Getting Started With dco0.9 And dcc0.9

Uwe Naumann

LuFG 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
email: naumann@stce.rwth-aachen.de

and

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

www: http://www.nag.co.uk
email: Uwe.Naumann@nag.co.uk

June 6, 2012

Preface

This document supports an introductory short course on first-order Algorithmic Differentiation (AD) using the AD software tools `dco 0.9` and `dcc 0.9`. It is based on

U. Naumann: *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.* Number 24 in Software, Environments, and Tools, SIAM, 2012.

and can only be a first step toward coverage of the material therein including

- motivating uses of derivatives in numerical algorithms
- second- and higher-order tangent-linear and adjoint code
- checkpointing through call tree reversal
- detection and exploitation of sparsity
- development and use of AD software tools.

Contents

Chapter 1

Introduction to Algorithmic Differentiation

This very compact introduction to Algorithmic Differentiation (AD) is based on [2]. It is meant to set the stage for the user guides of the AD software tools `dco 0.9` and `dcc 0.9` presented in Appendices A and B, respectively, and described in greater detail in [2]. Refer to the same book and [1] for comprehensive coverage of AD.

1.1 Functionality

In AD, we consider implementations of multivariate vector functions

$$F : \mathbb{R}^n \to \mathbb{R}^m : \quad \mathbf{y} = F(\mathbf{x})$$

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

```
void f(int n, int m, double *x, 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 $\mathbf{y} = (y_j)_{j=0,\ldots,m-1}$ with respect to the *independent* inputs $\mathbf{x} = (x_i)_{i=0,\ldots,n-1}$ 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 Routine The first-order tangent-linear routine $(\mathbf{y}^{(1)}, \mathbf{y}) = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$ computes the directional derivative $\mathbf{y}^{(1)}$ of F in direction $\mathbf{x}^{(1)}$ in addition to the function value:

$$\begin{aligned} \mathbf{y}^{(1)} &= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \\ \mathbf{y} &= F(\mathbf{x}) \quad , \end{aligned}$$

where $\nabla F(\mathbf{x}) \equiv \nabla_{\mathbf{x}}(\mathbf{x}) \in \mathbb{R}^{m \times n}$ denotes the Jacobian (matrix) of F. The signature of the tangent-linear 1st-order scalar routine becomes

```
void t1s_f(int n, int m, double *x, double *t1s_x
                         double *y, double *t1s_y)
```

where $\mathbf{x}^{(1)} \equiv$ t1s_x and $\mathbf{y}^{(1)} \equiv$ t1s_y. The underlying semantic modification of f can be implemented alternatively by source-to-source transformation or by operator and function overloading. dco 0.9 takes the latter approach. See Section 1.2.1 for details on the implementation of tangent-linear first-order scalar mode by overloading.

The entire Jacobian can be accumulated by letting $\mathbf{x}^{(1)}$ range over the Cartesian basis vectors in $\mathbb{R}^n$ with a computational cost of $O(n) \cdot Cost(F)$, where $Cost(F)$ 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 $O(1) \cdot Cost(F)$. 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.

Adjoint Routine The first-order adjoint routine $(\mathbf{x}_{(1)}, \mathbf{y}) = F_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)})$ increments the input value of $\mathbf{x}_{(1)}$ with the adjoint of F in direction $\mathbf{y}_{(1)}$ in addition to the computation of the function value:

$$\begin{aligned} \mathbf{y} &= F(\mathbf{x}) \\ \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + (\nabla F(\mathbf{x}))^T \cdot \mathbf{y}_{(1)} \\ \mathbf{y}_{(1)} &= 0 \quad . \end{aligned}$$

Adjoints $\mathbf{y}_{(1)}$ of the outputs are set to zero by the adjoint function. The signature of the adjoint 1st-order scalar routine becomes

```
void a1s_f(int n, int m, double *x, double *a1s_x
                         double *y, double *a1s_y)
```

where $\mathbf{x}_{(1)} \equiv$ a1s_x and $\mathbf{y}_{(1)} \equiv$ a1s_y. See Section 1.2.1 for details on the implementation of adjoint first-order scalar mode by overloading.

The entire Jacobian can be accumulated by setting $\mathbf{x}_{(1)} = 0$ on input followed by letting $\mathbf{y}_{(1)}$ range over the Cartesian basis vectors in $\mathbb{R}^m$ with a computational cost of $O(m) \cdot Cost(F)$. 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 two modes applied separately. Gradients of scalar ($m = 1$) multivariate functions can be obtained at a computational cost of $O(1) \cdot Cost(F)$. 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. The induced overhead depends on the quality of the given implementation of reverse mode.

1.1.2 Case Study

Consider the simple multivariate scalar function

$$f(\mathbf{x}) = \left(\sum_{i=0}^{n-1} x_i^2 \right)^2 \tag{1.1}$$

used in [2] to illustrate the superiority of adjoint mode AD in the context of nonlinear programming. A possible implementation is the following:

```
1 void f(int n, double *x, double &y) {
2   y=0;
3   for (int i=0;i<n;i++) y=y+x[i]*x[i];
```

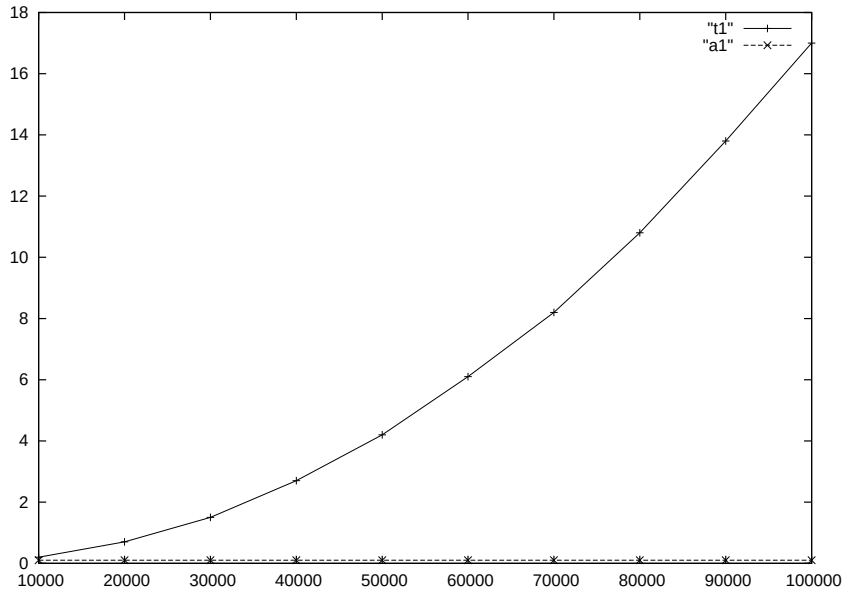

Figure 1.1: Run time of gradient accumulation using first-order tangent-linear (t1) and adjoint (a1) versions of the given implementation of Equation (1.1). The tangent-linear method yields a computational cost of $O(n) \cdot Cost(f)$. The run time of the single evaluation of the adjoint code is negligible for the problem sizes considered.

```
4     y=y*y;
5   }
```

Figure 1.1 compares the run times of tangent-linear and adjoint codes for the accumulation of the gradient $\nabla f(\mathbf{x}) \in \mathbb{R}^n$ for increasing values of n. While the run time cost ratio between the adjoint code and an original function evaluation turns out to be independent of n the same quantity grows with n for the tangent-linear code.

1.1.3 Second and Higher Derivative Code

Second derivatives can be computed by second-order tangent-linear and second-order adjoint code. The former is obtained by applying tangent-linear mode AD to a first-order tangent-linear code. The Hessian $\nabla^2 F(\mathbf{x}) \equiv \nabla^2_{\mathbf{x}} F(\mathbf{x}) \in \mathbb{R}^{m \times n \times n}$ can be accumulated at the computational cost of $O(n^2) \cdot Cost(F)$. The computational complexity is the same as that of second-order finite differences. Some run time savings result from the possible exploitation of symmetry and sparsity as in second-order finite differences.

Application of tangent-linear mode AD to a first-order adjoint code yields a second-order adjoint code. The entire Hessian can be accumulated at a computational cost of $O(n \cdot m) \cdot Cost(F)$. For $m = 1$, a single Hessian-vector product can be computed at the computational cost of $O(1) \cdot Cost(F)$, 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.

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 $\mathbf{y} = F(\mathbf{x})$ as described in Section 1.1 induces a directed acyclic graph (DAG) representing the *single assignment code* (SAC)

$$\begin{aligned} &\text{for } j = n, \ldots, n+p+m-1 \\ &\quad v_j = \varphi_j(v_i)_{i \prec j} \quad , \end{aligned} \tag{1.2}$$

where $i \prec j$ denotes a direct dependence of v_j on v_i. Within the SAC the result of each *elemental function* φ_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, $\forall i, j \in V : (i,j) \in E \Rightarrow i < j$.

Example 1 *For $n = 3$ the SAC of the given implementation of (1.1) becomes*

$$\begin{aligned} v_0 &= x_0 \\ v_1 &= x_1 \\ v_2 &= x_2 \\ v_3 &= v_0^2 \\ v_4 &= v_1^2 \\ v_5 &= v_2^2 \\ v_6 &= v_3 + v_4 \\ v_7 &= v_6 + v_5 \\ v_8 &= v_7^2 \\ y &= v_8 \end{aligned}$$

The corresponding DAG is shown in Figure 1.2.

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:

$$\begin{aligned} &\text{for } j = n, \ldots, n+p+m-1 \\ &\quad c_{j,k} = \frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k} \quad \text{for } k \prec j \\ &\quad v_j = \varphi_j(v_i)_{i \prec j} \quad . \end{aligned} \tag{1.3}$$

The linearized SAC induces a linearized DAG where local partial derivatives are attached to the corresponding edges in the DAG.

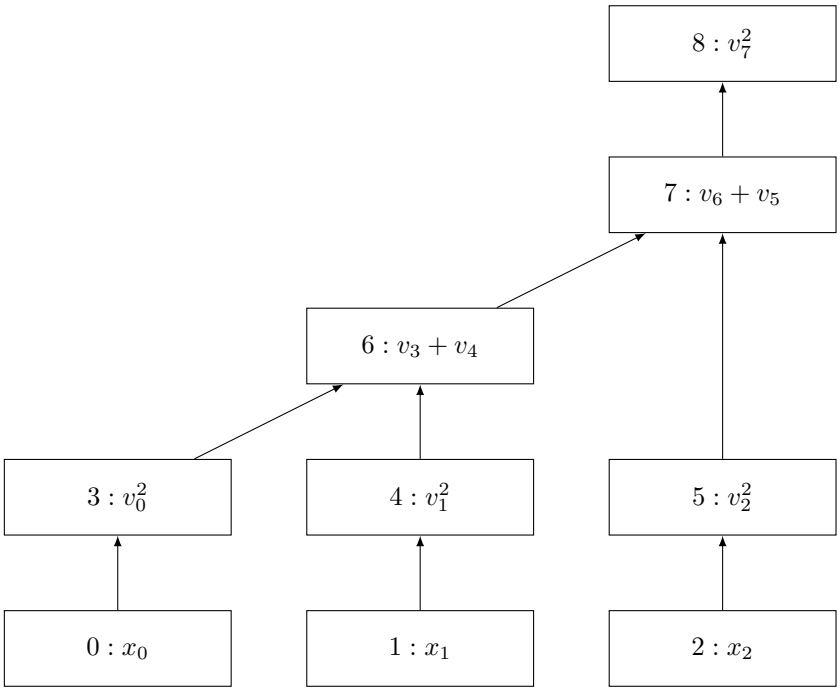

Figure 1.2: DAG of the given implementation of $y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$ for $n = 3$

Example 2 *For $n = 3$ the linearized SAC of the given implementation of (1.1) becomes*

$$\begin{aligned}
v_0 &= x_0 \\
v_1 &= x_1 \\
v_2 &= x_2 \\
c_{3,0} = 2 \cdot v_0;\ v_3 &= v_0^2 \\
c_{4,1} = 2 \cdot v_1;\ v_4 &= v_1^2 \\
c_{5,2} = 2 \cdot v_2;\ v_5 &= v_2^2 \\
c_{6,3} = 1;\ c_{6,4} = 1;\ v_6 &= v_3 + v_4 \\
c_{7,6} = 1;\ c_{7,5} = 1;\ v_7 &= v_6 + v_5 \\
c_{8,7} = 2 \cdot v_7;\ v_8 &= v_7^2 \\
y &= v_8
\end{aligned}$$

The corresponding linearized DAG is shown in Figure 1.3. Local partial derivatives are attached to the edges.

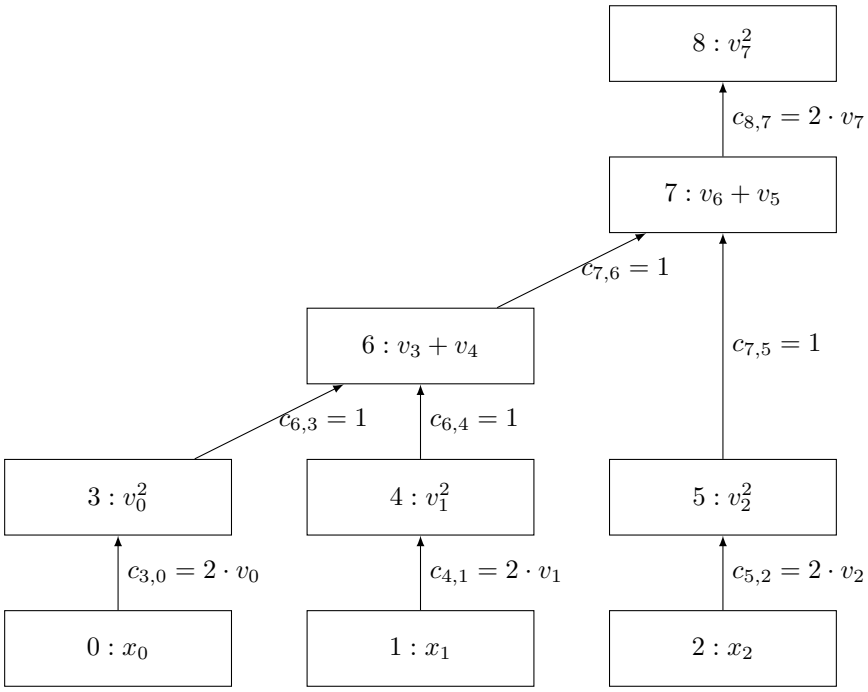

Figure 1.3: Linearized DAG of the given implementation of $y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$ for $n = 3$

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 (1.4):

$$\begin{aligned} &\text{for } j = n, \ldots, n+p+m-1 \\ &\qquad v_j^{(1)} = \sum_{k \prec j} \frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k} \cdot v_k^{(1)} = \sum_{k \prec j} c_{j,k} \cdot v_k^{(1)} \\ &\qquad v_j = \varphi_j(v_i)_{i \prec j} \quad . \end{aligned} \tag{1.4}$$

Implementation by overloading replaces all active floating-point variables[1] v_j with a pair $(v_j, v_j^{(1)})$ consisting of the original value and the associated directional derivative. All arithmetic operators and intrinsic functions are overloaded to implement Equation (1.4).

The Jacobian of $\mathbf{y}$ with respect to $\mathbf{x}$ can be computed by letting $\mathbf{x}^{(1)}$ range over the Cartesian basis vectors in $\mathbb{R}^n$.

[1] ... carrying potentially nonzero directional derivatives

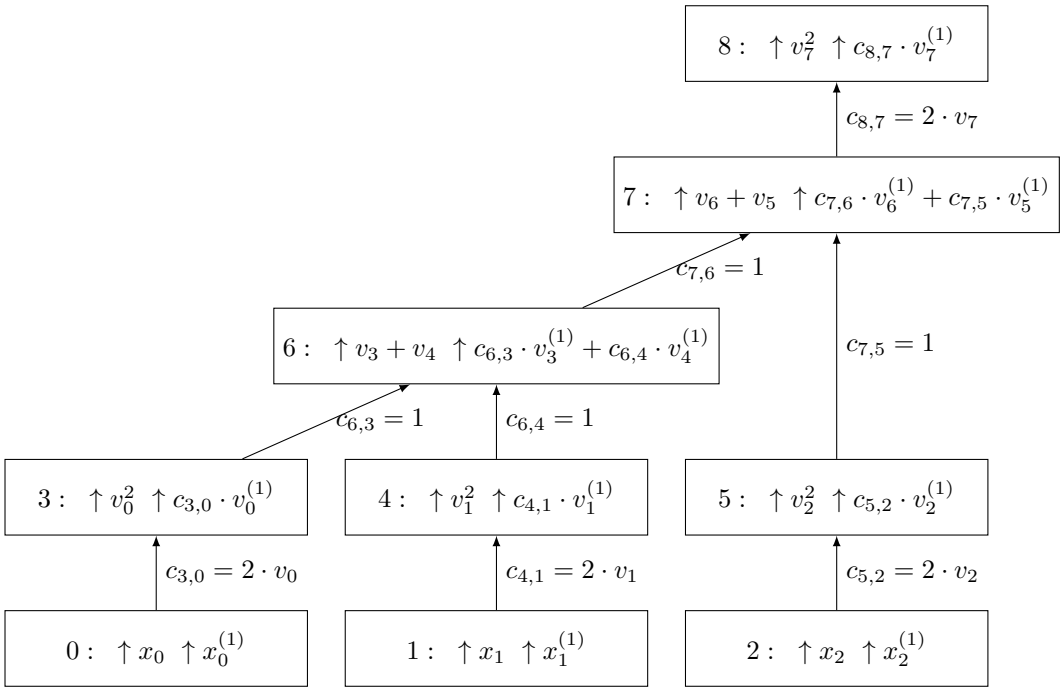

Figure 1.4: Tangent-linear DAG of the given implementation of $y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$ for $n = 3$

Example 3 *For $n = 3$ the tangent-linear SAC of the given implementation of (1.1) becomes*

$$
\begin{aligned}
v_0^{(1)} = x_0^{(1)};\ v_0 = x_0 \\
v_1^{(1)} = x_1^{(1)};\ v_1 = x_1 \\
v_2^{(1)} = x_2^{(1)};\ v_2 = x_2 \\
c_{3,0} = 2 \cdot v_0;\ v_3^{(1)} = c_{3,0} \cdot v_0^{(1)};\ v_3 = v_0^2 \\
c_{4,1} = 2 \cdot v_1;\ v_4^{(1)} = c_{4,1} \cdot v_1^{(1)};\ v_4 = v_1^2 \\
c_{5,2} = 2 \cdot v_2;\ v_5^{(1)} = c_{5,2} \cdot v_2^{(1)};\ v_5 = v_2^2 \\
c_{6,3} = 1;\ c_{6,4} = 1;\ v_6^{(1)} = c_{6,3} \cdot v_3^{(1)} + c_{6,4} \cdot v_4^{(1)};\ v_6 = v_3 + v_4 \\
c_{7,6} = 1;\ c_{7,5} = 1;\ v_7^{(1)} = c_{7,6} \cdot v_6^{(1)} + c_{7,5} \cdot v_5^{(1)};\ v_7 = v_6 + v_5 \\
c_{8,7} = 2 \cdot v_7;\ v_8^{(1)} = c_{8,7} \cdot v_7^{(1)};\ v_8 = v_7^2 \\
y^{(1)} = v_8^{(1)};\ y = v_8 \quad .
\end{aligned}
$$

The corresponding tangent-linear DAG is shown in Figure 1.4. 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:

$$
\begin{aligned}
(v_0, v_0^{(1)}) &= (x_0, x_0^{(1)}) \\
(v_1, v_1^{(1)}) &= (x_1, x_1^{(1)}) \\
(v_2, v_2^{(1)}) &= (x_2, x_2^{(1)}) \\
(v_3, v_3^{(1)}) &= (v_0^2, 2 \cdot v_0^{(1)}) \\
(v_4, v_4^{(1)}) &= (v_1^2, 2 \cdot v_1^{(1)}) \\
(v_5, v_5^{(1)}) &= (v_2^2, 2 \cdot v_2^{(1)}) \\
(v_6, v_6^{(1)}) &= (v_3 + v_4, v_3^{(1)} + v_4^{(1)}) \\
(v_7, v_7^{(1)}) &= (v_6 + v_5, v_6^{(1)} + v_5^{(1)}) \\
(v_8, v_8^{(1)}) &= (v_7^2, 2 \cdot v_7^{(1)}) \\
(y, y^{(1)}) &= (v_8, v_8^{(1)}) \quad .
\end{aligned}
$$

The gradient of the dependent output $y \equiv v_8$ *with respect to the independent input* $\mathbf{x} \equiv (v_0, v_1, v_2)^T$ *can be computed by letting* $\mathbf{x}^{(1)} \equiv (v_0^{(1)}, v_1^{(1)}, v_2^{(1)})^T$ *range over the Cartesian basis vectors in* $\mathbb{R}^3$ *yielding*

$$
\nabla f(\mathbf{x}) = \begin{pmatrix} 4 \cdot v_7 \cdot v_0 \\ 4 \cdot v_7 \cdot v_1 \\ 4 \cdot v_7 \cdot v_2 \end{pmatrix} \quad .
$$

Adjoint Code An adjoint code propagates adjoints $v_{(1)j}$ of all intermediate variables v_j (inner products of $\nabla_{v_j} f(\mathbf{x})$ with $\mathbf{y}_{(1)}$) 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 (1.5):

$$
\begin{aligned}
&\text{for } j = n, \ldots, n+p+m-1 \\
&\qquad v_j = \varphi_j(v_i)_{i \prec j} \\
&\text{for } j = n, \ldots, n+p-1 \\
&\qquad v_{(1)j} = 0 \\
&\quad \text{for } k = n+p+m-1, \ldots, n \\
&\qquad \text{for } j : j \prec k \\
&\qquad\quad v_{(1)j} = v_{(1)j} + \frac{\partial \varphi_k\left((v_i)_{i \prec k}\right)}{\partial v_j} \cdot v_{(1)k} = c_{k,j} \cdot v_{(1)k} \quad .
\end{aligned}
\tag{1.5}
$$

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 [3] 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 φ_j, v_j, and $\{i : i \prec j\}$ for all $j = 0, \ldots, n+p+m-1$ on a *tape*. Adjoints are propagated by reverse interpretation of the tape.

The (transposed) Jacobian of $\mathbf{y}$ with respect to $\mathbf{x}$ can be computed by initializing $\mathbf{x}_{(1)}$ to zero followed by letting $\mathbf{y}_{(1)}$ range over the Cartesian basis vectors in $\mathbb{R}^m$.

Example 4 *For $n = 3$ the adjoint SAC of the given implementation of (1.1) becomes*

$$
\begin{aligned}
v_0 &= x_0 \\
v_1 &= x_1 \\
v_2 &= x_2 \\
v_3 &= v_0^2 \\
v_4 &= v_1^2 \\
v_5 &= v_2^2 \\
v_6 &= v_3 + v_4 \\
v_7 &= v_6 + v_5 \\
v_8 &= v_7^2 \\
y &= v_8 \\
v_{(1)8} &= y_{(1)} \\
c_{8,7} = 2 \cdot v_7;\ v_{(1)7} &= c_{8,7} \cdot v_{(1)8} \\
c_{7,6} = 1;\ v_{(1)6} &= c_{7,6} \cdot v_{(1)7} \\
c_{7,5} = 1;\ v_{(1)5} &= c_{7,5} \cdot v_{(1)7} \\
c_{6,4} = 1;\ v_{(1)4} &= c_{6,4} \cdot v_{(1)6} \\
c_{6,3} = 1;\ v_{(1)3} &= c_{6,3} \cdot v_{(1)6} \\
c_{5,2} = 2 \cdot v_2;\ v_{(1)2} &= c_{5,2} \cdot v_{(1)5} \\
c_{4,1} = 2 \cdot v_1;\ v_{(1)1} &= c_{4,1} \cdot v_{(1)4} \\
c_{3,0} = 2 \cdot v_0;\ v_{(1)0} &= c_{3,0} \cdot v_{(1)3} \\
x_{(1)2} &= v_{(1)2} \\
x_{(1)1} &= v_{(1)1} \\
x_{(1)0} &= v_{(1)0} \quad .
\end{aligned}
$$

The corresponding adjoint DAG is shown in Figure 1.5. 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 overloading generates the following tape:

$$
\begin{aligned}
3:&\ (*, v_3, \{0\}) \\
4:&\ (*, v_4, \{1\}) \\
5:&\ (*, v_5, \{2\}) \\
6:&\ (+, v_6, \{3, 4\}) \\
7:&\ (+, v_7, \{6, 5\}) \\
8:&\ (*, v_8, \{7\})
\end{aligned}
$$

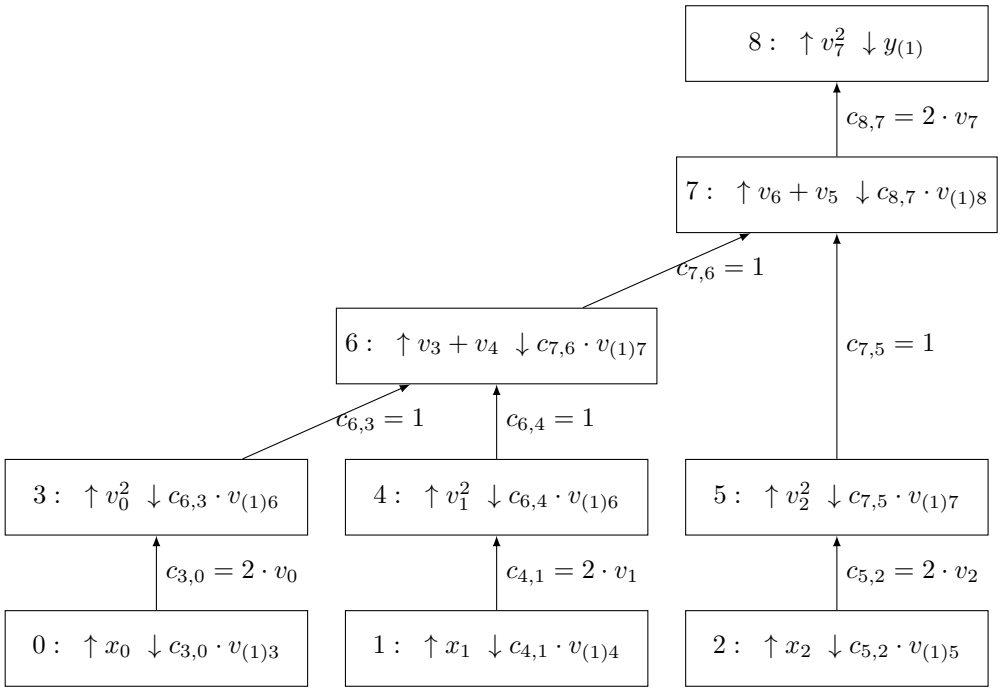

Figure 1.5: Adjoint DAG of the given implementation of $y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$ for $n = 3$

Square operations are represented as unary products. Interpretation yields

$$
\begin{aligned}
v_{(1)7} &= 2 \cdot v_7 \cdot v_{(1)8} \\
v_{(1)6} &= v_{(1)7} \\
v_{(1)5} &= v_{(1)7} \\
v_{(1)4} &= v_{(1)6} \\
v_{(1)3} &= v_{(1)6} \\
v_{(1)2} &= 2 \cdot v_2 \cdot v_{(1)5} \\
v_{(1)1} &= 2 \cdot v_1 \cdot v_{(1)4} \\
v_{(1)0} &= 2 \cdot v_0 \cdot v_{(1)3} \quad .
\end{aligned}
$$

1.2.2 Second and Higher Derivative Code

Application of Equation (1.4) 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 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

AD by Overloading with dco0.9

2.1 Tangent-Linear Code by Overloading with dco 0.9

A very natural and convenient way to implement forward mode AD is by definition of an augmented data type containing $v^{(1)}$ in addition to v for all variables (program variables as well as auxiliary variables generated by the compiler). Directional derivatives are propagated by replacing all arithmetic operations and intrinsic functions with routines for computing both the value and the derivative. A simple type change of all *active* floating-point variables carrying nontrivial derivative information to the new augmented data type is often the only source code modification if the target programming language (such as C++) supports function and operator overloading.[1] For example, $u = v \cdot w$ becomes $(u = v \cdot w, u^{(1)} = v^{(1)} \cdot w + v \cdot w^{(1)})$ and $u = \sin(v)$ is modified into $(u = \sin(v), u^{(1)} = \cos(v) \cdot v^{(1)})$, where u, v, w are floating-point variables.

AD by overloading is implemented by our C++ library `dco 0.9` (derivative code by overloading). The source code is listed in Appendix A. It serves as an illustration of the concepts discussed in Chapter 1. The production version 1.0 features a variety of advanced optimization techniques the discussion of which is beyond the scope of this introductory text. Its performance exceeds that of version 0.9 significantly.

For tangent-linear scalar mode AD, a class dco_t1s_type (`dco`'s tangent-linear 1st-order scalar type) is defined with **double** precision members v (value) and t (tangent).

```
class dco_t1s_type {
  public :
    double v;
    double t;
    dco_t1s_type(const double&);
    dco_t1s_type();
    dco_t1s_type& operator=(const dco_t1s_type&);
};
```

A special constructor (dco_t1s_type(**const double**&)) converts passive into active variables at run time. The provided standard constructor simply initializes the value and derivative components to zero. The assignment operator returns a copy of the right-hand side unless it is aliased with the left-hand side of the assignment.

```
dco_t1s_type& dco_t1s_type::operator=(const dco_t1s_type& x) {
  if (this==&x) return *this;
  v=x.v; t=x.t;
```

[1] More substantial modifications may become necessary in languages that do not have full support for object-oriented programming.

```
    return *this;
  }
```

Implementations of all relevant arithmetic operators and intrinsic functions are required, for example,

```
  dco_t1s_type operator*(const dco_t1s_type& x1,
                         const dco_t1s_type& x2) {
    dco_t1s_type tmp;
    tmp.v=x1.v*x2.v;
    tmp.t=x1.t*x2.v+x1.v*x2.t;
    return tmp;
  }
```

and

```
  dco_t1s_type sin(const dco_t1s_type& x) {
    dco_t1s_type tmp;
    tmp.v=sin(x.v);
    tmp.t=cos(x.v)*x.t;
    return tmp;
  }
```

Refer to Appendix A.1 for a more complete version of the source code. The driver program in Listing 2.1 uses the implementation of **class** dco_t1s_type to compute the gradient of Equation (1.1) for n = 4 at the point $x_i = 1$ for $i = 0, \ldots, 3$. Four evaluations of the tangent-linear routine

```
void f(dco_t1s_type *x, dco_t1s_type &y)
```

are performed with the derivative components of x initialized to the Cartesian basis vectors in $\mathbb{R}^4$.

Listing 2.1: Driver for Tangent-Linear Code by Overloading

```
#include<iostream>
using namespace std;
#include "dco_t1s_type.hpp"

const int n=4;

void f(dco_t1s_type *x, dco_t1s_type &y) {
  y=0;
  for (int i=0;i<n;i++) y=y+x[i]*x[i];
  y=y*y;
}

int main() {
  dco_t1s_type x[n], y;
  for (int i=0;i<n;i++) x[i]=1
  for (int i=0;i<n;i++) {
    x[i].t=1;
    f(x,y);
    x[i].t=0;
    cout << y.t << endl;
  }
  return 0;
}
```

Let **class** dco_t1s_type be defined in the C++ source files `dco_t1s_type.hpp` and `dco_t1s_type.cpp`, and let the driver program be stored as `main.cpp`. An executable is built by calling

```
$(CPPC) -c dco_t1s_type.cpp
$(CPPC) -c main.cpp
$(CPPL) -o main dco_t1s_type.o main.o
```

where `$(CPPC)` and `$(CPPL)` should be replaced by a C++ compiler and a corresponding linker, respectively (for example, `g++`).

2.2 Adjoint Code by Overloading with dco 0.9

The favored approach to a run-time version of the adjoint code is to build a *tape* (an augmented representation of the DAG) by overloading, followed by an interpretative reverse propagation of adjoints through the tape. In our case the tape is a statically allocated array of tape entries addressed by their position in the array. Each tape entry contains a code for the associated operation (oc), addresses of the operation's first and optional second arguments (arg1 and arg2), and two floating-point variables holding the current value (v) and the adjoint (a), respectively. The constructor marks the operation code and both arguments as undefined and it initializes both the value and the adjoint to zero.

```
class dco_a1s_tape_entry {
  public:
    int oc, arg1, arg2;
    double v,a;
    dco_a1s_tape_entry() :
      oc(DCO_A1S_UNDEF), arg1(DCO_A1S_UNDEF),
      arg2(DCO_A1S_UNDEF), v(0), a(0)
    {};
};
```

As in forward mode, an augmented data type is defined to replace the type of every active floating-point variable. The corresponding **class** dco_a1s_type (dco's adjoint 1st-order scalar type) contains the virtual address va (position in tape) of the current variable in addition to its value v.

```
class dco_a1s_type {
  public:
    int va;
    double v;
    dco_a1s_type() : va(DCO_A1S_UNDEF), v(0) {};
    dco_a1s_type(const double&);
    dco_a1s_type& operator=(const dco_a1s_type&);
};
```

Special constructors and a custom assignment operator are required. The latter either handles a self-assignment or generates a new tape entry with corresponding operation code and with copies of the right-hand side's value and virtual address. A global virtual address counter dco_a1s_vac is used to populate the tape.

```
dco_a1s_type& dco_a1s_type::operator=(const dco_a1s_type& x) {
  if (this==&x) return *this;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_ASG;
  dco_a1s_tape[dco_a1s_vac].v=v=x.v;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
```

```
  va=dco_a1s_vac++;
  return *this;
}
```

Passive values and constants are activated by a special constructor:

```
dco_a1s_type::dco_a1s_type(const double& x): v(x) {
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_CONST;
  dco_a1s_tape[dco_a1s_vac].v=x;
  va=dco_a1s_vac++;
};
```

All arithmetic operators and intrinsic functions make similar recordings on the tape, for example,

```
dco_a1s_type operator*(const dco_a1s_type& x1, const dco_a1s_type& x2)
    {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_MUL;
  dco_a1s_tape[dco_a1s_vac].arg1=x1.va;
  dco_a1s_tape[dco_a1s_vac].arg2=x2.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=x1.v*x2.v;
  tmp.va=dco_a1s_vac++;
  return tmp;
}
```

and

```
dco_a1s_type sin(const dco_a1s_type& x) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_SIN;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=sin(x.v);
  tmp.va=dco_a1s_vac++;
  return tmp;
}   .
```

The operation codes are implemented as macros (e.g., DCO_A1S_ASG, DCO_A1S_MUL, and DCO_A1S_SIN) to be replaced with some unique number by the C preprocessor.

The tape is constructed during a single execution of the overloaded original code; this is followed by an interpretation step for propagating adjoints through the tape in reverse order.

```
void dco_a1s_interpret_tape () {
  for (int i=dco_a1s_vac;i>=0;i--) {
    switch (dco_a1s_tape[i].oc) {
      case DCO_A1S_ASG : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_MUL : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=
          dco_a1s_tape[dco_a1s_tape[i].arg2].v*dco_a1s_tape[i].a;
        dco_a1s_tape[dco_a1s_tape[i].arg2].a+=
          dco_a1s_tape[dco_a1s_tape[i].arg1].v*dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_SIN : {
```

Tape:

```
 0: [ 0, -1, -1,  1.0, 0.0 ]
 1: [ 1,  0, -1,  1.0, 0.0 ]
 2: [ 0, -1, -1,  1.0, 0.0 ]
 3: [ 1,  2, -1,  1.0, 0.0 ]
 4: [ 0, -1, -1,  1.0, 0.0 ]
 5: [ 1,  4, -1,  1.0, 0.0 ]
 6: [ 0, -1, -1,  1.0, 0.0 ]
 7: [ 1,  6, -1,  1.0, 0.0 ]
 8: [ 0, -1, -1,  0.0, 0.0 ]
 9: [ 1,  8, -1,  0.0, 0.0 ]
10: [ 4,  1,  1,  1.0, 0.0 ]
11: [ 2,  9, 10,  1.0, 0.0 ]
12: [ 1, 11, -1,  1.0, 0.0 ]
13: [ 4,  3,  3,  1.0, 0.0 ]
14: [ 2, 12, 13,  2.0, 0.0 ]
15: [ 1, 14, -1,  2.0, 0.0 ]
16: [ 4,  5,  5,  1.0, 0.0 ]
17: [ 2, 15, 16,  3.0, 0.0 ]
18: [ 1, 17, -1,  3.0, 0.0 ]
19: [ 4,  7,  7,  1.0, 0.0 ]
20: [ 2, 18, 19,  4.0, 0.0 ]
21: [ 1, 20, -1,  4.0, 0.0 ]
22: [ 4, 21, 21, 16.0, 0.0 ]
23: [ 1, 22, -1, 16.0, 0.0 ]
```

(a)

Interpreted Tape:

```
[ 0, -1, -1,  1.0, 16.0 ]
[ 1,  0, -1,  1.0, 16.0 ]
[ 0, -1, -1,  1.0, 16.0 ]
[ 1,  2, -1,  1.0, 16.0 ]
[ 0, -1, -1,  1.0, 16.0 ]
[ 1,  4, -1,  1.0, 16.0 ]
[ 0, -1, -1,  1.0, 16.0 ]
[ 1,  6, -1,  1.0, 16.0 ]
[ 0, -1, -1,  0.0,  8.0 ]
[ 1,  8, -1,  0.0,  8.0 ]
[ 4,  1,  1,  1.0,  8.0 ]
[ 2,  9, 10,  1.0,  8.0 ]
[ 1, 11, -1,  1.0,  8.0 ]
[ 4,  3,  3,  1.0,  8.0 ]
[ 2, 12, 13,  2.0,  8.0 ]
[ 1, 14, -1,  2.0,  8.0 ]
[ 4,  5,  5,  1.0,  8.0 ]
[ 2, 15, 16,  3.0,  8.0 ]
[ 1, 17, -1,  3.0,  8.0 ]
[ 4,  7,  7,  1.0,  8.0 ]
[ 2, 18, 19,  4.0,  8.0 ]
[ 1, 20, -1,  4.0,  8.0 ]
[ 4, 21, 21, 16.0,  1.0 ]
[ 1, 22, -1, 16.0,  1.0 ]
```

(b)

Figure 2.1: dco_t2s_a1s_tape for the Computation of the Gradient of Equation (1.1) for n = 4 at the point $x_i = 1$ for $i = 0, \ldots, 3$; The five columns show for each tape entry with virtual addresses from 0 to 23 the operation code, the virtual addresses of the (up to two) arguments, the function value, and the adjoint value, where -1 ≡ *DCO_A1S_UNDEF* in the second and third columns and with operation codes 0 ≡ *DCO_A1S_CONST*, 1 ≡ *DCO_A1S_ASG*, 2 ≡ *DCO_A1S_ADD*, and 4 ≡ *DCO_A1S_MUL*.

```
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=
          cos(dco_a1s_tape[dco_a1s_tape[i].arg1].v)
            *dco_a1s_tape[i].a;
        break;
      }
      ...
    }
  }
}
```

The driver program in Listing 2.2 uses the implementation of **class** dco_a1s_type in connection with a tape of size DCO_A1S_TAPE_SIZE (to be replaced with an integer value by the C preprocessor). The tape is allocated statically in `dco_a1s_type.cpp` and is later linked to the object code of the driver program. The latter computes the gradient of the given implementation of Equation (1.1) for n = 4 at the point $x_i = 1$ for $i = 0, \ldots, 3$. Running the augmented function

```
void f(dco_a1s_type *x, dco_a1s_type &y)
```

followed by the tape interpretation yields the two tapes in Figure 2.1. Arguments are referenced by their virtual address within the tape. For example, tape entry 11 represents the sum (oc=2) of the two arguments represented by tape entries 9 and 10. The tape is structurally equivalent to the DAG. The propagation of adjoints is preceeded by the initialization of the adjoint of the

tape entry that corresponds to the dependent variable y (tape entry 23). The desired gradient is accumulated in the adjoint components of the four tape entries 1, 3, 5, and 7.

Listing 2.2: Driver for Adjoint Code by Overloading

```
1  #include <iostream>
2  #include "dco_a1s_type.hpp"
3  using namespace std;
4
5  const int n=4;
6
7  extern dco_a1s_tape_entry dco_a1s_tape[DCO_A1S_TAPE_SIZE];
8
9  void f(dco_a1s_type *x, dco_a1s_type &y) {
10   y=0;
11   for (int i=0;i<n;i++) y=y+x[i]*x[i];
12   y=y*y;
13 }
14
15 int main() {
16   dco_a1s_type x[n], y;
17   for (int i=0;i<n;i++) x[i]=1;
18   f(x,y);
19   dco_a1s_tape[y.va].a=1;
20   dco_a1s_interpret_tape();
21   for (int i=0;i<n;i++)
22     cout << i << "\t" << dco_a1s_tape[x[i].va].a << endl;
23   return 0;
24 }
```

Tape entries 0–7 correspond to the initialization of the x[j] in line 19. The initialization of y inside of f (line 10) yields tape entries 8 and 9. The loop in line 11 produces the following twelve (four triplets) entries 10–21. Squaring y in line 12 adds the last two tape entries 22 and 23.

The tape interpreter implements Equation (1.5) without modification. Starting from tape entry 23, the adjoint value 1 of the dependent variable y is propagated to the single argument of the underlying assignment. The adjoint of tape entry 22 is set to 1 as the local partial derivative of an assignment is equal to 1. Tape entry 22 represents the multiplication y=y∗y in line 12 of Listing 2.2, where the value of y on the right-hand side of the assignment is represented by tape entry 21. The value of the local partial derivative (2∗y=2*4=8) is multiplied with the adjoint of tape entry 22, followed by incrementing the adjoint of tape entry 21, whose initial value is equal to 0. This process continues until all tape entries have been visited. The gradient can be retrieved from tape entries 1, 3, 5, and 7. If none of the independent variables is overwritten, then their va components contain the correct virtual addresses after calling the overloaded version of f. This is the case in the given example. Hence, lines 19–22 deliver the correct gradient in Listing 2.2. Otherwise, the virtual addresses of the independent variables need to be stored in order to ensure a correct retrieval of the gradient. Listings of the full source code that implements adjoint mode AD by overloading can be found in Appendix A.2. If both **class** dco_a1s_type and the tape are implemented in the files `dco_a1s_type.hpp` and `dco_a1s_type.cpp`, and if the driver program is stored as `main.cpp`, then the build process is similar to that in Section 2.1.

Chapter 3

AD by Source Transformation with `dcc 0.9`

3.1 Functionality

`dcc 0.9` generates jth-order derivative code by arbitrary combinations of tangent-linear or adjoint modes. It takes a possibly preprocessed $(j-1)$th-order derivative code generated by itself as input. The original (0th-order derivative) code is expected to be written in a well-defined subset of C++ that is intentionally kept small. Still the accepted syntax and semantics are rich enough to be able to illustrate the topics discussed in [2] See Appendix B for a summary of the syntax accepted by `dcc 0.9`.

`dcc 0.9` operates on implementations of multivariate vector functions

$$F : \mathbb{R}^n \to \mathbb{R}^m, \quad \mathbf{y} = F(\mathbf{x}) \quad ,$$

as subroutines

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

Its results vary depending on whether certain inputs and outputs are *aliased* (represented by the same program variable) or not. Hence, the two cases

$$\mathbf{y} = F(\mathbf{x}) \quad \text{and} \quad \begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} = F(\mathbf{x}, \mathbf{z})$$

($\mathbf{x}$ and $\mathbf{y}$ unaliased) are considered separately in [2]. For $\mathbf{y} = F(\mathbf{x})$ and $\mathbf{x}$ and $\mathbf{y}$ not aliased the generated derivative code behaves similar to what has been presented Chapter 1.

3.1.1 Tangent-Linear Code by `dcc 0.9`

The tangent-linear version

$$F^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^m : \quad \begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(1)} \end{pmatrix} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$$

of the given implementation of $\mathbf{y} = F(\mathbf{x})$ computes

$$\begin{aligned} \mathbf{y}^{(1)} &= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \\ \mathbf{y} &= F(\mathbf{x}) \quad . \end{aligned}$$

For a given implementation of F as

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

dcc 0.9 generates a tangent-linear subroutine with the following signature:

```
void t1_f(int n, int m, double *x, double *t1_x,
                        double *y, double *t1_y)     .
```

All superscripts of the tangent-linear subroutine and variable names are replaced with the prefix t1_, that is, $\mathbf{v}^{(1)} \equiv$ t1_v.

3.1.2 Adjoint Code by dcc 0.9

Due to missing data flow analysis, dcc 0.9 cannot decide if a value that is overwritten within the forward section of the adjoint code is required by the reverse section. Conservatively, it stores all overwritten values on appropriately typed required data stacks. Hence, the straightforward application of reverse mode with data flow reversal stack s to $\mathbf{y} = F(\mathbf{x})$ yields

$$\begin{aligned} s[0] = \mathbf{y};\ \mathbf{y} &= F(\mathbf{x}) \\ \mathbf{y} = s[0];\ \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)} \\ \mathbf{y}_{(1)} &= 0 \quad . \end{aligned}$$

The adjoint code generated by dcc 0.9 does not return the correct function value $\mathbf{y}$ of F. It rather restores the (possibly undefined) input value of $\mathbf{y}$. To return the correct function value, code for storing a result checkpoint r must be provided by the user to save the value of $\mathbf{y}$ after the augmented forward sweep followed by recovering it after the reverse sweep:

$$\begin{aligned} s[0] = \mathbf{y};\ \mathbf{y} &= F(\mathbf{x}) \\ r[0] &= \mathbf{y} \\ \mathbf{y} = s[0];\ \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)} \\ \mathbf{y}_{(1)} &= 0 \\ \mathbf{y} &= r[0] \quad . \end{aligned}$$

Result checkpointing in dcc 0.9 will be discussed in further detail in Section 3.4.2. Hence, the adjoint

$$\begin{aligned} F_{(1)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m &\to \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^m : \\ \begin{pmatrix} \mathbf{x}_{(1)} \\ \mathbf{y} \\ \mathbf{y}_{(1)} \end{pmatrix} &= F_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)}) \end{aligned}$$

of an implementation of $\mathbf{y} = F(\mathbf{x})$ computes

$$\begin{aligned} \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)} \\ \mathbf{y} &= F(\mathbf{x}) \\ \mathbf{y}_{(1)} &= 0 \quad . \end{aligned}$$

For the given implementation of F as

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

dcc 0.9 generates an adjoint subroutine with the following signature:

```
void a1_f(int a1_mode, int n, int m,
                       double *x, double *a1_x,
                       double *y, double *a1_y)     .
```

All subscripts of the adjoint subroutine and variable names are replaced with the prefix a1_, that is, $\mathbf{v}_{(1)} \equiv$ a1_v. The integer parameter a1_mode selects between various modes required in the context of interprocedural adjoint code. Details will be discussed in Section 3.3.

3.2 Installation of dcc 0.9

The compiler has been tested on various Linux platforms. Its installation files come as a compressed tar archive file dcc-0.9.tar.gz. It is unpacked into a subdirectory ./dcc-0.9, e.g., by running

```
tar -xzvf dcc-0.9.tar.gz    .
```

To build the compiler enter the subdirectory ./dcc-0.9 and type

```
./configure --prefix=$(INSTALL_DIR)
make
make check
make install
```

The executable dcc can be found in $(INSTALL_DIR)/bin.

make check runs the compiler in both supported modes (tangent-linear and adjoint) on test input code stored in subdirectories of ./dcc-0.9/src/tests. The generated output is verified against a reference. An error message is generated for anything but identical matches.

Precompiled Windows binaries are available on www.siam.org/se24.

3.3 Use of dcc 0.9

Let the original source code reside in a file named f.c in subdirectory $(SRC_DIR) and let the top-level directory of the dcc 0.9 installation be $(DCC_DIR).

A first-order tangent-linear code is built in $(SRC_DIR) by typing

```
$(DCC_DIR)/dcc f.c 1 1
```

The name of the source file f.c is followed by two command-line parameters for setting tangent-linear mode (1) and the order of the derivative (1). The generated code is stored in a file named t1_f.c.

A first-order adjoint code is built in $(SRC_DIR) by typing

```
$(DCC_DIR)/dcc f.c 2 1
```

The first-order (third command-line parameter set to 1) adjoint (second command-line parameter set to 2) version of the code in f.c is stored in a file named a1_f.c.

Higher-order derivative code can be obtained by reapplying dcc 0.9 to a previously generated derivative code in either tangent-linear or adjoint mode. Reapplication of dcc 0.9 to a previously generated adjoint code a1_f.c requires running the C preprocessor on a1_f.c first as described in [2]. For example, the second-order adjoint code t2_a1_f.c results from running

```
$(DCC_DIR)/dcc a1_f.c 1 2
```

on the preprocessed version of a1_f.c. A third-order derivative code can be generated, for example, by running

```
$(DCC_DIR)/dcc t2_a1_f.c 2 3
```

The result is stored in a3_t2_a1_f.c. While reapplication of dcc 0.9 in adjoint mode to a previously generated first- or higher-order adjoint model is feasible, this feature is less likely to be used in practice for reasons outlined in [2]. A third-order adjoint model is best generated by running

```
$(DCC_DIR)/dcc t2_a1_f.c 1 3   .
```

3.4 Intraprocedural Derivative Code by dcc 0.9

Consider a file f.c with the following content

```
// a very simple input code
void f(double& x, double& y) {
  y=sin(x);
}
```

dcc 0.9 expects all **double** parameters to be passed by reference. Call by value is supported for integer parameters only. Single-line comments are not preserved in the output code. We use this trivial input code to take a closer look at the result of the semantic transformations performed by dcc 0.9. Larger inputs result in tangent-linear and adjoint code whose listing becomes unreasonable due to excessive length.

3.4.1 Tangent-Linear Code

The name t1_f of the tangent-linear routine is generated by prepending the prefix t1_ to the name of the original routine. The original parameter list is augmented with dummy variables holding directional derivatives of all **double** parameters. Both x and y receive respective tangent-linear versions t1_x and t1_y in line 1 of the following code listing.

```
1  void t1_f(double& x, double& t1_x, double& y, double& t1_y)
2  {
3      double v1_0=0;
4      double t1_v1_0=0;
5      double v1_1=0;
6      double t1_v1_1=0;
7      t1_v1_0=t1_x;
8      v1_0=x;
9      t1_v1_1=cos(v1_0)*t1_v1_0;
10     v1_1=sin(v1_0);
11     t1_y=t1_v1_1;
12     y=v1_1;
13 }
```

The original assignment is decomposed into the SAC (single assignment code; see Chapter 1) listed in lines 8, 10, and 12. Two auxiliary SAC variables v1_0 and v1_1 are declared in lines 3 and 5. dcc 0.9 expects a separate declaration for each variable as well as its initialization with some constant (e.g. 0). Tangent-linear versions of both auxiliary variables are declared and initialized in lines 4 and 6. All three SAC statements are augmented with local tangent-linear models (lines 7, 9, and 11).

Auxiliary variable names are built from the base string v by appending the order of differentiation (1) and a unique counter (0, 1, ...) separated by an underscore. Potential name clashes with variables present in the original source code could be avoided by a more sophisticated naming

strategy. dcc 0.9 does not support such a mechanism. Its source code would need to be edited in order to replace the base string v with some alternative. The native C++ compiler can be expected to eliminate most auxiliary variables as the result of copy propagation.

A driver program/function must be supplied by the user, for example,

```
1  #include<fstream>
2  #include<cmath>
3  using namespace std;
4
5  #include "t1_f.c"
6
7  int main() {
8    ofstream t1_out("t1.out");
9    double x=1, t1_x=1;
10   double y, t1_y;
11   t1_f(x, t1_x, y, t1_y);
12   t1_out << y << " " << t1_y << endl;
13   return 0;
14 }
```

It computes the partial derivative of the output y with respect to the input x at point x=1. Relevant parts of the C++ standard library are used for file i/o (fstream) and to provide an implementation for the intrinsic sine function (cmath). Global use of the std namespace is crucial as dcc 0.9 does neither accept nor generate namespace prefixes such as std::. The file t1_f.c is included into the driver in line 5 in order to make these preprocessor settings applicable to the tangent-linear output of dcc 0.9. Both the values of x and of its directional derivative t1_x are set to one at the time of their declaration in lines 9 and 10 followed by declarations of the outputs y and t1_y and the call of the tangent-linear function t1_f in lines 11 and 12, respectively. The results are written into the file t1.out for later validation. Optimistically, zero is returned to indicate an error-free execution of the driver program.

3.4.2 Adjoint Code

The adjoint routine a1_f has been edited slightly by removing parts without relevance to the intraprocedural case. Its signature is left unchanged despite the fact that the integer input parameter a1_mode could also be omitted in this situation.

```
1  int cs[10];
2  int csc=0;
3  double fds[10];
4  int fdsc=0;
5  int ids[10];
6  int idsc=0;
7  #include "declare_checkpoints.inc"
8
9  void a1_f(int a1_mode, double& x, double& a1_x,
10                        double& y, double& a1_y)
11 {
12    double v1_0=0;
13    double a1_v1_0=0;
14    double v1_1=0;
15    double a1_v1_1=0;
16    if (a1_mode==1) {
17      cs[csc]=0; csc=csc+1;
```

```
18        fds[fdsc]=y; fdsc=fdsc+1; y=sin(x);
19  #include "f_store_results.inc"
20        while (csc>0) {
21          csc=csc-1;
22          if (cs[csc]==0) {
23            fdsc=fdsc-1; y=fds[fdsc];
24            v1_0=x;
25            v1_1=sin(v1_0);
26            a1_v1_1=a1_y; a1_y=0;
27            a1_v1_0=cos(v1_0)*a1_v1_1;
28            a1_x=a1_x+a1_v1_0;
29          }
30        }
31  #include "f_restore_results.inc"
32      }
33  }
```

The adjoint function needs to be called in *first-order adjoint calling mode* a1_mode=1 to invoke the propagation of adjoints from the adjoint output a1_y to the adjoint input a1_x. Further calling modes will be added when considering call tree reversal in the interprocedural case in Section 3.5.

An augmented version of the original code enumerates basic blocks in the order of their execution (line 17; ref. Adjoint Code Generation Rule 5 in [2]) and it saves left-hand sides of assignments before they get overwritten (line 18; ref. Adjoint Code Generation Rule 3 in [2]). Three global stacks are declared for this purpose with default sizes set to 10 to be adapted by the user. The sizes of both the control flow stack cs and the required floating-point data stack fds can be reduced to 1 in the given example. Counter variables csc and fdsc are declared as references to the tops of the respective stacks. Missing integer assignments make the required integer data stack ids in line 5 as well as its counter variable idsc in line 6 obsolete. Code for allocating memory required for the potential storage of argument and/or result checkpoints needs to be provided by the user in a file named `declare_checkpoints.inc`. In `dcc 0.9`, all memory required for the data-flow reversal is allocated globally. Related issues such as thread safety of the generated adjoint code are the subject of ongoing research and development.

The reverse section of the adjoint code (lines 20 to 30) runs the adjoint basic blocks in reverse order driven by their indices retrieved one by one from the top of the control stack (lines 20 to 22). Processing of the original assignments within a basic block in reverse order starts with the recovery of the original value of the variable on the left-hand side of the assignment (line 23). An incomplete version of the assignment's SAC (without storage of the value of the right-hand-side expression in the variable on the left-hand side of the original assignment; lines 24 and 25; ref. Adjoint Code Generation Rule 4 in [2]) is built to ensure availability of all arguments of local partial derivatives potentially needed by the adjoint SAC (lines 26 to 28). The corresponding auxiliary SAC variables and their adjoints are declared in lines 12 to 15 (ref. Adjoint Code Generation Rule 1 in [2]). `dcc 0.9` expects all local variables to be initialized, e.g. to zero. Adjoints of variables present in the original code are incremented (line 28) while adjoints of (single-use) auxiliary variables are overwritten (lines 26 and 27). Adjoints of left-hand sides of assignments are set to zero after their use by the corresponding adjoint SAC statement (line 26; ref. Adjoint Code Generation Rule 2 in [2]).

The user is given the opportunity to ensure the return of the correct original function value through provision of three appropriate files to be included into the adjoint code. By default, the data flow reversal mechanism restores the input values of all parameters. For example, one could store y

```
rescp=y;
```

in `f_store_results.inc` and recover it

```
y=rescp;
```

in `f_restore_results.inc` in addition to the declaration and initialization of the checkpoint

```
double rescp=0;
```

in `declare_checkpoints.inc`. Automation of this kind of *checkpointing* is impossible if arrays are passed as pointer parameters due to missing size information in C/C++.

The determination of sufficiently large stack sizes may turn out to be not entirely trivial. For given values of the inputs, one could check the maxima of the stack counters csc, fdsc, and idsc by insertion of

```
cout << csc << " " << fdsc << " " << ids << endl;
```

in between the augmented forward and reverse sections of the adjoint code (right before or after line 19).

Again, a driver program/function needs to be supplied by the user. For our simple example, it looks very similar to the tangent-linear driver discussed in Section 3.4.1.

```
1  #include<fstream>
2  #include<cmath>
3  using namespace std;
4
5  #include "a1_f.c"
6
7  int main() {
8    ofstream a1_out("a1.out");
9    double x=1, a1_x=0;
10   double y, a1_y=1;
11   a1_f(1,x, a1_x, y, a1_y);
12   a1_out << y << " " << a1_x << endl;
13   return 0;
14 }
```

To compute the partial derivative of y with respect to x at point x=1, the value a1_y of the adjoint of the output is set to one while the adjoint a1_x of the input needs to be initialized to zero. The correct calling mode (1) is passed to the adjoint function a1_f in line 11. In line 12, the result a1_x is written to a file for later validation. Compilation of this driver followed by linking with the C++ standard library yields a program whose execution generates the same output as the tangent-linear driver in Section 3.4.1. A typical correctness check comes in the form of a comparison of the results obtained from the tangent-linear and adjoint code, for example running

```
diff t1.out a1.out   .
```

3.5 Interprocedural Derivative Code by `dcc 0.9`

For the generation of interprocedural derivative code, `dcc 0.9` expects all subroutines to be provided in a single file, for example,

```
void g(double& x) {
  x=sin(x);
}

void f(double& x, double& y) {
  g(x);
```

```
  y=sqrt(x);
}
```

This code implements a univariate vector function $x \mapsto (x, y)$.

3.5.1 Tangent-Linear Code

The generated tangent-linear code does not yield any surprises.

```
1  void t1_g(double& x, double& t1_x)
2  {
3      double v1_0=0;
4      double t1_v1_0=0;
5      double v1_1=0;
6      double t1_v1_1=0;
7      t1_v1_0=t1_x;
8      v1_0=x;
9      t1_v1_1=cos(v1_0)*t1_v1_0;
10     v1_1=sin(v1_0);
11     t1_x=t1_v1_1;
12     x=v1_1;
13 }
14 void t1_f(double& x, double& t1_x,
15            double& y, double& t1_y)
16 {
17     double v1_0=0;
18     double t1_v1_0=0;
19     double v1_1=0;
20     double t1_v1_1=0;
21     t1_g(x,t1_x);
22     t1_v1_0=t1_x;
23     v1_0=x;
24     t1_v1_1=1/(2*sqrt(v1_0))*t1_v1_0;
25     v1_1=sqrt(v1_0);
26     t1_y=t1_v1_1;
27     y=v1_1;
28 }
```

The original call of g is replaced by its tangent-linear version t1_g in line 21. Copy propagation (elimination of auxiliary variables) and the elimination of common subexpressions (for example, sqrt(v1_0) in lines 24 and 25) is again left to the native C++ compiler.

3.5.2 Adjoint Code

dcc 0.9 generates *fully joint call tree reversals*; see [2] and [1] for details. Due to the considerable length of the automatically generated interprocedural adjoint code, we split the listing into three parts.

The global declarations of required data and control stacks are followed by two **#include** C-preprocessor directives in lines 7 and 8 of the following listing.

```
1  int cs[10];
2  int csc=0;
3  double fds[10];
4  int fdsc=0;
```

```
5  int ids[10];
6  int idsc=0;
7  #include "declare_checkpoints.inc"
8  #include "f.c"
```

It is the user's responsibility to declare sufficiently large stacks. Moreover, name clashes with variables declared in the original program must be avoided. The preset sizes (here 10) need to be adapted accordingly. The file `declare_checkpoints.inc` is extended with variable declarations required for the implementation of the subroutine argument checkpointing scheme in joint call tree reversal mode. For example,

```
double rescp=0;
double argcp=0;
```

allocates memory for storing the input value x of g that is needed for running g "out of context" in joint call tree reversal mode. As in Section 3.4.2 these declarations need to be supplied by the user since the problem of generating correct checkpoints for C++-code is statically undecidable. Sizes of vector arguments passed as pointers are generally unknown due to missing array descriptors. While the scalar case could be treated automatically it is probably not worth the effort since in numerical simulation code handled by `dcc 0.9` most subroutine arguments are arrays. The inclusion of the original code in line 8 is necessary as g is called within the augmented forward section of the adjoint version a1_f of f.

Adjoint subroutines can be called in three modes selected by setting the integer parameter a1_mode . The prefix a1 indicates the order of differentiation. For example, if a third-order adjoint code is generated by reverse-over-forward-over-forward mode, then the name of this parameter becomes a3_mode. Let us first take a closer look at a1_g.

```
1  void a1_g(int a1_mode, double& x, double& a1_x) {
2     double v1_0=0;
3     double a1_v1_0=0;
4     double v1_1=0;
5     double a1_v1_1=0;
6     int save_csc=0;
7     save_csc=csc;
8     if (a1_mode==1) {
9        // augmented forward section
10       cs[csc]=0; csc=csc+1;
11       fds[fdsc]=x; fdsc=fdsc+1; x=sin(x);
12 #include "g_store_results.inc"
13       // reverse section
14       while (csc>save_csc) {
15         csc=csc-1;
16         if (cs[csc]==0) {
17           fdsc=fdsc-1; x=fds[fdsc];
18           v1_0=x;
19           v1_1=sin(v1_0);
20           a1_v1_1=a1_x; a1_x=0;
21           a1_v1_0=cos(v1_0)*a1_v1_1;
22           a1_x=a1_x+a1_v1_0;
23         }
24       }
25 #include "g_restore_results.inc"
26    }
27    if (a1_mode==2) {
28 #include "g_store_inputs.inc"
```

```
29          a1_mode=a1_mode;
30        }
31        if (a1_mode==3) {
32    #include "g_restore_inputs.inc"
33          a1_mode=a1_mode;
34        }
35    }
```

The three modes are represented by three **if** statements in lines 8, 27, and 31. If the adjoint subroutine is called with a1_mode set equal to one, then an adjoint code that is similar to the one discussed in Section 3.4.2 is executed. Note that the control flow reversal uses an additional auxiliary variable save_csc to store the state of the control stack counter csc in line 7 followed by stepping through the local csc − save_csc adjoint basic blocks in lines 14–24. Code for storage and recovery of g's results needs to be supplied by the user.

The two remaining adjoint calling modes invoke user-supplied code for storage (a1_mode==2) and recovery (a1_mode==3) of the subroutine's inputs. Two dummy assignments are generated in lines 29 and 33 to ensure correct syntax of the adjoint code even if no argument checkpointing code is provided, that is, if both files `g_store_inputs.inc` and `g_restore_inputs.inc` are left empty. These dummy assignments are eliminated by the optimizing native C++ compiler. In the current example the input value x of g is saved by

```
argcp=x
```

and restored by

```
x=argcp      .
```

Not saving x results in incorrect adjoints as its input value is overwritten by the call of g in line 13 of the following listing of a1_f. The adjoint a1_g would hence be called with the wrong input value for x in line 27.

```
1   void a1_f(int a1_mode, double& x, double& a1_x,
2                          double& y, double& a1_y) {
3     double v1_0=0;
4     double a1_v1_0=0;
5     double v1_1=0;
6     double a1_v1_1=0;
7     int save_csc=0;
8     save_csc=csc;
9     if (a1_mode==1) {
10      // augmented forward section
11      cs[csc]=0; csc=csc+1;
12      a1_g(2,x,a1_x);
13      g(x);
14      fds[fdsc]=y; fdsc=fdsc+1; y=sqrt(x);
15  #include "f_store_results.inc"
16      // reverse section
17      while (csc>save_csc) {
18        csc=csc-1;
19        if (cs[csc]==0) {
20          fdsc=fdsc-1; y=fds[fdsc];
21          v1_0=x;
22          v1_1=sqrt(v1_0);
23          a1_v1_1=a1_y; a1_y=0;
24          a1_v1_0=1/(2*sqrt(v1_0))*a1_v1_1;
25          a1_x=a1_x+a1_v1_0;
```

```
26              a1_g(3,x,a1_x);
27              a1_g(1,x,a1_x);
28            }
29          }
30  #include "f_restore_results.inc"
31      }
32      if (a1_mode==2) {
33  #include "f_store_inputs.inc"
34          a1_mode=a1_mode;
35      }
36      if (a1_mode==3) {
37  #include "f_restore_inputs.inc"
38          a1_mode=a1_mode;
39      }
40  }
```

Apart from the treatment of the subroutine call in lines 12, 13, 26, and 27 the adjoint version of f is structurally similar to a1_g. Subroutine calls are preceded by the storage of their argument checkpoints within the augmented forward section (lines 12 and 13). In the reverse section, the correct arguments are restored (line 26) before the adjoint subroutine is executed (line 27). The correct result y of f is preserved by the user-provided code for storing

```
rescp=y
```

in `f_store_results.inc` and restoring

```
y=rescp
```

in `f_restore_results.inc`. Arguments of f need not be stored and recovered as f is never executed "out of context."

Bibliography

[1] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*. Society for Industrial and Applied Mathematics (SIAM), 2008.

[2] U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation*. Number 24 in Software, Environments, and Tools, SIAM, 2012.

[3] U. Naumann. DAG Reversal is NP-complete. Journal of Discrete Algorithms, 7:402–410, Elsevier, 2009.

Appendix A

dco0.9 Source

We present parts of the dco 0.9 source code implementing the scalar tangent-linear (A.1) and adjoint (A.2), modes of AD. Listings are restricted to a few selected arithmetic operators and intrinsic functions. Extensions are reasonably straightforward. Refer to Sections 2.1 (for tangent-linear mode) and 2.2 (for adjoint mode) for explanation of the code.

A.1 Tangent-Linear Code

Listing A.1: dco_t1s_type.hpp

```
#ifndef DCO_T1S_INCLUDED_
#define DCO_T1S_INCLUDED_

  class dco_t1s_type {
    public :
      double v; double t;
      dco_t1s_type(const double&);
      dco_t1s_type();
      dco_t1s_type& operator=(const dco_t1s_type&);
  };
  dco_t1s_type operator*(const dco_t1s_type&,
                         const dco_t1s_type&);
  dco_t1s_type operator+(const dco_t1s_type&,
                         const dco_t1s_type&);
  dco_t1s_type operator-(const dco_t1s_type&,
                         const dco_t1s_type&);
  dco_t1s_type sin(const dco_t1s_type&);
  dco_t1s_type cos(const dco_t1s_type&);
  dco_t1s_type exp(const dco_t1s_type&);
#endif
```

Listing A.2: dco_t1s_type.cpp

```
#include <cmath>
using namespace std;
#include "dco_t1s_type.hpp"

  dco_t1s_type::dco_t1s_type(const double& x): v(x), t(0) { };
  dco_t1s_type::dco_t1s_type(): v(0), t(0) { };

  dco_t1s_type& dco_t1s_type::operator=(const dco_t1s_type& x) {
```

```
    if (this==&x) return *this;
    v=x.v; t=x.t;
    return *this;
  }

  dco_t1s_type operator*(const dco_t1s_type& x1, const dco_t1s_type& x2) {
    dco_t1s_type tmp;
    tmp.v=x1.v*x2.v;
    tmp.t=x1.t*x2.v+x1.v*x2.t;
    return tmp;
  }

  dco_t1s_type operator+(const dco_t1s_type& x1, const dco_t1s_type& x2) {
    dco_t1s_type tmp;
    tmp.v=x1.v+x2.v;
    tmp.t=x1.t+x2.t;
    return tmp;
  }

  dco_t1s_type operator-(const dco_t1s_type& x1, const dco_t1s_type& x2) {
    dco_t1s_type tmp;
    tmp.v=x1.v-x2.v;
    tmp.t=x1.t-x2.t;
    return tmp;
  }

  dco_t1s_type sin(const dco_t1s_type& x) {
    dco_t1s_type tmp;
    tmp.v=sin(x.v);
    tmp.t=cos(x.v)*x.t;
    return tmp;
  }

  dco_t1s_type cos(const dco_t1s_type& x) {
    dco_t1s_type tmp;
    tmp.v=cos(x.v);
    tmp.t=-sin(x.v)*x.t;
    return tmp;
  }

  dco_t1s_type exp(const dco_t1s_type& x) {
    dco_t1s_type tmp;
    tmp.v=exp(x.v);
    tmp.t=tmp.v*x.t;
    return tmp;
  }
```

A.2 Adjoint Code

Listing A.3: dco_a1s_type.hpp

```
#ifndef DCO_A1S_INCLUDED_
  #define DCO_A1S_INCLUDED_
  #define DCO_A1S_TAPE_SIZE 1000000

  #define DCO_A1S_UNDEF -1
  #define DCO_A1S_CONST 0
  #define DCO_A1S_ASG 1
```

```
  #define DCO_A1S_ADD 2
  #define DCO_A1S_SUB 3
  #define DCO_A1S_MUL 4
  #define DCO_A1S_SIN 5
  #define DCO_A1S_COS 6
  #define DCO_A1S_EXP 7

  class dco_a1s_tape_entry {
    public:
      int oc;
      int arg1;
      int arg2;
      double v;
      double a;
      dco_a1s_tape_entry() : oc(DCO_A1S_UNDEF), arg1(DCO_A1S_UNDEF), arg2(
          DCO_A1S_UNDEF), v(0), a(0) {};
  };

  class dco_a1s_type {
    public:
      int va;
      double v;
      dco_a1s_type() : va(DCO_A1S_UNDEF), v(0) {};
      dco_a1s_type(const double&);
      dco_a1s_type& operator=(const dco_a1s_type&);
  };

  dco_a1s_type operator*(const dco_a1s_type&, const dco_a1s_type&);
  dco_a1s_type operator+(const dco_a1s_type&, const dco_a1s_type&);
  dco_a1s_type operator-(const dco_a1s_type&, const dco_a1s_type&);
  dco_a1s_type sin(const dco_a1s_type&);
  dco_a1s_type cos(const dco_a1s_type&);
  dco_a1s_type exp(const dco_a1s_type&);
  void dco_a1s_print_tape();
  void dco_a1s_interpret_tape();
  void dco_a1s_reset_tape();

#endif
```

Listing A.4: dco_a1s_type.cpp

```
#include <cmath>
#include <iostream>
using namespace std;
#include "dco_a1s_type.hpp"

static int dco_a1s_vac=0;
dco_a1s_tape_entry dco_a1s_tape[DCO_A1S_TAPE_SIZE];

dco_a1s_type::dco_a1s_type(const double& x): v(x) {
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_CONST;
  dco_a1s_tape[dco_a1s_vac].v=x;
  va=dco_a1s_vac++;
};

dco_a1s_type& dco_a1s_type::operator=(const dco_a1s_type& x) {
  if (this==&x) return *this;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_ASG;
  dco_a1s_tape[dco_a1s_vac].v=v=x.v;
```

```
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  va=dco_a1s_vac++;
  return *this;
}

dco_a1s_type operator*(const dco_a1s_type& x1, const dco_a1s_type& x2) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_MUL;
  dco_a1s_tape[dco_a1s_vac].arg1=x1.va;
  dco_a1s_tape[dco_a1s_vac].arg2=x2.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=x1.v*x2.v;
  tmp.va=dco_a1s_vac++;
  return tmp;
}

dco_a1s_type operator+(const dco_a1s_type& x1, const dco_a1s_type& x2) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_ADD;
  dco_a1s_tape[dco_a1s_vac].arg1=x1.va;
  dco_a1s_tape[dco_a1s_vac].arg2=x2.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=x1.v+x2.v;
  tmp.va=dco_a1s_vac++;
  return tmp;
}

dco_a1s_type operator-(const dco_a1s_type& x1, const dco_a1s_type& x2) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_SUB;
  dco_a1s_tape[dco_a1s_vac].arg1=x1.va;
  dco_a1s_tape[dco_a1s_vac].arg2=x2.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=x1.v-x2.v;
  tmp.va=dco_a1s_vac++;
  return tmp;
}

dco_a1s_type sin(const dco_a1s_type& x) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_SIN;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=sin(x.v);
  tmp.va=dco_a1s_vac++;
  return tmp;
}

dco_a1s_type cos(const dco_a1s_type& x) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_COS;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=cos(x.v);
  tmp.va=dco_a1s_vac++;
  return tmp;
}

dco_a1s_type exp(const dco_a1s_type& x) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_EXP;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=exp(x.v);
```

```
  tmp.va=dco_a1s_vac++;
  return tmp;
}

void dco_a1s_print_tape () {
  cout << "tape:" << endl;
  for (int i=0;i<dco_a1s_vac;i++)
    cout << i << ":␣[␣" << dco_a1s_tape[i].oc << ",␣"
         << dco_a1s_tape[i].arg1 << ",␣"
         << dco_a1s_tape[i].arg2 << ",␣"
         << dco_a1s_tape[i].v << ",␣"
         << dco_a1s_tape[i].a << "␣]" << endl;
}

void dco_a1s_reset_tape () {
  for (int i=0;i<dco_a1s_vac;i++)
    dco_a1s_tape[i].a=0;
  dco_a1s_vac=0;
}

void dco_a1s_interpret_tape () {
  for (int i=dco_a1s_vac;i>=0;i--) {
    switch (dco_a1s_tape[i].oc) {
      case DCO_A1S_ASG : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_ADD : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=dco_a1s_tape[i].a;
        dco_a1s_tape[dco_a1s_tape[i].arg2].a+=dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_SUB : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=dco_a1s_tape[i].a;
        dco_a1s_tape[dco_a1s_tape[i].arg2].a-=dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_MUL : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=dco_a1s_tape[dco_a1s_tape[i].
            arg2].v*dco_a1s_tape[i].a;
        dco_a1s_tape[dco_a1s_tape[i].arg2].a+=dco_a1s_tape[dco_a1s_tape[i].
            arg1].v*dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_SIN : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=cos(dco_a1s_tape[dco_a1s_tape[
            i].arg1].v)*dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_COS : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a-=sin(dco_a1s_tape[dco_a1s_tape[
            i].arg1].v)*dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_EXP : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=dco_a1s_tape[i].v*dco_a1s_tape
            [i].a;
        break;
```

```
        }
      }
    }
}
```

Appendix B

dcc0.9 Syntax

This chapter contains a summary of the syntax accepted by version 0.9 of `dcc`. The same information can be obtained by running `flex` and `bison` in their diagnostic modes on the respective input files `scanner.l` and `parser.y`.

B.1 bison Grammar

```
code: sequence_of_global_declarations sequence_of_subroutines

sequence_of_subroutines: subroutine
                       | sequence_of_subroutines subroutine

subroutine: VOID SYMBOL '(' list_of_arguments ')' '{'
    sequence_of_local_declarations sequence_of_statements '}'

list_of_arguments: argument
                 | list_of_arguments ',' argument

sequence_of_asterixes: '*'
                     | sequence_of_asterixes '*'

argument: INT '&' SYMBOL
        | INT SYMBOL
        | FLOAT '&' SYMBOL
        | FLOAT sequence_of_asterixes SYMBOL
        | INT sequence_of_asterixes SYMBOL

sequence_of_global_declarations: /* empty */
                               | sequence_of_global_declarations
                                   global_declaration

sequence_of_local_declarations: /* empty */
                              | sequence_of_local_declarations
                                  local_declaration

local_declaration: FLOAT SYMBOL '=' CONSTANT ';'
                 | INT SYMBOL '=' CONSTANT ';'

global_declaration: INT SYMBOL '=' CONSTANT ';'
                  | INT SYMBOL '[' CONSTANT ']' ';'
```

```
                  | FLOAT SYMBOL '=' CONSTANT ';'
                  | FLOAT SYMBOL '[' CONSTANT ']' ';'

sequence_of_statements: statement
                      | sequence_of_statements statement

statement: assignment
         | if_statement
         | while_statement
         | subroutine_call_statement ';'

if_statement: IF '(' condition ')' '{' sequence_of_statements '}'
    else_branch

else_branch: /* empty */
           | ELSE '{' sequence_of_statements '}'

while_statement: WHILE '(' condition ')' '{' sequence_of_statements '}'

condition: memref_or_constant '<' memref_or_constant
         | memref_or_constant '>' memref_or_constant
         | memref_or_constant '=' '=' memref_or_constant
         | memref_or_constant '!' '=' memref_or_constant
         | memref_or_constant '>' '=' memref_or_constant
         | memref_or_constant '<' '=' memref_or_constant

subroutine_call_statement: SYMBOL '(' list_of_args ')'

assignment: memref '=' expression ';'

expression: '(' expression ')'
          | expression '*' expression
          | expression '/' expression
          | expression '+' expression
          | expression '-' expression
          | SIN '(' expression ')'
          | COS '(' expression ')'
          | EXP '(' expression ')'
          | SQRT '(' expression ')'
          | TAN '(' expression ')'
          | ATAN '(' expression ')'
          | LOG '(' expression ')'
          | POW '(' expression ',' SYMBOL ')'
          | memref
          | CONSTANT

list_of_args: memref_or_constant
            | memref_or_constant ',' list_of_args

memref_or_constant: memref
                  | CONSTANT

array_index: SYMBOL
           | CONSTANT

memref: SYMBOL
      | array_reference
```

```
array_reference: SYMBOL array_access

array_access: '[' array_index ']'

array_access: '[' array_index ']' array_access
```

B.2 flex Grammar

Whitespaces are ignored. Single-line comments are allowed starting with //. Some integer and floating-point constants are supported. Variable names start with lower or upper case letters followed by further letters, underscores, or digits.

int	$0 \mid [1-9][0-9]*$
float	$\{\textbf{int}\}\text{"."}[0-9]*$
const	$\{\textbf{int}\} \mid \{\textbf{float}\}$
symbol	$([\text{A–Z}] \mid [\text{a–z}])\ (([\text{A–Z}] \mid [\text{a–z}]) \mid \_ \mid \{\textbf{int}\})*$

Supported key words are the following: **static**, **double**, **int**, **void**, **if**, **else**, **while**, sin, cos, exp, sqrt, atan, tan, pow, and log.