cout << "STORE CHECKPOINT FOR SECTION "
35         << from << " ... " << to-1 << endl;
36         DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
            DCO_EAO_TYPE>(make_pair(1,1));
37         DCO_VALUE_TYPE xv=D->register_input(x);
38
39         // write argument checkpoint (FIFO)
40         D->write_data(from);
41         D->write_data(to);
42         if (D->state.empty()||from!=D->state.top().first) {
43             cout << "PUSHING (" << from << ", " << xv << ")" << endl;
44             D->state.push(make_pair(from,xv));
45         }
46
47         // call passive version of f
48         cout << "RUN SECTION " << from << " ... "
49         << to-1 << " PASSIVELY" << endl;
50         g(from,to,stride,xv);
51
52         // register output x with tape and store its
53         // position for retrieval of incoming adjoint required
54         // during interpretation
55         x=D->register_output(xv);
56         DCO_MODE::global_tape->insert_callback(g_fill_gap<DCO_MODE>,D);
57
58     } else if (m==GENERATE_TAPE) {
59         cout << "GENERATE TAPE FOR SECTION "
60         << from << " ... " << to-1 << endl;
61         stringstream s;
62         s << from << "GENERATE_TAPE-" << to-1 << ".dot";

```

```

63     DCO_MODE::global_tape->write_to_dot(s.str());
64     my_external_adjoint_object_t<DCO_MODE>::stride=stride;
65     // in taping mode, the interval is subdivided further if
66     // its length exceeds the desired stride ...
67     if (to-from>stride) {
68         g_make_gap<DCO_MODE>(from,from+(to-from)/2,stride,x,
69             CHECKPOINT_ARGUMENTS_AND_RUN_PASSIVELY);
70         g_make_gap<DCO_MODE>(from+(to-from)/2,to,stride,x,
71             CHECKPOINT_ARGUMENTS_AND_RUN_PASSIVELY);
72     }
73     else
74         // ... or the given number of iterations is performed
75         // actively and the corresponding tape is written
76         for (int i=from;i<to;i++) x=sin(x);
77 }
78
79 if (dco::size_of(DCO_MODE::global_tape) > max_tape_size)
80     max_tape_size = dco::size_of(DCO_MODE::global_tape);
81 }
82
83 template<typename DCO_MODE>
84 void g_fill_gap(
85     my_external_adjoint_object_t<DCO_MODE> *D
86 ) {
87     typedef typename DCO_MODE::type DCO_TYPE;
88     typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
89     typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
90
91     const int &from=D->template read_data<int>();
92     const int &to=D->template read_data<int>();
93     cout << "top=" << D->state.top().second << endl;
94     cout << "RESTORE CHECKPOINT FOR SECTION "
95         << from << " ... " << to-1 << endl;
96     DCO_TYPE x=D->state.top().second;
97     DCO_MODE::global_tape->register_variable(x);
98     DCO_TYPE x_in=x;
99     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
100    g_make_gap<DCO_MODE>(from,to,D->stride,x,GENERATE_TAPE);
101    derivative(x)=D->get_output_adjoint();
102    cout << "INTERPRET SECTION "
103        << from << " ... " << to-1 << endl;
104    DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
105    D->increment_input_adjoint(derivative(x_in));
106    if (to-from<=D->stride) {
107        cout << "popping " << D->state.top().first << ", " << D->state.top().second
108            << endl;
109        D->state.pop();
110    }
111 }

```

```

1  #include<vector>
2  using namespace std;
3
4

```

```

5  template<typename AD_TYPE>
6  void g(
7      int from,
8      int to,
9      int stride, // max number of consecutive tapings
10     AD_TYPE& x
11 ) {
12     if (to-from>stride) {
13         g(from,from+(to-from)/2,stride,x);
14         g(from+(to-from)/2,to,stride,x);
15     }
16     else
17         for (int i=from;i<to;i++) x=sin(x);
18 }

1  #include <iostream>
2  #include <cstdlib>
3  #include <cassert>
4  using namespace std;
5
6  #include "dco.hpp"
7  using namespace dco;
8
9  typedef gals<double> DCO_MODE;
10 typedef DCO_MODE::type DCO_TYPE;
11 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
12 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
13
14 #define STATISTICS
15 #define VERBOSE
16
17 #ifdef STATISTICS
18     dco::mem_long_t max_tape_size;
19     unsigned int max_checkpoint_size;
20 #endif
21 #include "f.hpp"
22
23 void driver(const int n, const int stride, double& xv, double& xa1) {
24     #ifdef STATISTICS
25         max_tape_size=0;
26         max_checkpoint_size=0;
27     #endif
28     DCO_TYPE x=xv;
29     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
30     DCO_MODE::global_tape->register_variable(x);
31     DCO_TYPE x_in=x;
32
33     f<DCO_MODE>(0,n,stride,x);
34
35     DCO_MODE::global_tape->register_output_variable(x);
36     derivative(x)=1;
37
38     #ifdef VERBOSE
39         cout << "INTERPRET SECTION: 0 ... " << n-1 << endl;

```

```

40 #endif
41 DCO_MODE::global_tape->write_to_dot("driver.dot");
42 DCO_MODE::global_tape->interpret_adjoint();
43
44 xv=value(x);
45 xa1=derivative(x_in);
46 #ifdef STATISTICS
47 cerr << "maximum tape size=" << max_tape_size*1024*1024 << "Byte" << endl;
48 cerr << "maximum checkpoint size=" << max_checkpoint_size*8 << "Byte" << endl;
49 #endif
50 DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
51 }

1 #include <cassert>
2 #include <cstdlib>
3 #include <iostream>
4 using namespace std;
5
6 #include "driver.hpp"
7
8 int main() {
9     cout.precision(15);
10    int n=10;
11    int stride=2;
12    double x=2.1,xa1=1.0;
13    driver(n,stride,x,xa1);
14    cout << "x=" << x << endl;
15    cout << "x_{{(1)}}=" << xa1 << endl;
16    return 0;
17 }

```

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
popping 8, 0.506149980527331
top=0.530714839456817
RESTORE CHECKPOINT FOR SECTION 7 ... 7
GENERATE TAPE FOR SECTION 7 ... 7
INTERPRET SECTION 7 ... 7
popping 7, 0.530714839456817
top=0.593714565454065
RESTORE CHECKPOINT FOR SECTION 5 ... 6
GENERATE TAPE FOR SECTION 5 ... 6
INTERPRET SECTION 5 ... 6
popping 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
popping 3, 0.688872566005681
top=0.75993256745268
RESTORE CHECKPOINT FOR SECTION 2 ... 2
GENERATE TAPE FOR SECTION 2 ... 2
INTERPRET SECTION 2 ... 2
popping 2, 0.75993256745268
top=2.1
RESTORE CHECKPOINT FOR SECTION 0 ... 1
GENERATE TAPE FOR SECTION 0 ... 1
INTERPRET SECTION 0 ... 1
popping 0, 2.1
maximum tape size=528Byte
```

```
maximum checkpoint size=32Byte  
x=0.466043407062983  
x_{(1)}=-0.0692765151830291
```

26.3 New dco/c++ Features

26.3.1 my_external_adjoint_object_t

Specialization of `DCO_MODE::external_adjoint_object_t`.

Chapter 27

Second-order Adjoint Mode: Recursive Checkpointing of Evolutions

27.1 Purpose

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

27.2 Example

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  #include "dco.hpp"
6
7  using namespace dco;
8
9  typedef gt1s<double> DCO_BASE_MODE;
10 typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
11 typedef gals<DCO_BASE_TYPE> DCO_MODE;
12 typedef DCO_MODE::type DCO_TYPE;
13 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
14 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_ITERATOR_TYPE;
15
16 #define VERBOSE
17
18 #include "f.hpp"
19
20 void driver(const int n, const int stride, double& x, double& xt2, double& xa1,
21            double& xt2a1) {
22     DCO_TYPE t2s_a1s_x;
23     DCO_BASE_TYPE v;
24     v = value(t2s_a1s_x);
```

```

24     value(v) = x;
25     derivative(v) = xt2;
26     value(t2s_a1s_x) = v;
27     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
28     DCO_MODE::global_tape->register_variable(t2s_a1s_x);
29     DCO_TYPE x_indep=t2s_a1s_x;
30     DCO_TAPE_ITERATOR_TYPE to_be_reset_to=DCO_MODE::global_tape->get_position();
31     f<DCO_MODE>(0,n,stride,t2s_a1s_x);
32     v = derivative(t2s_a1s_x);
33     value(v) = xa1;
34     derivative(v) = xt2a1;
35     derivative(t2s_a1s_x) = v;
36     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(to_be_reset_to);
37     v = value(t2s_a1s_x);
38     x = value(v);
39     xt2 = derivative(v);
40     v = derivative(x_indep);
41     xa1 = value(v);
42     xt2a1 = derivative(v);
43     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
44 }

1  #include <cassert>
2  #include <cstdlib>
3  #include <iostream>
4  using namespace std;
5
6  #include "driver.hpp"
7
8  int main() {
9      int n = 10; int stride=2; cout.precision(15);
10     double x=2.1,xa1=1.0,xt2=1.0,xt2a1=0.0;
11     driver(n,stride,x,xt2,xa1,xt2a1);
12     cout << "x=" << x << endl;
13     cout << "x^{(2)}=" << xt2 << endl;
14     cout << "x_{(1)}=" << xa1 << endl;
15     cout << "x_{(1)}^{(2)}=" << xt2a1 << endl;
16     return 0;
17 }

```

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
popping 8, 0.506149980527331
top=0.530714839456817
RESTORE CHECKPOINT FOR SECTION 7 ... 7
GENERATE TAPE FOR SECTION 7 ... 7
INTERPRET SECTION 7 ... 7
popping 7, 0.530714839456817
top=0.593714565454065
RESTORE CHECKPOINT FOR SECTION 5 ... 6
GENERATE TAPE FOR SECTION 5 ... 6
INTERPRET SECTION 5 ... 6
popping 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
popping 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
```

27.3 New dco/c++ Features

None.

Chapter 28

First-order Adjoint Mode: Checkpointing Ensembles

28.1 Purpose

This section illustrates checkpointing of ensembles in first-order adjoint mode. A global tape is used. The given solution allows for second and higher derivatives to be computed with minimal implementation effort; see Ch. 29.

28.2 Example

```
1  #include "g_gap.hpp"
2
3  template<typename DCO_TYPE>
4  void f(int n, int m, const DCO_TYPE& x, DCO_TYPE& y) {
5      double r;
6      DCO_TYPE sum;
7      for (int i=0;i<n;i++) {
8          r=rand();
9          g_make_gap(m,x,r,y);
10         sum+=y;
11     }
12     y=sum/n;
13 }
```



```
1  #include<iostream>
2  using namespace std;
3  #include "g.hpp"
4
5  template<typename DCO_MODE>
6  void g_fill_gap(typename DCO_MODE::external_adjoint_object_t*D) {
7      typedef typename DCO_MODE::type DCO_TYPE;
8      typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
9      typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
10
11     const int &m=D->template read_data<int>();
```

```

12     const DCO_TYPE* const &x_p=
13         D->template read_data<const DCO_TYPE*>();
14     const double &r=D->template read_data<const double>();
15     const int &c=D->template read_data<const int>();
16     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
17     DCO_TYPE y;
18     g(m,*x_p,r,y);
19     cerr << "ts" << c << "="
20         << dco::sizeof(DCO_MODE::global_tape) << "B" << endl;
21     derivative(y)=D->get_output_adjoint();
22     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
23 }
24
25 template<typename DCO_TYPE>
26 void g_make_gap(int m, const DCO_TYPE& x, const double& r, DCO_TYPE& y) {
27     typedef dco::mode<DCO_TYPE> DCO_MODE;
28     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
29     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
30     static int c=1;
31
32     DCO_EAO_TYPE *D=DCO_MODE::global_tape->
33         template create_callback_object<DCO_EAO_TYPE>();
34     D->write_data(m); D->write_data(&x); D->write_data(r); D->write_data(c++);
35     DCO_VALUE_TYPE yp;
36     g(m,value(x),r,yp);
37     y=D->register_output(yp,DCO_MODE::global_tape);
38     DCO_MODE::global_tape->
39         insert_callback(g_fill_gap<DCO_MODE>,D);
40 }
41
42 template<typename ATYPE>
43 void g(int m, const ATYPE& x, const double& r, ATYPE& y) {
44     y=0;
45     for (int i=0;i<m;i++) y+=sin(x+r);
46 }
47
48 #include "dco.hpp"
49 using namespace dco;
50
51 typedef dco::gals<double> DCO_MODE;
52 typedef DCO_MODE::type DCO_TYPE;
53 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
54 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
55
56 #include "f.hpp"
57
58 void driver(const int n, const int m,
59             const double& xv, double& xa,
60             double &yv, const double& ya) {
61     DCO_TYPE x=xv,y;
62     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
63     DCO_MODE::global_tape->register_variable(x);
64     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
65     f(n,m,x,y);

```

```

19     cerr << "ts0=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
20     yv=value(y);
21     DCO_MODE::global_tape->register_output_variable(y);
22     derivative(y)=ya; derivative(x)=xa;
23     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
24     //DCO_MODE::global_tape->interpret_adjoint();
25     xa=derivative(x);
26     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
27 }

1  #include <iostream>
2  using namespace std;
3
4  #include "driver.hpp"
5
6  int main() {
7      cout.precision(15);
8      int n=10; int m=10;
9      double xv=2.1,xa=0.0,yv,ya=1.0;
10     driver(n,m,xv,xa,yv,ya);
11     cout << "y=" << yv << endl;
12     cout << "x_{{1}}=" << xa << endl;
13     return 0;
14 }

```

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

```

28.3 New dco/c++ Features

None.

Chapter 29

Second-order Adjoint Mode: Checkpointing Ensembles

29.1 Purpose

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

29.2 Example

```
1  #include "dco.hpp"
2  using namespace dco;
3
4  typedef gt1s<double> DCO_BASE_MODE;
5  typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
6  typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
7  typedef DCO_MODE::type DCO_TYPE;
8  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
9  typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
10
11 #include "f.hpp"
12
13 void driver(const int n, const int m,
14             const double& xv, const double& xt2, double& xa1, double& xa1t2,
15             double &yv, double& yt2, const double& ya1, const double& ya1t2) {
16     DCO_TYPE x=xv,y;
17     derivative(value(x))=xt2;
18     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
19     DCO_MODE::global_tape->register_variable(x);
20     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
21     f(n,m,x,y);
22     cerr << "ts0=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
23     yv=passive_value(y);
24     yt2=derivative(value(y));
25     DCO_MODE::global_tape->register_output_variable(y);
26     value(derivative(y))=ya1;
27     derivative(derivative(y))=ya1t2;
```

```

28     value(derivative(x))=xa1;
29     derivative(derivative(x))=xa1t2;
30     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
31     xa1=value(derivative(x));
32     xa1t2=derivative(derivative(x));
33     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
34 }

1  #include <cassert>
2  #include <cstdlib>
3  #include <iostream>
4  using namespace std;
5
6  #include "driver.hpp"
7
8  int main() {
9      cout.precision(15);
10     int n=10; int m=10;
11     double x=2.1,xt2=1.0,xa1=0.0,xa1t2=0.0,y,yt2,ya1=1.0,ya1t2=0.0;
12     driver(n,m,x,xt2,xa1,xa1t2,y,yt2,ya1,ya1t2);
13     cout << "y=" << y << endl;
14     cout << "y^{(2)}=" << yt2 << endl;
15     cout << "x_{(1)}=" << xa1 << endl;
16     cout << "x_{(1)}^{(2)}=" << xa1t2 << endl;
17     return 0;
18 }

```

The following output is generated:

```

ts0=0.001678466796875MB
ts0=0.001678466796875MB
ts10=0.002349853515625MB
ts9=0.00218963623046875MB
ts8=0.0020294189453125MB
ts7=0.00186920166015625MB
ts6=0.001708984375MB
ts5=0.00154876708984375MB
ts4=0.0013885498046875MB
ts3=0.00122833251953125MB
ts2=0.001068115234375MB
ts1=0.0009307861328125MB

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

```

29.3 New dco/c++ Features

None.

Chapter 30

First-order Adjoint Mode: Embedding Adjoint Source Code

30.1 Purpose

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

30.2 Example

```
1  #include<vector>
2  #include "g_gap.hpp"
3
4  template<typename DCO_TYPE>
5  void f(std::vector<DCO_TYPE>& x, DCO_TYPE& y) {
6      for (unsigned i=0;i<x.size();i++) x[i]*=x[i];
7      // g(x,y);
8      g_make_gap(x,y);
9      y*=y;
10 }

```



```
1  #include<vector>
2
3  #include "g.hpp"
4  #include "g_a1s.hpp"
5
6  template<typename DCO_MODE>
7  void g_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
8      typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
9      const int &n=D->template read_data<int>();
10     vector<DCO_VALUE_TYPE> xa1(n);
11     DCO_VALUE_TYPE ya1=D->get_output_adjoint();
12     g_a1s<DCO_VALUE_TYPE>(xa1,ya1);
13     for (int i=0;i<n;i++) D->increment_input_adjoint(xa1[i]);
14 }
15
16 template<typename DCO_TYPE>
```

```

17 void g_make_gap(std::vector<DCO_TYPE>& x, DCO_TYPE &y) {
18     typedef dco::mode<DCO_TYPE> DCO_MODE;
19     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
20     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
21     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
22
23     size_t n=x.size();
24     vector<DCO_VALUE_TYPE> xv(n);
25     for (size_t i=0;i<n;i++) xv[i]=D->register_input(x[i]);
26     D->write_data(n);
27     DCO_VALUE_TYPE yv;
28     g<DCO_VALUE_TYPE>(xv,yv);
29     y=D->register_output(yv);
30     DCO_MODE::global_tape->insert_callback(g_fill_gap<DCO_MODE>,D);
31 }

1  #include<vector>
2
3  template<typename DCO_TYPE>
4  void g(std::vector<DCO_TYPE> x, DCO_TYPE& y) {
5      y=0;
6      for (unsigned i=0;i<x.size();i++) y+=x[i];
7  }

1  #include<vector>
2
3  template<typename DCO_TYPE>
4  void g_a1s(std::vector<DCO_TYPE>& xa1, const DCO_TYPE& ya1) {
5      typename std::vector<DCO_TYPE>::iterator i;
6      for (i=xa1.begin();i!=xa1.end();i++) *i+=ya1;
7  }

1  #include <vector>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6  typedef gals<double> DCO_MODE;
7  typedef DCO_MODE::type DCO_TYPE;
8  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
9  typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
10
11 #include "f.hpp"
12
13 void driver(
14     const vector<double>& xv,
15     vector<double>& xa1,
16     double& yv,
17     double& ya1
18 ) {
19     size_t n=xv.size();
20     vector<DCO_TYPE> x(n);
21     vector<DCO_TYPE> x_in(n);

```

```

22     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
23     for (size_t i=0;i<n;i++) {
24         DCO_MODE::global_tape->register_variable(x[i]);
25         value(x[i])=xv[i];
26         derivative(x[i])=xa1[i];
27         x_in[i]=x[i];
28     }
29     DCO_TYPE y;
30     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
31     f(x,y);
32     cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
33     yv=value(y);
34     derivative(y)=ya1;
35     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
36     for (size_t i=0;i<n;i++)
37         xa1[i]=derivative(x_in[i]);
38     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
39 }

1  #include <iostream>
2  #include <cstdlib>
3  #include <vector>
4  #include <cmath>
5  using namespace std;
6
7  #include "driver.hpp"
8
9  int main() {
10     cout.precision(15);
11     int n=5;
12     vector<double> x(n), xa1(n);
13     double y=0, ya1;
14     for (int i=0;i<n;i++) { x[i]=cos(double(i)); xa1[i]=0; }
15     ya1=1;
16     driver(x,xa1,y,ya1);
17     cout << "y=" << y << endl;
18     for (int i=0;i<n;i++)
19         cout << "x_{(1)}[" << i << "]= " << xa1[i] << endl;
20
21     return 0;
22 }

```

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
```

```
...
```

30.3 New dco/c++ Features

None.

Chapter 31

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

31.1 Purpose

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

31.2 Example

```
1  #include<vector>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  typedef gt1s<double> DCO_BASE_MODE;
8  typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
9  typedef gals<DCO_BASE_TYPE> DCO_MODE;
10 typedef DCO_MODE::type DCO_TYPE;
11 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
12 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
13
14 #include "../gals_external_manual/f.hpp"
15
16 void driver(
17     const vector<double>& xv,
18     const vector<double>& xt2,
19     vector<double>& xa1,
20     vector<double>& xa1t2,
21     double& yv,
22     double& yt2,
23     double& ya1,
24     double& ya1t2
```

```

25 ) {
26     gals<DCO_BASE_TYPE>::global_tape=gals<DCO_BASE_TYPE>::tape_t::create();
27     const size_t n=xv.size();
28     vector<DCO_TYPE> x(n), x_in(n); DCO_TYPE y;
29     for (size_t i=0;i<n;i++) {
30         gals<DCO_BASE_TYPE>::global_tape->register_variable(x[i]);
31         value(value(x[i]))=xv[i];
32         derivative(value(x[i]))=xt2[i];
33         x_in[i]=x[i];
34     }
35     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
36     f(x,y);
37     cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
38     for (size_t i=0;i<n;i++) {
39         value(derivative(x_in[i]))=xa1[i];
40         derivative(derivative(x_in[i]))=xa1t2[i];
41     }
42     yv=value(value(y));
43     yt2=derivative(value(y));
44     gals<DCO_BASE_TYPE>::global_tape->register_output_variable(y);
45     value(derivative(y))=ya1;
46     derivative(derivative(y))=ya1t2;
47     gals<DCO_BASE_TYPE>::global_tape->interpret_adjoint_and_reset_to(p);
48     for (size_t i=0;i<n;i++) {
49         xa1[i]=value(derivative(x_in[i]));
50         xa1t2[i]=derivative(derivative(x_in[i]));
51     }
52     gals<DCO_BASE_TYPE>::tape_t::remove(gals<DCO_BASE_TYPE>::global_tape);
53 }

1  #include<iostream>
2  #include<vector>
3  #include<cmath>
4  using namespace std;
5
6  #include "driver.hpp"
7
8  int main() {
9      cout.precision(15);
10     int n=5;
11     vector<double> x(n), xa1(n), xt2(n), xa1t2(n);
12     double y, ya1, yt2, ya1t2;
13     for (int i=0;i<n;i++) { x[i]=cos(double(i)); xt2[i]=1; xa1[i]=0; xa1t2[i]=0;
14         }
15     y=0, yt2=0, ya1=1; ya1t2=0;
16     driver(x,xt2,xa1,xa1t2,y,yt2,ya1,ya1t2);
17     cout << "y=" << y << endl;
18     for (int i=0;i<n;i++)
19         cout << "x_{(1)}[" << i << "]= " << xa1[i] << endl;
20     cout << "y^{(2)}=" << yt2 << endl;
21     for (int i=0;i<n;i++)
22         cout << "x_{(1)}^{(2)}[" << i << "]= " << xa1t2[i] << endl;
23     cout << "y_{(1)}=" << ya1 << endl;
24     cout << "y_{(1)}^{(2)}=" << ya1t2 << endl;

```

```

24     return 0;
25 }

```

The following output is generated:

```
ts=0.00077056884765625MB
```

```

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

```

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

```
ts=0.00109100341796875MB
```

```

y=8.25091096598783
x_{(1)}[0]=11.489759590862
...
y^{(2)}=-5.96870775926893
x_{(1)}^{(2)}[0]=7.33391440571752
...

```

31.3 New dco/c++ Features

None.

Chapter 32

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

32.1 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.

32.2 Example

```
1  #include "g_gap.hpp"
2
3  template<typename AD_TYPE>
4  void f(int n, AD_TYPE& x) {
5      g(n/3,x);
6      g(n/3,x);
7      // g_make_gap(n/3,x);
8      g(n-n/3*2,x);
9  }
10
11 #include "g.hpp"
12
13 template<typename DCO_BASE_TYPE>
14 void gt1(int n, DCO_BASE_TYPE& x, DCO_BASE_TYPE& xt1) {
15     for (int i=0;i<n;i++) {
16         xt1=cos(x)*xt1;
17         x=sin(x);
18     }
19 }
20
21 template<typename DCO_MODE>
22 void g_fill_gap(
23     typename DCO_MODE::external_adjoint_object_t *D
24 ) {
```

```

15     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
16
17     const DCO_VALUE_TYPE &p=D->template read_data<DCO_VALUE_TYPE>();
18     DCO_VALUE_TYPE r=D->get_output_adjoint();
19     D->increment_input_adjoint(r*p);
20 }
21
22 template<typename DCO_TYPE>
23 void g_make_gap(int n, DCO_TYPE &x) {
24     typedef typename dco::mode<DCO_TYPE> DCO_MODE;
25     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
26     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
27     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
28     DCO_VALUE_TYPE xv=D->register_input(x);
29     DCO_VALUE_TYPE xt1=1.;
30     gt1(n,xv,xt1);
31     D->write_data(xt1);
32     x=D->register_output(xv);
33     DCO_MODE::global_tape->insert_callback(g_fill_gap<DCO_MODE>,D);
34 }

1  #include<cmath>
2  using namespace std;
3
4  template<typename ATYPE>
5  void g(
6      int n,
7      ATYPE& x
8  ) {
9      for (int i=0;i<n;i++) x=sin(x);
10 }

1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  #include "dco.hpp"
6  using namespace dco;
7  typedef gals<double> DCO_MODE;
8  typedef DCO_MODE::type DCO_TYPE;
9  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
10 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
11
12 #include "f.hpp"
13
14 void driver(const int n, double& xv, double& xa) {
15     DCO_TYPE x=xv;
16     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
17     DCO_MODE::global_tape->register_variable(x);
18     DCO_TYPE x_in=x;
19     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
20     f(n,x);
21     cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;

```

```

22     derivative(x)=xa;
23     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
24     xv=value(x);
25     xa=derivative(x_in);
26     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
27 }

1  #include <cassert>
2  #include <cstdlib>
3  #include <iostream>
4  using namespace std;
5
6  #include "driver.hpp"
7
8  int main() {
9      cout.precision(15);
10     int n=10;
11     double x=2.1,xa1=1.0;
12     driver(n,x,xa1);
13     cout << "x=" << x << endl;
14     cout << "x_{(1)}=" << xa1 << endl;
15     return 0;
16 }

```

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
```

32.3 New dco/c++ Features

None.

Chapter 33

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

33.1 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.

33.2 Example

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  #include "dco.hpp"
6
7  using namespace dco;
8
9  typedef gt1s<double> DCO_BASE_MODE;
10 typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
11 typedef gals<DCO_BASE_TYPE> DCO_MODE;
12 typedef DCO_MODE::type DCO_TYPE;
13 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
14 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
15
16 #include "f.hpp"
17
18 void driver(const int n, double& xv, double& xt2, double& xa1, double& xt2a1) {
19     DCO_TYPE x;
20     passive_value(x)=xv;
21     derivative(value(x))=xt2;
22     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
23     DCO_MODE::global_tape->register_variable(x);
24     DCO_TYPE x_in=x;
```

```

25     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
26     f(n,x);
27     cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
28     xv=passive_value(x);
29     xt2=derivative(value(x));
30     value(derivative(x))=xa1;
31     derivative(derivative(x))=xt2a1;
32     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
33     xa1=value(derivative(x_in));
34     xt2a1=derivative(derivative(x_in));
35     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
36 }

1  #include <cassert>
2  #include <cstdlib>
3  #include <iostream>
4  using namespace std;
5
6  #include "driver.hpp"
7
8  int main() {
9      int n=10; cout.precision(15);
10     double x=2.1,xa1=1.0,xt2=1.0,xt2a1=0.0;
11     driver(n,x,xt2,xa1,xt2a1);
12     cout << "x=" << x << endl;
13     cout << "x^{(2)}=" << xt2 << endl;
14     cout << "x_{(1)}=" << xa1 << endl;
15     cout << "x_{(1)}^{(2)}=" << xt2a1 << endl;
16     return 0;
17 }

```

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

```

33.3 New dco/c++ Features

None.

Chapter 34

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

34.1 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.

34.2 Example

```
1  #include "g_gap.hpp"
2
3  template<typename AD_TYPE>
4  void f(int n, AD_TYPE& x) {
5      g(n/3,x);
6      g_make_gap(n/3,x);
7      g(n-n/3*2,x);
8  }
9
10
11 #include "g.hpp"
12
13 template<typename DCO_BASE_TYPE>
14 void gt1(int n, DCO_BASE_TYPE& x, DCO_BASE_TYPE& xt1) {
15     for (int i=0;i<n;i++) {
16         xt1=cos(x)*xt1;
17         x=sin(x);
18     }
19 }
20
21
22 template<typename DCO_TYPE>
23 void g_make_gap(int n, DCO_TYPE &x) {
24     typedef typename dco::mode<DCO_TYPE> DCO_MODE;
25     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
26     typedef typename DCO_MODE::local_gradient_t local_gradient_t;
```

```

16
17     DCO_TYPE xc = x; // copy required because output=input
18     local_gradient_t ylo(x);
19     // new tape index for output x
20
21     DCO_VALUE_TYPE xt1=1.;
22     gt1(n,value(x),xt1); // value of x changed
23
24     ylo.put(xc, xt1); // add edge from new output to input
25 }

1  #include<cmath>
2  using namespace std;
3
4  template<typename ATYPE>
5  void g(
6      int n,
7      ATYPE& x
8  ) {
9      for (int i=0;i<n;i++) x=sin(x);
10 }

1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  #include "dco.hpp"
6  using namespace dco;
7  typedef gals<double> DCO_MODE;
8  typedef DCO_MODE::type DCO_TYPE;
9  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
10 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
11
12 #include "f.hpp"
13
14 void driver(const int n, double& xv, double& xa) {
15     DCO_TYPE x=xv;
16     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
17     DCO_MODE::global_tape->register_variable(x);
18     DCO_TYPE x_in=x;
19     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
20     f(n,x);
21     cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
22     derivative(x)=xa;
23     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
24     xv=value(x);
25     xa=derivative(x_in);
26     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
27 }

1  #include <cassert>
2  #include <cstdlib>
3  #include <iostream>
4  using namespace std;

```

```

5
6 #include "driver.hpp"
7
8 int main() {
9     cout.precision(15);
10    int n=10;
11    double x=2.1,xa1=1.0;
12    driver(n,x,xa1);
13    cout << "x=" << x << endl;
14    cout << "x_{(1)}=" << xa1 << endl;
15    return 0;
16 }

```

The following output is generated:

```
ts=0.00032806396484375MB
```

```
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
```

34.3 New dco/c++ Features

34.3.1 `DCO_MODE::local_gradient_t`

Local gradient type associated with `DCO_MODE`.

34.3.2 `DCO_TAPE_TYPE::create_local_gradient_object`

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

34.3.3 `DCO_LOCAL_GRADIENT_TYPE::put`

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

34.3.4 `DCO_LOCAL_GRADIENT_TYPE::finalize`

Adds local gradient to tape.

Chapter 35

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

35.1 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.

35.2 Example

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  #include "dco.hpp"
6
7  using namespace dco;
8
9  typedef gt1s<double> DCO_BASE_MODE;
10 typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
11 typedef gals<DCO_BASE_TYPE> DCO_MODE;
12 typedef DCO_MODE::type DCO_TYPE;
13 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
14 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
15
16 #include "f.hpp"
17
18 void driver(const int n, double& xv, double& xt2, double& xa1, double& xt2a1) {
19     DCO_TYPE x;
20     passive_value(x)=xv;
21     derivative(value(x))=xt2;
22     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
23     DCO_MODE::global_tape->register_variable(x);
24     DCO_TYPE x_in=x;
```

```

25     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
26     f(n,x);
27     cerr << "ts=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
28     xv=passive_value(x);
29     xt2=derivative(value(x));
30     value(derivative(x))=xa1;
31     derivative(derivative(x))=xt2a1;
32     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p);
33     xa1=value(derivative(x_in));
34     xt2a1=derivative(derivative(x_in));
35     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
36 }

1  #include <cassert>
2  #include <cstdlib>
3  #include <iostream>
4  using namespace std;
5
6  #include "driver.hpp"
7
8  int main() {
9      int n=10; cout.precision(15);
10     double x=2.1,xa1=1.0,xt2=1.0,xt2a1=0.0;
11     driver(n,x,xt2,xa1,xt2a1);
12     cout << "x=" << x << endl;
13     cout << "x^{(2)}=" << xt2 << endl;
14     cout << "x_{(1)}=" << xa1 << endl;
15     cout << "x_{(1)}^{(2)}=" << xt2a1 << endl;
16     return 0;
17 }

```

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

```

35.3 New dco/c++ Features

None.

Chapter 36

First-order Adjoint Mode: User-Defined Adjoints

36.1 Purpose

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

36.2 Example

```
1  #include "g_gap.hpp"
2
3  template<typename DCO_TYPE>
4  void f(DCO_TYPE p, DCO_TYPE& x) {
5      p=exp(p);
6      // g(p,x);
7      g_make_gap(p,x);
8      x=x*p;
9  }
10
11 #include "g.hpp"
12
13 template<typename DCO_TYPE>
14 void a1s_g(DCO_TYPE& pa, const DCO_TYPE& xv, DCO_TYPE& xa) {
15     pa=xa/(2*xv);
16 }
17
18 template<typename DCO_MODE>
19 void g_fill_gap(typename DCO_MODE::external_adjoint_object_t* D) {
20     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
21     const DCO_VALUE_TYPE &xv = D->template read_data<DCO_VALUE_TYPE>();
22     DCO_VALUE_TYPE xa,pa;
23     xa=D->get_output_adjoint();
24     a1s_g(pa,xv,xa);
25     D->increment_input_adjoint(pa);
26 }
```

```

17
18 template<typename DCO_TYPE>
19 void g_make_gap(const DCO_TYPE& p, DCO_TYPE& x) {
20     typedef mode<DCO_TYPE> DCO_MODE;
21     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
22     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
23     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
24     DCO_VALUE_TYPE xv=value(x),pv;
25     pv=D->register_input(p);
26     g(pv,xv);
27     x=D->register_output(xv);
28     D->write_data(xv);
29     DCO_MODE::global_tape->insert_callback(g_fill_gap<DCO_MODE>,D);
30 }

1 template<class DCO_TYPE>
2 void g(
3     const DCO_TYPE& p,
4     DCO_TYPE& x
5 ) {
6     const DCO_TYPE eps=1e-12;
7     DCO_TYPE xp=x+1;
8     while (fabs(x-xp)>eps) {
9         xp=x;
10        x=xp-(xp*xp-p)/(2*xp);
11    }
12 }

1 #include<iostream>
2 using namespace std;
3
4 #include "dco.hpp"
5 using namespace dco;
6
7 typedef ga1s<double> DCO_MODE;
8 typedef DCO_MODE::type DCO_TYPE;
9 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
10 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
11
12 #include "f.hpp"
13
14 void driver (const double& pv, double& pa, double& xv, const double& xa) {
15     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
16     DCO_TYPE x=xv,p=pv;
17     DCO_MODE::global_tape->register_variable(p);
18     f(p,x);
19     cerr << "ts=" << dco::sizeof(DCO_MODE::global_tape) << "B" << endl;
20     xv=value(x);
21     DCO_MODE::global_tape->register_output_variable(x);
22     derivative(x)=xa;
23     DCO_MODE::global_tape->interpret_adjoint();
24     pa=derivative(p);
25     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);

```

```
26 }  
  
1  #include <iostream>  
2  using namespace std;  
3  
4  #include "driver.hpp"  
5  
6  int main() {  
7      double xv=1,pv=5,xa=1,pa=0;  
8      driver(pv,pa,xv,xa);  
9      cout.precision(15);  
10     cout << "x=" << xv << endl;  
11     cout << "p_{(1)}=" << pa << endl;  
12     return 0;  
13 }
```

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
```

36.3 New dco/c++ Features

None.

Chapter 37

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

37.1 Purpose

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

37.2 Example

```
1  #include<iostream>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  typedef gt1s<double> DCO_BASE_MODE;
8  typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
9  typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
10 typedef DCO_MODE::type DCO_TYPE;
11 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
12
13 #include "f.hpp"
14
15 void driver(const double& pv, const double& pt2, double& pa1, double& pa1t2,
16            double &xv, double& xt2, const double& xa1, const double& xa1t2) {
17     DCO_TYPE p,x;
18     passive_value(p)=pv;
19     derivative(value(p))=pt2;
20     passive_value(x)=xv;
21     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
22     DCO_MODE::global_tape->register_variable(p);
23     f(p,x);
24     cerr << "ts=" << dco::sizeof(DCO_MODE::global_tape) << "B" << endl;
25     xv=passive_value(x);
26     xt2=derivative(value(x));
```

```

27     DCO_MODE::global_tape->register_output_variable(x);
28     value(derivative(x))=xa1;
29     derivative(derivative(x))=xa1t2;
30     value(derivative(p))=pa1;
31     derivative(derivative(p))=pa1t2;
32     DCO_MODE::global_tape->interpret_adjoint();
33     pa1=value(derivative(p));
34     pa1t2=derivative(derivative(p));
35     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
36 }

1  #include <cassert>
2  #include <cstdlib>
3  #include <iostream>
4  using namespace std;
5
6  #include "driver.hpp"
7
8  int main() {
9      cout.precision(15);
10     double p=5,pt2=1,pa1=0,pa1t2=0,x=1,xt2=0,xa1=1,xa1t2=0;
11     driver(p,pt2,pa1,pa1t2,x,xt2,xa1,xa1t2);
12     cout << "x=" << x << endl;
13     cout << "x^{(2)}=" << xt2 << endl;
14     cout << "p_{(1)}=" << pa1 << endl;
15     cout << "p_{(1)}^{(2)}=" << pa1t2 << endl;
16     return 0;
17 }

```

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

```

37.3 New dco/c++ Features

None.

Chapter 38

First-order Adjoint Mode: Local Jacobian Preaccumulation

38.1 Purpose

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

38.2 Remarks

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

38.3 Example

38.3.1 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.

```
1  template<typename T>
2  void euler(const int from, const int to, T& x, const vector<T>& p) {
3      double dt=1./p.size();
4      for (int i=from;i<to;i++) {
5          T x_prev=x;
6          T f=-dt*p[i]*sin(x*i*dt);
7          while (abs(f)>1e-7) {
8              x=x-f/(1-dt*p[i]*i*dt*cos(x*i*dt));
9              f=x-x_prev-dt*p[i]*sin(x*i*dt);
10         }
11     }
12 }
```

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

```
1  #include<iostream>
2  #include<cmath>
```

```

3  #include<vector>
4  using namespace std;
5
6  #include "dco.hpp"
7
8  typedef dco::ga1s<double> DCO_MODE;
9  typedef DCO_MODE::type DCO_TYPE;
10 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
11
12 #include "f.h"
13
14 void driver() {
15     const int n=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.

38.3.2 Example Results

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

```

1  dx/dp[497]=0.000636433
2  dx/dp[498]=0.000637576
3  dx/dp[499]=0.000638718
4  dx/dp[500]=0.00063986
5  dx/dp[501]=0.000641
6  dx/dp[502]=0.00064214

```

38.4 New dco/c++ Features

38.4.1 `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.

38.4.2 `jacobian_preaccumulator_t::start()`

Start recording of tape used for preaccumulation.

38.4.3 `jacobian_preaccumulator_t::register_output(DCO_TYPE&)`

Register active outputs of computation subject to preaccumulation.

38.4.4 `jacobian_preaccumulator_t::finish()`

Preaccumulate local Jacobian and integrate into associated tape.

Chapter 39

Second-order Adjoint Mode: Local Jacobian Preaccumulation

39.1 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.

39.2 Example

The example from Sec. 38.3 is extended towards second-order adjoints.

39.2.1 Example Text

```
1  #include<iostream>
2  #include<cmath>
3  #include<vector>
4  using namespace std;
5
6  #include "dco.hpp"
7
8  typedef dco::gtls<double>::type DCO_BASE_TYPE;
9  typedef dco::gals<DCO_BASE_TYPE> DCO_MODE;
10 typedef DCO_MODE::type DCO_TYPE;
11 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
12
13 #include "f.h"
14
15 void driver() {
16     const int n=100, s=10;
17     const double x0=1;
18     vector<DCO_TYPE> p(n,1);
19     vector<vector<double>> > H(n,vector<double>(n,0));
20     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
21     for (int i=0;i<n;i++)
```

```

22     DCO_MODE::global_tape->register_variable(p[i]);
23     DCO_TAPE_TYPE::position_t tpos=DCO_MODE::global_tape->get_position();
24     for (int j=0;j<n;j++) {
25         DCO_TYPE x=x0;
26         dco::derivative(dco::value(p[j]))=1;
27         DCO_MODE::jacobian_preaccumulator_t jp(DCO_MODE::global_tape);
28         for (int i=0;i<n;i+=s) {
29             jp.start();
30             euler(i,min(i+s,n),x,p);
31             jp.register_output(x);
32             jp.finish();
33         }
34         DCO_MODE::global_tape->register_output_variable(x);
35         dco::derivative(x)=1;
36         DCO_MODE::global_tape->interpret_adjoint();
37         for (int i=0;i<n;i++) {
38             H[i][j]=dco::derivative(dco::derivative(p[i]));
39             dco::derivative(p[i])=0;
40         }
41         DCO_MODE::global_tape->reset_to(tpos);
42         dco::derivative(dco::value(p[j]))=0;
43     }
44     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
45     for (int j=n/2-1;j<n/2+1;j++)
46         for (int i=n/2-1;i<n/2+1;i++)
47             cout << "ddx/dp[" << j << "]dp[" << i << "]= " << H[j][i] << endl;
48 }

```

39.2.2 Example Results

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

```

1 ddx/dp[49]dp[49]=4.34577e-05
2 ddx/dp[49]dp[50]=1.71089e-05
3 ddx/dp[50]dp[49]=1.71089e-05
4 ddx/dp[50]dp[50]=4.47534e-05

```

39.3 New dco/c++ Features

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

Chapter 40

First-order Adjoint Mode: Multiple Adjoint Vectors

40.1 Purpose

Recording of a single tape can be followed by repeated (concurrent) interpretations in separate address spaces implemented by multiple adjoint vectors of potentially different types; see also DCO_ADJOINT_TYPE in Ch. 11.

40.2 Example

The implicit Euler case study from Sec. 38.3 is considered.

40.3 Example Text

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

```
1  #include<iostream>
2  #include<cmath>
3  using namespace std;
4  #include <omp.h>
5
6  #include "dco.hpp"
7  typedef dco::gals<double> DCO_MODE;
8  typedef DCO_MODE::type DCO_TYPE;
9  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
10
11 #include "f.h"
12
13 void driver() {
14     const int n=1000;
15     double x0=1;
16     vector<DCO_TYPE> p(n,1);
17     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
18     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
28     /** this is required, since the global_tape pointer is threadlocal;
29     /** alternatively, use compile-time flag -DDCO_NO_THREADLOCAL and
30     /** just use DCO_MODE::global_tape in both threads.
31     DCO_MODE::tape_t* shared_global_tape = DCO_MODE::global_tape;
32
33 #pragma omp parallel
34 {
35     switch (omp_get_thread_num()) {
36     case 0: {
37         dco::adjoint_vector<DCO_TAPE_TYPE,float>
38             av_float(shared_global_tape);
39         dco::derivative(x, av_float)=1;
40         av_float.interpret_adjoint();
41         adjoint_float = dco::derivative(p[n/2], av_float);
42         break;
43     }
44     case 1: {
45         dco::adjoint_vector<DCO_TAPE_TYPE,double>
46             av_double(shared_global_tape);
47         dco::derivative(x, av_double)=1;
48         av_double.interpret_adjoint();
49         adjoint_double = dco::derivative(p[n/2], av_double);
50     }
51     default: {}
52     }
53 }
54
55 cout << "float " << "dx/dp=" << adjoint_float << endl;
56 cout << "double " << "dx/dp=" << adjoint_double << endl;
57
58 DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
59 }

```

A single tape is recorded in lines 17-22 prior to entering the parallel regions in line 33. Thread 0 allocates an adjoint vector over `DCO_ADJOINT_TYPE=float` in lines 37-38. Similarly, an adjoint vector over `DCO_ADJOINT_TYPE=double` is allocated by thread 1 in lines 45-46. Access to derivative components requires specification of the associated adjoint vector (e.g., lines 41 and 49). Interpretation is called for the individual adjoint vectors in lines 40 and 48 followed by printing the respective results. Mind the use of `shared_global_tape` in line 31, 38, and 46.

40.4 Example Results

The following output is generated.

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

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

40.5 New dco/c++ Features

40.5.1 `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 an argument.

40.5.2 `derivative(DCO_T&,DCO_ADJOINT_VECTOR_TYPE&)`

Derivative access requires specification of the associated adjoint vector.

40.5.3 `DCO_ADJOINT_VECTOR_TYPE::interpret_adjoint()`

Interpreters are called directly on the adjoint vector.

Chapter 41

Modulo Adjoint Propagation

41.1 Purpose

For each adjoint mode X already shown (i.e. $X \in \text{ga1s}, \text{ga1sm}, \text{ga1v}, \text{ga1vm}$) 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 .

41.2 Optimization

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

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

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

Please check memory decrease with

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

see Sec. 3.1.6.

41.3 Example

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

$$\frac{dx}{dt} = f(x(t,p), t, p) \quad (41.1)$$

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`:

```

1  #include <cmath>
2
3  template<typename T>
4  void f(const int n, T& x, const T& p) {
5      double dt=1./n, t=0;
6      for (int i=0; i<n; i++, t+=dt) x+=dt*p*sin(x*t);
7  }
```

The main routine is given as follows.

```

1  #include "dco.hpp"
2
3  using namespace std;
4  using namespace dco;
5
6  typedef dco::gals_mod<double> DCO_MODE;
7  typedef DCO_MODE::type DCO_TYPE;
8  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
9
10 #include "timestepping.hpp"
11
12 int main(int c, char* v[]) {
13     int n = 10;
14     if (c==2) { n=atoi(v[1]); }
15
16     DCO_TYPE x=1.0, p=1.0;
17     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
18     DCO_MODE::global_tape->register_variable(p);
19
20     f(n,x,p);
21
22     dco::derivative(x)=1.0;
23     DCO_MODE::global_tape->interpret_adjoint();
24
25     cerr << "size of internal adjoint vector = "
26          << dco::size_of(DCO_MODE::global_tape, DCO_TAPE_TYPE::
27                          size_of_int_adjoint_vector)
28          << "b" << endl;
29     cerr << "size of tape memory = "
30          << dco::size_of(DCO_MODE::global_tape, DCO_TAPE_TYPE::size_of_stack)
31          << "b" << endl;
32
33     cout << "dp=" << dco::derivative(p) << endl;
34
35     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
36     return 0;
37 }
```

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.

41.4 New dco/c++ Features

41.4.1 `ga1s_mod<DC0_BASE_TYPE>`

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

Chapter 42

Sparse Tape Interpretation

42.1 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:

```
1  ...
2  DCO_M::global_tape=DCO_TAPE_T::create();
3  // record
4  DCO_M::global_tape->sparse_interpret() = true;
5  DCO_M::global_tape->interpret_adjoint();
6
7  DCO_M::global_tape->zero_adjoints();
8  DCO_M::global_tape->sparse_interpret() = false;
9  DCO_M::global_tape->interpret_adjoint();
10 ...
```

42.2 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.

```
1  #include "dco.hpp"
2
3  typedef dco::gals<double> DCO_MODE;
4  typedef DCO_MODE::type DCO_TYPE;
5  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
6
7  int main() {
8
9      DCO_MODE::global_tape = DCO_TAPE_TYPE::create();
10     DCO_TYPE x(0.0);
```

```

11
12   DCO_MODE::global_tape->register_variable(x);
13
14   DCO_TYPE t = sqrt(x*x); (void) t; // unused...
15   DCO_TYPE y = 2*x;
16
17   dco::derivative(y) = 1.0;
18   DCO_MODE::global_tape->interpret_adjoint();
19
20   std::cerr << "Adjoint of x without sparse interpretation: " << dco::derivative
      (x) << std::endl;
21
22   DCO_MODE::global_tape->zero_adjoints();
23   DCO_MODE::global_tape->sparse_interpret() = true;
24
25   dco::derivative(y) = 1.0;
26   DCO_MODE::global_tape->interpret_adjoint();
27
28   std::cout << "Adjoint of x with sparse interpretation: " << dco::derivative(x)
      << std::endl;
29
30   DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
31 }

```

42.3 New dco/c++ Features

42.3.1 TAPE_T::sparse_interpret()

Enable and disable sparse interpretation.

Chapter 43

Logging

43.1 Purpose

Logging might be useful for information and debugging purposes.

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

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

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

and fully enable by

```
g++ main.cpp -DDCO_LOG_MAX_LEVEL=7
```

By default, the logger writes to `stderr`.

43.2 Example

The user interface to logging at runtime is via a global `logger` object.

```
1 ...  
2 dco::logger::level()=dco::logINFO;  
3 dco::logger::stream()=stdout;  
4 ...
```

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`.

43.3 New `dco/c++` Features

43.3.1 `enum dco::log_level_enum`

Different verbosity levels, defined as

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

43.3.2 `dco::log_level_enum& dco::logger::level()`

Set/get the current logging level.

43.3.3 `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")`.

Chapter 44

Explicit Type Cast to Specified Type

44.1 Purpose

As already written in *General Remarks / Essentials*, Ch. 2: Usually, it is not desired to cast a `dco/c++` type to a non-dco type (e.g. `double`), since `dco/c++` is then unable to compute derivatives of that data flow. Nonetheless, there are cases, where a cast is actually what you want to do (e.g. cast to `int` for an index calculation). We therefore allow explicit cast operations to specified data types (implicit casts are not allowed). Use carefully.

44.1.1 Remarks

This is only available in C++11, since the implementation uses the `explicit` keyword on conversion functions. Please see https://en.cppreference.com/w/cpp/language/cast_operator for description of `explicit` keyword. This enables cast operations for explicit conversion and *direct-initialization*. The latter might be a source for errors, see Example.

44.2 Example

```
1  #include "dco.hpp"
2  DCO_ENABLE_EXPLICIT_TYPE_CAST_TO(int)
3
4  int main() {
5      dco::gtls<double>::type x(1.0);
6      // explicit conversion
7      int i = static_cast<int>(x);
8      // direct-initialization
9      int j(x);
10 }
```

44.3 New dco/c++ Features

44.3.1 DCO_ENABLE_EXPLICIT_TYPE_CAST_TO(TYPE)

This macro enables the explicit operator to type TYPE, i.e.

```
1 explicit operator TYPE() { return static_cast<TYPE>(dco::value(*this)); }
```

That means, it will automatically try recursively to cast to the specified type, if requested. The recursion is required for higher-order types.

Chapter 45

Huge Pages

This is currently only supported under Linux. Please contact NAG (support@nag.co.uk) if you want this feature for other platforms.

45.1 Purpose

See for example <https://wiki.debian.org/Hugepages> for a description of huge pages in Debian. By default, most systems use a page size of 4kB. Since `dco/c++` uses a lot of memory for the tape, a lot of pages are allocated. To get faster lookup, larger page sizes can be used (e.g. 2MB on usual Linux systems).

45.2 Usage

The `tape_options` object can be used to enable huge page allocation in `dco/c++`.

```
1  dco::tape_options o;  
2  o.alloc_use_huge_pages() = true;  
3  DCO_MODE::global_tape=DCO_TAPE_TYPE::create(o);
```

Please make sure you've enabled huge pages support in your system. Under Linux, memory for huge pages needs to be created at system level first by running

```
sudo sysctl -w vm.nr_hugepages=NUMBER
```

where `NUMBER` is the number of huge pages available for allocation. This memory will be directly blocked by the system (see e.g. `htop`). You can see how many pages are blocked by running

```
cat /proc/meminfo | grep Huge
```

A sample output would look like the following:

```
AnonHugePages:      0 kB  
ShmemHugePages:     0 kB  
HugePages_Total:    1024  
HugePages_Free:     1024  
HugePages_Rsvd:      0  
HugePages_Surp:      0  
Hugepagesize:       2048 kB
```

Appendix A

Status of User Interface

Definitions:

- **Gold:** tested, signature fix, fully documented
- **Silver:** tested, signature fix
- **Bronze:** tested
- **Undefined:** experimental

We assume global use of namespace **dco**.

Disclaimer: The following code listings do not show the exact dco source. They are meant to give you an overview of the available features. Refer to the example programs for more precise information.

A.1 Gold

We are in the process of discussing the level of detail required for features to be considered “fully documented.”

A.2 Silver

- The tangent first-order scalar mode over the given DCO_BASE_TYPE.

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

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

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

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

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

- The adjoint first-order scalar mode over the given DCO_VALUE_TYPE, DCO_PARTIAL_TYPE, and DCO_ADJOINT_TYPE.

```
typedef ga1s< DCO_VALUE_TYPE, DCO_PARTIAL_TYPE, DCO_ADJOINT_TYPE >
        DCO_GA1S_MODE;
```

- The chunk size of tape determined by the user.

```
dco::tape_options o;
o.set_chunk_size_in_byte(1024*1024*1024);
o.set_chunk_size_in_kbyte(1024*1024);
o.set_chunk_size_in_mbyte(1024);
o.set_chunk_size_in_gbyte(1);
std::cout << o.chunk_size_in_byte() << std::endl;
DCO_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 the given DCO_GA1S_MODE.

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

- The tangent first-order vector mode over the given DCO_BASE_TYPE with the vector length v_size.

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

- The tangent first-order vector type for the given DCO_GT1V_MODE.

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

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

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

- The adjoint first-order vector type for the given DCO_GA1V_MODE.

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

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

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

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

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

- The adjoint first-order scalar mode over the given DCO_BASE_TYPE using modulo adjoint propagation.

```
typedef ga1s_mod< DCO_BASE_TYPE > DCO_GA1S_MOD_MODE;
```

- The adjoint first-order vector mode over the given DCO_BASE_TYPE using modulo adjoint propagation.

```
typedef ga1v_mod< DCO_BASE_TYPE, v_size=1 > DCO_GA1SV_MOD_MODE;
```

- The adjoint first-order scalar mode with support for multiple tapes using modulo adjoint propagation over the given DCO_BASE_TYPE.

```
typedef ga1sm_mod< DCO_BASE_TYPE > DCO_GA1SM_MOD_MODE;
```

- The adjoint first-order vector mode with support for multiple tapes using modulo adjoint propagation over the given DCO_BASE_TYPE.

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

From now on:

- DCO_MODE \in DCO_x_MODE with $x \in \{\text{GT1S}, \text{GA1S}, \text{GA1SM}, \text{GT1V}, \text{GA1V}, \text{GA1S_MOD}, \text{GA1SM_MOD}, \text{GA1V_MOD}, \text{GA1VM_MOD}\}$ 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 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 the given DCO_TYPE.

```
typedef mode<DCO_TYPE> DCO_MODE;
```

- Proxy types to access passive value, value and derivative component of one entry in the passed vector.

```
typedef vector_reference <PASSIVE_VALUE, DCO_TYPE,CONTAINER_T> VEC_REF_PVAL;
typedef vector_reference <VALUE          , DCO_TYPE,CONTAINER_T> VEC_REF_VAL;
typedef vector_reference <DERIVATIVE    , DCO_TYPE,CONTAINER_T> VEC_REF_DER;
```

Individual elements can be accessed through member operators

```
double&          VEC_REF_PVAL::operator[] (int);
DCO_BASE_TYPE&   VEC_REF_VAL::operator[] (int);
DCO_DERIVATIVE_TYPE& VEC_REF_DER::operator[] (int);
```

- Returns proxy object for passive value components of x.

```
VEC_REF_PVAL passive_value ( std::vector< DCO_TYPE > &x );
```

- Returns proxy object for value components of x.

```
VEC_REF_VAL value ( std::vector< DCO_TYPE > &x );
```

- Returns proxy object for derivative (tangent or adjoint) components of x.

```
VEC_REF_DER derivative ( std::vector<DCO_TYPE> &x );
```

The following description is only valid for adjoint modes.

- Local gradient for direct insertion into tape.

```
typedef DCO_TAPE_TYPE::local_gradient_t DCO_LOCAL_GRADIENT_TYPE;
```

- Creates local gradient of y for direct insertion into tape; gradient contains n elements.

```
DCO_LOCAL_GRADIENT_TYPE
DCO_MODE::create_local_gradient_object<DCO_LOCAL_GRADIENT_TYPE> (
    DCO_TYPE &y, size_t n
);
```

- Inserts 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::ostream& 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 $T = T(t, x, c) : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ as the solution of an initial and boundary value problem for the one-dimensional heat equation

$$\frac{\partial T}{\partial t} = c \cdot \frac{\partial^2 T}{\partial x^2} \quad (\text{B.1})$$

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 within 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 $t \rightarrow \infty$ to a straight line that connects the two points $(0, 300)$ and $(1, 1700)$. The dependence on c vanishes asymptotically. Hence, we consider the calibration of the parameter c at time $t = 1$.

For the given value of c and given observations $O(x)$ at time $t = 1$ we aim to solve the unconstrained least squares problem

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

The observations are obtained through the measurement procedure illustrated in the lower row of pictures in Figure B.1. We aim to estimate the unknown/uncertain material property, c , of the stick such that the real-world measurements are reproduced as closely as possible by the implementation of the mathematical model.

B.1.2 Numerical Method

The continuous mathematical model needs to be translated into a discrete formulation in order to be able to solve it on today's computers. We use *finite difference quotients* to replace both the spatial (see Sec. B.1.2) and the temporal (see Sec. B.1.2) differential terms in Equation (B.1). The integral in Equation (B.2) is discretized using a simple *quadrature rule* (see Sec. B.1.2). See, for example, [5] for a gentle introduction to finite difference discretization methods.

Spatial Discretization

The given stick of unit length $[0, 1]$ is divided into $n - 1$ sub-intervals of equal length

$$\Delta x = \frac{1}{n - 1} \quad (\text{B.3})$$

yielding a spatial discretization with $n - 2$ inner points x_1, \dots, x_{n-2} in addition to the two end (boundary) points $x_0 = 0$ and $x_{n-1} = 1$. 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

$$a_j \equiv \frac{x_{j-1} + x_j}{2} \quad (\text{B.4})$$

¹approximate temperature of a candle flame

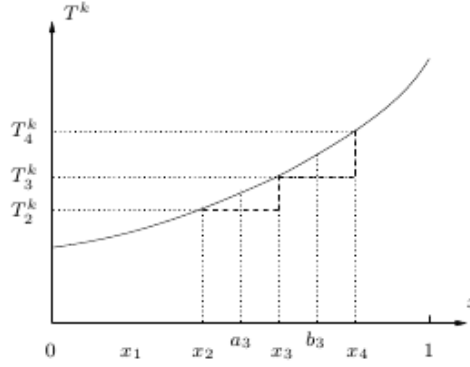


Figure B.2: Spatial Discretization: Central Finite Differences

and

$$b_j \equiv \frac{x_j + x_{j+1}}{2} \quad (\text{B.5})$$

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

$$\frac{\partial T}{\partial x}(t, a_j, c) \approx \frac{T(t, x_j, c) - T(t, x_{j-1}, c)}{\Delta x} = \frac{T_j - T_{j-1}}{\Delta x} \quad (\text{B.6})$$

for $j = 1, \dots, n-2$ and

$$\frac{\partial T}{\partial x}(t, b_j, c) \approx \frac{T(t, x_{j+1}, c) - T(t, x_j, c)}{\Delta x} = \frac{T_{j+1} - T_j}{\Delta x} \quad (\text{B.7})$$

for $j = 1, \dots, n-2$ follows

$$\frac{\partial^2 T}{\partial x^2}(t, x_j, c) \approx \frac{\frac{\partial T}{\partial x}(t, b_j, c) - \frac{\partial T}{\partial x}(t, a_j, c)}{\Delta x} = \frac{T_{j+1} - 2 \cdot T_j + T_{j-1}}{(\Delta x)^2} \quad (\text{B.8})$$

for $j = 1, \dots, n-2$ and hence the system of ordinary differential equations (ODE)

$$\frac{\partial T_j}{\partial t} = r_j(c, \Delta x, T), \quad j = 0, \dots, n-1, \quad (\text{B.9})$$

where

$$r_j = 0 \quad j \in \{0, n-1\} \quad (\text{B.10})$$

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

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 $j = 1, \dots, n-2$) as follows:

$$\frac{\partial T}{\partial t} = \underbrace{r(c, \Delta x, 0)}_{=0} + \frac{\partial r}{\partial T}(c, \Delta x) \cdot (T - 0). \quad (\text{B.12})$$

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 Δx a directional derivative in direction $T = (T_j)_{j=0, \dots, n-1}$ is computed by the second term.

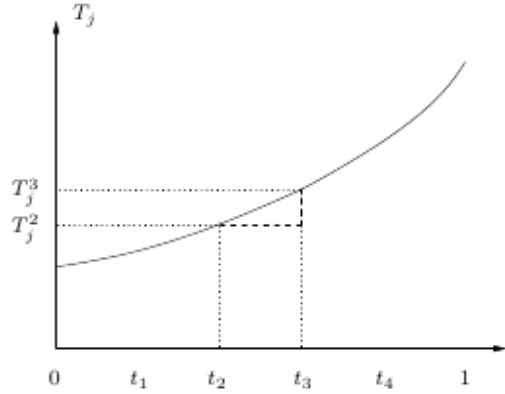


Figure B.3: Temporal Discretization: Backward Finite Difference

Example For $n = 6$ the Jacobian of the residual becomes

$$\frac{\partial r}{\partial T}(c, \Delta x) = \frac{c}{(\Delta x)^2} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}. \quad (\text{B.13})$$

The first and the last rows are identically equal to zero due to the constant Dirichlet boundary conditions (see Equation (B.10)).

The residual $r(c, \Delta x, T)$ is evaluated by the function

```
template <typename TYPE>
inline void residual(const vector<TYPE>& c, vector<TYPE>& T) {
    size_t n=T.size();
    vector<TYPE> T_new(n);
    for (size_t i=1; i<n-1; ++i)
        T_new[i]=c[i]*(n-1)*(n-1)*(T[i-1]-2*T[i]+T[i+1]));
    T[0]=0;
    for (size_t i=0; i<n; ++i) T[i]=T_new[i];
    T[n-1]=0;
}
```

and using Equation (B.3).

Temporal Discretization

The differential left-hand side of the ODE in Equation (B.9) is approximated by a backward finite difference approximation yielding the *backward Euler method* (see, for example, [5]). Similar to Sec. B.1.2, the unit time interval $[0, 1]$ is decomposed into m sub-intervals of equal length

$$\Delta t = \frac{1}{m} \quad (\text{B.14})$$

yielding a temporal discretization with m states T_j^1, \dots, T_j^m , and a given initial state T_j^0 for $j = 0, \dots, n-1$. 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

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

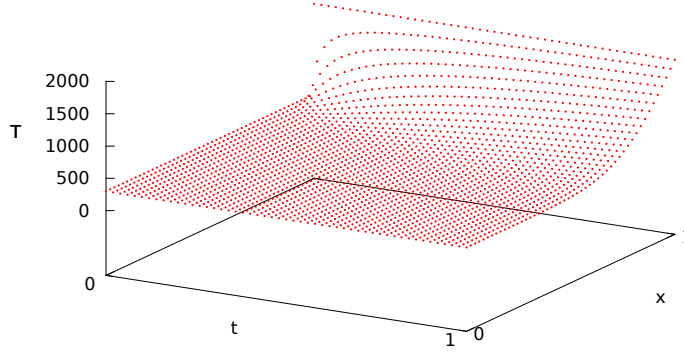


Figure B.4: Discrete Simulation of Temperature T as a Function of Time t and Position x .

follows

$$\frac{T^{k+1} - T^k}{\Delta t} = r(c, \Delta x, T^{k+1}) = \frac{\partial r}{\partial T}(c, \Delta x) \cdot T^{k+1} \quad (\text{B.16})$$

yielding the recurrence

$$\left(\Delta t \cdot \frac{\partial r}{\partial T}(c, \Delta x) - I \right) \cdot T^{k+1} = -T^k \quad (\text{B.17})$$

with a constant system matrix on the left-hand side for given c , Δx , and Δt and where I denotes the identity in \mathbb{R}^n .

Methods for computing the Jacobian $\frac{\partial r}{\partial T}(c, \Delta x)$ of the residual by the function

```
template <typename TYPE>
inline void residual_jacobian(const vector<TYPE>& c, vector<TYPE>& J);
```

are discussed in Sec. B.1.3. Moreover, we require an LU decomposition

```
template <class TYPE>
inline void LUDecomp(vector<TYPE>& A);
```

and a linear solver for a given LU decomposition of A

```
template <class TYPE>
inline void Solve(const vector<TYPE>& LU, vector<TYPE>& b);
```

for the solution of the system of linear equations

$$\underbrace{\left(\Delta t \cdot \frac{\partial r}{\partial T}(c, \Delta x) - I \right)}_{=:A} \cdot T^{k+1} = \underbrace{-T^k}_{:=b} \quad (\text{B.18})$$

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) {
```

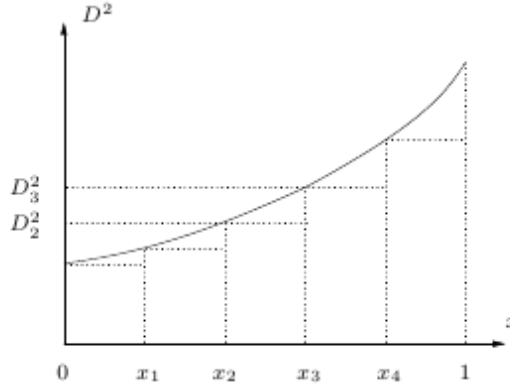


Figure B.5: Discretization of Objective: Quadrature

```

    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

$$y = f(c, n, m, T, O) = \frac{1}{n-1} \cdot \sum_{j=0}^{n-2} (T_j^m(c) - O_j)^2. \quad (\text{B.19})$$

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 (α) 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 $T^{(1)} \triangleq \mathbf{t1_T} \in \mathbb{R}^n$ without prior accumulation of the Jacobian itself. It computes a *matrix-free* directional derivative.

Jacobian of the Residual

The Jacobian $\frac{\partial r}{\partial T} \triangleq \mathbf{J} = \mathbf{J}[:, :] \in \mathbb{R}^{n \times n}$ of the residual is computed column-wise by letting the input vector $T^{(1)} \triangleq \mathbf{t1_T} \in \mathbb{R}^n$ range over the Cartesian basis vectors in \mathbb{R}^n .

```

template <typename TYPE>
inline void residual_jacobian(const vector<TYPE>& c, vector<TYPE>& J) {
    size_t n=c.size();
    vector<TYPE> t1_T(n);
    for (size_t i=0;i<n;++i) t1_T[i]=0;
    for (size_t i=0;i<n;++i) {
        t1_T[i]=1;
        residual(c,t1_T);
        for (size_t j=0;j<n;++j) {
            J[j*n+i]=t1_T[j];
            t1_T[j]=0;
        }
    }
}

```

Linearity of the residual enables the use of the primal function `residual(...)` for the computation of the directional derivative. Sparsity of the (in this case tridiagonal) Jacobian can and should be exploited. Refer to [4] or [9] for details on corresponding compression techniques. Associated graph colouring problems are discussed in [2].

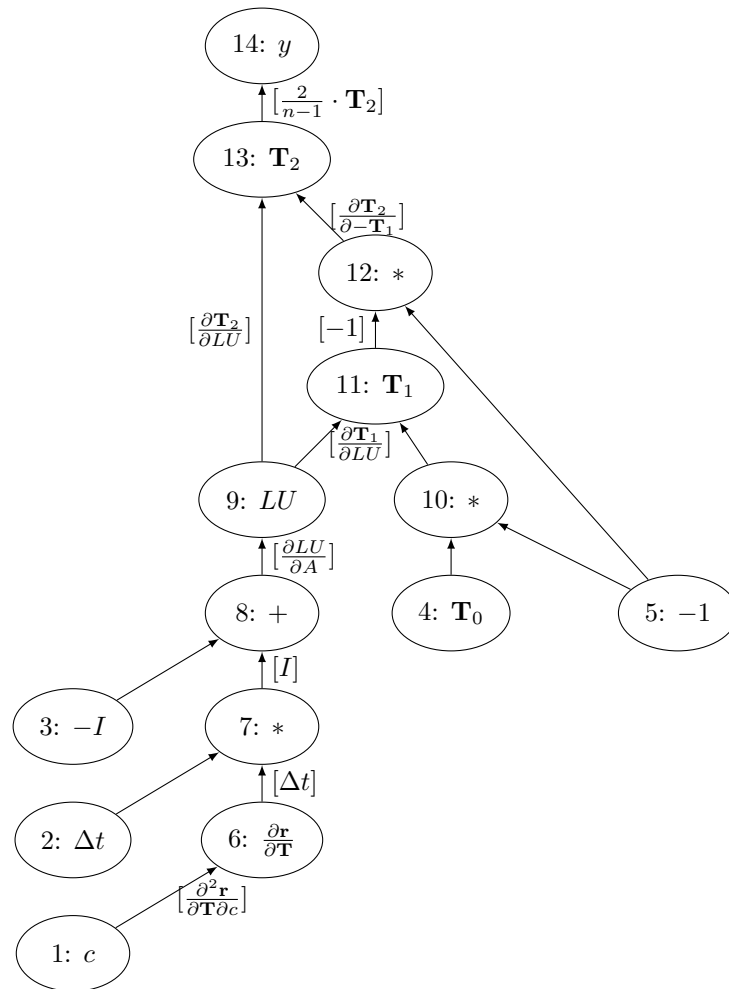


Figure B.6: Linearized DAG of Simulation Method

Gradient of the Objective

The gradient of the objective with respect to the free parameters that enter the simulation ($c \in \mathbb{R}$ in our case; see Sec. B.1.4 for comments on the non-scalar case $c = c(x)$) is required by the steepest descent algorithm. It can be obtained by overloading the entire simulation and the following evaluation of the objective for either a tangent or an adjoint AD type. Adjoint mode should be preferred for large numbers of parameters where the actual number depends on the performance of the AD implementation provided by the given AD tool.

AD of \mathbf{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 \mathbf{f} in tangent mode implies the evaluation of a tangent model of the Jacobian computation (see Sec. B.1.3). Implicitly, second-order derivatives are computed. The computation of

$$A = \Delta t \cdot \frac{\partial r}{\partial T}(c, \Delta x) - I$$

yields the tangent projection

$$A^{(2)} = \left\langle \frac{\partial A}{\partial c}, c^{(2)} \right\rangle + \left\langle \frac{\partial A}{\partial T}, T^{(2)} \right\rangle = \Delta t \cdot \left\langle \frac{\partial^2 r}{\partial T \partial c}(c, \Delta x), c^{(2)} \right\rangle,$$

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

$$\begin{pmatrix} T_{(2)} \\ c_{(2)} \end{pmatrix} = \left\langle A_{(2)}, \frac{\partial A}{\partial(T, c)} \right\rangle \quad (\text{B.20})$$

$$= \Delta t \cdot \left\langle A_{(2)}, \frac{\partial^2 r}{\partial T \partial(T, c)}(c, \Delta x) \right\rangle \quad (\text{B.21})$$

$$= \begin{pmatrix} 0 \\ \left\langle A_{(2)}, \frac{\partial^2 r}{\partial T \partial c}(c, \Delta x) \right\rangle \end{pmatrix}. \quad (\text{B.22})$$

Linearity of r in T yields $T_{(2)} = 0 \in \mathbb{R}^n$.

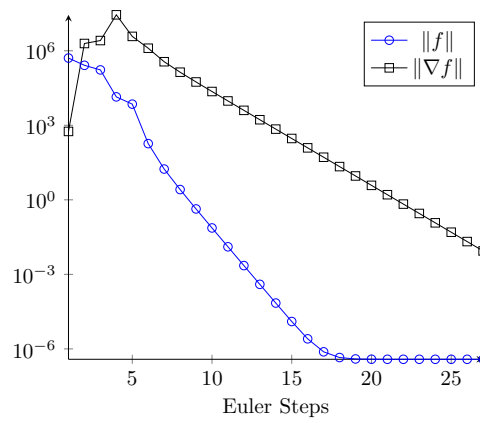
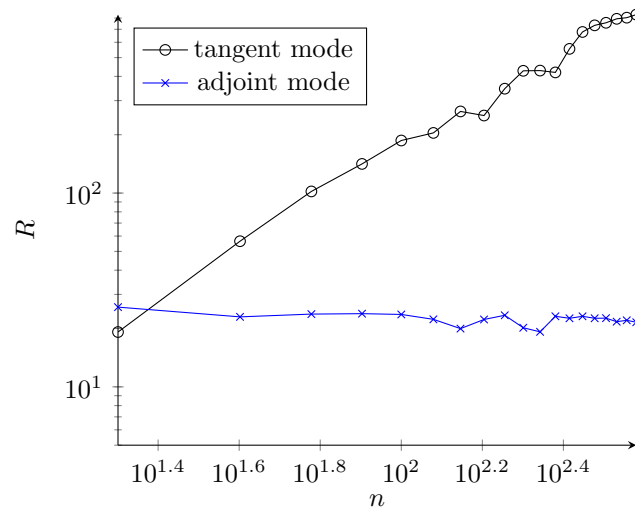
B.1.4 Optimization

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 $c = (c_i)_{i=0, \dots, n-1}$ 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 .

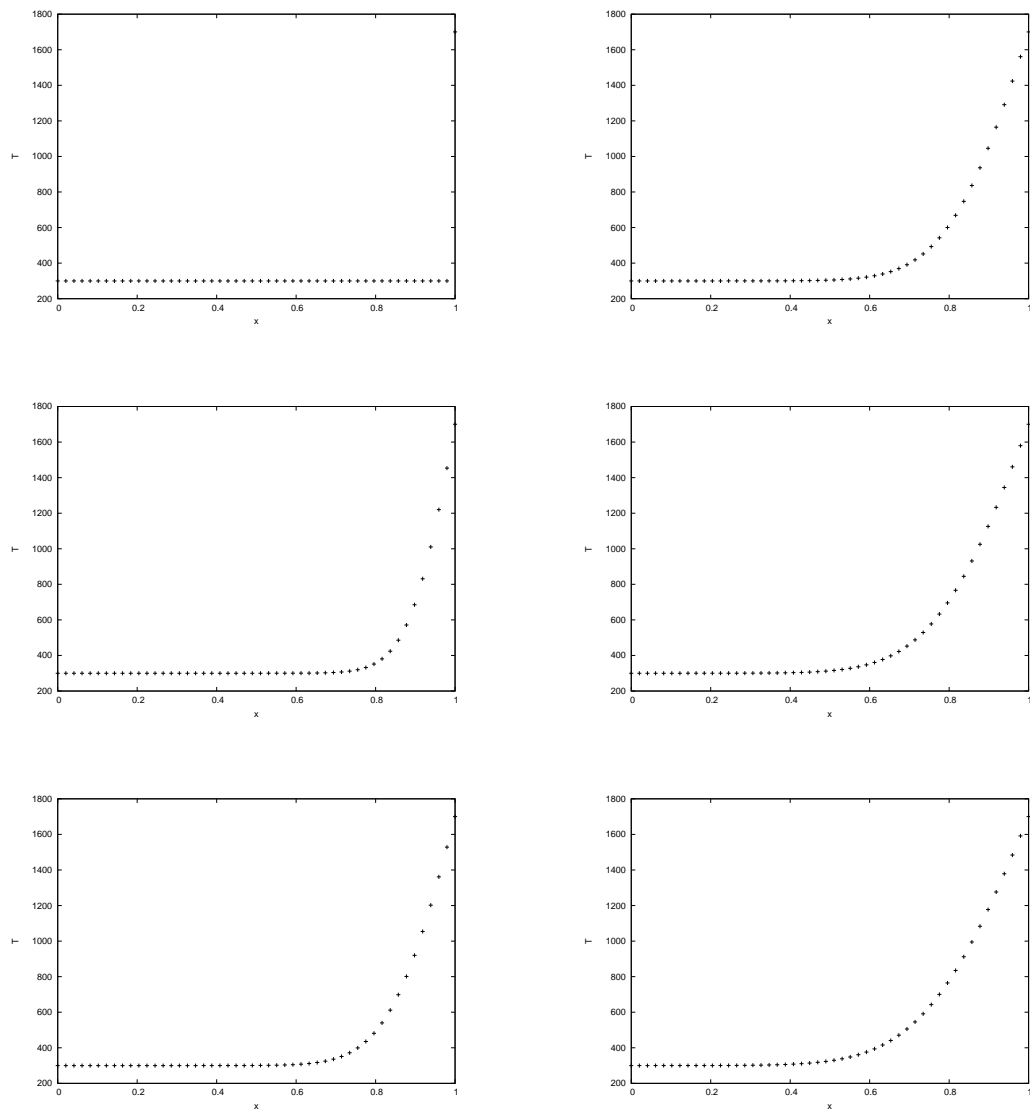


Figure B.9: Simulated Temperature Distribution for $t = 0, 0.2, 0.4$ (Left Column) and $t = 0.6, 0.8, 1$ (Right Column)

B.1.5 Code

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

```

1  #include "dco.hpp"
2  using namespace dco;
3
4  template <typename cTYPE, typename TTYPE>
5  inline void residual(const vector<cTYPE>& c, vector<TTYPE>& T) {
6      size_t n=T.size();
7      vector<TTYPE> T_new(n);
8      for (size_t i=1;i<n-1;++i)
9          T_new[i]=c[i]*static_cast<double>(n-1)*static_cast<double>(n-1)*(T[i-1]-2*T[
10             i]+T[i+1]);
11      T[0]=0;
12      for (size_t i=0;i<n;++i) T[i]=T_new[i];
13      T[n-1]=0;
14  }
15
16  /*
17  template <typename TYPE>
18  inline void residual_jacobian(const vector<TYPE>& c, vector<TYPE>& J) {
19      size_t n=c.size();
20      vector<TYPE> t1_T(n);
21      for (size_t i=0;i<n;++i) t1_T[i]=0;
22      for (size_t i=0;i<n;++i) {
23          t1_T[i]=1;
24          residual(c,t1_T);
25          for (size_t j=0;j<n;++j) {
26              J[j*n+i]=t1_T[j];
27              t1_T[j]=0;
28          }
29      }
30  }
31
32  template <typename TYPE>
33  inline void residual_jacobian(const vector<TYPE>& c, const vector<TYPE>& T,
34      vector<TYPE>& J) {
35      typedef typename dco::gtls<TYPE>::type DCO_T1S_TYPE;
36      size_t n=T.size();
37      vector<DCO_T1S_TYPE> t1_T(n);
38      for (size_t i=0;i<n;i++) t1_T[i]=T[i];
39      for (size_t i=0;i<n;i++) {
40          derivative(t1_T[i])=1;
41          residual(c,t1_T);
42          for (size_t j=0;j<n;++j) {
43              J[j*n+i]=derivative(t1_T[j]);
44              derivative(t1_T[j])=0;
45          }
46      }
47  }
48  */
49
50  template <typename TYPE>

```

```

49 inline void residual_jacobian(const vector<TYPE>& c, const vector<TYPE>& T,
    vector<TYPE>& J) {
50     typedef typename dco::gtls<TYPE>::type DCO_T1S_TYPE;
51     size_t n=T.size();
52     vector<DCO_T1S_TYPE> t1T(n);
53     const size_t bw=3;
54     for (size_t i=0;i<n;i++) t1T[i]=T[i];
55     for (size_t i=0;i<bw;i++) {
56         for (size_t j=i;j<n;j+=bw) derivative(t1T[j])=1;
57         residual(c,t1T);
58         for (size_t j=i;j<n;j+=bw) {
59             for (size_t k=(j<(bw-1)/2)? 0 : j-(bw-1)/2;k<= min(n-1,j+(bw-1)/2);k++)
60                 J[k*n+j]=derivative(t1T[k]);
61             derivative(t1T[j])=0;
62         }
63         for (size_t j=0;j<n;j++) derivative(t1T[j])=0;
64     }
65 }
66
67 template <class TYPE>
68 inline void LUDecomp(vector<TYPE>& A) {
69     size_t n = static_cast<size_t>(sqrt(double(A.size())));
70     for (size_t k=0;k<n;k++) {
71         for (size_t i=k+1;i<n;i++)
72             A[i*n+k]=A[i*n+k]/A[k*n+k];
73         for (size_t j=k+1;j<n;j++)
74             for (size_t i=k+1;i<n;i++)
75                 A[i*n+j]=A[i*n+j]-A[i*n+k]*A[k*n+j];
76     }
77 }
78
79 // L*y=b
80 template <class TYPE>
81 inline void FSubst(const vector<TYPE>& LU, vector<TYPE>& b) {
82     size_t n=b.size();
83     for (size_t i=0;i<n;i++)
84         for (size_t j=0;j<i;j++)
85             b[i]=b[i]-LU[i*n+j]*b[j];
86 }
87
88 // U*x=y
89 template <class TYPE>
90 inline void BSubst(const vector<TYPE>& LU, vector<TYPE>& y) {
91     size_t n=y.size();
92     for (size_t k=n,i=n-1;k>0;k--,i--) {
93         for (size_t j=n-1;j>i;j--)
94             y[i]=y[i]-LU[i*n+j]*y[j];
95         y[i]=y[i]/LU[i*n+i];
96     }
97 }
98
99 template <class TYPE>
100 inline void Solve(const vector<TYPE>& LU, vector<TYPE>& b) {
101     FSubst(LU,b);

```

```

102     BSubst(LU,b);
103 }
104
105 template <typename TYPE>
106 inline void sim(const vector<TYPE>& c, const size_t m, vector<TYPE>& T) {
107     size_t n=T.size();
108     vector<TYPE> A(n*n,0);
109
110     residual_jacobian(c,T,A);
111
112     for (size_t i=0;i<n;++i) {
113         for (size_t j=0;j<n;++j)
114             A[i*n+j]=A[i*n+j]/static_cast<double>(m);
115         A[i+i*n]=A[i+i*n]-1;
116     }
117
118     LUDecomp(A);
119
120     for (size_t j=0;j<m;++j) {
121         for (size_t i=0;i<n;++i)
122             T[i]=-T[i];
123         Solve(A,T);
124     }
125 }
126
127 template <typename TYPE, typename OTYPE>
128 inline void f(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
129     vector<OTYPE>& O, TYPE& v) {
130     size_t n=T.size();
131     sim(c,m,T);
132
133     v=0;
134     for (size_t i=0;i<n-1;++i)
135         v=v+(T[i]-O[i])*(T[i]-O[i]);
136     v=v/(static_cast<double>(n)-1);
137 }

```

B.2 Function

This section shows the evaluation of the objective.

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <fstream>
4  #include <vector>
5  using namespace std;
6
7  //#include "../include/f_residual_Jacobian_by_hand.hpp"
8  //#include "../include/f_residual_Jacobian_by_dco_dense.hpp"
9  #include "../include/f.hpp"
10
11 int main(int argc, char* argv[]){
12     cout.precision(15);

```

```

13  if (argc!=3) {
14      cerr << "2 parameters expected:" << endl
15      << "  1. number of spatial finite difference grid points" << endl
16      << "  2. number of implicit Euler steps" << endl;
17      return -1;
18  }
19  size_t n=atoi(argv[1]), m=atoi(argv[2]);
20  vector<double> c(n);
21  for (size_t i=0;i<n;i++) c[i]=0.01;
22  vector<double> T(n);
23  for (size_t i=0;i<n-1;i++) T[i]=300.;
24  T[n-1]=1700.;
25  ifstream ifs("0.txt");
26  vector<double> O(n);
27  for (size_t i=0;i<n;i++) ifs >> O[i] ;
28  double v;
29  f(c, m, T, O, v);
30  cout << "v=" << v << endl;
31  return 0;
32 }

```

B.3 Observations

This section shows the generation of the “observations.”

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <fstream>
4  #include <vector>
5  using namespace std;
6
7  #include "../include/f.hpp"
8
9  int main(int argc, char* argv[]){
10     cout.precision(15);
11     if (argc!=3) {
12         cerr << "2 parameters expected:" << endl
13         << "  1. number of spatial finite difference grid points" << endl
14         << "  2. number of implicit Euler steps" << endl;
15         return -1;
16     }
17     size_t n=atoi(argv[1]), m=atoi(argv[2]);
18     vector<double> c(n);
19     for (size_t i=0;i<n;i++) c[i]=0.01;
20     vector<double> T(n);
21     for (size_t i=0;i<n-1;i++) T[i]=300.;
22     T[n-1]=1700.;
23     vector<double> O(n,0);
24     double v;
25     ofstream ofs("0.txt");
26     f(c,m,T,O,v);
27     for (size_t i=0;i<n;i++) ofs << T[i]+(double) rand()/RAND_MAX << endl;
28     return 0;

```

29 }

B.4 Gradient

B.4.1 Finite Differences

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

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <fstream>
4  #include <vector>
5  #include <cfloat>
6  using namespace std;
7
8  #include "../include/f.hpp"
9
10 int main(int argc, char* argv[]){
11     cout.precision(15);
12     if (argc!=3) {
13         cerr << "2 parameters expected:" << endl
14         << " 1. number of spatial finite difference grid points" << endl
15         << " 2. number of implicit Euler steps" << endl;
16         return -1;
17     }
18     size_t n=atoi(argv[1]), m=atoi(argv[2]);
19     vector<double> c(n);
20     for (size_t i=0;i<n;i++) c[i]=0.01;
21     vector<double> T(n);
22     for (size_t i=0;i<n-1;i++) T[i]=300.;
23     T[n-1]=1700.;
24     ifstream ifs("0.txt");
25     vector<double> O(n);
26     for (size_t i=0;i<n;i++) ifs >> O[i] ;
27     vector<double> dvdc(n,0),cph(n,0),cmh(n,0);
28     double vmh;
29     double vph;
30     for(size_t j=0;j<n;j++) {
31         double h=(c[j]==0) ? sqrt(DBL_EPSILON) : sqrt(DBL_EPSILON)*abs(c[j]);
32         for (size_t i=0;i<n;i++) cmh[i]=c[i];
33         for (size_t i=0;i<n-1;i++) T[i]=300.;
34         T[n-1]=1700.;
35         cmh[j]-=h;
36         f(cmh,m,T,0,vmh);
37         for (size_t i=0;i<n;i++) cph[i]=c[i];
38         for (size_t i=0;i<n-1;i++) T[i]=300.;
39         T[n-1]=1700.;
40         cph[j]+=h;
41         f(cph,m,T,0,vph);
42         dvdc[j]=(vph-vmh)/(2*h);
43     }
44     cout.precision(15);

```

```

45     for(size_t i=0;i<n;i++)
46         cout << "dvdc[" << i << "]= " << dvdc[i] << endl;
47     return 0;
48 }

```

B.4.2 `gtls<double>`

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

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5  #include "dco.hpp"
6  using namespace dco;
7
8  typedef gtl<double>::type DCO_TYPE;
9
10 #include "../include/f.hpp"
11
12 int main(int argc, char* argv[]){
13     if (argc!=3) {
14         cerr << "2 parameters expected:" << endl
15         << " 1. number of spatial finite difference grid points" << endl
16         << " 2. number of implicit Euler steps" << endl;
17         return -1;
18     }
19     size_t n=atoi(argv[1]), m=atoi(argv[2]);
20     vector<double> c(n);
21     for (size_t i=0;i<n;i++) c[i]=0.01;
22     vector<double> T(n);
23     for (size_t i=0;i<n-1;i++) T[i]=300.;
24     T[n-1]=1700.;
25     ifstream ifs("0.txt");
26     vector<double> O(n);
27     for (size_t i=0;i<n;i++) ifs >> O[i] ;
28     vector<double> dvdc(n);
29     vector<DCO_TYPE> ca(n);
30     vector<DCO_TYPE> Ta(n);
31     DCO_TYPE va;
32     for(size_t i=0;i<n;i++) {
33         for(size_t j=0;j<n;j++) { ca[j]=c[j]; Ta[j]=T[j]; }
34         derivative(ca[i])=1.0;
35         f(ca,m,Ta,0,va);
36         dvdc[i]=derivative(va);
37     }
38     cout.precision(15);
39     for(size_t i=0;i<n;i++)
40         cout << "dvdc[" << i << "]= " << dvdc[i] << endl;
41     return 0;
42 }

```

B.4.3 `ga1s< double >`

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

```

1  #include <cstdlib>
2  #include <iostream>
3  using namespace std;
4  #include "dco.hpp"
5  using namespace dco;
6  typedef ga1s<double> DCO_MODE;
7  typedef DCO_MODE::type DCO_TYPE;
8  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
9
10 #include "../include/f.hpp"
11
12 int main(int argc, char* argv[]){
13     if (argc!=3) {
14         cerr << "2 parameters expected:" << endl
15         << " 1. number of spatial finite difference grid points" << endl
16         << " 2. number of implicit Euler steps" << endl;
17         return -1;
18     }
19     size_t n=atoi(argv[1]), m=atoi(argv[2]);
20     vector<double> c(n);
21     for (size_t i=0;i<n;i++) c[i]=0.01;
22     vector<double> T(n);
23     for (size_t i=0;i<n-1;i++) T[i]=300.;
24     T[n-1]=1700.;
25     ifstream ifs("0.txt");
26     vector<double> O(n);
27     for (size_t i=0;i<n;i++) ifs >> O[i] ;
28     vector<double> dvdc(n);
29
30     vector<DCO_TYPE> ca(n);
31     vector<DCO_TYPE> Ta(n);
32     DCO_TYPE va;
33     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
34     for(size_t i=0;i<n;i++) {
35         ca[i]=c[i];
36         Ta[i]=T[i];
37         DCO_MODE::global_tape->register_variable(ca[i]);
38         DCO_MODE::global_tape->register_variable(Ta[i]);
39     }
40     f(ca,m,Ta,O,va);
41     cerr << "record (0," << m << ")=" << dco::size_of(DCO_MODE::global_tape) << "B"
42         << endl;
43     derivative(va)=1.0;
44     DCO_MODE::global_tape->register_output_variable(va);
45     DCO_MODE::global_tape->interpret_adjoint();
46     for (size_t i=0;i<n;i++) dvdc[i]=derivative(ca[i]);
47     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
48     cout.precision(15);
49     for(size_t i=0;i<n;i++)
50         cout << "dvdc[" << i << "]= " << dvdc[i] << endl;

```

```

50     return 0;
51 }

```

B.4.4 `gals<double>` + Equidistant Checkpointing

This section shows the computation of the gradient in adjoint mode with equidistant checkpointing.

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <cassert>
4  using namespace std;
5  #include "dco.hpp"
6  using namespace dco;
7  typedef gals<double> DCO_MODE;
8  typedef DCO_MODE::type DCO_TYPE;
9  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
10
11 #include "../include/f.hpp"
12
13 template<typename TYPE>
14 void EulerSteps(const size_t m, const vector<TYPE>& A, vector<TYPE>& T){
15     size_t n=T.size();
16     for (size_t j=0;j<m;j++) {
17         for (size_t i=0;i<n;++i) T[i]=-T[i];
18         Solve(A,T);
19     }
20 }
21
22 template<typename DCO_MODE>
23 void EulerSteps_fill_gap(typename DCO_MODE::external_adjoint_object_t *D){
24     typedef typename DCO_MODE::type DCO_TYPE;
25     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
26     typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
27     typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
28
29     const DCO_TAPE_POSITION_TYPE p0 = DCO_MODE::global_tape->get_position();
30     const size_t& m=D->template read_data<size_t>();
31     const size_t& n = D->template read_data<size_t>();
32     vector<DCO_TYPE>* A_p=D->template read_data<vector<DCO_TYPE>*>();
33     vector<DCO_TYPE> T(n);
34     for (size_t i=0;i<n;i++) {
35         T[i]=D->template read_data<DCO_VALUE_TYPE>();
36         DCO_MODE::global_tape->register_variable(T[i]);
37     }
38     vector<DCO_TYPE> T_in(n);
39     for (size_t i=0;i<n;i++) T_in[i]=T[i];
40     DCO_TAPE_POSITION_TYPE p1=DCO_MODE::global_tape->get_position();
41     EulerSteps(m,*A_p,T);
42     cerr << "record =" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
43     for (size_t i=0;i<n;i++) {
44         DCO_MODE::global_tape->register_output_variable(T[i]);
45         derivative(T[i])=D->get_output_adjoint();
46     }

```

```

47     DCO_MODE::global_tape->interpret_adjoint_to(p1);
48     for (size_t i=0; i<n; i++)
49         D->increment_input_adjoint(derivative(T_in[i]));
50     DCO_MODE::global_tape->reset_to(p0);
51 }
52
53 template<typename DCO_TYPE>
54 void EulerSteps_make_gap(const size_t m, const vector<DCO_TYPE>& A, vector<
    DCO_TYPE>& T){
55     typedef mode<DCO_TYPE> DCO_MODE;
56     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
57     size_t n=T.size();
58
59     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
60     D->write_data(m); D->write_data(n); D->write_data(&A);
61     for (size_t i=0; i<n; i++) {
62         D->register_input(T[i]);
63         D->write_data(value(T[i]));
64     }
65     DCO_MODE::global_tape->switch_to_passive();
66     EulerSteps(m,A,T);
67     DCO_MODE::global_tape->switch_to_active();
68     for (size_t i=0; i<n; i++) T[i]=D->register_output(value(T[i]));
69     DCO_MODE::global_tape->insert_callback(EulerSteps_fill_gap<DCO_MODE>,D);
70 }
71
72 template <typename TYPE>
73 inline void sim(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
    size_t cs) {
74     size_t n=T.size();
75     static vector<TYPE> A(n*n); // static avoids copy
76     residual_jacobian(c,T,A);
77     for (size_t i=0; i<n; ++i) {
78         for (size_t j=0; j<n; ++j)
79             A[i*n+j]=A[i*n+j]/m;
80         A[i+i*n]=A[i+i*n]-1;
81     }
82     LUDecomp(A);
83     for (size_t j=0; j<m; j+=cs) {
84         size_t s=(j+cs<m) ? cs : m-j;
85         EulerSteps_make_gap<DCO_TYPE>(s,A,T);
86     }
87 }
88
89 template <typename TYPE, typename OTYPE>
90 inline void f(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
    vector<OTYPE>& O, TYPE& v, const size_t cs) {
91     sim(c,m,T,cs);
92     v=0;
93     size_t n=T.size();
94     for (size_t i=0; i<n-1; ++i)
95         v=v+(T[i]-O[i])*(T[i]-O[i]);
96     v=v/(n-1);

```

```

97 }
98
99 int main(int argc, char* argv[]){
100     if (argc!=4) {
101         cerr << "2 parameters expected:" << endl
102         << " 1. number of spatial finite difference grid points" << endl
103         << " 2. number of implicit Euler steps" << endl
104         << " 3. distance between checkpoints" << endl;
105         return -1;
106     }
107     size_t n=atoi(argv[1]), m=atoi(argv[2]), cs=atoi(argv[3]);
108     assert(cs<=m);
109     vector<double> c(n);
110     for (size_t i=0;i<n;i++) c[i]=0.01;
111     vector<double> T(n);
112     for (size_t i=0;i<n-1;i++) T[i]=300.;
113     T[n-1]=1700.;
114     ifstream ifs("0.txt");
115     vector<double> O(n);
116     for (size_t i=0;i<n;i++) ifs >> O[i] ;
117     vector<double> dvdc(n);
118
119     vector<DCO_TYPE> ca(n);
120     vector<DCO_TYPE> Ta(n);
121     DCO_TYPE va;
122     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
123     for(size_t i=0;i<n;i++) {
124         ca[i]=c[i];
125         Ta[i]=T[i];
126         DCO_MODE::global_tape->register_variable(ca[i]);
127         DCO_MODE::global_tape->register_variable(Ta[i]);
128     }
129     f(ca,m,Ta,O,va,cs);
130     cerr << "record=" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
131     derivative(va)=1.0;
132     DCO_MODE::global_tape->register_output_variable(va);
133     DCO_MODE::global_tape->interpret_adjoint();
134     for (size_t i=0;i<n;i++) dvdc[i]=derivative(ca[i]);
135     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
136     cout.precision(15);
137     for(size_t i=0;i<n;i++)
138         cout << "dvdc[" << i << "]= " << dvdc[i] << endl;
139     return 0;
140 }

```

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.

```

1 #include <cstdlib>
2 #include <iostream>
3 #include <cassert>

```

```

4 using namespace std;
5 #include "dco.hpp"
6 using namespace dco;
7 typedef gals<double> DCO_MODE;
8 typedef DCO_MODE::type DCO_TYPE;
9 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
10
11 #include "../include/f.hpp"
12
13 template<typename DCO_MODE>
14 void LUDecomp_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
15     (void) D;
16 }
17
18 template<typename DCO_MODE, typename TYPE>
19 void LUDecomp_make_gap(vector<TYPE>& A){
20     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
21
22     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
23
24     size_t n=static_cast<size_t>(sqrt(double(A.size())));
25     vector<double> Ap(n*n);
26     for(size_t i=0;i<n*n;i++) Ap[i]=value(A[i]);
27     LUDecomp(Ap);
28     for(size_t i=0;i<n;i++)
29         for(size_t j=0;j<n;j++)
30             value(A[i*n+j])=Ap[i*n+j];
31
32     DCO_MODE::global_tape->insert_callback(LUDecomp_fill_gap<DCO_MODE>,D);
33 }
34
35 //  $U^T * y = b$ 
36 template <class TYPE>
37 inline void FSubstT(const vector<TYPE>& LU, vector<TYPE>& y) {
38     size_t n=y.size();
39     for (size_t i=0;i<n;i++){
40         for (size_t j=0;j<i;j++)
41             y[i]=y[i]-LU[j*n+i]*y[j];
42         y[i]=y[i]/LU[i*n+i];
43     }
44 }
45
46 //  $L^T * x = y$ 
47 template <class TYPE>
48 inline void BSubstT(const vector<TYPE>& LU, vector<TYPE>& b) {
49     size_t n=b.size();
50     for (size_t k=n,i=n-1;k>0;k--,i--)
51         for (size_t j=n-1;j>i;j--)
52             b[i]=b[i]-LU[j*n+i]*b[j];
53 }
54
55 //  $LU^T * x = y$ 
56 template <class TYPE>

```

```

57 inline void SolveT(const vector<TYPE>& LU, vector<TYPE>& b) {
58     FSubstT(LU,b);
59     BSubstT(LU,b);
60 }
61
62 template<typename DCO_MODE>
63 void Solve_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
64     typedef typename DCO_MODE::type DCO_TYPE;
65     vector<DCO_TYPE>* LU_p = D->template read_data<vector<DCO_TYPE>*>();
66     size_t n=static_cast<size_t>(sqrt(double(LU_p->size())));
67     vector<DCO_TYPE> T(n), a1T(n);
68
69     for (size_t i=0;i<n;i++) a1T[i]=D->get_output_adjoint();
70     DCO_MODE::global_tape->switch_to_passive();
71     SolveT(*LU_p,a1T);
72     for (size_t i=0;i<n;i++) T[i]=D->template read_data<double>();
73     for (size_t i=0;i<n;i++)
74         for (size_t j=0;j<n;j++)
75             D->increment_input_adjoint(value(-a1T[i]*T[j]));
76     for (size_t i=0;i<n;i++)
77         D->increment_input_adjoint(value(a1T[i]));
78     DCO_MODE::global_tape->switch_to_active();
79 }
80
81 template<typename DCO_MODE, typename TYPE>
82 void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b){
83     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
84
85     const size_t n=b.size();
86     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
87         DCO_EAO_TYPE>();
88
89     D->write_data(&LU);
90     for (size_t i=0;i<n*n;i++) D->register_input(LU[i]);
91     for (size_t i=0;i<n;i++) D->register_input(b[i]);
92     DCO_MODE::global_tape->switch_to_passive();
93     Solve(LU,b);
94     DCO_MODE::global_tape->switch_to_active();
95     for(size_t i=0;i<n;i++) {
96         D->write_data(value(b[i]));
97         b[i]=D->register_output(value(b[i]));
98     }
99     DCO_MODE::global_tape->insert_callback(Solve_fill_gap<DCO_MODE>,D);
100 }
101
102 template <>
103 inline void sim(const vector<DCO_TYPE>& c, const size_t m, vector<DCO_TYPE>& T)
104 {
105     const size_t n=c.size();
106     static vector<DCO_TYPE> A(n*n);
107     residual_jacobian(c,T,A);
108     for (size_t i=0;i<n;++i) {
109         for (size_t j=0;j<n;++j)
110             A[i*n+j]=A[i*n+j] / static_cast<double>(m);

```

```

109     A[i+i*n]=A[i+i*n]-1;
110 }
111 LUDecomp_make_gap<DCO_MODE>(A);
112 for (size_t j=0;j<m;++j) {
113     for (size_t i=0;i<n;++i) T[i]=-T[i];
114     Solve_make_gap<DCO_MODE>(A,T);
115 }
116 }
117
118 int main(int argc, char* argv[]){
119     if (argc!=3) {
120         cerr << "2 parameters expected:" << endl
121         << " 1. number of spatial finite difference grid points" << endl
122         << " 2. number of implicit Euler steps" << endl;
123         return -1;
124     }
125     size_t n=atoi(argv[1]), m=atoi(argv[2]);
126     vector<double> c(n);
127     for (size_t i=0;i<n;i++) c[i]=0.01;
128     vector<double> T(n);
129     for (size_t i=0;i<n-1;i++) T[i]=300.;
130     T[n-1]=1700.;
131     ifstream ifs("0.txt");
132     vector<double> O(n);
133     for (size_t i=0;i<n;i++) ifs >> O[i] ;
134     vector<double> dvdc(n);
135
136     vector<DCO_TYPE> ca(n);
137     vector<DCO_TYPE> Ta(n);
138     DCO_TYPE va;
139     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
140     for(size_t i=0;i<n;i++) {
141         ca[i]=c[i];
142         Ta[i]=T[i];
143         DCO_MODE::global_tape->register_variable(ca[i]);
144     }
145     f(ca,m,Ta,O,va);
146     cerr << "record (0," << m << ")=" << dco::size_of(DCO_MODE::global_tape) << "B"
147         << endl;
148     derivative(va)=1.0;
149     DCO_MODE::global_tape->register_output_variable(va);
150     DCO_MODE::global_tape->interpret_adjoint();
151     DCO_MODE::global_tape->write_to_dot();
152
153     for (size_t i=0;i<n;i++) dvdc[i]=derivative(ca[i]);
154     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
155     cout.precision(15);
156     for(size_t i=0;i<n;i++)
157         cout << "dvdc[" << i << "]= " << dvdc[i] << endl;
158     return 0;
159 }

```

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.

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <cassert>
4  using namespace std;
5  #include "dco.hpp"
6  using namespace dco;
7  typedef ga1s<double> DCO_MODE;
8  typedef DCO_MODE::type DCO_TYPE;
9  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
10
11 #include "../include/f.hpp"
12
13 template<typename DCO_MODE, typename TYPE>
14 void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b);
15
16 void EulerSteps(const size_t m, const vector<DCO_TYPE>& A, vector<DCO_TYPE>& T){
17     size_t n=T.size();
18     for (size_t j=0;j<m;j++) {
19         for (size_t i=0;i<n;++i) T[i]=-T[i];
20         if (DCO_MODE::global_tape->is_active())
21             Solve_make_gap<DCO_MODE>(A,T);
22         else
23             Solve(A,T);
24     }
25 }
26
27 template<typename DCO_MODE>
28 void EulerSteps_fill_gap(typename DCO_MODE::external_adjoint_object_t *D){
29     typedef typename DCO_MODE::type DCO_TYPE;
30     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
31     typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
32     typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
33
34     const DCO_TAPE_POSITION_TYPE p0 = DCO_MODE::global_tape->get_position();
35     const size_t& m=D->template read_data<size_t>();
36     const size_t& n = D->template read_data<size_t>();
37
38     vector<DCO_TYPE>* A_p=D->template read_data<vector<DCO_TYPE>*>();
39     vector<DCO_TYPE> T(n);
40     for (size_t i=0;i<n;i++) {
41         T[i]=D->template read_data<DCO_VALUE_TYPE>();
42         DCO_MODE::global_tape->register_variable(T[i]);
43     }
44
45     //save position of overwritten inputs
46     vector<DCO_TYPE> T_in(n);
47     for (size_t i=0;i<n;i++) T_in[i]=T[i];
48

```

```

49  //save this position
50  DCO_TAPE_POSITION_TYPE p1=DCO_MODE::global_tape->get_position();
51
52  //forward run
53  EulerSteps(m,*A_p,T);
54  cerr << "record =" << dco::sizeof(DCO_MODE::global_tape) << "B" << endl;
55
56  //get output adjoints (seeds for this section)
57  for (size_t i=0;i<n;i++) {
58      DCO_MODE::global_tape->register_output_variable(T[i]);
59      derivative(T[i])=D->get_output_adjoint();
60  }
61
62  //reverse run and reset to position p1 (so only reverse run of Solve)
63  DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p1);
64
65  //increment input adjoint
66  for (size_t i=0;i<n;i++) {
67      D->increment_input_adjoint(derivative(T_in[i]));
68  }
69
70  //reset to position p0
71  DCO_MODE::global_tape->reset_to(p0);
72 }
73 template<typename DCO_TYPE>
74 void EulerSteps_make_gap(const size_t m, const vector<DCO_TYPE>& A, vector<
    DCO_TYPE>& T){
75     typedef mode<DCO_TYPE> DCO_MODE;
76     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
77     size_t n=T.size();
78
79     // create call back object
80     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
81
82     D->write_data(m);
83     D->write_data(n);
84     D->write_data(&A);
85     for (size_t i=0;i<n;i++) {
86         D->register_input(T[i]);
87         D->write_data(value(T[i]));
88     }
89
90     // forward run
91     DCO_MODE::global_tape->switch_to_passive();
92     EulerSteps(m,A,T);
93     DCO_MODE::global_tape->switch_to_active();
94
95     // register output
96     for (size_t i=0;i<n;i++) T[i]=D->register_output(value(T[i]));
97
98     DCO_MODE::global_tape->insert_callback(EulerSteps_fill_gap<DCO_MODE>,D);
99 }
100

```

```

101
102 template<typename DCO_MODE>
103 void LUDecomp_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
104     (void) D;
105 }
106
107 template<typename DCO_MODE, typename TYPE>
108 void LUDecomp_make_gap(vector<TYPE>& A){
109     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
110
111     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
112
113     size_t n=sqrt(double(A.size()));
114     vector<double> Ap(n*n);
115     for(size_t i=0;i<n*n;i++) Ap[i]=value(A[i]);
116     LUDecomp(Ap);
117     for(size_t i=0;i<n;i++)
118         for(size_t j=0;j<n;j++)
119             value(A[i*n+j])=Ap[i*n+j];
120
121     DCO_MODE::global_tape->insert_callback(LUDecomp_fill_gap<DCO_MODE>,D);
122 }
123
124 //  $U^T y = b$ 
125 template <class TYPE>
126 inline void FSubstT(const vector<TYPE>& LU, vector<TYPE>& y) {
127     size_t n=y.size();
128     for (size_t i=0;i<n;i++){
129         for (size_t j=0;j<i;j++)
130             y[i]=y[i]-LU[j*n+i]*y[j];
131         y[i]=y[i]/LU[i*n+i];
132     }
133 }
134
135 //  $L^T x = y$ 
136 template <class TYPE>
137 inline void BSubstT(const vector<TYPE>& LU, vector<TYPE>& b) {
138     size_t n=b.size();
139     for (size_t k=n,i=n-1;k>0;k--,i--)
140         for (size_t j=n-1;j>i;j--)
141             b[i]=b[i]-LU[j*n+i]*b[j];
142 }
143
144 //  $LU^T x = y$ 
145 template <class TYPE>
146 inline void SolveT(const vector<TYPE>& LU, vector<TYPE>& b) {
147     FSubstT(LU,b);
148     BSubstT(LU,b);
149 }
150
151 template<typename DCO_MODE>
152 void Solve_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
153     typedef typename DCO_MODE::type DCO_TYPE;

```

```

154     vector<DCO_TYPE>* LU_p = D->template read_data<vector<DCO_TYPE>*>();
155     size_t n=sqrt(double(LU_p->size()));
156     vector<DCO_TYPE> T(n), a1T(n);
157
158     for (size_t i=0;i<n;i++) a1T[i]=D->get_output_adjoint();
159     DCO_MODE::global_tape->switch_to_passive();
160     SolveT(*LU_p,a1T);
161     for (size_t i=0;i<n;i++) T[i]=D->template read_data<double>();
162     for (size_t i=0;i<n;i++)
163         for (size_t j=0;j<n;j++)
164             D->increment_input_adjoint(value(-a1T[i]*T[j]));
165     for (size_t i=0;i<n;i++)
166         D->increment_input_adjoint(value(a1T[i]));
167     DCO_MODE::global_tape->switch_to_active();
168 }
169
170 template<typename DCO_MODE, typename TYPE>
171 void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b){
172     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
173
174     const size_t n=b.size();
175     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
176
177     D->write_data(&LU);
178     for (size_t i=0;i<n*n;i++) D->register_input(LU[i]);
179     for (size_t i=0;i<n;i++) D->register_input(b[i]);
180     DCO_MODE::global_tape->switch_to_passive();
181     Solve(LU,b);
182     DCO_MODE::global_tape->switch_to_active();
183     for(size_t i=0;i<n;i++) {
184         D->write_data(value(b[i]));
185         b[i]=D->register_output(value(b[i]));
186     }
187     DCO_MODE::global_tape->insert_callback(Solve_fill_gap<DCO_MODE>,D);
188 }
189
190 inline void sim(const vector<DCO_TYPE>& c, const size_t m, vector<DCO_TYPE>& T,
        const size_t cs) {
191     const size_t n=c.size();
192     static vector<DCO_TYPE> A(n*n);
193     residual_jacobian(c,T,A);
194     for (size_t i=0;i<n;++i) {
195         for (size_t j=0;j<n;++j)
196             A[i*n+j]=A[i*n+j]/m;
197         A[i+i*n]=A[i+i*n]-1;
198     }
199     LUDecomp_make_gap<DCO_MODE>(A);
200     for (size_t j=0;j<m;j+=cs) {
201         size_t s=(j+cs<m) ? cs : m-j;
202         EulerSteps_make_gap<DCO_TYPE>(s,A,T);
203     }
204 }
205

```

```

206 template <typename TYPE, typename OTYPE>
207 inline void f(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
    vector<OTYPE>& O, TYPE& v, const size_t cs) {
208     sim(c,m,T,cs);
209     v=0;
210     size_t n=T.size();
211     for (size_t i=0;i<n-1;++i)
212         v=v+(T[i]-O[i])*(T[i]-O[i]);
213     v=v/(n-1);
214 }
215
216 int main(int argc, char* argv[]){
217     if (argc!=4) {
218         cerr << "3 parameters expected:" << endl
219         << " 1. number of spatial finite difference grid points" << endl
220         << " 2. number of implicit Euler steps" << endl
221         << " 3. distance between checkpoints " << endl;
222         return -1;
223     }
224     size_t n=atoi(argv[1]), m=atoi(argv[2]), cs=atoi(argv[3]);
225     vector<double> c(n);
226     for (size_t i=0;i<n;i++) c[i]=0.01;
227     vector<double> T(n);
228     for (size_t i=0;i<n-1;i++) T[i]=300.;
229     T[n-1]=1700.;
230     ifstream ifs("0.txt");
231     vector<double> O(n);
232     for (size_t i=0;i<n;i++) ifs >> O[i] ;
233     vector<double> dvdc(n);
234
235     vector<DCO_TYPE> ca(n);
236     vector<DCO_TYPE> Ta(n);
237     DCO_TYPE va;
238     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
239     for(size_t i=0;i<n;i++) {
240         ca[i]=c[i];
241         Ta[i]=T[i];
242         DCO_MODE::global_tape->register_variable(ca[i]);
243     }
244     f(ca,m,Ta,O,va,cs);
245     cerr << "record (0," << m << ")=" << dco::size_of(DCO_MODE::global_tape) << "B
        " << endl;
246     derivative(va)=1.0;
247     DCO_MODE::global_tape->register_output_variable(va);
248     DCO_MODE::global_tape->interpret_adjoint();
249     for (size_t i=0;i<n;i++) dvdc[i]=derivative(ca[i]);
250     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
251     cout.precision(15);
252     for(size_t i=0;i<n;i++)
253         cout << "dvdc[" << i << "]= " << dvdc[i] << endl;
254     return 0;
255 }

```

B.5 Hessian

B.5.1 Finite Differences

This section shows the approximation of the Hessian by finite difference.

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <fstream>
4  #include <vector>
5  #include <cfloat>
6  using namespace std;
7
8  #include "../include/f.hpp"
9
10 int main(int argc, char* argv[]){
11     cout.precision(15);
12     if (argc!=3) {
13         cerr << "2 parameters expected:" << endl
14         << " 1. number of spatial finite difference grid points" << endl
15         << " 2. number of implicit Euler steps" << endl;
16         return -1;
17     }
18     size_t n=atoi(argv[1]), m=atoi(argv[2]);
19     vector<double> c(n);
20     for (size_t i=0;i<n;i++) c[i]=0.01;
21     vector<double> T(n);
22     for (size_t i=0;i<n-1;i++) T[i]=300.;
23     T[n-1]=1700.;
24     ifstream ifs("0.txt");
25     vector<double> O(n);
26     for (size_t i=0;i<n;i++) ifs >> O[i] ;
27     vector<double> cp(n,0),Tp(n,0);
28     vector<vector<double> > d2vdc2(n,vector<double>(n,0));
29     double vp1, vp2;
30     for (size_t i=0;i<n;i++) {
31         for (size_t j=0;j<n;j++) {
32             double h=(c[j]==0) ? sqrt(sqrt(DBL_EPSILON)) : sqrt(sqrt(DBL_EPSILON))*abs
33                 (c[j]);
34             for (size_t k=0;k<n;k++) { cp[k]=c[k]; Tp[k]=T[k]; }
35             cp[i]+=h; cp[j]+=h;
36             f(cp,m,Tp,0,vp2);
37             for (size_t k=0;k<n;k++) { cp[k]=c[k]; Tp[k]=T[k]; }
38             cp[i]-=h; cp[j]+=h;
39             f(cp,m,Tp,0,vp1);
40             vp2-=vp1;
41             for (size_t k=0;k<n;k++) { cp[k]=c[k]; Tp[k]=T[k]; }
42             cp[i]+=h; cp[j]-=h;
43             f(cp,m,Tp,0,vp1);
44             vp2-=vp1;
45             for (size_t k=0;k<n;k++) { cp[k]=c[k]; Tp[k]=T[k]; }
46             cp[i]-=h; cp[j]-=h;
47             f(cp,m,Tp,0,vp1);
48             vp2+=vp1;

```

```

48     d2vdc2[i][j]=vp2/(4*h*h);
49 }
50 }
51 cout.precision(15);
52 for (size_t i=0;i<n;i++)
53     for (size_t j=0;j<n;j++)
54         cout << "d2vdc2[" << i << "]" << j << "]=\"
55             << d2vdc2[i][j] << endl;
56 return 0;
57 }

```

B.5.2 `gt1s<gt1s<double>::type>`

This section shows the computation of the Hessian in second-order tangent mode.

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5  #include "dco.hpp"
6  using namespace dco;
7
8  typedef gt1s<gt1s<double>::type>::type DCO_TYPE;
9
10 #include "../include/f.hpp"
11
12 int main(int argc, char* argv[]){
13     if (argc!=3) {
14         cerr << "2 parameters expected:" << endl
15             << " 1. number of spatial finite difference grid points" << endl
16             << " 2. number of implicit Euler steps" << endl;
17         return -1;
18     }
19     size_t n=atoi(argv[1]), m=atoi(argv[2]);
20     vector<double> c(n);
21     for (size_t i=0;i<n;i++) c[i]=0.01;
22     vector<double> T(n);
23     for (size_t i=0;i<n-1;i++) T[i]=300.;
24     T[n-1]=1700.;
25     ifstream ifs("0.txt");
26     vector<double> O(n);
27     for (size_t i=0;i<n;i++) ifs >> O[i] ;
28     vector<vector<double> > d2vdc2(n,vector<double>(n,0));
29     vector<DCO_TYPE> ca(n);
30     vector<DCO_TYPE> Ta(n);
31     DCO_TYPE va;
32     for(size_t i=0;i<n;i++) {
33         for(size_t j=0;j<n;j++) {
34             for(size_t k=0;k<n;k++) { ca[k]=c[k]; Ta[k]=T[k]; }
35             value(derivative(ca[i]))=1.0; derivative(value(ca[j]))=1.0;
36             f(ca,m,Ta,0,va);
37             d2vdc2[i][j]=derivative(derivative(va));
38         }
39     }

```

```

39     }
40     cout.precision(15);
41     for(size_t i=0;i<n;i++)
42         for(size_t j=0;j<n;j++)
43             cout << "d2vdc[" << i << "][" << j << "]= " << d2vdc2[i][j] << endl;
44     return 0;
45 }

```

B.5.3 `gals<gtls<double>::type>`

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

```

1  #include <cstdlib>
2  #include <iostream>
3  using namespace std;
4  #include "dco.hpp"
5  using namespace dco;
6  typedef gals<gtls<double>::type> DCO_MODE;
7  typedef DCO_MODE::type DCO_TYPE;
8  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
9  typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
10
11 #include "../include/f.hpp"
12
13 int main(int argc, char* argv[]){
14     if (argc!=3) {
15         cerr << "2 parameters expected:" << endl
16         << " 1. number of spatial finite difference grid points" << endl
17         << " 2. number of implicit Euler steps" << endl;
18         return -1;
19     }
20     size_t n=atoi(argv[1]), m=atoi(argv[2]);
21     vector<double> c(n);
22     for (size_t i=0;i<n;i++) c[i]=0.01;
23     vector<double> T(n);
24     for (size_t i=0;i<n-1;i++) T[i]=300.;
25     T[n-1]=1700.;
26     ifstream ifs("0.txt");
27     vector<double> O(n);
28     for (size_t i=0;i<n;i++) ifs >> O[i] ;
29     vector<vector<double> > d2vdc2(n,vector<double>(n,0));
30
31     vector<DCO_TYPE> ca(n);
32     vector<DCO_TYPE> Ta(n);
33     DCO_TYPE va;
34     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
35
36     for(size_t i=0;i<n;i++) {
37         ca[i]=c[i];
38         Ta[i]=T[i];
39         DCO_MODE::global_tape->register_variable(ca[i]);
40     }
41     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();

```

```

42     for(size_t i=0;i<n;i++) {
43         for(size_t j=0;j<n;j++) Ta[j]=T[j];
44         if (i>0) DCO_MODE::global_tape->zero_adjoints();
45         derivative(value(ca[i]))=1;
46         f(ca,m,Ta,0,va);
47         value(derivative(va))=1;
48         DCO_MODE::global_tape->interpret_adjoint();
49         for(size_t j=0;j<n;j++) derivative(value(ca[j]))=0;
50         for(size_t j=0;j<n;j++) d2vdc2[j][i]=derivative(derivative(ca[j]));
51         DCO_MODE::global_tape->reset_to(p);
52     }
53     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
54     cout.precision(15);
55     for(size_t i=0;i<n;i++)
56         for(size_t j=0;j<n;j++)
57             cout << "d2vdc2[" << i << "]" << j << "]= " << d2vdc2[i][j] << endl;
58
59     return 0;
60 }

```

B.5.4 `ga1s<gt1s<double>::type>` + Equidistant Checkpointing

This section shows the computation of the Hessian in second-adjoint mode with equidistant checkpointing.

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <cassert>
4  using namespace std;
5  #include "dco.hpp"
6  using namespace dco;
7  typedef ga1s<gt1s<double>::type> DCO_MODE;
8  typedef DCO_MODE::type DCO_TYPE;
9  typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
10 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
11
12 #include "../include/f.hpp"
13
14 template<typename TYPE>
15 void EulerSteps(const size_t m, const vector<TYPE>& A, vector<TYPE>& T) {
16     size_t n=T.size();
17     for (size_t j=0;j<m;j++) {
18         for (size_t i=0;i<n;++i) T[i]=-T[i];
19         Solve(A,T);
20     }
21 }
22
23 template<typename DCO_MODE>
24 void EulerSteps_fill_gap(typename DCO_MODE::external_adjoint_object_t *D){
25     typedef typename DCO_MODE::type DCO_TYPE;
26     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
27     typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
28     typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;

```

```

29
30     const DCO_TAPE_POSITION_TYPE p0 = DCO_MODE::global_tape->get_position();
31     const size_t& m=D->template read_data<size_t>();
32     const size_t& n = D->template read_data<size_t>();
33     vector<DCO_TYPE>* A_p=D->template read_data<vector<DCO_TYPE>*>();
34     vector<DCO_TYPE> T(n);
35     for (size_t i=0;i<n;i++) {
36         T[i]=D->template read_data<DCO_VALUE_TYPE>();
37         DCO_MODE::global_tape->register_variable(T[i]);
38     }
39     vector<DCO_TYPE> T_in(n);
40     for (size_t i=0;i<n;i++) T_in[i]=T[i];
41     DCO_TAPE_POSITION_TYPE p1=DCO_MODE::global_tape->get_position();
42     EulerSteps(m,*A_p,T);
43     cerr << "record =" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
44     for (size_t i=0;i<n;i++) {
45         DCO_MODE::global_tape->register_output_variable(T[i]);
46         derivative(T[i])=D->get_output_adjoint();
47     }
48     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p1);
49     for (size_t i=0;i<n;i++)
50         D->increment_input_adjoint(derivative(T_in[i]));
51     DCO_MODE::global_tape->reset_to(p0);
52 }
53
54 template<typename DCO_TYPE>
55 void EulerSteps_make_gap(const size_t m, const vector<DCO_TYPE>& A, vector<
    DCO_TYPE>& T){
56     typedef mode<DCO_TYPE> DCO_MODE;
57     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
58     size_t n=T.size();
59
60     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
61     D->write_data(m);
62     D->write_data(n);
63     D->write_data(&A);
64     for (size_t i=0;i<n;i++) {
65         D->register_input(T[i]);
66         D->write_data(value(T[i]));
67     }
68     DCO_MODE::global_tape->switch_to_passive();
69     EulerSteps(m,A,T);
70     DCO_MODE::global_tape->switch_to_active();
71     for (size_t i=0;i<n;i++) T[i]=D->register_output(value(T[i]));
72     DCO_MODE::global_tape->insert_callback(EulerSteps_fill_gap<DCO_MODE>,D);
73 }
74
75 template <typename TYPE>
76 inline void sim(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
    size_t cs) {
77     size_t n=T.size();
78     static vector<TYPE> A(n*n); // static avoids copy
79

```

```

80     residual_jacobian(c,T,A);
81     for (size_t i=0;i<n;++i) {
82         for (size_t j=0;j<n;++j)
83             A[i*n+j]=A[i*n+j]/m;
84         A[i+i*n]=A[i+i*n]-1;
85     }
86     LUDecomp(A);
87     for (size_t j=0;j<m;j+=cs) {
88         size_t s=(j+cs<m) ? cs : m-j;
89         EulerSteps_make_gap<DCO_TYPE>(s,A,T);
90     }
91 }
92
93 template <typename TYPE, typename OTYPE>
94 inline void f(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
95     vector<OTYPE>& O, TYPE& v, const size_t cs) {
96     sim(c,m,T,cs);
97     v=0;
98     size_t n=T.size();
99     for (size_t i=0;i<n-1;++i)
100         v=v+(T[i]-O[i])*(T[i]-O[i]);
101     v=v/(n-1);
102 }
103
104 int main(int argc, char* argv[]){
105     if (argc!=4) {
106         cerr << "3 parameters expected:" << endl
107         << " 1. number of spatial finite difference grid points" << endl
108         << " 2. number of implicit Euler steps" << endl
109         << " 3. distance between checkpoints" << endl;
110         return -1;
111     }
112     size_t n=atoi(argv[1]), m=atoi(argv[2]), cs=atoi(argv[3]);
113     assert(cs<=m);
114     vector<double> c(n);
115     for (size_t i=0;i<n;i++) c[i]=0.01;
116     vector<double> T(n);
117     for (size_t i=0;i<n-1;i++) T[i]=300.;
118     T[n-1]=1700.;
119     ifstream ifs("0.txt");
120     vector<double> O(n);
121     for (size_t i=0;i<n;i++) ifs >> O[i] ;
122     vector<vector<double>> > d2vdc2(n,vector<double>(n,0));
123     vector<DCO_TYPE> ca(n);
124     vector<DCO_TYPE> Ta(n);
125     DCO_TYPE va;
126
127     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
128     for(size_t i=0;i<n;i++) {
129         ca[i]=c[i];
130         Ta[i]=T[i];
131         DCO_MODE::global_tape->register_variable(ca[i]);
132     }
133     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();

```

```

133     for(size_t i=0;i<n;i++) {
134         for(size_t j=0;j<n;j++) Ta[j]=T[j];
135         if (i>0) DCO_MODE::global_tape->zero_adjoints();
136         derivative(value(ca[i]))=1;
137         f(ca,m,Ta,0,va,cs);
138         value(derivative(va))=1;
139         DCO_MODE::global_tape->interpret_adjoint();
140         for(size_t j=0;j<n;j++) derivative(value(ca[j]))=0;
141         for(size_t j=0;j<n;j++) d2vdc2[j][i]=derivative(derivative(ca[j]));
142         DCO_MODE::global_tape->reset_to(p);
143     }
144     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
145     cout.precision(15);
146     for(size_t i=0;i<n;i++)
147         for(size_t j=0;j<n;j++)
148             cout << "d2vdc2[" << i << "]" << j << "]= " << d2vdc2[i][j] << endl;
149     return 0;
150 }

```

B.5.5 `gals<gtls< double >::type>` + Symbolic Derivative of Linear Solver

This section shows the computation of the Hessian in second-adjoint mode with symbolically differentiated linear solver.

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <cassert>
4  using namespace std;
5  #include "dco.hpp"
6  using namespace dco;
7  typedef gtls<double>::type DCO_BASE_TYPE;
8  typedef gals<DCO_BASE_TYPE> DCO_MODE;
9  typedef DCO_MODE::type DCO_TYPE;
10 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
11 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
12
13 #include "../include/f.hpp"
14
15 template<typename DCO_MODE>
16 void LUDecomp_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
17     (void) D;
18 }
19
20 template<typename DCO_MODE, typename TYPE>
21 void LUDecomp_make_gap(vector<TYPE>& A){
22     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
23
24     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
25
26     size_t n=sqrt(double(A.size()));
27     vector<DCO_BASE_TYPE> Ap(n*n);
28     for(size_t i=0;i<n*n;i++) Ap[i]=value(A[i]);

```

```

29     LUDecomp(Ap);
30     for(size_t i=0;i<n;i++)
31         for(size_t j=0;j<n;j++)
32             value(A[i*n+j])=Ap[i*n+j];
33
34     DCO_MODE::global_tape->insert_callback(LUDecomp_fill_gap<DCO_MODE>,D);
35 }
36
37 // U^T*y=b
38 template <class TYPE>
39 inline void FSubstT(const vector<TYPE>& LU, vector<TYPE>& y) {
40     size_t n=y.size();
41     for (size_t i=0;i<n;i++){
42         for (size_t j=0;j<i;j++)
43             y[i]=y[i]-LU[j*n+i]*y[j];
44         y[i]=y[i]/LU[i*n+i];
45     }
46 }
47
48 // L^T*x=y
49 template <class TYPE>
50 inline void BSubstT(const vector<TYPE>& LU, vector<TYPE>& b) {
51     size_t n=b.size();
52     for (size_t k=n,i=n-1;k>0;k--,i--)
53         for (size_t j=n-1;j>i;j--)
54             b[i]=b[i]-LU[j*n+i]*b[j];
55 }
56
57 // LU^T*x=y
58 template <class TYPE>
59 inline void SolveT(const vector<TYPE>& LU, vector<TYPE>& b) {
60     FSubstT(LU,b);
61     BSubstT(LU,b);
62 }
63
64 template<typename DCO_MODE>
65 void Solve_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
66     typedef typename DCO_MODE::type DCO_TYPE;
67     vector<DCO_TYPE>* LU_p = D->template read_data<vector<DCO_TYPE>*>();
68     size_t n=sqrt(double(LU_p->size()));
69     vector<DCO_TYPE> T(n), a1T(n);
70
71     for (size_t i=0;i<n;i++) a1T[i]=D->get_output_adjoint();
72     DCO_MODE::global_tape->switch_to_passive();
73     SolveT(*LU_p,a1T);
74     for (size_t i=0;i<n;i++) T[i]=D->template read_data<DCO_BASE_TYPE>();
75     for (size_t i=0;i<n;i++)
76         for (size_t j=0;j<n;j++)
77             D->increment_input_adjoint(value(-a1T[i]*T[j]));
78     for (size_t i=0;i<n;i++)
79         D->increment_input_adjoint(value(a1T[i]));
80     DCO_MODE::global_tape->switch_to_active();
81 }
82

```

```

83 template<typename DCO_MODE, typename TYPE>
84 void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b){
85     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
86
87     const size_t n=b.size();
88     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
89
90     D->write_data(&LU);
91     for (size_t i=0;i<n*n;i++) D->register_input(LU[i]);
92     for (size_t i=0;i<n;i++) D->register_input(b[i]);
93     DCO_MODE::global_tape->switch_to_passive();
94     Solve(LU,b);
95     DCO_MODE::global_tape->switch_to_active();
96     for(size_t i=0;i<n;i++) {
97         D->write_data(value(b[i]));
98         b[i]=D->register_output(value(b[i]));
99     }
100     DCO_MODE::global_tape->insert_callback(Solve_fill_gap<DCO_MODE>,D);
101 }
102
103 template <>
104 inline void sim(const vector<DCO_TYPE>& c, const size_t m, vector<DCO_TYPE>& T)
105 {
106     const size_t n=c.size();
107     static vector<DCO_TYPE> A(n*n);
108     residual_jacobian(c,T,A);
109     for (size_t i=0;i<n;++i) {
110         for (size_t j=0;j<n;++j)
111             A[i*n+j]=A[i*n+j]/m;
112         A[i+i*n]=A[i+i*n]-1;
113     }
114     LUDecomp_make_gap<DCO_MODE>(A);
115     for (size_t j=0;j<m;++j) {
116         for (size_t i=0;i<n;++i) T[i]=-T[i];
117         Solve_make_gap<DCO_MODE>(A,T);
118     }
119 }
120 int main(int argc, char* argv[]){
121     if (argc!=3) {
122         cerr << "2 parameters expected:" << endl
123         << " 1. number of spatial finite difference grid points" << endl
124         << " 2. number of implicit Euler steps" << endl;
125         return -1;
126     }
127     size_t n=atoi(argv[1]), m=atoi(argv[2]);
128     vector<double> c(n);
129     for (size_t i=0;i<n;i++) c[i]=0.01;
130     vector<double> T(n);
131     for (size_t i=0;i<n-1;i++) T[i]=300.;
132     T[n-1]=1700.;
133     ifstream ifs("0.txt");
134     vector<double> O(n);

```

```

135     for (size_t i=0;i<n;i++) ifs >> O[i] ;
136     vector<vector<double> > d2vdc2(n,vector<double>(n,0));
137
138     vector<DCO_TYPE> ca(n);
139     vector<DCO_TYPE> Ta(n);
140     DCO_TYPE va;
141     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
142
143     for(size_t i=0;i<n;i++) {
144         ca[i]=c[i];
145         Ta[i]=T[i];
146         DCO_MODE::global_tape->register_variable(ca[i]);
147     }
148     DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
149     for(size_t i=0;i<n;i++) {
150         for(size_t j=0;j<n;j++) Ta[j]=T[j];
151         if (i>0) DCO_MODE::global_tape->zero_adjoints();
152         derivative(value(ca[i]))=1;
153         f(ca,m,Ta,0,va);
154         value(derivative(va))=1;
155         DCO_MODE::global_tape->interpret_adjoint();
156         for(size_t j=0;j<n;j++) derivative(value(ca[j]))=0;
157         for(size_t j=0;j<n;j++) d2vdc2[j][i]=derivative(derivative(ca[j]));
158         DCO_MODE::global_tape->reset_to(p);
159     }
160     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
161     cout.precision(15);
162     for(size_t i=0;i<n;i++)
163         for(size_t j=0;j<n;j++)
164             cout << "d2vdc2[" << i << "]" << j << "]= " << d2vdc2[i][j] << endl;
165
166     return 0;
167 }

```

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.

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <cassert>
4  using namespace std;
5  #include "dco.hpp"
6  using namespace dco;
7  typedef gt1s<double>::type DCO_BASE_TYPE;
8  typedef ga1s<DCO_BASE_TYPE> DCO_MODE;
9  typedef DCO_MODE::type DCO_TYPE;
10 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
11 typedef DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
12
13 #include "../include/f.hpp"

```

```

14
15 template<typename DCO_MODE, typename TYPE>
16 void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b);
17
18 void EulerSteps(const size_t m, const vector<DCO_TYPE>& A, vector<DCO_TYPE>& T){
19     size_t n=T.size();
20     for (size_t j=0;j<m;j++) {
21         for (size_t i=0;i<n;++i) T[i]=-T[i];
22         if (DCO_MODE::global_tape->is_active())
23             Solve_make_gap<DCO_MODE>(A,T);
24         else
25             Solve(A,T);
26     }
27 }
28
29 template<typename DCO_MODE>
30 void EulerSteps_fill_gap(typename DCO_MODE::external_adjoint_object_t *D){
31     typedef typename DCO_MODE::type DCO_TYPE;
32     typedef typename DCO_MODE::value_t DCO_VALUE_TYPE;
33     typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
34     typedef typename DCO_TAPE_TYPE::iterator_t DCO_TAPE_POSITION_TYPE;
35
36     const DCO_TAPE_POSITION_TYPE p0 = DCO_MODE::global_tape->get_position();
37     const size_t& m=D->template read_data<size_t>();
38     const size_t& n = D->template read_data<size_t>();
39
40     vector<DCO_TYPE>* A_p=D->template read_data<vector<DCO_TYPE>*>();
41     vector<DCO_TYPE> T(n);
42     for (size_t i=0;i<n;i++) {
43         T[i]=D->template read_data<DCO_VALUE_TYPE>();
44         DCO_MODE::global_tape->register_variable(T[i]);
45     }
46
47     //save position of overwritten inputs
48     vector<DCO_TYPE> T_in(n);
49     for (size_t i=0;i<n;i++) T_in[i]=T[i];
50
51     //save this position
52     DCO_TAPE_POSITION_TYPE p1=DCO_MODE::global_tape->get_position();
53
54     //forward run
55     EulerSteps(m,*A_p,T);
56     cerr << "record =" << dco::size_of(DCO_MODE::global_tape) << "B" << endl;
57
58     //get output adjoints (seeds for this section)
59     for (size_t i=0;i<n;i++) {
60         DCO_MODE::global_tape->register_output_variable(T[i]);
61         derivative(T[i])=D->get_output_adjoint();
62     }
63
64     //reverse run and reset to position p1 (so only reverse run of Solve)
65     DCO_MODE::global_tape->interpret_adjoint_and_reset_to(p1);
66
67     //increment input adjoint

```

```

68     for (size_t i=0;i<n;i++) {
69         D->increment_input_adjoint(derivative(T_in[i]));
70     }
71
72     //reset to position p0
73     DCO_MODE::global_tape->reset_to(p0);
74 }
75 template<typename DCO_TYPE>
76 void EulerSteps_make_gap(const size_t m, const vector<DCO_TYPE>& A, vector<
    DCO_TYPE>& T){
77     typedef mode<DCO_TYPE> DCO_MODE;
78     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
79     size_t n=T.size();
80
81     // create call back object
82     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
83
84     D->write_data(m);
85     D->write_data(n);
86     D->write_data(&A);
87     for (size_t i=0;i<n;i++) {
88         D->register_input(T[i]);
89         D->write_data(value(T[i]));
90     }
91
92     // forward run
93     DCO_MODE::global_tape->switch_to_passive();
94     EulerSteps(m,A,T);
95     DCO_MODE::global_tape->switch_to_active();
96
97     // register output
98     for (size_t i=0;i<n;i++) T[i]=D->register_output(value(T[i]));
99
100    DCO_MODE::global_tape->insert_callback(EulerSteps_fill_gap<DCO_MODE>,D);
101 }
102
103
104 template<typename DCO_MODE>
105 void LUDecomp_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
106     (void) D;
107 }
108
109 template<typename DCO_MODE, typename TYPE>
110 void LUDecomp_make_gap(vector<TYPE>& A){
111     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
112
113     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
114
115     size_t n=sqrt(double(A.size()));
116     vector<DCO_BASE_TYPE> Ap(n*n);
117     for(size_t i=0;i<n*n;i++) Ap[i]=value(A[i]);
118     LUDecomp(Ap);

```

```

119     for(size_t i=0;i<n;i++)
120         for(size_t j=0;j<n;j++)
121             value(A[i*n+j])=Ap[i*n+j];
122
123     DCO_MODE::global_tape->insert_callback(LUDecomp_fill_gap<DCO_MODE>,D);
124 }
125
126 // U^T*y=b
127 template <class TYPE>
128 inline void FSubstT(const vector<TYPE>& LU, vector<TYPE>& y) {
129     size_t n=y.size();
130     for (size_t i=0;i<n;i++){
131         for (size_t j=0;j<i;j++){
132             y[i]=y[i]-LU[j*n+i]*y[j];
133             y[i]=y[i]/LU[i*n+i];
134         }
135     }
136
137 // L^T*x=y
138 template <class TYPE>
139 inline void BSubstT(const vector<TYPE>& LU, vector<TYPE>& b) {
140     size_t n=b.size();
141     for (size_t k=n,i=n-1;k>0;k--,i--){
142         for (size_t j=n-1;j>i;j--){
143             b[i]=b[i]-LU[j*n+i]*b[j];
144         }
145     }
146
147 // LU^T*x=y
148 template <class TYPE>
149 inline void SolveT(const vector<TYPE>& LU, vector<TYPE>& b) {
150     FSubstT(LU,b);
151     BSubstT(LU,b);
152 }
153
154 template<typename DCO_MODE>
155 void Solve_fill_gap(typename DCO_MODE::external_adjoint_object_t *D) {
156     typedef typename DCO_MODE::type DCO_TYPE;
157     vector<DCO_TYPE>* LU_p = D->template read_data<vector<DCO_TYPE>*>();
158     size_t n=sqrt(double(LU_p->size()));
159     vector<DCO_TYPE> T(n), a1T(n);
160
161     for (size_t i=0;i<n;i++) a1T[i]=D->get_output_adjoint();
162     DCO_MODE::global_tape->switch_to_passive();
163     SolveT(*LU_p,a1T);
164     for (size_t i=0;i<n;i++) T[i]=D->template read_data<DCO_BASE_TYPE>();
165     for (size_t i=0;i<n;i++)
166         for (size_t j=0;j<n;j++)
167             D->increment_input_adjoint(value(-a1T[i]*T[j]));
168     for (size_t i=0;i<n;i++)
169         D->increment_input_adjoint(value(a1T[i]));
170     DCO_MODE::global_tape->switch_to_active();
171 }
172
173 template<typename DCO_MODE, typename TYPE>

```

```

173 void Solve_make_gap(const vector<TYPE>& LU, vector<TYPE>& b){
174     typedef typename DCO_MODE::external_adjoint_object_t DCO_EAO_TYPE;
175
176     const size_t n=b.size();
177     DCO_EAO_TYPE *D = DCO_MODE::global_tape->template create_callback_object<
        DCO_EAO_TYPE>();
178
179     D->write_data(&LU);
180     for (size_t i=0;i<n*n;i++) D->register_input(LU[i]);
181     for (size_t i=0;i<n;i++) D->register_input(b[i]);
182     DCO_MODE::global_tape->switch_to_passive();
183     Solve(LU,b);
184     DCO_MODE::global_tape->switch_to_active();
185     for(size_t i=0;i<n;i++) {
186         D->write_data(value(b[i]));
187         b[i]=D->register_output(value(b[i]));
188     }
189     DCO_MODE::global_tape->insert_callback(Solve_fill_gap<DCO_MODE>,D);
190 }
191
192 inline void sim(const vector<DCO_TYPE>& c, const size_t m, vector<DCO_TYPE>& T,
        const size_t cs) {
193     const size_t n=c.size();
194     static vector<DCO_TYPE> A(n*n);
195     residual_jacobian(c,T,A);
196     for (size_t i=0;i<n;++i) {
197         for (size_t j=0;j<n;++j)
198             A[i*n+j]=A[i*n+j]/m;
199         A[i+i*n]=A[i+i*n]-1;
200     }
201     LUDecomp_make_gap<DCO_MODE>(A);
202     for (size_t j=0;j<m;j+=cs) {
203         size_t s=(j+cs<m) ? cs : m-j;
204         EulerSteps_make_gap<DCO_TYPE>(s,A,T);
205     }
206 }
207
208 template <typename TYPE, typename OTYPE>
209 inline void f(const vector<TYPE>& c, const size_t m, vector<TYPE>& T, const
        vector<OTYPE>& O, TYPE& v, const size_t cs) {
210     sim(c,m,T,cs);
211     v=0;
212     size_t n=T.size();
213     for (size_t i=0;i<n-1;++i)
214         v=v+(T[i]-O[i])*(T[i]-O[i]);
215     v=v/(n-1);
216 }
217
218 int main(int argc, char* argv[]){
219     if (argc!=4) {
220         cerr << "3 parameters expected:" << endl
221         << " 1. number of spatial finite difference grid points" << endl
222         << " 2. number of implicit Euler steps" << endl
223         << " 3. distance between checkpoints" << endl;

```

```

224     return -1;
225 }
226 size_t n=atoi(argv[1]), m=atoi(argv[2]), cs=atoi(argv[3]);
227 assert(cs<=m);
228 vector<double> c(n);
229 for (size_t i=0;i<n;i++) c[i]=0.01;
230 vector<double> T(n);
231 for (size_t i=0;i<n-1;i++) T[i]=300.;
232 T[n-1]=1700.;
233 ifstream ifs("0.txt");
234 vector<double> O(n);
235 for (size_t i=0;i<n;i++) ifs >> O[i] ;
236 vector<vector<double> > d2vdc2(n,vector<double>(n,0));
237 vector<DCO_TYPE> ca(n);
238 vector<DCO_TYPE> Ta(n);
239 DCO_TYPE va;
240
241 DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
242 for(size_t i=0;i<n;i++) {
243     ca[i]=c[i];
244     Ta[i]=T[i];
245     DCO_MODE::global_tape->register_variable(ca[i]);
246 }
247 DCO_TAPE_POSITION_TYPE p=DCO_MODE::global_tape->get_position();
248 for(size_t i=0;i<n;i++) {
249     for(size_t j=0;j<n;j++) Ta[j]=T[j];
250     if (i>0) DCO_MODE::global_tape->zero_adjoints();
251     derivative(value(ca[i]))=1;
252     f(ca,m,Ta,0,va,cs);
253     value(derivative(va))=1;
254     DCO_MODE::global_tape->interpret_adjoint();
255     for(size_t j=0;j<n;j++) derivative(value(ca[j]))=0;
256     for(size_t j=0;j<n;j++) d2vdc2[j][i]=derivative(derivative(ca[j]));
257     DCO_MODE::global_tape->reset_to(p);
258 }
259 DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
260 cout.precision(15);
261 for(size_t i=0;i<n;i++)
262     for(size_t j=0;j<n;j++)
263         cout << "d2vdc2[" << i << "][" << j << "]= " << d2vdc2[i][j] << endl;
264 return 0;
265 }

```

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:

```
1 template<class T>
2 void f(const vector<T> x, T& y) {
3     y=0;
4     for (size_t i=0;i<x.size();i++) y=y+x[i]*x[i];
5     y=y*y;
6 }
```

We compare various methods for computing the gradient and the Hessian of f .

C.2 Gradient

n	cf _d	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

```

1  #include<iostream>
2  #include<cmath>
3  #include<cassert>
4  #include<cstdlib>
5  #include<vector>
6  using namespace std;
7
8
9  #include "../x22.hpp"
10
11 template<typename T>
12 void cfd_driver(const vector<T> &x, T &y, vector<T> &g) {
13     size_t n=x.size();
14     vector<T> x_ph(n), x_mh(n);
15     T y_ph, y_mh;
16     f(x,y);
17     for (size_t i=0;i<n;i++) {
18         for (size_t j=0;j<n;j++) x_ph[j]=x_mh[j]=x[j];
19         T h=(x[i]==0) ? sqrt(DBL_EPSILON) : sqrt(DBL_EPSILON)*abs(x[i]);
20         x_ph[i]+=h;
21         f(x_ph,y_ph);
22         x_mh[i]-=h;
23         f(x_mh,y_mh);
24         g[i]=(y_ph-y_mh)/(2*h);
25     }
26 }
27
28 int main(int c, char* v[]) {
29     assert(c==2); (void)c;
30     cout.precision(15);
31     size_t n=atoi(v[1]);
32     vector<double> x(n), g(n); double y;
33     for (size_t i=0;i<n;i++) x[i]=cos(double(i));
34     cfd_driver(x,y,g);
35     cout << y << endl;
36     for (size_t i=0;i<n;i++)
37         cout << g[i] << endl;
38     return 0;
39 }

```

C.2.2 First-order Tangent Mode

```

1  #include<vector>
2  #include<iostream>
3  using namespace std;
4  #include "dco.hpp"
5  using namespace dco;
6  #include "../x22.hpp"
7
8  template<typename T>
9  void gt1s_driver(const vector<T>& xv, T& yv, vector<T>& g) {
10     typedef typename gt1s<T>::type DCO_TYPE;
11     size_t n=xv.size();
12     vector<DCO_TYPE> x(n); DCO_TYPE y;
13     for (size_t i=0;i<n;i++) {
14         for (size_t j=0;j<n;j++) x[j]=xv[j];
15         derivative(x[i])=1.;
16         f(x,y);
17         g[i]=derivative(y);
18     }
19     yv=value(y);
20 }
21
22 int main(int c, char* v[]) {
23     assert(c==2); (void)c;
24     cout.precision(15);
25     size_t n=atoi(v[1]);
26     vector<double> x(n), g(n); double y;
27     for (size_t i=0;i<n;i++) x[i]=cos(double(i));
28     gt1s_driver(x,y,g);
29     cout << y << endl;
30     for (size_t i=0;i<n;i++)
31         cout << g[i] << endl;
32     return 0;
33 }

```

C.2.3 First-order Tangent Mode: Vector Mode

```

1  #include<vector>
2  #include<iostream>
3  using namespace std;
4  #include "dco.hpp"
5  using namespace dco;
6  #include "../x22.hpp"
7
8  template<typename T>
9  void gt1v_driver(const vector<T>& xv, T& yv, vector<T>& g) {
10     size_t n=xv.size(); const size_t l=100; assert(!(n%l));
11     typedef typename gt1v<T,l>::type DCO_TYPE;
12     vector<DCO_TYPE> x(n); DCO_TYPE y;
13     for (size_t j=0;j<n/l;j++) {
14         for (size_t i=0;i<n;i++) x[i]=xv[i];
15         for (size_t i=0;i<l;i++) derivative(x[j*l+i])[i]=1.;
16         f(x,y);
17         for (size_t i=0;i<l;i++) g[j*l+i]=derivative(y)[i];
18     }

```

```

19     yv=value(y);
20 }
21
22 int main(int c, char* v[]) {
23     assert(c==2); (void)c;
24     cout.precision(15);
25     size_t n=atoi(v[1]);
26     vector<double> x(n), g(n); double y;
27     for (size_t i=0;i<n;i++) x[i]=cos(double(i));
28     gtlv_driver(x,y,g);
29     cout << y << endl;
30     for (size_t i=0;i<n;i++)
31         cout << g[i] << endl;
32     return 0;
33 }

```

C.2.4 First-order Adjoint Mode

```

1  #include<vector>
2  #include<iostream>
3  using namespace std;
4  #include "dco.hpp"
5  using namespace dco;
6  #include "../x22.hpp"
7
8  template<typename T>
9  void gals_driver(const vector<T>& xv, T& yv, vector<T>& g) {
10     typedef gals<T> DCO_MODE;
11     typedef typename DCO_MODE::type DCO_TYPE;
12     typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
13     size_t n=xv.size();
14     vector<DCO_TYPE> x(n); DCO_TYPE y;
15
16     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
17     for (size_t j=0;j<n;j++) {
18         DCO_MODE::global_tape->register_variable(x[j]);
19         value(x[j])=xv[j];
20     }
21     f(x,y);
22     DCO_MODE::global_tape->register_output_variable(y);
23     yv=value(y);
24     derivative(y)=1;
25     DCO_MODE::global_tape->interpret_adjoint();
26     for (size_t j=0;j<n;j++) g[j]=derivative(x[j]);
27     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
28 }
29
30 int main(int c, char* v[]) {
31     assert(c==2); (void)c;
32     cout.precision(15);
33     size_t n=atoi(v[1]);
34     vector<double> x(n), g(n); double y;
35     for (size_t i=0;i<n;i++) x[i]=cos(double(i));
36     gals_driver(x,y,g);

```

```
37     cout << y << endl;
38     for (size_t i=0;i<n;i++)
39         cout << g[i] << endl;
40     return 0;
41 }
```

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

```

1  #include<iostream>
2  #include<cfloat>
3  #include<cmath>
4  #include<cassert>
5  #include<cstdlib>
6  #include<vector>
7  using namespace std;
8
9  #include "../x22.hpp"
10
11 template<typename T>
12 void cfd_driver(const vector<T> &x, T &y, vector<T> &g) {
13     size_t n=x.size();
14     vector<T> x_ph(n), x_mh(n);
15     T y_ph, y_mh;
16     f(x,y);
17     for (size_t i=0;i<n;i++) {
18         for (size_t j=0;j<n;j++) x_ph[j]=x_mh[j]=x[j];
19         T h=(x[i]==0) ? sqrt(sqrt(DBL_EPSILON)) : sqrt(sqrt(DBL_EPSILON))*abs(x[i]);
20         x_ph[i]+=h;
21         f(x_ph,y_ph);
22         x_mh[i]-=h;
23         f(x_mh,y_mh);
24         g[i]=(y_ph-y_mh)/(2*h);
25     }
26 }
27
28 template<typename T>
29 void socfd_driver(const vector<T> &x, T &y, vector<T> &g, vector<vector<T> >& H)
30 {
31     size_t n=x.size();
32     vector<T> x_ph(n), x_mh(n), g_ph(n), g_mh(n);
33     T y_ph, y_mh;
34     cfd_driver(x,y,g);
35     for (size_t i=0;i<n;i++) {
36         for (size_t j=0;j<n;j++) x_ph[j]=x_mh[j]=x[j];
37         T h=(x[i]==0) ? sqrt(sqrt(DBL_EPSILON)) : sqrt(sqrt(DBL_EPSILON))*abs(x[i]);
38         x_ph[i]+=h;
39         f(x_ph,y_ph);
40         cfd_driver(x_ph,y_ph,g_ph);
41         x_mh[i]-=h;

```

```

41     cfd_driver(x_mh,y_mh,g_mh);
42     for (size_t j=0;j<n;j++) H[i][j]=(g_ph[j]-g_mh[j])/(2*h);
43 }
44 }
45
46 int main(int c, char* v[]) {
47     assert(c==2); (void)c;
48     cout.precision(15);
49     size_t n=atoi(v[1]);
50     vector<vector<double> > H(n, vector<double>(n));
51     vector<double> x(n), g(n); double y;
52     for (size_t i=0;i<n;i++) x[i]=cos(double(i));
53     socfd_driver(x,y,g,H);
54     cout << y << endl;
55     for (size_t i=0;i<n;i++)
56         cout << g[i] << endl;
57     for (size_t i=0;i<n;i++)
58         for (size_t j=0;j<n;j++)
59             cout << H[i][j] << endl;
60     return 0;
61 }

```

C.3.2 Second-order Tangent Mode

```

1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  #include "dco.hpp"
6  using namespace dco;
7
8  typedef gt1s<double> DCO_BASE_MODE;
9  typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
10 typedef gt1s<DCO_BASE_TYPE> DCO_MODE;
11 typedef DCO_MODE::type DCO_TYPE;
12
13 #include "../x22.hpp"
14
15 template<typename T>
16 void driver(const vector<T>& xv, T& yv, vector<T>& g, vector<vector<T> >& h) {
17     size_t n=xv.size();
18     vector<DCO_TYPE> x(n); DCO_TYPE y;
19     for (size_t i=0;i<n;i++) {
20         for (size_t j=0;j<n;j++) {
21             for (size_t k=0;k<n;k++) x[k]=xv[k];
22             derivative(value(x[i]))=1;
23             value(derivative(x[j]))=1;
24             f(x,y);
25             h[i][j]=derivative(derivative(y));
26         }
27         g[i]=derivative(value(y));
28     }
29     yv=passive_value(y);
30 }

```

```

31
32 int main(int c, char* v[]) {
33     assert(c==2); (void)c;
34     cout.precision(15);
35     size_t n=atoi(v[1]);
36     vector<vector<double> > H(n, vector<double>(n));
37     vector<double> x(n), g(n); double y;
38     for (size_t i=0;i<n;i++) x[i]=cos(double(i));
39     driver(x,y,g,H);
40     cout << y << endl;
41     for (size_t i=0;i<n;i++)
42         cout << g[i] << endl;
43     for (size_t i=0;i<n;i++)
44         for (size_t j=0;j<n;j++)
45             cout << H[i][j] << endl;
46     return 0;
47 }

```

C.3.3 Second-order Adjoint Mode (Tangent over Adjoint)

```

1  #include<vector>
2  #include<iostream>
3  using namespace std;
4  #include "dco.hpp"
5  using namespace dco;
6  #include "../x22.hpp"
7
8  template<typename T>
9  void driver(const vector<T>& xv, T& yv, vector<T>& g, vector<vector<T> >& H) {
10     typedef typename gtls<T>::type DCO_BASE_TYPE;
11     typedef gals<DCO_BASE_TYPE> DCO_MODE;
12     typedef typename DCO_MODE::type DCO_TYPE;
13     typedef typename DCO_MODE::tape_t DCO_TAPE_TYPE;
14     size_t n=xv.size();
15     vector<DCO_TYPE> x(n); DCO_TYPE y;
16
17     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
18     for (size_t i=0;i<n;i++) {
19         for (size_t j=0;j<n;j++) {
20             DCO_MODE::global_tape->register_variable(x[j]);
21             passive_value(x[j])=xv[j];
22             derivative(value(x[j]))=0;
23         }
24         derivative(value(x[i]))=1;
25         f(x,y);
26         DCO_MODE::global_tape->register_output_variable(y);
27         yv=passive_value(y);
28         g[i]=derivative(value(y));
29         value(derivative(y))=1;
30         DCO_MODE::global_tape->interpret_adjoint();
31         for (size_t j=0;j<n;j++) H[i][j]=derivative(derivative(x[j]));
32         DCO_MODE::global_tape->reset();
33     }
34     DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);

```

```

35 }
36
37 int main(int c, char* v[]) {
38     assert(c==2); (void)c;
39     cout.precision(15);
40     size_t n=atoi(v[1]);
41     vector<vector<double> > H(n, vector<double>(n));
42     vector<double> x(n), g(n); double y;
43     for (size_t i=0;i<n;i++) x[i]=cos(double(i));
44     driver(x,y,g,H);
45     cout << y << endl;
46     for (size_t i=0;i<n;i++)
47         cout << g[i] << endl;
48     for (size_t i=0;i<n;i++)
49         for (size_t j=0;j<n;j++)
50             cout << H[i][j] << endl;
51     return 0;
52 }

```

C.3.4 Second-order Adjoint Mode (Adjoint over Tangent)

```

1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  #include "dco.hpp"
6  using namespace dco;
7
8  typedef ga1s<double> DCO_BASE_MODE;
9  typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
10 typedef gt1s<DCO_BASE_TYPE> DCO_MODE;
11 typedef DCO_MODE::type DCO_TYPE;
12 typedef DCO_BASE_MODE::tape_t DCO_TAPE_TYPE;
13
14 #include "../x22.hpp"
15
16 template<typename T>
17 void driver(const vector<T>& xv, T& yv, vector<T>& g, vector<vector<T> >& h) {
18     DCO_BASE_MODE::global_tape=DCO_TAPE_TYPE::create();
19     size_t n=xv.size();
20     vector<DCO_TYPE> x(n),x_in(n);
21     DCO_TYPE y;
22     for (size_t i=0;i<n;i++) {
23         for (size_t j=0;j<n;j++) {
24             x[j]=xv[j];
25             DCO_BASE_MODE::global_tape->register_variable(value(x[j]));
26             DCO_BASE_MODE::global_tape->register_variable(derivative(x[j]));
27             x_in[j]=x[j];
28         }
29         value(derivative(x[i]))=1;
30         f(x,y);
31         DCO_BASE_MODE::global_tape->register_output_variable(value(y));
32         DCO_BASE_MODE::global_tape->register_output_variable(derivative(y));
33         derivative(derivative(y))=1;

```

```

34     DCO_BASE_MODE::global_tape->interpret_adjoint();
35     for (size_t j=0;j<n;j++)
36         h[j][i] = derivative(value(x_in[j]));
37     DCO_BASE_MODE::global_tape->reset();
38     g[i]=value(derivative(y));
39 }
40 yv=passive_value(y);
41 DCO_TAPE_TYPE::remove(DCO_BASE_MODE::global_tape);
42 }
43
44 int main(int c, char* v[]) {
45     assert(c==2); (void)c;
46     cout.precision(15);
47     size_t n=atoi(v[1]);
48     vector<vector<double>> > H(n, vector<double>(n));
49     vector<double> x(n), g(n); double y;
50     for (size_t i=0;i<n;i++) x[i]=cos(double(i));
51     driver(x,y,g,H);
52     cout << y << endl;
53     for (size_t i=0;i<n;i++)
54         cout << g[i] << endl;
55     for (size_t i=0;i<n;i++)
56         for (size_t j=0;j<n;j++)
57             cout << H[i][j] << endl;
58     return 0;
59 }

```

C.3.5 Second-order Adjoint Mode (Adjoint over Adjoint)

```

1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  #include "dco.hpp"
6  using namespace dco;
7
8  typedef gals<double> DCO_BASE_MODE;
9  typedef DCO_BASE_MODE::type DCO_BASE_TYPE;
10 typedef DCO_BASE_MODE::tape_t DCO_BASE_TAPE_TYPE;
11 typedef gals<DCO_BASE_TYPE> DCO_MODE;
12 typedef DCO_MODE::type DCO_TYPE;
13 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
14
15 #include "../x22.hpp"
16
17 template<typename T>
18 void driver(const vector<T>& xv, T& yv, vector<T>& g, vector<vector<T>>& h) {
19     size_t n=xv.size();
20     vector<DCO_TYPE> x(n),x_in(n);
21     DCO_TYPE y;
22     DCO_BASE_MODE::global_tape=DCO_BASE_TAPE_TYPE::create();
23     DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
24     for (size_t j=0;j<n;j++) {
25         x[j]=xv[j];

```

```

26     DCO_BASE_MODE::global_tape->register_variable(value(x[j]));
27     DCO_MODE::global_tape->register_variable(x[j]);
28     x_in[j]=x[j];
29 }
30 f(x,y);
31 derivative(y)=1.0;
32 DCO_BASE_MODE::global_tape->register_variable(derivative(y));
33 DCO_MODE::global_tape->interpret_adjoint();
34 for (size_t j=0;j<n;j++)
35     g[j]=value(derivative(x_in[j]));
36 for (size_t i=0;i<n;i++) {
37     derivative(derivative(x_in[i]))=1;
38     DCO_BASE_MODE::global_tape->interpret_adjoint();
39     for (size_t j=0;j<n;j++)
40         h[i][j]=derivative(value(x_in[j]));
41     DCO_BASE_MODE::global_tape->zero_adjoints();
42 }
43 yv=passive_value(y);
44 DCO_BASE_TAPE_TYPE::remove(DCO_BASE_MODE::global_tape);
45 DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);
46 }
47
48 int main(int c, char* v[]) {
49     assert(c==2); (void)c;
50     cout.precision(15);
51     size_t n=atoi(v[1]);
52     vector<vector<double>> > H(n, vector<double>(n));
53     vector<double> x(n), g(n); double y;
54     for (size_t i=0;i<n;i++) x[i]=cos(double(i));
55     driver(x,y,g,H);
56     cout << y << endl;
57     for (size_t i=0;i<n;i++)
58         cout << g[i] << endl;
59     for (size_t i=0;i<n;i++)
60         for (size_t j=0;j<n;j++)
61             cout << H[i][j] << endl;
62     return 0;
63 }

```

Index

DCO_ADJOINT_TYPE, 151
DCO_BASE_TYPE, 29
DCO_EAO_TYPE::get_output_adjoint, 85, 155
DCO_EAO_TYPE::increment_input_adjoint, 85, 155
DCO_EAO_TYPE::read_data, 85, 155
DCO_EAO_TYPE::register_input, 85, 155
DCO_EAO_TYPE::register_output, 85, 155
DCO_EAO_TYPE::write_data, 85, 155
DCO_LOCAL_GRADIENT_TYPE::finalize, 124
DCO_LOCAL_GRADIENT_TYPE::put, 124
DCO_MODE::derivative_t, 152
DCO_MODE::external_adjoint_object_t, 85, 154
DCO_MODE::global_tape, 33, 152
DCO_MODE::jacobian_preaccumulator_t::finish, 155
DCO_MODE::jacobian_preaccumulator_t::register_output, 155
DCO_MODE::jacobian_preaccumulator_t::start, 155
DCO_MODE::jacobian_preaccumulator_t, 155
DCO_MODE::local_gradient_t, 124
DCO_MODE::value_t, 84, 152
DCO_MODE, 30, 150–152
DCO_PARTIAL_TYPE, 151
DCO_TAPE_TYPE::create_callback_object, 85, 155
DCO_TAPE_TYPE::create_local_gradient_object, 124
DCO_TAPE_TYPE::create, 33, 43, 74, 152
DCO_TAPE_TYPE::get_position, 84
DCO_TAPE_TYPE::insert_callback, 85, 155
DCO_TAPE_TYPE::interpret_adjoint_and_reset_to, 84
DCO_TAPE_TYPE::interpret_adjoint, 33, 153
DCO_TAPE_TYPE::iterator_t, 84
DCO_TAPE_TYPE::register_output_variable, 33, 153
DCO_TAPE_TYPE::register_variable, 33, 153
DCO_TAPE_TYPE::remove, 33, 74, 153
DCO_TAPE_TYPE::reset_to, 85
DCO_TAPE_TYPE, 33, 152
DCO_TYPE, 30, 150, 151
DCO_VALUE_TYPE, 151
dco::adjoint_vector::interpret_adjoint, 155
dco::adjoint_vector, 155
dco::derivative(DCO_TYPE,dco::adjoint_vector), 155
dco::log_level_enum, 145
dco::logger::level, 146
dco::logger::stream, 146

[dco::mode<DCO_TYPE>](#), 84
[dco::size_of\(DCO_TAPE_TYPE*\)](#), 85
[derivative](#), 30, 68, 152
[ga1s<DCO_BASE_TYPE>::type](#), 33
[ga1sm<DCO_BASE_TYPE>::type](#), 73
[gt1s<DCO_BASE_TYPE>::type](#), 30
[mode<DCO_TYPE>](#), 156
[passive_value](#), 50, 152
[value](#), 30, 152

adjoint

- checkpointing
 - ensemble, 105
 - joint call reversal, 82
- embedding adjoint source code, 110
- first-order
 - [DCO_ADJOINT_TYPE](#), 46
 - [DCO_ADJOINT_VECTOR_TYPE::interpret_adjoint\(\)](#), 139
 - [DCO_ADJOINT_VECTOR_TYPE](#), 139
 - [DCO_MODE::gbcp_order](#), 81
 - [DCO_MODE::order](#), 80
 - [DCO_PARTIAL_TYPE](#), 46
 - [DCO_VALUE_TYPE](#), 46
 - [adjoint_vector<DCO_TAPE_TYPE,DCO_ADJOINT_TYPE>](#), 139
 - [create_gbcp<GBCP_TYPE>\(ACTIVE_TYPE\)](#), 80
 - [derivative\(DCO_T&,DCO_ADJOINT_VECTOR_TYPE&\)](#), 139
 - [gbcp<DCO_ACTIVE_TYPE>::type](#), 80
 - [jacobian_preaccumulator_t::finish\(\)](#), 134
 - [jacobian_preaccumulator_t::register_output\(DCO_TYPE&\)](#), 134
 - [jacobian_preaccumulator_t::start\(\)](#), 134
 - [jacobian_preaccumulator_t](#), 134
 - blob tape, 42
 - chunk tape, 42
 - file tape, 44
 - preaccumulation, 132
 - scalar, 31, 72
 - vector, 69, 75
- second-order
 - checkpointing ensemble, 108
 - joint call reversal, 86
 - preaccumulation, 135
 - scalar, 51, 54
- third-order
 - scalar, 61

casting, 147

dco.hpp, 29

[dco::tape_options](#), 43, 151

logging, 145

[static_cast](#), 147

tangent

- first-order
 - scalar, [28](#)
 - vector, [66](#)
- second-order
 - scalar, [48](#)
- third-order
 - scalar, [57](#)
- type cast, [147](#)
- vector type, [34](#)
- vectorization, [39](#)
- vectorized
 - tangent
 - type, [37](#)

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.