The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/
Abstract. dco/c++ is a highly flexible and efficient implementation of first- and higher-order tangent-linear and adjoint Algorithmic Differentiation (AD) by operator overloading in C++. It combines a cache-optimized internal representation based on C++ expression templates with an intuitive and powerful application programmer interface (API). Support for externally differentiated functions and for optimized data flow reversal through checkpointing is provided. dco/c++ has been applied successfully to a number of numerical simulations in the context of, for example, large-scale parameter estimation and shape optimization.

Starting with an introduction to the fundamentals of AD this user guide describes the various modes of dco/c++ in terms of their APIs and with the help of very simple examples.

dco/c++ is actively developed resulting in a steady evolution of the associated user guide.
## 1 Introduction to Algorithmic Differentiation

1.1 Functionality .................................................. 7

1.1.1 First Derivative Code ........................................ 7

1.1.2 Second Derivative Code ..................................... 9

1.1.3 Higher Derivative Code ...................................... 10

1.2 Implementation by Overloading ................................. 10

1.2.1 First Derivative Code ........................................ 14

1.2.2 Second and Higher Derivative Code ......................... 17

## 2 Basic Tangent-Linear Mode (dco::t1s)

2.1 Purpose ......................................................... 21

2.2 Specification .................................................. 21

2.2.1 Interface ..................................................... 21

2.2.2 Classes ...................................................... 21

2.2.3 Constructors ................................................ 21

2.2.4 Functions .................................................... 21

2.3 Description ..................................................... 22

2.4 Interface Description ......................................... 22

2.4.1 Constructors ................................................ 22

2.4.2 Functions .................................................... 22

2.5 Error Indicators and Warnings ................................. 23

2.6 Example ......................................................... 23

## 3 Basic Adjoint Mode (dco::a1s)

3.1 Purpose ......................................................... 27

3.2 Specification .................................................. 27

3.2.1 Interface ..................................................... 27

3.2.2 Classes ...................................................... 27

3.2.3 Global Data ................................................ 27

3.2.4 Constructors ................................................ 28

3.2.5 Functions .................................................... 28
3.3 Description .......................................................... 28
3.4 Interface Description .............................................. 28
  3.4.1 Constructors ................................................... 28
  3.4.2 Functions ...................................................... 29
3.5 Error Indicators and Warnings .................................. 30
3.6 Example ............................................................ 30

4 Basic Second-Order Tangent-Linear Mode (dco::t2s_t1s) .............. 33
  4.1 Purpose .......................................................... 33
  4.2 Specification .................................................... 33
    4.2.1 Interface ................................................... 33
    4.2.2 Classes .................................................... 33
    4.2.3 Constructors .............................................. 34
    4.2.4 Functions .................................................. 34
  4.3 Description ...................................................... 34
  4.4 Interface Description ........................................... 34
    4.4.1 Constructors .............................................. 34
    4.4.2 Functions .................................................. 34
  4.5 Error Indicators and Warnings .................................. 35
  4.6 Example .......................................................... 35

5 Basic Second-Order Adjoint Mode (dco::t2s_a1s) ........................... 39
  5.1 Purpose .......................................................... 39
  5.2 Specification .................................................... 39
    5.2.1 Classes .................................................... 39
    5.2.2 Constructors .............................................. 40
    5.2.3 Functions .................................................. 40
  5.3 Description ...................................................... 40
  5.4 Interface Description ........................................... 40
    5.4.1 Constructors .............................................. 40
    5.4.2 Functions .................................................. 41
  5.5 Error Indicators and Warnings .................................. 42
  5.6 Example .......................................................... 42

6 Basic Tangent-Linear Vector Mode (dco::t1v) ............................... 47
  6.1 Purpose .......................................................... 47
  6.2 Specification .................................................... 47
    6.2.1 Interface ................................................... 47
    6.2.2 Classes .................................................... 47
    6.2.3 Constructors .............................................. 47
    6.2.4 Functions .................................................. 48
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3 Description</td>
<td>48</td>
</tr>
<tr>
<td>6.4 Interface Description</td>
<td>48</td>
</tr>
<tr>
<td>6.4.1 Constructors</td>
<td>48</td>
</tr>
<tr>
<td>6.4.2 Functions</td>
<td>48</td>
</tr>
<tr>
<td>6.5 Error Indicators and Warnings</td>
<td>49</td>
</tr>
<tr>
<td>6.6 Example</td>
<td>49</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction to Algorithmic Differentiation

This very compact introduction to Algorithmic Differentiation (AD) is based on the book

The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation

by Uwe Naumann [2]. Refer to the same book or to [1] for further details.

1.1 Functionality

In AD, we consider implementations of multivariate vector functions

\[ F : \mathbb{R}^n \rightarrow \mathbb{R}^m : \quad y = F(x) \]

as functions/subroutines written in some programming language, for example

\[ \text{void } f(\text{int } n, \text{int } m, \text{double } *x, \text{double } *y) \]

in C/C++. We are interested in the accurate computation of first (gradients for \( m = 1 \) or Jacobians for \( m > 1 \)), second (Hessians), or higher derivatives of the dependent outputs \( y = (y_j)_{j=0,\ldots,m-1} \) with respect to the independent inputs \( x = (x_i)_{i=0,\ldots,n-1} \) at points for which the given implementation \( f \) of \( F \) is once, twice, or more often continuously differentiable. For notational simplicity, passive arguments that are neither independent inputs nor dependent outputs are not taken into account for the time being. Conceptually, their presence adds nothing to the formal framework outlined in the following. Throughout this introductory chapter we assume that the sets of input and output variables are disjoint. Refer to [2] for a discussion of the general case.

1.1.1 First Derivative Code

**Tangent-Linear Code** The first-order tangent-linear routine

\[ (y^{(1)}, y) = F^{(1)}(x, x^{(1)}) \]

computes the directional derivative \( y^{(1)} \) of \( F \) in direction \( x^{(1)} \) in addition to the function value:

\[ y := F(x) \]

\[ y^{(1)} := \nabla F(x) \cdot x^{(1)} \]

7
where
\[ \nabla F(x) \equiv \frac{\partial y}{\partial x} \in \mathbb{R}^{m \times n} \]
denotes the Jacobian (matrix) of \( F \).

The entire Jacobian can be accumulated by letting \( x^{(1)} \) range over the Cartesian basis vectors in \( \mathbb{R}^n \) with a computational cost of \( O(n) \cdot \text{Cost}(F) \), where \( \text{Cost}(F) \) denotes the computational cost of an evaluation of the given implementation \( f \) of \( F \). Sparsity can and should be exploited. Single directional derivatives can be obtained at a computational cost of \( O(1) \cdot \text{Cost}(F) \). The induced overhead is typically small, that is \( < 2 \). This complexity result is of particular interest in the context of matrix-free solvers for nonlinear systems used, for example, for the solution of nonlinear (partial) differential equations.

See Chapter ?? for a detailed explanation of an implementation of tangent-linear mode with dco/c++

**Adjoint Code**  The first-order adjoint routine
\[
(x^{(1)}, y) = F^{(1)}(x, x^{(1)}, y^{(1)})
\]
increments the input value of \( x^{(1)} \) with the adjoint of \( F \) in direction \( y^{(1)} \) in addition to the computation of the function value:
\[
y := F(x)
\]
\[
x^{(1)} := x^{(1)} + (\nabla F(x))^T \cdot y^{(1)}.
\]

The entire Jacobian can be accumulated by setting \( x^{(1)} = 0 \) on input followed by letting \( y^{(1)} \) range over the Cartesian basis vectors in \( \mathbb{R}^m \) with a computational cost of \( O(m) \cdot \text{Cost}(F) \). Again, sparsity can and should be exploited. Combinations of tangent-linear and adjoint modes may be more effective in the sparse case than any of the two modes applied separately. Gradients of scalar \( (m = 1) \) multivariate functions can be obtained at a computational cost of \( O(1) \cdot \text{Cost}(F) \). The induced overhead depends on the quality of the given implementation of reverse mode. For most overloading tools it rarely undercuts a factor of 10. More likely, it should be expected to range between 20 and 50 for nontrivial numerical simulation programs. dco/c++ aims to push the limits of overloading solutions to adjoint mode AD. It often results in a factor of less than 10, thus outperforming its competitors. This complexity result is of particular interest in the context of first-order nonlinear programming methods used, for example, in the context of parameter estimation or optimal control.

See Chapter ?? for a detailed explanation of an implementation of adjoint mode with dco/c++

**Case Study**  We consider the following implementation of the multivariate scalar function
\[
y = F(x) = \left( \sum_{i=0}^{n-1} x_i^2 \right)^2
\]  
(1.1)
in C/C++:

```cpp
template<typename T>
void f(int n, T *x, T &y) {
    y=0;
    for (int i=0;i<n;i++) y=y+x[i]*x[i];
    y=y*y;
}
```

```cpp
y = F(x) = \left( \sum_{i=0}^{n-1} x_i^2 \right)^2
```

(1.1)
1.1. FUNCTIONALITY

All floating-point data is made generic through the use of the template mechanism. Thus, the function $f$ can be instantiated with both built-in float or double data types or with any of the AD data types provided by dco/c++.

In Table 1.1 we compare the performance of tangent-linear and adjoint modes of dco/c++ for the computation of the gradient $\nabla F(x) = \frac{\partial y}{\partial x} \in \mathbb{R}^n$ for increasing values of $n$. Qualitatively this behaviour is independent of the given computing platform. Generally, we observe linear growth with $n$ of the relative run time of gradient accumulation with respect to the run time of a single function evaluation in tangent-linear mode. Adjoint mode yields a constant relative run time.

<table>
<thead>
<tr>
<th>$n$</th>
<th>tangent-linear mode</th>
<th>adjoint mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>10000</td>
<td>0.2</td>
<td>0.02</td>
</tr>
<tr>
<td>100000</td>
<td>16.0</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 1.1: Run time (in sec.) of gradient accumulation for Equation (1.1) in tangent-linear and adjoint modes.

1.1.2 Second Derivative Code

Second-Order Tangent-Linear Code  The second-order tangent-linear routine

$$(y, y^{(2)}, y^{(1)}, y^{(1,2)}) = F^{(1,2)}(x, x^{(2)}, x^{(1)}, x^{(1,2)})$$

is obtained by application of tangent-linear mode to the first-order tangent-linear routine $F^{(1)}$:

$$y^{(1,2)} := <\nabla F(x), x^{(1,2)} > + <\nabla^2 F(x), x^{(1)}, x^{(2)} >$$

$$y^{(1)} := <\nabla F(x), x^{(1)} >$$

$$y^{(2)} := <\nabla F(x), x^{(2)} >$$

$$y := F(x).$$

(1.2)

The Hessian

$$\nabla^2 F(x) \equiv \frac{\partial^2 y}{\partial x^2} \in \mathbb{R}^{m \times n \times n}$$

can be computed by setting $x^{(1,2)} = 0$ and by letting $x^{(1)}$ and $x^{(2)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^n$. The computational cost amounts to $O(n^2) \cdot Cost(F)$ where $Cost(F)$ denotes the computational cost of a single evaluation of $F$. The computations of the function value and of the first directional derivatives are implied. A Hessian vector product can be evaluated at the cost of $O(n) \cdot Cost(F)$. See Chapter ?? for a detailed explanation of an implementation of second-order tangent-linear mode with dco/c++.

Second-Order Adjoint Code  The second-order adjoint routine

$$(y, y^{(2)}, x^{(1)}, x^{(1,2)}) = F^{(2)}(x, x^{(2)}, x^{(1)}, y^{(2)}, y^{(1)}, y^{(1)})$$

is obtained by application of tangent-linear mode to the first-order adjoint routine $F^{(1)}$:

$$x^{(2)} := x^{(2)} + <y^{(2)}, \nabla F(x) > + <y^{(1)}, \nabla^2 F(x), x^{(2)} >$$

$$x^{(1)} := x^{(1)} + <y^{(1)}, \nabla F(x) >$$

(1.3)

$$y^{(2)} := <\nabla F(x), x^{(2)} >$$

$$y := F(x).$$
The Hessian can be computed by setting $x^{(2)}_{(1)} = y^{(2)}_{(1)} = 0$ and by letting $y^{(1)}_{(1)}$ and $x^{(2)}_{(2)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively. The computational cost amounts to $O(m \cdot n) \cdot \text{Cost}(F)$. For $m = 1$, a single Hessian-vector product can be computed at the computational cost of $O(n) \cdot \text{Cost}(F)$, that is, at a constant multiple of the cost of evaluating $F$. This complexity result is of particular interest in the context of matrix-free second-order nonlinear programming methods. The computations of the function value and of the first directional and adjoint derivatives are implied. See Chapter ?? for a detailed explanation of an implementation of second-order adjoint mode with dco/c++

**Case Study** We consider the given implementation of Equation (1.1) from Section 1.1.1. In Table 1.2 we compare the performance of second-order tangent-linear and adjoint modes of dco/c++ for the computation of the Hessian

$$\nabla^2 F(x) = \frac{\partial^2 y}{\partial x^2} \in \mathbb{R}^{n \times n}$$

for increasing values of $n$. Generally, we observe quadratic growth with $n$ of the relative run time of Hessian accumulation in second-order tangent-linear mode. Second-order adjoint mode yields a linear relative run time.

### 1.1.3 Higher Derivative Code

Higher derivative code is defined recursively. $k$-th-order tangent-linear or adjoint code is obtained by applying tangent-linear mode AD to a $(k-1)$-th-order tangent-linear or adjoint code, respectively.

### 1.2 Implementation by Overloading

Implementations of AD by overloading are best explained in terms of the linearized directed acyclic graph. The execution of a numerical routine that implements $y = F(x)$ as described in Section 1.1 induces a directed acyclic graph (DAG) representing the single assignment code (SAC)

$$\text{for } j = n, \ldots, n + p + m - 1$$

$$v_j = \varphi_j(v_i)_{i \prec j}$$  \hspace{1cm} (1.4)

where $i \prec j$ denotes a direct dependence of $v_j$ on $v_i$. Within the SAC the result of each elemental function $\varphi_j$ is assigned to a unique auxiliary variable $v_j$. The $n$ independent inputs $x_i = v_i$, for $i = 0, \ldots, n - 1$, are mapped onto $m$ dependent outputs $y_j = v_{n+p+j}$, for $j = 0, \ldots, m - 1$. The values of $p$ intermediate variables $v_k$ are computed for $k = n, \ldots, n+p-1$. The corresponding DAG $G = (V, E)$ consists of integer vertices $V = \{0, \ldots, n+p+m-1\}$ and edges $E = \{(i,j) | i \prec j\}$. The vertices are sorted topologically with respect to variable dependence, that is, $\forall i, j \in V : (i,j) \in E \Rightarrow i < j$.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
$n$ & second-order tangent-linear mode & second-order adjoint mode \\
\hline
100 & 0.01 & 0.01 \\
1000 & 2.7 & 0.3 \\
2000 & 21.6 & 1.0 \\
3000 & 76.7 & 2.5 \\
\hline
\end{tabular}
\caption{Run time (in sec.) of Hessian accumulation for Equation (1.1) in second-order tangent-linear and adjoint modes.}
\end{table}
For illustration we consider the lighthouse example $F : \mathbb{R}^4 \rightarrow \mathbb{R}^2$ from [1]
\[
y = F(x) = \begin{pmatrix} x_0 \cdot \tan (x_2 \cdot x_3) & x_1 \cdot x_0 \cdot \tan (x_2 \cdot x_3) \\ x_1 - \tan (x_2 \cdot x_3) & x_1 - \tan (x_2 \cdot x_3) \end{pmatrix}
\] (1.5) implemented as
\[
\text{void } f(\text{double } *x, \text{ double } *y) 
\begin{align*}
&v = \tan(x[2] \cdot x[3]) ; \\
y[0] = x[0] \cdot \frac{v}{(x[1] - v)} ; \\
y[1] = x[1] \cdot y[0] ; \\
\end{align*}
\]

**Example 1** The SAC of the lighthouse example becomes
\[
\begin{align*}
v_0 &= x_0; \quad v_1 = x_1; \quad v_2 = x_2; \quad v_3 = x_3 \\
v_4 &= v_2 \cdot v_3 \\
v_5 &= \tan(v_4) \\
v_6 &= v_1 - v_5 \\
v_7 &= \frac{v_5}{v_6} \\
v_8 &= v_0 \cdot v_7 \\
v_9 &= v_1 \cdot v_8 \\
y_0 &= v_8; \quad y_1 = v_9
\end{align*}
\]
The corresponding DAG is shown in Figure 1.1.

The SAC is linearized by augmenting each SAC assignment with the computation of local partial derivatives of the variable on the left-hand side with respect to all variables on the right-hand side:
\[
c_{j,k} = \frac{\partial \phi_j(v_i)}{\partial v_k} \quad \text{for } k < j
\] (1.6)
\[
v_j = \phi_j(v_i)_{i \prec} .
\]
The linearized SAC induces a linearized DAG where local partial derivatives are attached to the corresponding edges in the DAG.

**Example 2** The linearized SAC of the given implementation of (1.5) becomes
\[
\begin{align*}
v_0 &= x_0; \quad v_1 = x_1; \quad v_2 = x_2; \quad v_3 = x_3 \\
c_{4,2} &= v_3; \quad c_{4,3} = v_2; \quad v_4 = v_2 \cdot v_3 \\
c_{5,4} &= 1 + \tan^2(v_4); \quad v_5 = \tan(v_4) \\
c_{6,1} &= 1; \quad c_{6,5} = -1; \quad v_6 = v_1 - v_5 \\
c_{7,5} &= \frac{1}{v_6}; \quad c_{7,6} = \frac{v_5}{v_6}; \quad v_7 = \frac{v_5}{v_6} \\
c_{8,0} &= v_7; \quad c_{8,7} = v_6; \quad v_8 = v_0 \cdot v_7 \\
c_{9,1} &= v_8; \quad c_{9,8} = v_1; \quad v_9 = v_1 \cdot v_8 \\
y_0 &= v_8; \quad y_1 = v_9
\end{align*}
\]
The corresponding linearized DAG is shown in Figure 1.2. Local partial derivatives are attached to the edges.
Figure 1.1: DAG of the given implementation of Equation (1.5)
1.2. IMPLEMENTATION BY OVERLOADING

Figure 1.2: Linearized DAG of the given implementation of Equation (1.5)
1.2.1 First Derivative Code

Tangent-Linear Code  A tangent-linear code propagates directional derivatives at point \( \mathbf{x} \) in direction \( \mathbf{x}(1) \) of all intermediate variables alongside with their values as in Equation (1.7):

\[
\begin{align*}
    v_j^{(1)} = & \sum_{k<j} \frac{\partial \varphi_j(v_i)}{\partial v_k} \cdot v_k^{(1)} = \sum_{k<j} c_{j,k} \cdot v_k^{(1)} \\
    v_j = & \varphi_j(v_i)_{i<j}.
\end{align*}
\]

(1.7)

Implementation by overloading replaces all active floating-point variables\(^1\) \( v_j \) with a pair \( (v_j^{(1)}, v_j) \) consisting of the original value and the associated directional derivative. All arithmetic operators and intrinsic functions are overloaded to implement Equation (1.7).

Example 3  The tangent-linear SAC of the given implementation of (1.5) becomes

\[
\begin{align*}
    v_0^{(1)} = v_0^{(1)}; & \quad v_0 = x_0; \quad v_1^{(1)} = x_1^{(1)}; \quad v_1 = x_1; \quad v_2^{(1)} = x_2^{(1)}; \quad v_2 = x_2; \quad v_3^{(1)} = x_3^{(1)}; \quad v_3 = x_3 \\
    c_{4,2} = v_3; & \quad c_{4,3} = v_2; \quad v_4^{(1)} = c_{4,2} \cdot v_2^{(1)} + c_{4,3} \cdot v_3^{(1)}; \quad v_4 = v_2 \cdot v_3 \\
    c_{5,4} = 1 + \tan(v_4); & \quad v_5^{(1)} = c_{5,4} \cdot v_4^{(1)}; \quad v_5 = \tan(v_4) \\
    c_{6,1} = 1; & \quad c_{6,5} = -1; \quad v_6^{(1)} = c_{6,1} \cdot v_1^{(1)} + c_{6,5} \cdot v_5^{(1)}; \quad v_6 = v_1 - v_5 \\
    c_{7,5} = \frac{1}{v_5}; & \quad c_{7,6} = -\frac{v_5}{v_6}; \quad v_7^{(1)} = c_{7,5} \cdot v_5^{(1)} + c_{7,6} \cdot v_6^{(1)}; \quad v_7 = v_5 \cdot v_6 \\
    c_{8,0} = v_7; & \quad c_{8,7} = v_0; \quad v_8^{(1)} = c_{8,0} \cdot v_0^{(1)} + c_{8,7} \cdot v_7^{(1)}; \quad v_8 = v_0 \cdot v_7 \\
    c_{9,1} = v_8; & \quad c_{9,8} = v_1; \quad v_9^{(1)} = c_{9,1} \cdot v_1^{(1)} + c_{9,8} \cdot v_8^{(1)}; \quad v_9 = v_1 \cdot v_8 \\
    y_0^{(1)} = & \quad v_8^{(1)}; \quad y_0 = v_8; \quad y_1^{(1)} = v_0^{(1)}; \quad y_1 = v_0
\end{align*}
\]

The corresponding tangent-linear DAG is shown in Figure 1.3. Both the SAC variables’ values and their directional derivatives can be computed forward during the evaluation of the augmented (with local partial derivatives and tangent-linear statements) numerical routine. The data flow of the original routine is preserved.

An implementation of tangent-linear mode AD by overloading yields the following sequence of computations:

\[
\begin{align*}
    (v_0, v_0^{(1)}) = & \quad (x_0, x_0^{(1)}); \quad (v_1, v_1^{(1)}) = (x_1, x_1^{(1)}); \quad (v_2, v_2^{(1)}) = (x_2, x_2^{(1)}); \quad (v_3, v_3^{(1)}) = (x_3, x_3^{(1)}) \\
    (v_4, v_4^{(1)}) = & \quad (v_2 \cdot v_3, v_3 \cdot v_2^{(1)} + v_2 \cdot v_3^{(1)}) \\
    (v_5, v_5^{(1)}) = & \quad (\tan(v_4), (1 + \tan^2(v_4)) \cdot v_4^{(1)}) \\
    (v_6, v_6^{(1)}) = & \quad (v_1 - v_5, v_1^{(1)} - v_5^{(1)}) \\
    (v_7, v_7^{(1)}) = & \quad \left(\frac{v_5}{v_6}, \frac{1}{v_6} \cdot v_5^{(1)} - \frac{v_5}{v_6} \cdot v_6^{(1)}\right) \\
    (v_8, v_8^{(1)}) = & \quad (v_0, v_0 \cdot v_7^{(1)} - v_0 \cdot v_7^{(1)}) \\
    (v_9, v_9^{(1)}) = & \quad (v_1 \cdot v_8, v_8 \cdot v_1^{(1)} + v_1 \cdot v_8^{(1)}) \\
    (y_0, y_0^{(1)}) = & \quad (v_8, v_8^{(1)}); \quad (y_1, y_1^{(1)}) = (v_0, v_0^{(1)})
\end{align*}
\]

The Jacobian of the dependent outputs \( y_0 \equiv v_8 \) and \( y_1 \equiv v_0 \) with respect to the independent input \( \mathbf{x} \equiv (v_0, v_1, v_2, v_3)^T \) can be computed by letting \( \mathbf{x}(1) \equiv (v_0^{(1)}, v_1^{(1)}, v_2^{(1)}, v_3^{(1)})^T \) range over the

\(^1\)... carrying potentially nonzero directional derivatives
1.2. IMPLEMENTATION BY OVERLOADING

Figure 1.3: Tangent-linear DAG of the given implementation of Equation (1.5)
Cartesian basis vectors in $\mathbb{R}^4$ yielding
\[
\nabla F(x) = \begin{pmatrix}
v_7 & -\frac{v_6}{v_5} \cdot v_0 & v_1 \cdot a & v_2 \cdot a \\
v_7 \cdot v_8 & -\frac{v_5}{v_6} \cdot v_0 \cdot v_1 + v_8 & v_1 \cdot a & v_2 \cdot v_1 \cdot a
\end{pmatrix},
\]
where $a := (1 + \tan^2(v_4)) \cdot (\frac{v_5}{v_6} + \frac{1}{v_6}) \cdot v_0$.

**Adjoint Code** An adjoint code propagates adjoints $v_{(1)j}$ of all intermediate variables $v_j$ (inner products of $\nabla v_j f(x)$ with $y_{(1)}$) for $j = n + p + m - 1, \ldots, 0$, that is, in reverse order with respect to the original data flow. In the adjoint SAC the generation of the SAC is followed by the use of the (nonlinearly used) intermediate values for the computation of the local partial derivatives of the elemental functions as in Equation (1.8):

\begin{align}
&\text{for } j = n, \ldots, n + p + m - 1 \\
&v_j = \varphi_j(v_i)_{i < j} \\
&\text{for } j = n, \ldots, n + p - 1 \\
&v_{(1)j} = 0 \\
&\text{for } k = n + p + m - 1, \ldots, n \\
&\text{for } j : j < k \\
&v_{(1)j} = v_{(1)j} + \frac{\partial \varphi_k ((v_i)_{i < k})}{\partial v_j} \cdot v_{(1)k} = v_{(1)j} + c_{k,j} \cdot v_{(1)k}.
\end{align}

This *incremental* adjoint mode distributes in its reverse section the scaled contributions of the gradients of all elemental functions to the adjoints of their arguments. Adjoints of intermediate variables are initialized to zero. The values of nonlinearly used SAC variables need to be recorded for random access within the reverse section. Solutions to the underlying DAG Reversal problem [4] store these values if sufficient persistent memory is available. Otherwise, checkpointing techniques store only selected values in order to recompute the remaining required values. Considerable insight into the given program’s structure and a deep understanding of adjoint mode AD in general are crucial prerequisites for the construction of a robust and efficient practical solution to the DAG Reversal problem. Throughout this introductory chapter we assume that the memory requirement of the given adjoint code does not exceed the available resources.

Implementation of adjoint mode AD by overloading replaces all active floating-point variables $v_j$ with a pair $(v_j, j)$ that consists of the original value and its associated SAC index. All arithmetic operators and intrinsic functions are overloaded to record $\varphi_j$, $v_j$, and $\{i : i < j\}$ for all $j = 0, \ldots, n + p + m - 1$ on a *tape*. Adjoints are propagated by reverse interpretation of the tape.

**Example 4** The adjoint SAC of the given implementation of (1.5) becomes

\[
\begin{align*}
v_0 &= x_0; & v_1 &= x_1; & v_2 &= x_2; & v_3 &= x_3 \\
v_4 &= v_2 \cdot v_3 \\
v_5 &= \tan(v_4) \\
v_6 &= v_1 - v_5 \\
v_7 &= \frac{v_5}{v_6} \\
v_8 &= v_0 \cdot v_7 \\
v_9 &= v_1 \cdot v_8 \\
y_0 &= v_8; & y_1 &= v_9 \\
v_{(1)9} &= y_{(1)1}; & v_{(1)8} &= y_{(1)0} \\
c_{9,8} &= v_1; & v_{(1)8} &= c_{9,8} + v_{(1)9}
\end{align*}
\]
1.2. IMPLEMENTATION BY OVERLOADING

\[ c_{9,1} = v_8; \quad v_{(1)1} = v_{(1)1} + c_{9,1} \cdot v_{(1)9} \]
\[ c_{8,7} = v_0; \quad v_{(1)7} = v_{(1)7} + c_{8,7} \cdot v_{(1)8} \]
\[ c_{8,0} = v_7; \quad v_{(1)0} = v_{(1)0} + c_{8,0} \cdot v_{(1)8} \]
\[ c_{7,6} = -\frac{v_7}{v_6}; \quad v_{(1)6} = v_{(1)6} + c_{7,6} \cdot v_{(1)7} \]
\[ c_{7,5} = \frac{1}{v_6}; \quad v_{(1)5} = v_{(1)5} + c_{7,5} \cdot v_{(1)7} \]
\[ c_{6,5} = -1; \quad v_{(1)5} = v_{(1)5} + c_{6,5} \cdot v_{(1)6} \]
\[ c_{6,1} = 1; \quad v_{(1)1} = v_{(1)1} + c_{6,1} \cdot v_{(1)6} \]
\[ c_{5,4} = 1 + \tan^2(v_4); \quad v_{(1)4} = v_{(1)4} + c_{5,4} \cdot v_{(1)5} \]
\[ c_{4,3} = v_2; \quad v_{(1)3} = v_{(1)3} + c_{4,3} \cdot v_{(1)4} \]
\[ c_{4,2} = v_3; \quad v_{(1)2} = v_{(1)2} + c_{4,2} \cdot v_{(1)4} \]
\[ x_{(1)3} = v_{(1)3}; \quad x_{(1)2} = v_{(1)2}; \quad x_{(1)1} = v_{(1)1}; \quad x_{(1)0} = v_{(1)0}. \]

The corresponding adjoint DAG is shown in Figure 1.4. All required SAC variable values are computed during the evaluation of the SAC within the forward section of the adjoint code. Adjoints are propagated from the outputs toward the inputs. The data flow of the original routine is reversed. A conceptual implementation of adjoint mode AD by overloadi ng generates the following tape:

4 : \((\cdot, v_4, \{2, 3\})\)
5 : \((\tan, v_5, \{4\})\)
6 : \((-, v_6, \{1, 5\})\)
7 : \((/, v_7, \{5, 6\})\)
8 : \((\cdot, v_8, \{0, 7\})\)
9 : \((\cdot, v_9, \{1, 8\})\)

Interpretation yields

\[ v_{(1)8} = v_1 \cdot v_{(1)9} \]
\[ v_{(1)7} = v_0 \cdot v_{(1)8} \]
\[ v_{(1)6} = -\frac{v_7}{v_6} \cdot v_{(1)7} \]
\[ v_{(1)5} = -1 \cdot v_{(1)6} + \frac{1}{v_6} \cdot v_{(1)7} \]
\[ v_{(1)4} = (1 + \tan^2) \cdot v_{(1)5} \]
\[ v_{(1)3} = v_2 \cdot v_{(1)4} \]
\[ v_{(1)2} = v_3 \cdot v_{(1)4} \]
\[ v_{(1)1} = v_8 \cdot v_{(1)9} + 1 \cdot v_{(1)6} \]
\[ v_{(1)0} = v_7 \cdot v_{(1)8}. \]

1.2.2 Second and Higher Derivative Code

Application of Equation (1.7) to the tangent-linear and adjoint SACs yields second-order tangent-linear and adjoint code, respectively. An implementation of second-order tangent-linear mode AD is obtained by considering \((v_j, v_j^{(1)})\) as a pair of first-order tangent-linear variables yielding \(((v_j, v_j^{(2)}), (v_j^{(1)}, v_j^{(1,2)}))\) and by overloading all arithmetic operators and intrinsic functions for this
Figure 1.4: Adjoint DAG of the given implementation of Equation (1.5)
new quadruple based on the given first-order tangent-linear overloading library. Both the recording
and the interpretation steps in a given implementation of first-order adjoint mode AD need to be
overloaded in first-order tangent-linear mode in order to obtain second-order adjoint mode by
overloading. Similar statements apply to third- and higher-order tangent-linear and adjoint code.
Chapter 2

Basic Tangent-Linear Mode
(dco::t1s)

2.1 Purpose

The dco::t1s namespace implements tangent-linear first-order scalar mode AD as outlined in Chapter 1. It provides the data type dco::t1s::type and appropriately overloaded versions of the arithmetic operators and intrinsic functions in C++. A given implementation of a multivariate vector function

$$y = F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$$  \hspace{1cm} (2.1)

is transformed into code for evaluating

$$y^{(1)} := < \nabla F(x), x^{(1)} >$$

$$y := F(x).$$  \hspace{1cm} (2.2)

2.2 Specification

2.2.1 Interface

#include "dco.hpp"

2.2.2 Classes

dco::t1s::type

Description: active data type; all arithmetic operations and built-in functions are overloaded for the computation of directional derivatives

2.2.3 Constructors

dco::t1s::type();
dco::t1s::type(double);
dco::t1s::type(const dco::t1s::type&);

2.2.4 Functions
void dco::t1s::get(const dco::t1s::type&, double&);
void dco::t1s::get(const dco::t1s::type&, double&, int);
void dco::t1s::set(dco::t1s::type&, const double&);
void dco::t1s::set(dco::t1s::type&, const double&, int);

2.3 Description

To obtain a tangent-linear first-order scalar version of the given implementation

```cpp
void f(int n, int m, double *x, double *y)
```

of a multivariate vector function given by Equation (2.1) the user needs to change the data types
of all active parameters and of all active local program variables to dco::t1s::type yielding

```cpp
void f(int n, int m, dco::t1s::type *x, dco::t1s::type *y).
```

The resulting code can be used to compute directional derivatives in addition to the function value
as defined in Equation (2.2) and as illustrated by an example in Section 2.6.

Further preprocessing of the user code may become necessary if, for example, active variables are
written to and read from files. The data layout would have to be adapted accordingly.

2.4 Interface Description

2.4.1 Constructors

```cpp
dco::t1s::type();
```

**Description** allocates an active variable a; sets both its value a.v and its directional
derivative a.v(1) equal to zero

```cpp
dco::t1s::type(double p);
```

**Description** allocates an active variable a; sets a.v = p and a.v(1) = 0

```cpp
dco::t1s::type(const dco::t1s::type &a);
```

**Description** allocates an active variable as a copy of the active variable a

2.4.2 Functions

```cpp
void dco::t1s::get(const dco::t1s::type &a, double &p)
```

**Description** returns p = a.v

```cpp
void dco::t1s::get(const dco::t1s::type &a, double &p, int k)
```

**Constraints** k = 1

**Description** returns p = a.v(1)

```cpp
void dco::t1s::set(dco::t1s::type &a, const double &p)
```
2.5. ERROR INDICATORS AND WARNINGS

Description sets a.\(v = p\)

```cpp
void dco::t1s::set(dco::t1s::type &a, const double &p, int k)
```

Constraints \(k = 1\)

Description sets \(a.v(1) = p\)

2.5 Error Indicators and Warnings

dco::exception thrown if illegal parameter \(k\) is passed to set or get

2.6 Example

We consider an implementation \(f\) of the lighthouse example \(F : \mathbb{R}^4 \rightarrow \mathbb{R}^2\) from [1] given in lines 6-16 of the following code listing. All active variables \((x,y,v,w)\) are declared as dco::t1s::type in order to enable the propagation of directional derivatives by overloading of the built-in arithmetic operators and intrinsic functions. No further preprocessing of \(f\) is required in this example. \LaTeX \ syntax is used in some of the comments.

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

#include "dco.hpp" // definition of dco::t1s::type

void f(
    int n, // number of active inputs
    int m, // number of active outputs
    const dco::t1s::type * const x, // active inputs
    dco::t1s::type * const y // active outputs
) {
    dco::t1s::type v=tan(x[2]*x[3]);
    dco::t1s::type w=x[1]-v;
    y[0]=x[0]*v/w;
    y[1]=y[0]*x[1];
}

void t1s_driver(
    int n, // number of active inputs
    int m, // number of active outputs
    const double * const x, // active inputs x
    const double * const xt1, // x'\{1\}
    double * const y, // active outputs y
    double * const yt1 // y'\{1\}
) {
    dco::t1s::type *t1s_x=new dco::t1s::type[n];
    dco::t1s::type *t1s_y=new dco::t1s::type[m];
    // initialization of values and first directional
    // derivatives of active inputs
    for (int i=0;i<n;i++) {
        dco::t1s::set(t1s_x[i],x[i]);
    }
```
Inclusion of the `dco/c++` header in line 4 ensures availability of the definition of the active data type `dco::t1s::type`. The driver in lines 18-43 computes the value $y = y$ of the function $F$ and a product $yt1 = y^{(1)}$ of the Jacobian $\nabla F \in \mathbb{R}^{2 \times 4}$ with the vector $xt1 = x^{(1)} \in \mathbb{R}^{4}$ of directional derivatives of the input vector $x = x \in \mathbb{R}^{4}$ as defined in Equation (2.2). Both the entries of $x$ and of $xt1$ are set equal to 1 in line 52 yielding the sum of the columns of the Jacobian at point $x^T = (1, 1, 1, 1)$ as the output $yt1$ of the driver in line 54.

Tangent-linear instances `t1s_x` of the active input $x$ and `t1s_y` of the active output $y$ are allocated in lines 26 and 27. Value and directional derivative components of the active variable `t1s_x` are set equal to the entries of $x$ and $xt1$ in lines 30-33 followed by calling the overloaded version of $f$ in line 35. Function values and directional derivatives are extracted from the active output `t1s_y` in lines 38-41 and they are stored in $y$ and $yt1$, respectively.

A single call of the driver yields the function values $y$ and their directional derivatives $yt1$ with respect to $x$ in direction $xt1$.

If the above code is stored as a file `t1s_basic.cpp`, then the executable `t1s_basic` is build by

```
$CC -I$PATH_TO_DCO_HPP -o t1s_basic t1s_basic.cpp $DCO_LIB
```

where `$CC` references the native C++ compiler and where `$PATH_TO_DCO_HPP` contains the path to the `dco/c++` interface `dco.hpp`. `$DCO_LIB` denotes the static `dco/c++` library.

Running the executable...
tis_basic

yields the following output:

\[ y[0]=-2.79402 \]
\[ y[1]=-2.79402 \]
\[ y^{(1)}[0]=14.2435 \]
\[ y^{(1)}[1]=11.4495 \]

Six significant digits are displayed. The Jacobian of \( F \) at point \( x = (1,1,1)^T \) is equal to

\[
\nabla F(x) = \begin{pmatrix}
-2.79402 & -5.01252 & 11.025 & 11.025 \\
-2.79402 & -7.80654 & 11.025 & 11.025 \\
\end{pmatrix}
\]

Consequently, the sum of its columns is computed as the first directional derivative of \( F \) with respect to \( x \) in direction \( x^{(1)} = (1,1,1,1) \), that is,

\[
y^{(1)} := \langle \nabla F(x), x^{(1)} \rangle \equiv \nabla F(x) \cdot x^{(1)}
\]

\[
= \begin{pmatrix}
-2.79402 & -5.01252 & 11.025 & 11.025 \\
-2.79402 & -7.80654 & 11.025 & 11.025 \\
\end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 14.2435 \\ 11.4495 \end{pmatrix}.
\]
Chapter 3

Basic Adjoint Mode (dco::a1s)

3.1 Purpose

The dco::a1s namespace implements adjoint first-order scalar mode AD as outlined in Chapter 1. It provides the data type dco::a1s::type and appropriately overloaded versions of the arithmetic operators and intrinsic functions in C++. The overloaded arithmetic operators and intrinsic functions generate an image of the computation (a tape) of type dco::a1s::tape the interpretation of which computes the desired adjoints incrementally. A given implementation of a multivariate vector function

\[ y = F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m \]  

(3.1)

is transformed into code for evaluating

\[
\begin{align*}
  x_{(1)} &:= x_{(1)} + < y_{(1)}, \nabla F(x) > \\
y &:= F(x).
\end{align*}
\]

(3.2)

3.2 Specification

3.2.1 Interface

```cpp
#include "dco.hpp"
```

3.2.2 Classes

- **dco::a1s::type**
  - Description: active data type; all arithmetic operations and built-in functions are overloaded to generate a tape

- **dco::a1s::tape**
  - Description: tape; grows dynamically within the available memory

3.2.3 Global Data

```cpp
dco::a1s::tape* dco::a1s::global_tape
```

- Description: global tape for use in single-tape adjoint computation
3.2.4 Constructors

\texttt{dco \::\: a1s \::\: type();}
\texttt{dco \::\: a1s \::\: type(double);} 
\texttt{dco \::\: a1s \::\: type(const dco \::\: a1s \::\: type&);} 

3.2.5 Functions

\texttt{void dco \::\: a1s \::\: get(const dco \::\: a1s \::\: type&, double&);} 
\texttt{void dco \::\: a1s \::\: get(const dco \::\: a1s \::\: type&, double&, int);} 
\texttt{void dco \::\: a1s \::\: set(dco \::\: a1s \::\: type&, const double&);} 
\texttt{void dco \::\: a1s \::\: set(dco \::\: a1s \::\: type&, const double&, int);} 

\texttt{static dco \::\: a1s \::\: tape* dco \::\: a1s \::\: tape \::\: create();} 
\texttt{static void dco \::\: a1s \::\: tape \::\: remove(dco \::\: a1s \::\: tape*);} 
\texttt{void dco \::\: a1s \::\: tape \::\: register_variable(dco \::\: a1s \::\: type&);} 
\texttt{void dco \::\: a1s \::\: tape \::\: register_output_variable(dco \::\: a1s \::\: type&);} 
\texttt{void dco \::\: a1s \::\: tape \::\: interpret_adjoint();} 

3.3 Description

To obtain an adjoint 1st-order scalar version of the given implementation 
\texttt{void f(int n, int m, double *x, double *y)}
of a multivariate vector function given by Equation (3.1) the user needs to switch the data types of all active parameters and of all active local program variables to \texttt{dco::a1s::type} yielding 
\texttt{void f(int n, int m, dco::a1s::type *x, dco::a1s::type *y)} .

The resulting code can be used to generate a tape the interpretation of which yields the desired adjoints in addition to the function value as defined in Equation (3.2) and as illustrated by an example in Section 3.6.

Further preprocessing of the user code may become necessary if, for example, active variables are written to and read from files. The data layout would have to be adapted accordingly.

3.4 Interface Description

3.4.1 Constructors

\texttt{dco::a1s::type();} 
\textbf{Description} allocates an active variable \( a \); sets both its value \( a.v \) and its adjoint \( a.v(1) \) equal to zero

\texttt{dco::a1s::type(double p);} 
\textbf{Description} allocates an active variable \( a \); sets \( a.v = p \) and \( a.v(1) = 0 \)

\texttt{dco::a1s::type(const dco::a1s::type &a);} 
\textbf{Description} allocates an active variable as a copy of the active variable \( a \)
3.4. INTERFACE DESCRIPTION

3.4.2 Functions

\texttt{dco::a1s::tape* dco::a1s::tape::create();}

Description allocates a tape; chunk-wise growth up to physical memory bound; predefined chunk size

\texttt{dco::a1s::tape::remove(dco::a1s::tape *my_tape);}  

Description deallocates the tape

\texttt{void dco::a1s::tape::register_variable(dco::a1s::type &a)}

Description registers an independent variable a with the tape; a corresponding tape entry is created

\texttt{void dco::a1s::tape::register_output_variable(dco::a1s::type &a)}

Description registers a dependent variable a with the tape; a new tape entry is created through an auxiliary assignment

\texttt{void dco::a1s::tape::interpret_adjoint();}

Description propagates adjoints from the dependent to the independent variables backwards through the tape

\texttt{void dco::a1s::get(const dco::a1s::type &a, double &p)}

Description returns \( p = a.v \)

\texttt{void dco::a1s::get(const dco::a1s::type &a, double &p, int k)}

Constraints \( k = -1 \)

Description returns \( p = a.v_{(1)} \)

\texttt{void dco::a1s::set(dco::a1s::type &a, const double &p)}

Description sets \( a.v = p \)

\texttt{void dco::a1s::set(dco::a1s::type &a, const double &p, int k)}

Constraints \( k = -1 \)

Description sets \( a.v_{(1)} = p \)
3.5 Error Indicators and Warnings

dco::exception thrown if illegal parameter k is passed to set/get or if a variable of type dco::als::type is referenced after deallocation of the tape.

3.6 Example

We consider an implementation f of the lighthouse example $F : \mathbb{R}^4 \rightarrow \mathbb{R}^2$ from [1] given in lines 6-16 of the following code listing. All active variables (x,y,v,w) are declared as dco::als::type in order to enable the recording of the tape by overloading of the built-in arithmetic operators and intrinsic functions. The tape is interpreted for the computation of adjoints of the active inputs x as a function of adjoints of the active outputs y. No further preprocessing of f is required in this example. LaTeX syntax is used in some of the comments.

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

#include "dco.hpp" // definition of dco::als::type and
// of dco::als::tape

dco::als::tape *dco::als::global_tape = 0;

void f(
   int n,        // number of active inputs
   int m,        // number of active outputs
   const dco::als::type * const x, // active inputs
   dco::als::type * const y        // active outputs
) {
   dco::als::type v=tan(x[2]*x[3]);
   dco::als::type w=x[1]-v;
   y[0]=x[0]*v/w;
   y[1]=y[0]*x[1];
}

void als_driver(
   int n,        // number of active inputs
   int m,        // number of active inputs
   const double * const x, // active inputs x
   double * const xal, // x_{(1)}
   double * const yal // y_{(1)}
) { // allocation of chunk tape (dynamic growth to
   // physical memory bound)
   dco::als::global_tape = dco::als::tape::create();
   dco::als::type *als_x = new dco::als::type[n];
   dco::als::type *als_y = new dco::als::type[m];
   // initialization of values of active inputs
   // and registration with tape
   for (int i=0;i<n;i++) {
      dco::als::global_tape->register_variable(als_x[i]);
      dco::als::set(als_x[i], x[i]);
   }
   //...
   delete [] als_x;
   delete [] als_y;
   dco::als::global_tape->close();
}
```


3.6. EXAMPLE

```cpp
40 // overloaded function evaluation (creates tape)
41 f(n,m,als_x,als_y);
42 for (int i=0;i<m;i++) {
43 // marking of active outputs within tape
44 dco::als::global_tape->register_output_variable(als_y[i]);
45 // extraction of values of active outputs
46 dco::als::get(als_y[i],y[i]);
47 // initialization of adjoints of active outputs
48 dco::als::set(als_y[i],y[i],-1);
49 }
50 // initialization of adjoints of active inputs
51 for (int i=0;i<n;i++)
52 dco::als::set(als_x[i],x[i],-1);
53 // interpretation of tape
54 dco::als::global_tape->interpret_adjoint();
55 // extraction of adjoints of active inputs
56 for (int i=0;i<n;i++) dco::als::get(als_x[i],x[i],-1);
57 // extraction of adjoints of active outputs
58 for (int i=0;i<m;i++) dco::als::get(als_y[i],y[i],-1);
59 delete [] als_y; delete [] als_x;
60 // deallocation of tape
61 dco::als::tape::remove(dco::als::global_tape);
62 }
63
64 int main() {
65 const int n=4, m=2;
66 double *x=new double[n], *xal=new double[n];
67 double *y=new double[m], *yal=new double[m];
68 // initialization of inputs
69 for (int i=0;i<n;i++) { x[i]=1; xal[i]=1; }
70 for (int i=0;i<m;i++) yal[i]=1;
71 // driver
72 als_driver(n,m,x,xal,y,yal);
73 // results
74 for (int i=0;i<m;i++)
75 cout << "y[" << i << "]=" << y[i] << endl;
76 for (int i=0;i<n;i++)
77 cout << "x[" << (i) << "]=" << xal[i] << endl;
78 for (int i=0;i<n;i++)
79 cout << "y[" << (i) << "]=" << yal[i] << endl;
80 delete [] yal; delete [] y; delete [] xal; delete [] x;
81 return 0;
82 }
```

Inclusion of the `dco/c++` header in line 4 ensures availability of the definition of the active data type `dco::als::type`. The driver in lines 18-59 computes the value `y=x` of the function `F` and the value of `xal=x(1)` incremented by the product of the transposed Jacobian `\nabla F^T \in \mathbb{R}^{4 \times 2}` with the vector `yal=y(1) \in \mathbb{R}^2` of adjoints of the output vector `y \in \mathbb{R}^2` as defined in Equation (3.2). The input values of `x=x`, `xal` as well as the values of `yal` are set equal to 1 in lines 66-67, yielding the shifted (each entry increased by one) sum of the rows of the Jacobian at point `x^T = (1, 1, 1, 1)` as the output `xal` of the driver in line 69.

The tape is created in line 28. It grows in chunks of predefined size and it is limited by the given bound on the available memory. Adjoint instances `als_x` of the active input `x` and `als_y` of the
active output $y$ are allocated in lines 29 and 30. The values of the active input $a_{1\cdot}$x are set equal to the entries of x following the registration of the respective independent variables with the tape in lines 33–36. A tape is recorded by the call of the overloaded function f in line 38 followed by marking the dependent variables in $a_{1\cdot}$y as active outputs. Function values are extracted from the active output in line 43. The adjoints of all active inputs and outputs are set equal to the respective values from xa1 and ya1 in lines 49 and 45. Valid adjoints of the inputs can be extracted in line 55 only after calling the tape interpreter in line 51. All active data and the tape data structure are deallocated in lines 56–58.

A single call of the driver in line 69 evaluates Equation (3.2). The adjoints $y_{(1)}$ of the active outputs remain unchanged provided that the program variable $y$ is a pure output of f that holds values of active outputs exclusively.

If the above code is stored as a file a1s_basic.cpp, then the executable a1s_basic is build by

```
$CC -I$PATH_TO_DCO_HPP -o a1s_basic a1s_basic.cpp $DCO_LIB
```

where $CC$ references the native C++ compiler and where $PATH_TO_DCO_HPP$ contains the path to the dco/c++ interface dco.hpp. $DCO_LIB$ denotes the static dco/c++ library.

Running the executable a1s_basic yields the following output:

```
y[0]=-2.79402
y[1]=-2.79402
x_{(1)}[0]=-4.58804
x_{(1)}[1]=-11.8191
x_{(1)}[2]=23.0501
x_{(1)}[3]=23.0501
y_{(1)}[0]=1
y_{(1)}[1]=1
```

Six significant digits are displayed.

The Jacobian of $F$ at point $x = (1, 1, 1)^T$ is equal to

$$
\nabla F(x) = \begin{pmatrix}
-2.79402 & -5.01252 & 11.025 & 11.025 \\
-2.79402 & -7.80654 & 11.025 & 11.025
\end{pmatrix}.
$$

Consequently, the shifted sum of its rows is computed as the first-order adjoint of $F$ with respect to $x$ in direction $y_{(1)} = (1, 1)^T$ added to $x_{(1)} = (1, 1)^T$, that is,

$$
x_{(1)} := x_{(1)} + < y_{(1)}, \nabla F > = x_{(1)} + (\nabla F)^T \cdot y_{(1)}
$$

$$
= \begin{pmatrix}1 \\ 1 \\ 1 \\ 1\end{pmatrix} + \begin{pmatrix}-2.79402 & -2.79402 \\
-5.01252 & -7.80654 \\
11.025 & 11.025 \\
11.025 & 11.025
\end{pmatrix} \begin{pmatrix}1 \\ 1 \\ 1 \\ 1\end{pmatrix}
= \begin{pmatrix}-4.58804 \\ -11.8191 \\ 23.0501 \\ 23.0501\end{pmatrix}.
$$
Chapter 4

Basic Second-Order Tangent-Linear Mode (dco::t2s_t1s)

4.1 Purpose

The dco::t2s_t1s namespace implements tangent-linear second-order scalar mode AD by overloading of the tangent-linear first-order scalar arithmetic in tangent-linear mode as outlined in Chapter 1. It provides the data type dco::t2s_t1s::type and appropriately overloaded versions of the arithmetic operators and intrinsic functions in C++. A given implementation of a multivariate vector function

\[
\begin{align*}
y &= F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m
\end{align*}
\]  

is transformed into code for evaluating

\[
\begin{align*}
y^{(1,2)} &= \langle \nabla F(x), x^{(1,2)} \rangle + \langle \nabla^2 F(x), x^{(1)}, x^{(2)} \rangle \\
y^{(1)} &= \langle \nabla F(x), x^{(1)} \rangle \\
y^{(2)} &= \langle \nabla F(x), x^{(2)} \rangle \\
y &= F(x).
\end{align*}
\]  

4.2 Specification

4.2.1 Interface

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

**Description** defines dco::t2s_t1s::type

4.2.2 Classes

dco::t2s_t1s::type

**Description** active data type; all arithmetic operators and built-in functions are overloaded for the computation of second-order directional derivatives
4.2.3 Constructors

dco::t2s_t1s::type();
dco::t2s_t1s::type(const dco::t2s_t1s::type&);

4.2.4 Functions

void dco::t2s_t1s::get (const dco::t2s_t1s::type&, double&);
void dco::t2s_t1s::get (const dco::t2s_t1s::type&, double&, int);
void dco::t2s_t1s::get (const dco::t2s_t1s::type&, double&, int, int);

void dco::t2s_t1s::set (dco::t2s_t1s::type&, const double&);
void dco::t2s_t1s::set (dco::t2s_t1s::type&, const double&, int);
void dco::t2s_t1s::set (dco::t2s_t1s::type&, const double&, int, int);

4.3 Description

To obtain a second-order tangent-linear scalar version of the given implementation

void f(int n, int m, double *x, double *y)

of a multivariate vector function given by Equation (4.1) the user needs to switch the data types
of all active parameters and of all active local program variables to dco::t2s_t1s::type yielding

void f(int n, int m, dco::t2s_t1s::type* x, dco::t2s_t1s::type* y).

The resulting code can be used to compute second-order directional derivatives in addition to first-
order directional derivatives and the function value as defined in Equation (4.2) and as illustrated
by an example in Section 4.6.

Further preprocessing of the user code may become necessary if, for example, active variables are
written to and read from files. The data layout would have to be adapted accordingly.

4.4 Interface Description

4.4.1 Constructors

dco::t2s_t1s::type();

Description allocates an active variable a; sets its value a.v and its first- and
second-order directional derivatives a.v^{(1)}, a.v^{(2)}, and a.v^{(1,2)} equal to zero

dco::t2s_t1s::type(const dco::t2s_t1s::type &a);

Description allocates an active variable as a copy of the active variable a

4.4.2 Functions

void dco::t2s_t1s::get (const dco::t2s_t1s::type& a, double& p)

Description returns p = a.v

void dco::t2s_t1s::get (const dco::t2s_t1s::type& a, double& p, int k)
4.5 Error Indicators and Warnings

- **Constraints**: $k \in \{1, 2\}$
- **Description**: $k = 1$ returns $p = a.v(1)$; $k = 2$ returns $p = a.v(2)$;

```cpp
void dco::t2s_t1s::get (const dco::t2s_t1s::type& a, double& p, int k, int l)
```
- **Constraints**: $k = 1$ and $l = 2$
- **Description**: returns $p = a.v(1,2)$

```cpp
void dco::t2s_t1s::set (dco::t2s_t1s::type& a, const double& p)
```
- **Description**: sets $a.v = p$

```cpp
void dco::t2s_t1s::set (dco::t2s_t1s::type& a, const double& p, int k)
```
- **Constraints**: $k \in \{1, 2\}$
- **Description**: $k = 1$ sets $a.v(1) = p$; $k = 2$ sets $a.v(2) = p$;

```cpp
void dco::t2s_t1s::set (dco::t2s_t1s::type& a, const double& p, int k, int l)
```
- **Constraints**: $k = 1$ and $l = 2$
- **Description**: sets $a.v(1,2) = p$

4.5 Error Indicators and Warnings

- **dco::exception**: thrown if illegal values for $k$ and/or $l$ are passed to set or get

4.6 Example

We consider an implementation $f$ of the lighthouse example $F : \mathbb{R}^4 \to \mathbb{R}^2$ from [1] given in lines 6–16 of the following code listing. All active variables $(x,y,v,w)$ are declared as `dco::t2s_t1s::type` in order to enable the propagation of first and second directional derivatives by overloading of the built-in arithmetic operators and intrinsic functions. No further preprocessing of $f$ is required in this example. \LaTeX{} syntax is used in some of the comments.

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

#include "dco.hpp" // definition of dco::t2s_t1s::type

void f(
  int n, // number of active inputs
  int m, // number of active outputs
  const dco::t2s_t1s::type * const x, // active inputs
```
```cpp
dco::t2s_t1s::type * const y // active outputs
}
dco::t2s_t1s::type v=tan(x[2]*x[3]);
dco::t2s_t1s::type w=x[1]-v;
y[0]=x[0]*v/w;
y[1]=y[0]*x[1];
}
void t2s_t1s_driver(
int n, // number of active inputs
int m, // number of active outputs
const double * const x, // active inputs x
const double * const xt1, // xˆ{(1)}
const double * const xt2, // xˆ{(2)}
const double * const xt1t2, // xˆ{(1,2)}
double * const y, // active outputs y
double * const yt1, // yˆ{(1)}
double * const yt2, // yˆ{(2)}
double * const yt1t2 // yˆ{(1,2)}
)
{
dco::t2s_t1s::type *t2s_t1s_x=new dco::t2s_t1s::type[n];
dco::t2s_t1s::type *t2s_t1s_y=new dco::t2s_t1s::type[m];
// initialization of values, first and second directional
derivatives of active inputs
for (int i=0;i<n;i++) {
dco::t2s_t1s::set(t2s_t1s_x[i],x[i]);
dco::t2s_t1s::set(t2s_t1s_x[i],xt1[i],1);
dco::t2s_t1s::set(t2s_t1s_x[i],xt2[i],2);
dco::t2s_t1s::set(t2s_t1s_x[i],xt1t2[i],1,2);
}
// overloaded function evaluation
f(n,m,t2s_t1s_x,t2s_t1s_y);
// extraction of values, first and second directional
derivatives from active outputs
for (int i=0;i<m;i++) {
dco::t2s_t1s::get(t2s_t1s_y[i],y[i]);
dco::t2s_t1s::get(t2s_t1s_y[i],yt1[i],1);
dco::t2s_t1s::get(t2s_t1s_y[i],yt2[i],2);
dco::t2s_t1s::get(t2s_t1s_y[i],yt1t2[i],1,2);
}
delete [] t2s_t1s_y; delete [] t2s_t1s_x;
}
int main() {
const int n=4, m=2;
double *x=new double[n], *x1=new double[n];
double *xt2=new double[n], *xt1t2=new double[n];
double *y=new double[m], *yt1t2=new double[m];
// initialization of inputs
for (int i=0;i<n;i++) { x[i]=1; xt1[i]=1; xt2[i]=1; xt1t2[i]=1;}
// driver
t2s_t1s_driver(n,m,x,xt1,xt2,xt1t2,y,yt1,yt2,yt1t2);
// results
```
Inclusion of the \texttt{dco/c++} header in line 4 ensures availability of the definition of the active data type \texttt{dco::t2s::t1s::type}. The driver in lines 18–51 computes the value $y^* = y$ of the function $F$, products $yt1^* = y^{(1)} \in \mathbb{R}^2$ and $yt2^* = y^{(2)} \in \mathbb{R}^2$ of the Jacobian the $\nabla F \in \mathbb{R}^{2 \times 4}$ with vectors $xt1^* = x^{(1)} \in \mathbb{R}^4$ and $xt2^* = x^{(2)} \in \mathbb{R}^4$, respectively, and a projection of the Jacobian $\nabla F$ in direction $xt1t2^* = x^{(1,2)} \in \mathbb{R}^4$ added to a projection $yt1t2^* = y^{(1,2)} \in \mathbb{R}^{2 \times 4 \times 4}$ of the Hessian $\nabla^2 F \in \mathbb{R}^{2 \times 4 \times 4 \times 4}$ in directions $xt1$ and $xt2$ as defined in Equation (4.2). The entries of $x^* = x$, $xt1^*$, $xt2^*$ and $xt1t2^*$ are set equal to 1 in line 60 yielding the sum of the columns of the Jacobian at point $x^T = (1, 1, 1, 1)$ in $yt1$ as well as in $yt2$. Similarly, the projection of the Hessian in direction $xt1$ results in $< \nabla^2 F, xt1 > \in \mathbb{R}^{2 \times 4}$ that is the sum of the four $2 \times 4$ matrices forming the trailing dimension of the 3-tensor $\nabla^2 F \in \mathbb{R}^{2 \times 4 \times 4 \times 4}$. The subsequent projection of the result in direction $xt2$ yields the sum of the columns of $< \nabla^2 F, xt1 >$ added to the sum of the columns of the Jacobian in $yt1t2$.

Second-order tangent-linear instances $t2s_t1s_x$ of the active input $x$ and $t2s_t1s_y$ of the active output $y$ are allocated in lines 30 and 31. Value and first and second directional derivative components of the active input $t2s_t1s_x$ are set equal to the entries of $x$, $xt1$, $xt2$, and $xt1t2$ in lines 34 – 39 followed by calling the overloaded version of $f$ in line 41. Function values and first and second directional derivatives are extracted from the active output $t1s_y$ in lines 44 – 49 and they are stored in $y$, $yt1$, $yt2$, and $yt1t2$, respectively.

A single call of the driver in line 62 evaluates Equation (4.2). Initialization of $x$, $xt1$, $xt2$ and $xt1t2$ with different values leads to correspondingly modified linear combinations of projections of the first and second derivative tensors.

If the above code is stored as a file \texttt{t2s\_t1s\_basic.cpp}, then the executable \texttt{t2s\_t1s\_basic} is build by

\begin{verbatim}
$CC -I$PATH_TO_DCO_HPP -o t2s_t1s_basic t2s_t1s_basic.cpp $DCO_LIB
\end{verbatim}

where $\texttt{CC}$ references the native C++ compiler and where $\texttt{PATH_TO_DCO_HPP}$ contains the path to the \texttt{dco/c++} interface \texttt{dco.hpp}. $\texttt{DCO_LIB}$ denotes the static \texttt{dco/c++} library.

Running the executable \texttt{t2s_t1s_basic}

yields the following output:

\begin{verbatim}
y[0]=-2.79402
y[1]=-2.79402
y^{(1)}[0]=14.2435
y^{(1)}[1]=11.4495
y^{(2)}[0]=14.2435
y^{(2)}[1]=11.4495
y^{(1,2)}[0]=-149.95
y^{(1,2)}[1]=-124.257
\end{verbatim}
Six significant digits are displayed. 
The Jacobian of \( F \) at point \( x = (1, 1, 1)^T \) is equal to 
\[
\nabla F(x) = \begin{pmatrix}
-2.79402 & -5.01252 & 11.025 & 11.025 \\
-2.79402 & -7.80654 & 11.025 & 11.025
\end{pmatrix}
\]
yielding for \( x^{(1)} = x^{(2)} = (1, 1, 1)^T \) 
\[
< \nabla F(x), x^{(1)} > = < \nabla F(x), x^{(2)} > = \begin{pmatrix}
14.2435 \\
11.4495
\end{pmatrix}.
\]
A serialization of the trailing dimension of the corresponding Hessian \( \nabla^2 F(x) \in \mathbb{R}^{2\times 4\times 4} \) is equal to 
\[
\nabla^2 F(x) = \begin{pmatrix}
0 & -5.01252 & 11.025 & 11.025 \\
0 & -7.80654 & 11.025 & 11.025 \\
-5.01252 & -17.9851 & 50.5833 & 50.5833 \\
-7.80654 & -28.0102 & 61.6084 & 61.6084 \\
11.025 & 50.5833 & -101.167 & -90.1416 \\
11.025 & 61.6084 & -101.167 & -90.1416 \\
11.025 & 50.5833 & -90.1416 & -101.167 \\
\end{pmatrix}.
\]
The sum of the above four \( (2 \times 4) \)-matrices can be computed for \( x^{(1)} = (1, 1, 1)^T \) as 
\[
< \nabla^2 F(x), x^{(1)} > = \begin{pmatrix}
17.0376 & 78.169 & -129.7 & -129.7 \\
14.2435 & 87.4 & -118.675 & -118.675
\end{pmatrix}.
\]
The following product with \( x^{(2)} = (1, 1, 1)^T \) yields 
\[
< \nabla^2 F(x), x^{(1)}, x^{(2)} >= < < \nabla^2 F(x), x^{(1)} >, x^{(2)} > 
= \begin{pmatrix}
17.0376 & 78.169 & -129.7 & -129.7 \\
14.2435 & 87.4 & -118.675 & -118.675
\end{pmatrix} \begin{pmatrix}
1 \\
1 \\
1
\end{pmatrix} = \begin{pmatrix}
-164.193 \\
-135.706
\end{pmatrix}.
\]
Consequently, 
\[
y^{(1, 2)} := < \nabla F(x), x^{(1, 2)} > + < \nabla^2 F(x), x^{(1)}, x^{(2)} > 
= \begin{pmatrix}
14.2435 \\
11.4495
\end{pmatrix} + \begin{pmatrix}
-164.193 \\
-135.706
\end{pmatrix} = \begin{pmatrix}
-149.95 \\
-124.257
\end{pmatrix}
\]
for \( x^{(1, 2)} = (1, 1, 1)^T \).
Chapter 5

Basic Second-Order Adjoint Mode (dco::t2s_a1s)

5.1 Purpose

The dco::t2s_a1s namespace implements adjoint second-order scalar mode AD by overloading the adjoint first-order scalar arithmetic in tangent-linear mode as outlined in Chapter 1. It provides the data type dco::t2s_a1s::type and appropriately overloaded versions of the arithmetic operators and intrinsic functions in C++. A given implementation of a multivariate vector function

\[ y = F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m \]  

is transformed into code for evaluating

\[ x^{(2)}(1) := x^{(2)}(1) + \langle y^{(2)}(1), \nabla F(x) \rangle + \langle y(1), \nabla^2 F(x), x^{(2)} \rangle \]
\[ x(1) := x(1) + \langle y(1), \nabla F(x) \rangle \]
\[ y^{(2)} := \langle \nabla F(x), x^{(2)} \rangle \]
\[ y := F(x). \]  

5.2 Specification

#include <dco.hpp>

Description defines dco::t2s_a1s::type

5.2.1 Classes

dco::t2s_a1s::type

Description active data type; all arithmetic operators and built-in functions are overloaded for the computation of second-order adjoint derivatives

dco::t2s_a1s::tape

Description tangent-linear global tape
5.2.2 Constructors

\begin{verbatim}
dco::t2s_a1s::type();
dco::t2s_a1s::type(const dco::t2s_a1s::type&);\end{verbatim}

5.2.3 Functions

\begin{verbatim}
static dco::t2s_a1s::tape* dco::t2s_a1s::tape::create();
static void dco::t2s_a1s::tape::remove(dco::t2s_a1s::tape*);
void dco::t2s_a1s::tape::register_variable(dco::t2s_a1s::type&);
void dco::t2s_a1s::tape::register_output_variable(dco::t2s_a1s::type&);
void dco::t2s_a1s::tape::interpret_adjoint();
void dco::t2s_a1s::get(const dco::t2s_a1s::type&, double&);
void dco::t2s_a1s::get(const dco::t2s_a1s::type&, double&, int);
void dco::t2s_a1s::get(const dco::t2s_a1s::type&, double&, int, int);
void dco::t2s_a1s::set(dco::t2s_a1s::type&, const double&);
void dco::t2s_a1s::set(dco::t2s_a1s::type&, const double&, int);
void dco::t2s_a1s::set(dco::t2s_a1s::type&, const double&, int, int);
\end{verbatim}

5.3 Description

To obtain a second-order adjoint scalar version of the given implementation

\begin{verbatim}
void f(int n, int m, double *x, double *y)
\end{verbatim}

of a multivariate vector function given in Equation (5.1) the user needs to switch the data types of all active parameters and of all active local program variables to \texttt{dco::t2s_a1s::type} yielding

\begin{verbatim}
void f(int n, int m, dco::t2s_a1s::type* x, dco::t2s_a1s::type* y).
\end{verbatim}

The resulting code can be used to compute second-order adjoint derivatives in addition to first-order directional and adjoint derivatives and the function value as defined in Equation (5.2) and as illustrated by an example in Section 5.6.

Further preprocessing of the user code may become necessary if, for example, active variables are written to and read from files. The data layout would have to be adapted accordingly.

5.4 Interface Description

5.4.1 Constructors

\begin{verbatim}
dco::t2s_a1s::type();
\end{verbatim}

\begin{verbatim}
dco::t2s_a1s::type(const dco::t2s_a1s::type &a);
\end{verbatim}

\begin{verbatim}
Description allocates an active variable a; sets its value \texttt{a.v} and its first and second
derivatives \texttt{a.v(1)}, \texttt{a.v(2)}, and \texttt{a.v(2)(1)} equal to zero
\end{verbatim}
5.4. INTERFACE DESCRIPTION

5.4.2 Functions

```cpp
static dco::t2s_a1s::tape* dco::t2s_a1s::tape::create();
Description  allocates a tape; chunk-wise growth up to physical memory bound;
predefined chunk size

static void dco::t2s_a1s::tape::remove(dco::t2s_a1s::tape*);
Description  deallocates the tape

void dco::t2s_a1s::tape::register_variable(dco::t2s_a1s::type&);
Description  registers an independent variable a with the tape; a corresponding
tape entry is created

void dco::t2s_a1s::tape::register_output_variable(dco::t2s_a1s::type&);
Description  registers a dependent variable a with the tape; a new tape entry is
created through an auxiliary assignment

void dco::t2s_a1s::tape::interpret_adjoint();
Description  propagates first- and second-order adjoints from the dependent to the
independent variables backwards through the tape

void dco::t2s_a1s::get(const dco::t2s_a1s::type& a, double& p)
Description  returns p = a.v

void dco::t2s_a1s::get(const dco::t2s_a1s::type& a, double& p, int k)
Constraints  k ∈ {−1, 2}
Description  k = −1 returns p = a.v(1); k = 2 returns p = a.v(2);

void dco::t2s_a1s::get(const dco::t2s_a1s::type& a, double& p, int k, int l)
Constraints  k = −1 and l = 2
Description  returns p = a.v_l(1)

void dco::t2s_a1s::set(dco::t2s_a1s::type& a, const double& p)
Description  sets a.v = p

void dco::t2s_a1s::set(const dco::t2s_a1s::type& a, double& p, int k)
```

CHAPTER 5. BASIC SECOND-ORDER ADJOINT MODE (DCO::T2S A1S)

Constraints $k \in \{-1, 2\}$

Description $k = -1$ sets $a.v(1) = p$; $k = 2$ sets $a.v(2) = p$;

```cpp
void dco::t2s_a1s::set(const dco::t2s_a1s::type& a, double& p, int k, int l)
```

Constraints $k = -1$ and $l = 2$

Description sets $a.v^{(2)}(1) = p$

5.5 Error Indicators and Warnings

$dco::exception$ thrown if illegal values for $k$ and/or $l$ are passed to set/get or if a variable of type $dco::t2s_a1s::type$ is referenced after deallocation of the tape

5.6 Example

We consider an implementation $f$ of the lighthouse example $F: \mathbb{R}^4 \to \mathbb{R}^2$ from [1] given in lines 6–16 of the following code listing. All active variables $(x, y, v, w)$ are declared as $dco::t2s_a1s::type$ in order to enable the propagation of first- and second-order adjoints by overloading of the built-in arithmetic operators and intrinsic functions. No further preprocessing of $f$ is required in this example. \LaTeX syntax is used in some of the comments.

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

#include "dco.hpp" // definition of dco::t2s_a1s::type

dco::t2s_a1s::tape * dco::t2s_a1s::global_tape = 0;

void f(  
    int n,  // number of active inputs  
    int m,  // number of active outputs  
    const dco::t2s_a1s::type * const x,  // active inputs  
    const dco::t2s_a1s::type * const y  // active inputs  
)  
{
    dco::t2s_a1s::type v=tan(x[2]*x[3]);
    dco::t2s_a1s::type w=x[1]-v;
    y[0]=x[0]*v/w;
    y[1]=y[0]*x[1];
}

void t2s_a1s_driver(  
    int n,  // number of active inputs  
    int m,  // number of active outputs  
    const double * const x,  // active inputs x  
    const double * const xt2,  // x'^{(2)}  
    double * const xal1,  // x_{(1)}  
    double * const xalt2,  // x_{(1)}^{(2)}
```
5.6. EXAMPLE

```cpp
27 double * const y, // active outputs y
28 double * const yt2, // y^\{(2)\}
29 double * const ya1, // y_{(1)}
30 double * const ya1t2 // y_{(1)}^\{\{(2)\}\}
31 )
32 // creation of chunk tape (dynamic growth to
33 // physical memory bound)
34 dco::t2s_als::global_tape=dco::t2s_als::tape::create();
35 dco::t2s_als::type *t2s_als_x=new dco::t2s_als::type[n];
36 dco::t2s_als::type *t2s_als_y=new dco::t2s_als::type[m];
37 // initialization of active inputs
38 // and registration with tape
39 for (int i=0; i<n; i++)
40 dco::t2s_als::global_tape->register_variable(t2s_als_x[i]);
41 dco::t2s_als::set(t2s_als_x[i],x[i]);
42 dco::t2s_als::set(t2s_als_x[i],xl[i],-1);
43 dco::t2s_als::set(t2s_als_x[i],xt2[i],2);
44 dco::t2s_als::set(t2s_als_x[i],xa1[i],-1,2);
45 }
46 // overloaded function evaluation
47 // (creates tangent-linear tape)
48 f(n,m,t2s_als_x,t2s_als_y);
49 for (int i=0; i<m; i++)
50 // extraction of values of active outputs
51 dco::t2s_als::get(t2s_als_y[i],y[i]);
52 // extraction of directional derivatives of
53 // active outputs with respect to active inputs
54 dco::t2s_als::get(t2s_als_y[i],yt2[i],2);
55 // marking of active outputs within tangent-linear tape
56 dco::t2s_als::global_tape->register_output_variable(t2s_als_y[i]);
57 // initialization of adjoints of active outputs
58 dco::t2s_als::set(t2s_als_y[i],ya1[i],-1);
59 // initialization of second-order adjoints of active outputs
60 dco::t2s_als::set(t2s_als_y[i],ya1t2[i],-1,2);
61 }
62 // interpretation of tangent-linear tape
63 dco::t2s_als::global_tape->interpret_adjoint();
64 // extraction of first- and second-order adjoints of
65 // active outputs with respect to active inputs
66 for (int i=0; i<n; i++)
67 dco::t2s_als::get(t2s_als_x[i],xl[i],-1);
68 dco::t2s_als::get(t2s_als_x[i],xa1t2[i],-1,2);
69 }
70 // first- and second-order adjoints of
71 // active outputs remain unchanged (just checking)
72 for (int i=0; i<m; i++)
73 dco::t2s_als::get(t2s_als_y[i],ya1[i],-1);
74 dco::t2s_als::get(t2s_als_y[i],ya1t2[i],-1,2);
75 }
76 delete [] t2s_als_y; delete [] t2s_als_x;
77 // deallocation of tape
78 dco::t2s_als::tape::remove(dco::t2s_als::global_tape);
79 }
```
CHAPTER 5. BASIC SECOND-ORDER ADJOINT MODE (DCO::T2S_A1S)

44

int main() {
  const int n=4,m=2;
  double *x=new double[n], *xal=new double[n];
  double *xt2=new double[n], *xal2t=new double[n];
  double *y=new double[m], *yal=new double[m];
  double *yt2=new double[m], *yal2t=new double[m];

  // initialization of inputs
  for (int i=0;i<n;i++) { x[i]=1; xt2[i]=1; xal[i]=1; xal2[i]=1; }
  for (int i=0;i<m;i++) { yal[i]=1; yal2[i]=1; }

  // driver
  t2s_a1s_driver (n,m,x,xt2,xal,xal2,y,yt2,yal,yal2);

  // results
  for (int i=0;i<m;i++)
    cout << "y[" << i << "]=" << y[i] << endl;
  for (int i=0;i<n;i++)
    cout << "x[" << i << "]=" << xal[i] << endl;
  for (int i=0;i<m;i++)
    cout << "y[" << i << "]=" << yal2[i] << endl;

  delete [] x; delete [] xt2; delete [] xal; delete [] xal2t;
  delete [] y; delete [] yt2; delete [] yal; delete [] yal2t;

  return 0;
}

Inclusion of the dco/c++ header in line 4 ensures availability of the definition of the active data type dco::t2s_t1s::type. The driver in lines 18–78 computes ya1t2=\( y^{(2)}_1 \), xal=\( x_1 \), yt2=\( y^{(2)} \), and y=\( y \) as defined in Equation (5.2). The entries of \( x=x_1, x_2=xt2^{(2)}, xal2t=x_1^{(3)}, yal=y_1 \), and yal2t are set equal to 1 in lines 87–88 yielding the shifted sum (by adding 1 to each entry) of the rows of the Jacobian at point \( x^T=(1,1,1,1) \) in xal as well as the sum of the columns of the Jacobian in yt2 as defined in Equation (5.2). Moreover, the projection of the Hessian in direction xt2 results in \( < \nabla^2 F, xt2 > \in \mathbb{R}^{3\times4} \) that is the sum of the four \( 2\times4 \) matrices forming the trailing dimension of the 3-tensor \( \nabla^2 F \in \mathbb{R}^{2\times4\times4} \). The subsequent projection of the result in direction yal yields the sum of the rows of \( < \nabla^2 F, xt2 > \) added to the shifted sum (by adding 1 to each entry) of the rows of the Jacobian in xal2t.

Second-order adjoint instances t2s_a1s_x of the active input x and t2s_a1s_y of the active output y are allocated in lines 33–34 following the creation of the dynamically growing tape in line 32. The registration of the elements of t2s_a1s_x as active inputs within the tape in line 38 is followed by the initialization of its value and first and second derivative components in lines 39–42. Calling the overloaded function f in line 47 yields the function value extracted from t2s_a1s_y into y in line 50 and first-order directional derivative information extracted from t2s_a1s_y into yt2 in line 53. A tangent-linear tape is generated. Its active outputs are registered in line 55 followed by the initialization of its first- and second-order adjoints in lines 57 and 59, respectively. First- and second-order adjoints of the inputs are obtained by interpretation of the tangent-linear tape. The respective values are extracted from t2s_a1s_x in lines 65–68.

A single call of the driver in line 90 evaluates Equation (5.2). Initialization of its arguments with different values leads to correspondingly modified linear combinations of projections of the first
and second derivative tensors.

If the above code is stored as a file `t2s_a1s_basic.cpp`, then the executable `t2s_a1s_basic` is build by

```bash
$CC -I$PATH_TO_DCO_HPP -o t2s_a1s_basic t2s_a1s_basic.cpp $DCO_LIB
```

where `$CC` references the native C++ compiler and where `$PATH_TO_DCO_HPP` contains the path to the `dco/c++` interface `dco.hpp`. `$DCO_LIB` denotes the static `dco/c++` library.

Running the executable `t2s_a1s_basic` yields the following output:

```plaintext
y[0]=-2.79402
y[1]=-2.79402
x_{(1)}[0]=-4.58804
x_{(1)}[1]=-11.8191
x_{(1)}[2]=23.0501
x_{(1)}[3]=23.0501
y^{(2)}[0]=14.2435
y^{(2)}[1]=11.4495
x_{(1)}^{(2)}[0]=26.6931
x_{(1)}^{(2)}[1]=153.75
x_{(1)}^{(2)}[2]=-225.325
x_{(1)}^{(2)}[3]=-225.325
y_{(1)}[0]=1
y_{(1)}[1]=1
y_{(1)}^{(2)}[0]=1
y_{(1)}^{(2)}[1]=1
```

Six significant digits are displayed.

The Jacobian of $F$ at point $x = (1, 1, 1)^T$ is equal to

$$
\nabla F(x) = \begin{pmatrix} -2.79402 & -5.01252 & 11.025 & 11.025 \\ -2.79402 & -7.80654 & 11.025 & 11.025 \end{pmatrix}
$$

yielding for $x_{(1)} = x^{(2)} = (1, 1, 1, 1)^T$ and $y_{(1)} = (1, 1)^T$

$$
y^{(2)} := \langle \nabla F(x), x^{(2)} \rangle = \begin{pmatrix} 14.2435 \\ 11.4495 \end{pmatrix}
$$

and

$$
x_{(1)} := x_{(1)} + \langle y_{(1)}, \nabla F(x) \rangle = \begin{pmatrix} -4.58804 \\ -11.8191 \\ 23.0501 \\ 23.0501 \end{pmatrix}
$$

A serialization of the trailing dimension of the corresponding Hessian $\nabla^2 F(x) \in \mathbb{R}^{2 \times 4 \times 4}$ is equal to

$$
\nabla^2 F(x) = \begin{pmatrix}
0 & -5.01252 & 11.025 & 11.025 \\
-5.01252 & -17.9851 & 50.5833 & 50.5833 \\
11.025 & 50.5833 & -101.167 & -90.1416 \\
11.025 & 61.6084 & -101.167 & -90.1416 \\
11.025 & 61.6084 & -90.1416 & -101.167 \\
11.025 & 61.6084 & -90.1416 & -101.167 \\
\end{pmatrix}
$$
CHAPTER 5. BASIC SECOND-ORDER ADJOINT MODE (DCO::T2S_A1S)

The sum of the above four \((2 \times 4)\)-matrices can be computed for \(\mathbf{x}^{(2)} = (1, 1, 1, 1)^T\) as

\[
< \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} > = \begin{pmatrix}
17.0376 & 78.169 & -129.7 & -129.7 \\
14.2435 & 87.4 & -118.675 & -118.675
\end{pmatrix}.
\]

The following product of its transpose with \(\mathbf{y}^{(1)} = (1, 1)^T\) yields

\[
< \mathbf{y}^{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} > \equiv < \mathbf{y}^{(1)}, < \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} >>
\]

\[
= \begin{pmatrix}
17.0376 & 14.2435 \\
78.169 & 87.4 \\
-129.7 & -118.675 \\
-129.7 & -118.675
\end{pmatrix} \cdot \begin{pmatrix}1 \\ 1 \end{pmatrix} = \begin{pmatrix}31.2811 \\ 165.569 \\ -248.375 \\ -248.375 \end{pmatrix}.
\]

Consequently,

\[
\mathbf{x}^{(2)}_{(1)} := \mathbf{x}^{(2)}_{(1)} + < \mathbf{y}^{(2)}_{(1)}, \nabla F(\mathbf{x}) > + < \mathbf{y}^{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} >
\]

\[
= \begin{pmatrix}1 \\ 1 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix}-5.58804 \\ -12.8191 \\ 22.0501 \\ 22.0501 \end{pmatrix} + \begin{pmatrix}31.2811 \\ 165.569 \\ -248.375 \\ -248.375 \end{pmatrix} = \begin{pmatrix}26.6931 \\ 153.75 \\ -225.325 \\ -225.325 \end{pmatrix}
\]

for \(\mathbf{y}^{(2)}_{(1)} = (1, 1, 1, 1)^T\).
Chapter 6

Basic Tangent-Linear Vector Mode (dco::t1v)

6.1 Purpose

The dco::t1v namespace implements tangent-linear first-order vector mode AD as outlined in Chapter 1. It provides the data type dco::t1v::type and appropriately overloaded versions of the arithmetic operators and intrinsic functions in C++. A given implementation of a multivariate vector function

\[ y = F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m \]  

(6.1)

is transformed into code for evaluating

\[ Y^{(1)} := \langle \nabla F(x), X^{(1)} \rangle \]

\[ y := F(x), \]  

(6.2)

where \( X^{(1)} \in \mathbb{R}^{n \times l} \) and, hence, \( Y^{(1)} \in \mathbb{R}^{m \times l} \).

6.2 Specification

6.2.1 Interface

#include "dco.hpp"

6.2.2 Classes

dco::t1v::type

Description  active data type; all arithmetic operations and built-in functions are overloaded for the computation of directional derivatives

6.2.3 Constructors

dco::t1v::type();
dco::t1v::type(double);
dco::t1v::type(const dco::t1v::type&);
6.2.4 Functions

```cpp
void dco::t1v::get(const dco::t1v::type&, double&);
void dco::t1v::get(const dco::t1v::type&, double&, int, int);
void dco::t1v::set(dco::t1v::type&, const double&);
void dco::t1v::set(dco::t1v::type&, const double&, int, int);
```

6.3 Description

To obtain a tangent-linear first-order vector version of the given implementation

```cpp
void f(int n, int m, double *x, double *y)
```

of a multivariate vector function given by Equation (6.1) the user needs to change the data types of all active parameters and of all active local program variables to `dco::t1v::type` yielding

```cpp
void f(int n, int m, dco::t1v::type *x, dco::t1v::type *y).
```

The resulting code can be used to compute directional derivatives in addition to the function value as defined in Equation (6.2) and as illustrated by an example in Section 6.6.

Further preprocessing of the user code may become necessary if, for example, active variables are written to and read from files. The data layout would have to be adapted accordingly.

6.4 Interface Description

6.4.1 Constructors

```cpp
dco::t1v::type();
```

**Description**  allocates an active variable `a`; sets both its value `a.v` and its directional derivatives `a.v_i(1)` for `i = 0,...,l - 1`, equal to zero

```cpp
dco::t1v::type(double p);
```

**Description**  allocates an active variable `a`; sets `a.v = p` and `a.v_i(1) = 0, i = 0,...,l - 1`

```cpp
dco::t1v::type(dco::t1v::type a);
```

**Description**  allocates an active variable as a copy of the active variable `a`

6.4.2 Functions

```cpp
void dco::t1v::get(const dco::t1v::type &a, double &p)
```

**Description**  returns `p = a.v`

```cpp
void dco::t1v::get(const dco::t1v::type &a, double &p, int k, int i)
```

**Constraints**  `k = 1`
6.5. ERROR INDICATORS AND WARNINGS

Description
returns \( p = a \cdot v \)

\[
\text{void dco::t1v::set(dco::t1v::type \&a, const double \&p)}
\]

Description
sets \( a \cdot v = p \)

\[
\text{void dco::t1v::set(dco::t1v::type \&a, const double \&p, int \ k, int \ i)}
\]

Constraints
\( k = 1 \)

Description
sets \( a \cdot v_{1} = p \)

6.6 Example

We consider an implementation \( f \) of the lighthouse example \( F : \mathbb{R}^4 \rightarrow \mathbb{R}^2 \) from [1] given in lines 6-16 of the following code listing. All active variables \( (x,y,v,w) \) are declared as \( \text{dco::t1v::type} \) in order to enable the propagation of directional derivatives by overloading of the built-in arithmetic operators and intrinsic functions. No further preprocessing of \( f \) is required in this example. LaTeX syntax is used in some of the comments.

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

#include "dco.hpp" // definition of dco::t1v::type

void f(int n, // number of active inputs
         int m, // number of active outputs
         const dco::t1v::type * const x, // active inputs
         dco::t1v::type * const y // active outputs
         ) {
    dco::t1v::type v=tan(x[2]*x[3]);
    dco::t1v::type w=x[1]-v;
    y[0]=x[0]*v/w;
    y[1]=y[0]*x[1];
}

void t1v_driver{
    int n, // number of active inputs
    int m, // number of active outputs
    int l, // number of directional derivatives
    const double * const x, // active inputs x \in \mathbb{R}^n
    const double * const * const xtl, // x \{(1)\} \in \mathbb{R}^\{n \times l\}
    double * const y, // active outputs y \in \mathbb{R}^m
    double * const * const ytl // y \{(1)\} \in \mathbb{R}^\{m \times l\}
    ) {
```

```cpp
```
```
27  dco::t1v::type *t1v_x=new dco::t1v::type[n];
28  dco::t1v::type *t1v_y=new dco::t1v::type[m];
29  // initialization of values and first directional derivatives of active inputs
30  for (int i=0;i<n;i++) {
31    dco::t1v::set(t1v_x[i],x[i],0);
32    for (int j=0;j<l;j++)
33      dco::t1v::set(t1v_x[i],xt1[i][j],1,j);
34  }
35  // overloaded function evaluation
36  f(n,m,t1v_x,t1v_y);
37  // extraction of values and first directional derivatives from active outputs
38  for (int i=0;i<m;i++) {
39    dco::t1v::get(t1v_y[i],y[i],0);
40    for (int j=0;j<l;j++)
41      dco::t1v::get(t1v_y[i],yt1[i][j],1,j);
42  }
43  delete [] t1v_y; delete [] t1v_x;
44  }
45
46  int main() {
47    const int m=2, n=4, l=n;
48    double *x=new double[n];
49    double **xt1=new double*[n];
50    for (int i=0;i<n;i++) xt1[i]=new double[l];
51    double *y=new double[m];
52    double **yt1=new double*[m];
53    for (int i=0;i<m;i++) yt1[i]=new double[l];
54    // initialization of inputs
55    for (int i=0;i<n;i++) {
56      x[i]=1;
57      for (int j=0;j<l;j++) xt1[i][j]=0;
58      xt1[i][i]=1;
59    }
60    // driver
61    t1v_driver(n,m,l,x,xt1,y,yt1);
62    // results
63    for (int i=0;i<m;i++)
64      cout << "y[" << i << "]"="y[i]" << endl;
65    for (int i=0;i<n;i++)
66      for (int j=0;j<l;j++)
67        cout << "y`{((1)}"[" << i << "]"[" << j << "]"="
68          << yt1[i][j]" << endl;
69    for (int i=0;i<m;i++) delete [] yt1[i];
70    delete [] yt1; delete [] y;
71    for (int i=0;i<n;i++) delete [] xt1[i];
72    delete [] xt1; delete [] x;
73    return 0;
74  }
```

Inclusion of the dco/c++ header in line 4 ensures availability of the definition of the active data type dco::t1v::type. The driver in lines 18-46 computes the value \(\hat{y}\) of the function \(F\) and the Jacobian \(\nabla F \in \mathbb{R}^{2 \times 4}\) in \(\hat{y} = y^{(1)} \in \mathbb{R}^{2 \times 4}\) as a collection of derivatives in the direction of the
Cartesian basis vectors stored in \( x_{1} = X^{(1)} \in \mathbb{R}^{4 \times 4} \) in lines 59–60. All the entries of \( x \) are set equal to 1 in line 58 yielding the Jacobian at point \( x^T = (1, 1, 1, 1) \) as the output \( y_{1} \) of the driver in line 63.

Tangent-linear instances \( t_{1v,x} \) of the active input \( x \) and \( t_{1v,y} \) of the active output \( y \) are allocated in lines 27 and 28. Value and directional derivative components of the active variable \( t_{1v,x} \) are set equal to the entries of \( x \) and \( x_{1} \) in lines 31–35 followed by calling the overloaded version of \( f \) in line 37. Function values and directional derivatives are extracted from the active output \( t_{1v,y} \) in lines 40–44 and they are stored in \( y \) and \( y_{1} \), respectively. A single call of the driver yields the function values \( y \) and their directional derivatives \( y_{1} \) with respect to \( x \) in direction \( x_{1} \).

If the above code is stored as a file \( t_{1v} \_basic.cpp \), then the executable \( t_{1v} \_basic \) is build by

\[
\$CC$ -I$\text{PATH\_TO\_DCO\_HPP}$ -DDCO\_T1V\_SIZE=4 -o $t_{1v} \_basic \ $t_{1v} \_basic.cpp $\text{DCO\_LIB}
\]

where \$CC\$ references the native C++ compiler and where \$PATH\_TO\_DCO\_HPP\$ contains the path to the dco/c++ interface dco.hpp. The preprocessor variable DCO\_T1V\_SIZE needs to be set to the number of directional derivatives to be computed. Its default value is 5. \$\text{DCO\_LIB}\$ denotes the static dco/c++ library.

Running the executable

\( t_{1v} \_basic \)

yields the following output:

\[
\begin{align*}
    y[0] &= -2.79402 \\
    y[1] &= -2.79402 \\
    y^{(1)}[0][0] &= -2.79402 \\
    y^{(1)}[0][1] &= 5.01252 \\
    y^{(1)}[0][2] &= 11.025 \\
    y^{(1)}[0][3] &= 11.025 \\
    y^{(1)}[1][0] &= -2.79402 \\
    y^{(1)}[1][1] &= -7.80654 \\
    y^{(1)}[1][2] &= 11.025 \\
    y^{(1)}[1][3] &= 11.025
\end{align*}
\]

Six significant digits of the function value and of the Jacobian are displayed.
Bibliography


Aachener Informatik-Berichte

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

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

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

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

2010-01 * Fachgruppe Informatik: Jahresbericht 2010
2010-02 Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03 Holger Krahn: MontiCore: Agile Entwicklung von domänenpezifischen Sprachen im Software-Engineering
2010-04 René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme
2010-05 Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme
2010-06 Taohe Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata
2010-07 George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms
2010-08 Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting
2010-09 George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs
2010-10 Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut
2010-11 Martin Zimmermann: Parametric LTL Games
2010-12 Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut
2010-13 Ashraf Armouch: Design Patterns for Safety-Critical Embedded Systems
2010-14 Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination
2010-15 Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode
2010-17 Hans Grönninger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten
2010-18 Ibrahim Armouch: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit
2010-19 Felix Reidl: Experimental Evaluation of an Independent Set Algorithm
2010-20 Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games
2011-01 * Fachgruppe Informatik: Jahresbericht 2011
2011-02 Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting
2011-03 Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems
2011-04 Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars
2011-06 Johannes Lotz, Klaus Leppkes, and Uwe Naumann: dco/c++ - Derivative Code by Overloading in C++
2011-07 Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV
2011-08 Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog
2011-09 Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing
2011-10 Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations
2011-12 Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems
2011-13 Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP
2011-14 Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games
2011-16 Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM
2011-17 Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis
2011-18 Kamal Barakat: Introducing Timers to pi-Calculus
2011-24 Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations
2011-25 Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations
2011-26 Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata
2012-01 Fachgruppe Informatik: Annual Report 2012
2012-02 Thomas Heer: Controlling Development Processes
2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems
2012-04 Marcus Gelderie: Strategy Machines and their Complexity
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-05</td>
<td>Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs</td>
<td>Automated Complexity Analysis for Prolog by Term Rewriting</td>
</tr>
<tr>
<td>2012-06</td>
<td>Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl</td>
<td>Automated Termination Proofs for Java Programs with Cyclic Data</td>
</tr>
<tr>
<td>2012-07</td>
<td>André Egners, Björn Marschollek, and Ulrike Meyer</td>
<td>Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms</td>
</tr>
<tr>
<td>2012-08</td>
<td>Hongfei Fu</td>
<td>Computing Game Metrics on Markov Decision Processes</td>
</tr>
<tr>
<td>2012-09</td>
<td>Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer</td>
<td>Quantitative Timed Analysis of Interactive Markov Chains</td>
</tr>
<tr>
<td>2012-10</td>
<td>Uwe Naumann and Johannes Lotz</td>
<td>Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations</td>
</tr>
<tr>
<td>2012-11</td>
<td>Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs</td>
<td>Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs</td>
</tr>
<tr>
<td>2012-12</td>
<td>Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara</td>
<td>Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations</td>
</tr>
<tr>
<td>2012-14</td>
<td>Viet Yen Nguyen</td>
<td>Trustworthy Spacecraft Design Using Formal Methods</td>
</tr>
<tr>
<td>2012-15</td>
<td>Michael Reke</td>
<td>Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen</td>
</tr>
<tr>
<td>2012-16</td>
<td>Markus Towara and Uwe Naumann</td>
<td>A Discrete Adjoint Model for OpenFOAM</td>
</tr>
<tr>
<td>2012-17</td>
<td>Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes</td>
<td>Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries</td>
</tr>
<tr>
<td>2012-18</td>
<td>Andreas Rausch and Marc Sihling</td>
<td>Software &amp; Systems Engineering Essentials 2013</td>
</tr>
<tr>
<td>2013-01</td>
<td>Marc Brockschmidt, Byron Cook, and Carsten Fuhs</td>
<td>Better termination proving through cooperation</td>
</tr>
<tr>
<td>2013-02</td>
<td>André Stollenwerk</td>
<td>Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung</td>
</tr>
<tr>
<td>2013-03</td>
<td>Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám</td>
<td>On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers</td>
</tr>
<tr>
<td>2013-05</td>
<td>Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl</td>
<td>Alternating Runtime and Size Complexity Analysis of Integer Programs</td>
</tr>
</tbody>
</table>


* These reports are only available as a printed version.

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