RWTH Aachen

Department of Computer Science
Technical Report

Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations

Uwe Naumann and Johannes Lotz

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2012-10

RWTH Aachen · Department of Computer Science · June 2012

Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations

Uwe Naumann and Johannes Lotz

RWTH Aachen University LuFG Informatik 12
Software and Tools for Computational Engineering
Department of Computer Science, RWTH Aachen University
52062 Aachen, Germany
Email: {naumann, lotz}@stce.rwth-aachen.de

Abstract. We consider the Algorithmic Differentiation (also know as Automatic Differentiation; AD) of numerical simulation programs that contain calls to direct solvers for systems of n linear equations. AD of the linear solvers yields a local overhead of $O(n^3)$ for the computation of directional derivatives or adjoints of the solution vector with respect to the system matrix and right-hand side. The local memory requirement is of the same order in adjoint mode AD. Mathematical insight yields a reduction of the local computational complexity to $O(n^2)$. The memory overhead can be reduced to at least $O(n^2)$ in adjoint mode. We derive efficient tangent-linear and adjoint direct linear solvers and illustrate their use within tangent-linear and adjoint versions of the enclosing numerical simulation.

1 Motivation

Algorithmic Differentiation (AD) [GW08,Nau12a] is a semantic program transformation technique that yields robust and efficient derivative code. Its reverse or adjoint mode is of particular interest in large-scale nonlinear optimization due to the independence of its computational cost on the number of free parameters. AD tools for compile- (source code transformation) and run-time (operator and function overloading) solutions have been developed many of which are listed on the AD community's web portal `www.autodiff.org`. Traditionally, AD tools transform the source code at the level of arithmetic operators and built-in functions. Potentially complex numerical kernels, for example, matrix products [Gil08] or the solvers for systems of linear equations to be discussed in this paper, are typically not considered as intrinsic functions often resulting in suboptimal computational performance. Ideally, one would like to re-use intermediate results of the evaluation of the original kernel for the evaluation of directional derivatives and/or adjoints, thus, potentially reducing the computational overhead induced by the differentiation. For direct linear solvers mathematical insight yields a reduction of the overhead from $O(n^3)$ to $O(n^2)$. Applicability of this theoretical result is facilitated by the integration of direct linear solvers as intrinsic functions into AD software tools. For a given programming language a generally applicable solution would require built-in numerical kernels. However, currently these kernels are provided through various run time support libraries. A practical implementation needs to focus on a given library. A large number of technical issues need to be addressed that do not add to the conceptual understanding of the subject. For this reason we chose to present a reference implementation based on a simple custom Gauss solver in the appendix while focusing on the mathematical/algorithmic details in the main paper. Readers are welcome to download the sources of all implementations that are referenced in the following from

`www.stce.rwth-aachen.de/publications/Naumann2012DLS`.

2 Foundations

We consider the computation of directional derivatives (tangents)

$$\mathbb{R}^n \ni \mathbf{x}^{(1)} =< \frac{\partial \mathbf{x}}{\partial A}, A^{(1)} > + < \frac{\partial \mathbf{x}}{\partial \mathbf{b}}, \mathbf{b}^{(1)} > \tag{1}$$

and of adjoints

$$\mathbb{R}^{n\times n} \ni A_{(1)} =< \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial A} > \tag{2}$$

and

$$\mathbb{R}^n \ni \mathbf{b}_{(1)} =< \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial \mathbf{b}} > \tag{3}$$

for direct solvers of systems of linear equations

$$A \cdot \mathbf{x} = \mathbf{b}, \tag{4}$$

where $A = A(\mathbf{z}) \in \mathbb{R}^{n\times n}$, $\mathbf{b} = \mathbf{b}(\mathbf{z}) \in \mathbb{R}^n$, $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{z} \in \mathbb{R}^m$. The *continuous approach* to the computation of derivatives of the solution $\mathbf{x}$ with respect to the right-hand side $\mathbf{b}$ (see Sections 3.2 and 4.2) has been proposed previously (see, for example, [TFK06]). We are not aware of a corresponding description of differentiation with respect to the system matrix A. Moreover, to the best of our knowledge, there does not exist a formal description of the special treatment of (direct) linear solvers in the context of AD tool development.

For the purpose of illustration, the linear solvers are assumed to be embedded into the convex nonlinear programming (NLP) problem

$$\min_{\mathbf{z}\in\mathbb{R}^m} f(\mathbf{z})$$

for a given objective function $f : \mathbb{R}^m \to \mathbb{R}$. If first-order gradient-based methods shall be used for its solution, then the gradient of $y = f(\mathbf{z}) \in \mathbb{R}$ with respect to $\mathbf{z} \in \mathbb{R}^m$ needs to be computed.

We adopt the notation from [Nau12a]. Individual entries of a *projection* $\mathbb{R}^n \ni \mathbf{x}_A^{(1)} \equiv< \frac{\partial \mathbf{x}}{\partial A}, A^{(1)} >$ of the three-tensor $\frac{\partial \mathbf{x}}{\partial A} \in \mathbb{R}^{n\times(n\times n)}$ in direction $A^{(1)} \in \mathbb{R}^{n\times n}$ are defined as inner products

$$\left[\mathbf{x}_A^{(1)}\right]_i = \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} \left[\frac{\partial \mathbf{x}}{\partial A}\right]_{i,j,k} \cdot \left[A^{(1)}\right]_{j,k} \tag{5}$$

for $i = 0, \ldots, n-1$ and given *serializations* (referred to as *vectorizations* in [MN02]) of A and $A^{(1)} \in \mathbb{R}^{n\times n}$. We denote the element with indexes $i_0, \ldots, i_{k-1}$ within a k-tensor T by $[T]_{i_0,\ldots,i_{k-1}}$. Tensors up to fourth order need to be considered in the following. The matrix-vector product $\mathbb{R}^n \ni \mathbf{x}_\mathbf{b}^{(1)} \equiv< \frac{\partial \mathbf{x}}{\partial \mathbf{b}}, \mathbf{b}^{(1)} >$ (a projection of the two-tensor $\frac{\partial \mathbf{x}}{\partial \mathbf{b}} \in \mathbb{R}^{n\times n}$ in direction $\mathbf{b}^{(1)} \in \mathbb{R}^n$) becomes

$$\left[\mathbf{x}_\mathbf{b}^{(1)}\right]_i = \sum_{j=0}^{n-1} \left[\frac{\partial \mathbf{x}}{\partial \mathbf{b}}\right]_{i,j} \cdot \left[\mathbf{b}^{(1)}\right]_j \tag{6}$$

for $i = 0, \ldots, n-1$ and given $\mathbf{b}^{(1)} \in I\!R^n$. Adjoints are defined as projections of $\frac{\partial \mathbf{x}}{\partial A}$ in direction $\mathbf{x}_{(1)}$, that is,

$$\left[A_{(1)}\right]_{i,j} = \sum_{k=0}^{n-1} \left[\mathbf{x}_{(1)}\right]_k \cdot \left[\frac{\partial \mathbf{x}}{\partial A}\right]_{k,i,j} \tag{7}$$

for $i, j = 0, \ldots, n-1$ and, similarly, as projections of $\frac{\partial \mathbf{x}}{\partial \mathbf{b}}$ in the same direction, that is,

$$\left[\mathbf{b}_{(1)}\right]_i = \sum_{j=0}^{n-1} \left[\mathbf{x}_{(1)}\right]_j \cdot \left[\frac{\partial \mathbf{x}}{\partial \mathbf{b}}\right]_{j,i} \tag{8}$$

for $i = 0, \ldots, n-1$ and given $\mathbf{x}_{(1)} \in I\!R^n$. The above projections of the derivative tensors $\frac{\partial \mathbf{x}}{\partial A}$ and $\frac{\partial \mathbf{x}}{\partial \mathbf{b}}$ shall be evaluated in tensor-free fashion, that is, without accumulation of the tensors themselves.

For further illustration, f is decomposed into a sequence of three successive function evaluations

$$y = f(\mathbf{z}) = p(S(P(\mathbf{z}))), \tag{9}$$

where $P : I\!R^m \to I\!R^{n \times n} \times I\!R^n$ denotes the part of the computation that precedes the direct linear solver $S : I\!R^{n \times n} \times I\!R^n \to I\!R^n$ and where $p : I\!R^n \to I\!R$ maps the result $\mathbf{x}$ onto the scalar objective y. The direct linear solver $\mathbf{x} = S(A, \mathbf{b})$ solves the system of linear equations for $A = A(\mathbf{z})$ and $\mathbf{b} = \mathbf{b}(\mathbf{z})$, for example, by LU, QR, or LL^T factorization of A as described in any standard textbook on numerical linear algebra; see, for example, [TB97].

While we use unconstrained NLP as the motivating context, the results in this paper are applicable to arbitrary problems that involve the solution of linear systems including the solution of systems of nonlinear equations and general NLP problems, for example, with constraints given as (partial) differential equations [HPUU09]. Our arguments will be based on the following algorithmic description of Equation (9):

$$\begin{pmatrix} A \\ \mathbf{b} \end{pmatrix} := P(\mathbf{z}) \tag{10}$$

$$\mathbf{x} := S(A, \mathbf{b}) \tag{11}$$

$$y := p(\mathbf{x}). \tag{12}$$

This structure occurs, for example, in the context of parameter calibration problems for (partial) differential equations the solution of which is to be fitted to given observations. See Section 5 for a case study.

AD yields semantic transformations of implementations of, in general, multivariate vector functions $F : I\!R^n \to I\!R^m$ as computer programs. In the following we use the notation from [Nau12a] that is partially inspired by the notation used in [GW08]. For AD to become applicable, the given implementation of F is assumed to decompose into a *single assignment code* (SAC) as follows:

$$\begin{aligned} &\text{for } j = n, \ldots, n+p+m-1 \\ &\quad v_j = \varphi_j(v_i)_{i \prec j}, \end{aligned} \tag{13}$$

where $i \prec j$ denotes a direct dependence of v_j on v_i. The result of each *intrinsic function*[1] 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 SAC induces a directed acyclic graph (DAG) $G = (V, E)$ with 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, $\forall i, j \in V : (i,j) \in E \Rightarrow i < j$. Intrinsic functions φ_j are assumed to posses jointly continuous partial derivatives with respect to their arguments. Association of the local partial derivatives with their corresponding edges in the DAG yields the *linearized DAG*. The linearized DAG of our reference problem is shown in Fig. 1 (a) with (high-level) intrinsic functions P, S, and p.

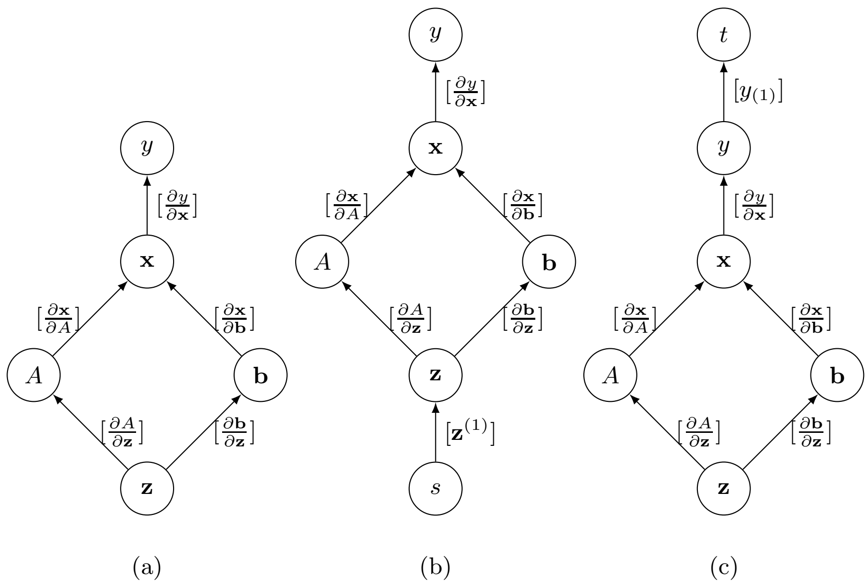


Fig. 1. Reference Problem: (a) Linearized DAG; (b) Tangent-Linear Extension; (c) Adjoint Extension

By the chain rule of differential calculus, the entries of the Jacobian $A = (a_{i,j}) \equiv \nabla F(\mathbf{x})$ of F can be computed as

$$a_{i,j} = \sum_{\pi \in [i \rightarrow n+p+j]} \prod_{(k,l) \in \pi} c_{l,k} \tag{14}$$

[1] Intrinsic functions can range from fundamental arithmetic operations $(+, *, \ldots)$ and built-in (into the used programming language) functions $(\sin, \exp, \ldots)$ to potentially highly complex numerical algorithms such as routines for interpolation, numerical integration, or the solution of systems of linear or nonlinear equations. In its basic form, AD is defined for the arithmetic operators and built-in functions. A formal extension of this concept to higher-level intrinsics turns out to be straight forward. For a complex algorithm to become an intrinsic function we require existence and availability of the partial derivatives of its results with respect to its arguments.

with local partial derivatives

$$c_{l,k} \equiv \frac{\partial \varphi_l}{\partial v_k}(v_q)_{q \prec l}$$

and where $[i \to n+p+j]$ denotes the set of all paths that connect the independent vertex i with the dependent vertex $n+p+j$ [Bau74]. For example, according to Fig. 1

$$\frac{\partial f}{\partial \mathbf{z}} \equiv \frac{\partial y}{\partial \mathbf{z}} = \frac{\partial y}{\partial \mathbf{x}} \cdot \frac{\partial \mathbf{x}}{\partial A} \cdot \frac{\partial A}{\partial \mathbf{z}} + \frac{\partial y}{\partial \mathbf{x}} \cdot \frac{\partial \mathbf{x}}{\partial \mathbf{b}} \cdot \frac{\partial \mathbf{b}}{\partial \mathbf{z}} = \frac{\partial y}{\partial \mathbf{x}} \cdot \left(\frac{\partial \mathbf{x}}{\partial A} \cdot \frac{\partial A}{\partial \mathbf{z}} + \frac{\partial \mathbf{x}}{\partial \mathbf{b}} \cdot \frac{\partial \mathbf{b}}{\partial \mathbf{z}} \right) .$$

The minimization of the computational cost of Jacobian accumulation is known to be NP-hard [Nau08]. Elimination techniques on linearized DAGs that facilitate approximate solutions of the combinatorial *Optimal Jacobian Accumulation* problem have been developed for several years; see, for example, [GR91,GV03,Nau04].

The Jacobian $\nabla F = \nabla F(\mathbf{x})$ induces a linear mapping $\nabla F : I\!R^n \to I\!R^m$ defined by

$$\mathbf{x}^{(1)} \mapsto < \nabla F, \mathbf{x}^{(1)} > .$$

The function $F^{(1)} : I\!R^{2 \cdot n} \to I\!R^m$, defined as

$$\mathbf{y}^{(1)} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}) \equiv < \nabla F, \mathbf{x}^{(1)} > = \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}, \tag{15}$$

is referred to as the *tangent-linear model* of F. The directional derivative $\mathbf{y}^{(1)}$ can be regarded as the partial derivative of $\mathbf{y}$ with respect to an auxiliary scalar variable s, where

$$\mathbf{x}^{(1)} = \frac{\partial \mathbf{x}}{\partial s}.$$

Interpretation of the chain rule on the corresponding linearized DAG (the *tangent-linear extension* of the original linearized DAG) yields

$$\mathbf{y}^{(1)} \equiv \frac{\partial \mathbf{y}}{\partial s} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \cdot \frac{\partial \mathbf{x}}{\partial s} = < \nabla F(\mathbf{x}), \mathbf{x}^{(1)} > .$$

The tangent-linear extension of the linearized DAG of our reference problem is shown in Fig. 1 (b). Equation (14) yields $y^{(1)} = \frac{\partial y}{\partial \mathbf{z}} \cdot \mathbf{z}^{(1)} = < \frac{\partial y}{\partial \mathbf{z}}, \mathbf{z}^{(1)} >$. Note that $\frac{\partial y}{\partial \mathbf{z}} \in I\!R^{1 \times m}$.

The adjoint of a linear operator is its transpose [DS88]. Consequently, the transposed Jacobian $\nabla F^T = \nabla F(\mathbf{x})^T$ induces a linear mapping $I\!R^m \to I\!R^n$ defined by

$$\mathbf{y}_{(1)} \mapsto \nabla F^T \cdot \mathbf{y}_{(1)}.$$

The function $F_{(1)} : I\!R^{n+m} \to I\!R^n$ defined as

$$\mathbf{x}_{(1)} = F_{(1)}(\mathbf{x}, \mathbf{y}_{(1)}) \equiv < \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) > = \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)} \tag{16}$$

is referred to as the *adjoint model* of F. Adjoints can be defined as partial derivatives of an auxiliary scalar variable t with respect to $\mathbf{y}$ and $\mathbf{x}$, where

$$\mathbf{y}_{(1)} \equiv \frac{\partial t}{\partial \mathbf{y}} \quad \text{and} \quad \mathbf{x}_{(1)} \equiv \frac{\partial t}{\partial \mathbf{x}}.$$

By the chain rule, we get

$$\mathbf{x}_{(1)} \equiv \left(\frac{\partial t}{\partial \mathbf{x}}\right)^T = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \cdot \left(\frac{\partial t}{\partial \mathbf{y}}\right)^T = \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)}.$$

The corresponding *adjoint extension* of the linearized DAG of our reference problem is shown in Fig. 1 (c). Equation (14) yields $\mathbf{z}_{(1)} = \frac{\partial y}{\partial \mathbf{z}}^T \cdot y_{(1)} =< y_{(1)}, \frac{\partial y}{\partial \mathbf{z}} >$.

Projections of derivative tensors in vector-valued directions are defined to be invariant with respect to transposition of the vector argument, that is,

$$< \nabla F, \mathbf{x}^{(1)} >=< \nabla F, \mathbf{x}^{(1)^T} > \tag{17}$$

and

$$< \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) >=< \mathbf{y}_{(1)}^T, \nabla F(\mathbf{x}) > . \tag{18}$$

2.1 Example

For illustration of the algorithmic details behind basic AD we consider the simple scalar multivariate function

$$y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$$

that is used in [Nau12a] to illustrate the superiority of adjoint over tangent-linear mode AD in the context of unconstrained nonlinear programming. Fig. 2 shows the linearized DAG for $n = 3$ annotated with the tangent-linear SAC statements. The corresponding tangent-linear SAC itself is shown in the upper left corner of Fig. 2. A total of $n = 3$ runs with $\mathbf{x}^{(1)}$ ranging over the Cartesian basis vectors in $I\!R^n$ return the individual entries of the gradient of y with respect to $\mathbf{x}$ at the current point in $y^{(1)}$, respectively. Neither the v_i nor their tangent-linear versions $v_i^{(1)}$ need to be stored persistently. Consequently, the memory requirement of the tangent-linear code is about twice as large as that of the original program.

Fig. 3 shows the same linearized DAG annotated with the adjoint SAC statements. The corresponding adjoint SAC is shown in the upper left corner of Fig. 3. A single run with $y_{(1)} = 1$ returns the gradient of y with respect to $\mathbf{x}$ at the current point in $\mathbf{x}_{(1)}$. The v_i (or a subset thereof; see [HNP05]) need to be stored persistently in order to be able to access them in reverse order in the reverse section of the adjoint code. Thus, the memory requirement of the adjoint code is of the same order as the number of operations performed by the original program. Refer to [Nau12a] for in-depth information on the fundamental structure of tangent-linear and adjoint code.

3 Tangent-Linear Direct Linear Solver

The chain rule applied to Fig. 1 (b) yields the following tangent-linear version of the algorithm in Equations (10)–(12):

$$\begin{pmatrix} A \\ \mathbf{b} \end{pmatrix} := P(\mathbf{z}) \tag{19}$$

$$\begin{pmatrix} A^{(1)} \\ \mathbf{b}^{(1)} \end{pmatrix} := P^{(1)}(\mathbf{z}, \mathbf{z}^{(1)}) = \begin{pmatrix} < \frac{\partial A}{\partial \mathbf{z}}, \mathbf{z}^{(1)} > \\ < \frac{\partial \mathbf{b}}{\partial \mathbf{z}}, \mathbf{z}^{(1)} > \end{pmatrix} \tag{20}$$

$v_0 = x_0;\ v_0^{(1)} = x_0^{(1)}$
$v_1 = x_1;\ v_1^{(1)} = x_1^{(1)}$
$v_2 = x_2;\ v_2^{(1)} = x_2^{(1)}$
$v_3 = v_0^2;\ v_3^{(1)} = 2v_0 \cdot v_3^{(1)}$
$v_4 = v_1^2;\ v_4^{(1)} = 2v_1 \cdot v_4^{(1)}$
$v_5 = v_2^2;\ v_5^{(1)} = 2v_2 \cdot v_5^{(1)}$
$v_6 = v_3 + v_4;\ v_6^{(1)} = v_3^{(1)} + v_4^{(1)}$
$v_7 = v_6 + v_5;\ v_7^{(1)} = v_6^{(1)} + v_5^{(1)}$
$v_8 = v_7^2;\ 2v_7 \cdot v_7^{(1)}$
$y = v_8;\ y^{(1)} = v_8^{(1)}$

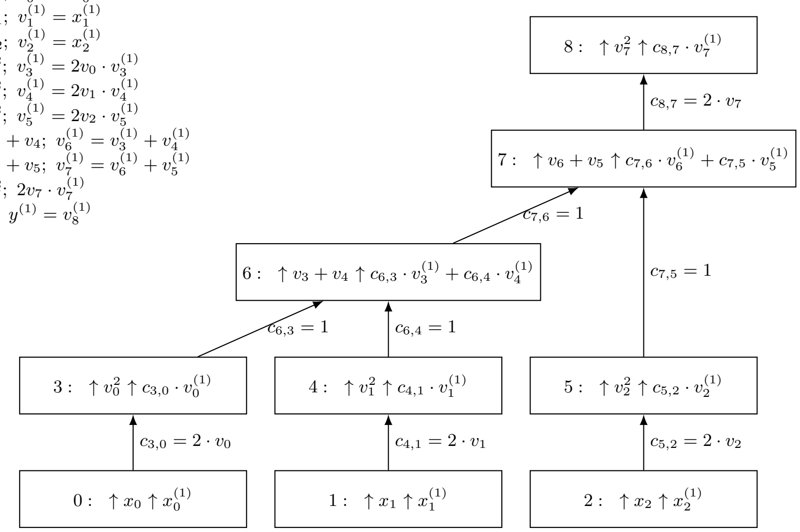


Fig. 2. Tangent-Linear AD illustrated for $y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$ and $n = 3$; The tangent-linear SAC shown in the upper left corner is derived from the displayed tangent-linear linearized DAG.

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

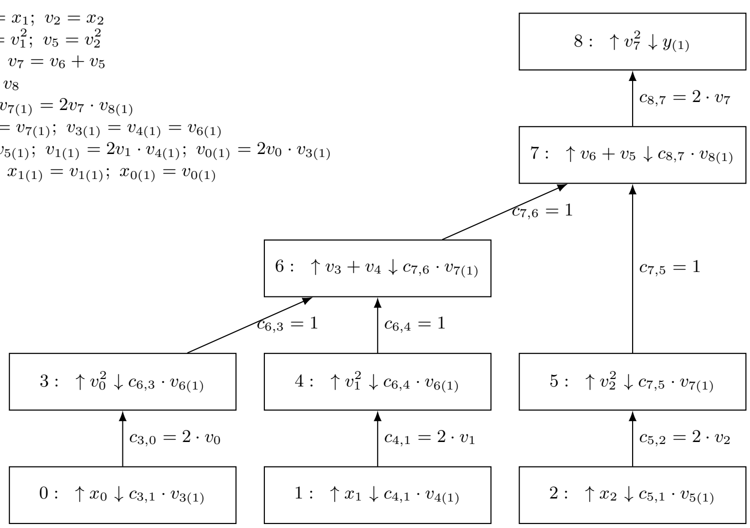


Fig. 3. Adjoint AD illustrated for $y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$ and $n = 3$; The adjoint SAC shown in the upper left corner is derived from the displayed adjoint linearized DAG.

$$
\begin{aligned}
\mathbf{x} &:= S(A, \mathbf{b}) && (21)\\
\mathbf{x}^{(1)} &:= S^{(1)}(A, A^{(1)}, \mathbf{b}, \mathbf{b}^{(1)}) =< \frac{\partial \mathbf{x}}{\partial A}, A^{(1)} > + < \frac{\partial \mathbf{x}}{\partial \mathbf{b}}, \mathbf{b}^{(1)} > && (22)\\
y &:= p(\mathbf{x}) && (23)\\
y^{(1)} &:= p^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}) =< \frac{\partial y}{\partial \mathbf{x}}, \mathbf{x}^{(1)} > . && (24)
\end{aligned}
$$

3.1 Discrete Version

A fully discrete version of Equations (19)–(24) is obtained by applying tangent-linear mode AD to the implementation of Equations (10)–(12). Conceptually, each operation is augmented with its tangent-linear counterpart resulting roughly in a duplication of the computational cost as well as the memory requirement. An example that illustrates the application of our AD tool `dco` (version 0.9; [Nau12b]) introduced in [Nau12a] to an *LU*-factorization and the following forward and backward substitutions is presented in Appendix A. Note that the computational cost of evaluating Equation (22) becomes $O(n^3)$, which will be improved through the exploitation of mathematical insight in the following section.

3.2 Continuous Version

Tangent-linear versions of P and p, called in Equation (20) and Equation (24), respectively, are assumed to be available. For example, they can be obtained through application of `dco` in tangent-linear mode to the given implementations of P and p similar to Section 3.1. Our focus is on the efficient evaluation of Equation (22).

Computation of $< \frac{\partial \mathbf{x}}{\partial \mathbf{b}}, \mathbf{b}^{(1)} >$ Partial differentiation of Equation (4) with respect to $\mathbf{b}$ yields

$$
< \frac{\partial (A\mathbf{x})}{\partial A}, \underbrace{\frac{\partial A}{\partial \mathbf{b}}}_{(=0)} > + \underbrace{\frac{\partial (A\mathbf{x})}{\partial \mathbf{x}}}_{(=A)}, \frac{\partial \mathbf{x}}{\partial \mathbf{b}} >= \frac{\partial \mathbf{b}}{\partial \mathbf{b}} \tag{25}
$$

and, hence,

$$
A \cdot \frac{\partial \mathbf{x}}{\partial \mathbf{b}} = I_n, \tag{26}
$$

where $I_n \in I\!R^{n \times n}$ denotes the identity in $I\!R^n$. The three-tensor $\frac{\partial A}{\partial \mathbf{b}} \in I\!R^{(n \times n) \times n}$ and, hence, the projection $< \frac{\partial (A\mathbf{x})}{\partial A}, \frac{\partial A}{\partial \mathbf{b}} >$ vanish identically. The term $A \cdot \frac{\partial \mathbf{x}}{\partial \mathbf{b}}$ is the usual product of two $n \times n$ matrices.

Multiplication of both sides of Equation (26) with $\mathbf{b}^{(1)}$ from the right yields

$$
A \cdot \frac{\partial \mathbf{x}}{\partial \mathbf{b}} \cdot \mathbf{b}^{(1)} = A \cdot < \frac{\partial \mathbf{x}}{\partial \mathbf{b}}, \mathbf{b}^{(1)} >= \mathbf{b}^{(1)}. \tag{27}
$$

Consequently, the second term in Equation (22) can be computed by replacing the original right-hand side in Equation (4) with $\mathbf{b}^{(1)}$. A previously computed factorization of A should be reused, thus, reducing the computational cost locally from $O(n^3)$ to $O(n^2)$.

Computation of $<\frac{\partial \mathbf{x}}{\partial A}, A^{(1)}>$ Partial differentiation of Equation (4) with respect to A yields

$$< \frac{\partial (A\mathbf{x})}{\partial A}, \underbrace{\frac{\partial A}{\partial A}}_{(=I_{n\times n})} > + < \underbrace{\frac{\partial (A\mathbf{x})}{\partial \mathbf{x}}}_{(=A)}, \frac{\partial \mathbf{x}}{\partial A} > = \underbrace{\frac{\partial \mathbf{b}}{\partial A}}_{(=0)}, \tag{28}$$

where $I_{n\times n} \in I\!R^{(n\times n)\times(n\times n)}$ denotes the identity in $I\!R^{n\times n}$, which yields

$$\frac{\partial (A\mathbf{x})}{\partial A} + < \frac{\partial (A\mathbf{x})}{\partial \mathbf{x}}, \frac{\partial \mathbf{x}}{\partial A} > = \underbrace{\frac{\partial \mathbf{b}}{\partial A}}_{(=0)}. \tag{29}$$

and, hence,

$$< \frac{\partial (A\mathbf{x})}{\partial \mathbf{x}}, \frac{\partial \mathbf{x}}{\partial A} > = -\frac{\partial (A\mathbf{x})}{\partial A}. \tag{30}$$

Projection of the two trailing dimensions of both sides of Equation (30) in direction $A^{(1)}$ yields

$$\begin{aligned}
<< \frac{\partial (A\mathbf{x})}{\partial \mathbf{x}}, \frac{\partial \mathbf{x}}{\partial A} >, A^{(1)} > &\overset{(Assoc.)}{=} < A, < \frac{\partial \mathbf{x}}{\partial A}, A^{(1)} >> && (31)\\
&\overset{(Eq.(15))}{=} A\cdot < \frac{\partial \mathbf{x}}{\partial A}, A^{(1)} > && (32)\\
&\overset{(Eq.(30))}{=} - < \frac{\partial (A\mathbf{x})}{\partial A}, A^{(1)} > && (33)\\
&\overset{(Lemma\ 1)}{=} -A^{(1)} \cdot \mathbf{x} && (34)
\end{aligned}$$

with $(Assoc.)$ denoting the associativity of the projection operation. Consequently, the first term in Equation (22) can be computed by replacing the original right-hand side in Equation (4) with $-A^{(1)}\cdot\mathbf{x}$. A previously computed factorization of A can be reused thus reducing the computational cost locally from $O(n^3)$ to $O(n^2)$.

Lemma 1. *Let $A, A^{(1)} \in I\!R^{n\times n}$ and $\mathbf{x} \in I\!R^n$. Then*

$$< \frac{\partial (A\mathbf{x})}{\partial A}, A^{(1)} > = A^{(1)} \cdot \mathbf{x}.$$

Proof. The entries of the three-tensor $\frac{\partial (A\mathbf{x})}{\partial A}$ are the following:

$$\left[\frac{\partial (A\mathbf{x})}{\partial A}\right]_{i,j,k} = \frac{\partial [A\mathbf{x}]_i}{\partial [A]_{j,k}} = \frac{\partial \sum_{l=0}^{n-1}([A]_{i,l}\cdot[\mathbf{x}]_l)}{\partial [A]_{j,k}} = \begin{cases} [\mathbf{x}]_k & \text{for } i=j \\ 0 & \text{otherwise.}\end{cases} \tag{35}$$

It follows that the sum

$$\left[< \frac{\partial (A\mathbf{x})}{\partial A}, A^{(1)} >\right]_i = \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} \left[\frac{\partial (A\mathbf{x})}{\partial A}\right]_{i,j,k} \cdot [A^{(1)}]_{j,k} \tag{36}$$

collapses to

$$\sum_{k=0}^{n-1} [\mathbf{x}]_k \cdot [A^{(1)}]_{i,k} = [A^{(1)}\cdot\mathbf{x}]_i.$$

3.3 Examples

Discrete tangents of LU, QR, and LL^T factorizations followed by the corresponding substitution procedures can be obtained in a straight-forward fashion by applying tangent-linear mode AD to the given implementations as outlined in Section 3. In the following we focus on the continuous versions.

LU-Factorization Equations (19) and (20) are followed by

$$(\mathbf{x}, L, U) = S(A, \mathbf{b}) \tag{37}$$

$$\mathbf{x}^{(1)} = B(U, F(L, \mathbf{b}^{(1)})) + B(U, F(L, -\mathbf{x}^T \cdot A^{(1)})) \tag{38}$$

and by Equations (23) and (24), where $F, B : I\!R^{n\cdot(n+1)/2} \times I\!R^n \to I\!R^n$ denote solvers for lower and upper triangular systems by forward and backward substitution, respectively. Refer to Appendix B for a corresponding implementation that has been verified against the discrete tangent-linear version from Appendix A.

QR-Factorization Fully discrete tangent-linear QR factorization adds little insight and is hence omitted. Its computational complexity is $O(n^3)$ to be reduced to $O(n^2)$ by the following continuous approach, where Equations (19) and (20) are followed by

$$(\mathbf{x}, Q, R) = S(A, \mathbf{b}) \tag{39}$$

$$\mathbf{x}^{(1)} = B(R, Q^T \cdot \mathbf{b}^{(1)}) + B(R, -Q^T \cdot \mathbf{x}^T \cdot A^{(1)}) \tag{40}$$

and by Equations (23) and (24). The reduced computational complexity follows from $\mathbf{b}^{(1)} = A\cdot < \frac{\partial \mathbf{x}}{\partial \mathbf{b}}, \mathbf{b}^{(1)} >= Q \cdot R\cdot < \frac{\partial \mathbf{x}}{\partial \mathbf{b}}, \mathbf{b}^{(1)} >$ and, hence, $R\cdot < \frac{\partial \mathbf{x}}{\partial \mathbf{b}}, \mathbf{b}^{(1)} >= Q^{-1} \cdot \mathbf{b}^{(1)} = Q^T \cdot \mathbf{b}^{(1)}$.

LL^T-Factorization If A is symmetric positive definite within the range of P, then $\mathbf{x}$ depends only on the lower (or upper) triangular submatrix of A. Hence, U can simply be replaced by L^T in Section 3.3.

4 Adjoint Direct Linear Solver

An adjoint code starts with an (augmented) forward section to record all data required for the data flow reversal due to the propagation of adjoints in the reverse section. See [Nau12a] for details.

4.1 Discrete Version

The chain rule as illustrated in Fig. 1 (c) yields the following discrete adjoint version of the algorithm in Equations (10)–(12):

forward section:

$$\left(\begin{pmatrix} A \\ \mathbf{b} \end{pmatrix}, \tau_0 \right) := P_\downarrow(\mathbf{z}) \tag{41}$$

$$(\mathbf{x}, \tau_1) := S_\downarrow(A, \mathbf{b}) \tag{42}$$

$$(y, \tau_2) := p_\downarrow(\mathbf{x}) \tag{43}$$

reverse section:

$$\mathbf{x}_{(1)} := p_{(1)}(\tau_2, y_{(1)}) \equiv < y_{(1)}, \frac{\partial y}{\partial \mathbf{x}} > \tag{44}$$

$$\begin{pmatrix} A_{(1)} \\ \mathbf{b}_{(1)} \end{pmatrix} := S_{(1)}(\tau_1, \mathbf{x}_{(1)}) \equiv \begin{pmatrix} < \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial A} > \\ < \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial \mathbf{b}} > \end{pmatrix} \tag{45}$$

$$\mathbf{z}_{(1)} := P_{(1)}(\tau_0, A_{(1)}, \mathbf{b}_{(1)}) \equiv < \begin{pmatrix} A_{(1)} \\ \mathbf{b}_{(1)} \end{pmatrix}, \begin{pmatrix} \frac{\partial A}{\partial \mathbf{z}} \\ \frac{\partial \mathbf{b}}{\partial \mathbf{z}} \end{pmatrix} > . \tag{46}$$

Data that is required for the correct evaluation of adjoints in the reverse section is recorded on a *tape* $\tau = (\tau_0, \tau_1, \tau_2)$ by running $P_\downarrow$, $S_\downarrow$, and $p_\downarrow$ in the forward section. Typically, the term *tape* is used in the context of implementations of AD by overloading. Here we allow for a relaxed interpretation that includes required data stored in the augmented forward sections of adjoint codes generated by source code transformation.

The size of the memory occupied by τ_1 is $O(n^3)$. The computational complexity of its interpretation in Equation (45) is also $O(n^3)$. Both can be reduced to $O(n^2)$ through the exploitation of mathematical insight in the following section.

4.2 Continuous Version

We aim to avoid the recording of the tape τ_1 in Equation (42) and hence its interpretation in Equation (45) yielding the following adjoint code:

forward section:

$$\left(\begin{pmatrix} A \\ \mathbf{b} \end{pmatrix}, \tau_0 \right) := P_\downarrow(\mathbf{z}) \tag{47}$$

$$\mathbf{x} := S(A, \mathbf{b}) \tag{48}$$

$$(y, \tau_2) := p_\downarrow(\mathbf{x}) \tag{49}$$

reverse section:

$$\mathbf{x}_{(1)} := p_{(1)}(\tau_2, y_{(1)}) \equiv < y_{(1)}, \frac{\partial y}{\partial \mathbf{x}} > \tag{50}$$

$$\begin{pmatrix} A_{(1)} \\ \mathbf{b}_{(1)} \end{pmatrix} := S_{(1)}(A, \mathbf{b}, \mathbf{x}_{(1)}) \equiv \begin{pmatrix} < \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial A} > \\ < \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial \mathbf{b}} > \end{pmatrix} \tag{51}$$

$$\mathbf{z}_{(1)} := P_{(1)}(\tau_0, A_{(1)}, \mathbf{b}_{(1)}) \equiv < \begin{pmatrix} A_{(1)} \\ \mathbf{b}_{(1)} \end{pmatrix}, \begin{pmatrix} \frac{\partial A}{\partial \mathbf{z}} \\ \frac{\partial \mathbf{b}}{\partial \mathbf{z}} \end{pmatrix} > . \tag{52}$$

It remains to show how to evaluate Equation (51).

Computation of $\mathbf{b}_{(1)} \equiv < \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial \mathbf{b}} >$ From Equation (26) follows

$$\frac{\partial \mathbf{x}}{\partial \mathbf{b}} = A^{-1}. \tag{53}$$

Multiplication of both sides of Equation (53) with $\mathbf{x}_{(1)}^T$ from the left yields

$$\mathbf{x}_{(1)}^T \cdot \frac{\partial \mathbf{x}}{\partial \mathbf{b}} = < \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial \mathbf{b}} >^T = \mathbf{x}_{(1)}^T \cdot A^{-1} \tag{54}$$

and hence

$$< \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial \mathbf{b}} >^T \cdot A = \mathbf{x}_{(1)}^T. \tag{55}$$

Transposing Equation (55) gives

$$A^T \cdot < \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial \mathbf{b}} >= A^T \cdot \mathbf{b}_{(1)} = \mathbf{x}_{(1)} \tag{56}$$

whose solution can be obtained at the computational cost of $O(n^2)$ using the previously computed factorization of A.

Computation of $A_{(1)} \equiv < \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial A} >$ Projection of the leading dimension of Equation (30) in direction $\mathbf{x}_{(1)}$ yields

$$< \mathbf{x}_{(1)}, < \frac{\partial (A\mathbf{x})}{\partial \mathbf{x}}, \frac{\partial \mathbf{x}}{\partial A} >>=< \mathbf{x}_{(1)}, \frac{\partial (A\mathbf{x})}{\partial A} > . \tag{57}$$

Transposition and exploitation of associativity on the left-hand side and application of Equation (56) to the right-hand side yields as an immediate consequence of Equation (18)

$$A^T \cdot \underbrace{< \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial A} >}_{(=A_{(1)})} = - << A^T, \mathbf{b}_{(1)} >, \frac{\partial (A\mathbf{x})}{\partial A} > \tag{58}$$

and, hence,

$$A_{(1)} = - < \mathbf{b}_{(1)}, \frac{\partial (A\mathbf{x})}{\partial A} >= -\mathbf{b}_{(1)} \cdot \mathbf{x}^T \tag{59}$$

as shown in the following lemma.

Lemma 2. *Let $A \in I\!R^{n\times n}$ and $\mathbf{b}_{(1)}, \mathbf{x} \in I\!R^n$. Then*

$$< \mathbf{b}_{(1)}, \frac{\partial (A\mathbf{x})}{\partial A} >= \mathbf{b}_{(1)} \cdot \mathbf{x}^T$$

Proof. With the three-tensor $\frac{\partial (A\mathbf{x})}{\partial A}$ as in Lemma 1 we get

$$\begin{aligned}
\left[< \mathbf{b}_{(1)}, \frac{\partial (A\mathbf{x})}{\partial A} >\right]_{j,k} &= \sum_{i=0}^{n-1} [\mathbf{b}_{(1)}]_i \cdot \left[\frac{\partial (A\mathbf{x})}{\partial A}\right]_{i,j,k} && (60)\\
&\overset{Eq.(35))}{=} [\mathbf{b}_{(1)}]_j \cdot \left[\frac{\partial (A\mathbf{x})}{\partial A}\right]_{j,j,k} && (61)\\
&= [\mathbf{b}_{(1)}]_j \cdot [\mathbf{x}]_k = [\mathbf{b}_{(1)} \cdot \mathbf{x}^T]_{j,k}. && (62)
\end{aligned}$$

Note that the memory overhead induced by the continuous adjoint can even be reduced to $O(n)$ through exploitation of the unit rank of $A_{(1)}$. The integration of this feature into our AD software tool set is the subject of an ongoing development effort.

4.3 Examples

***LU*-Factorization** A reference implementation of the discrete adjoint using `dco` is listed in Appendix C. In the continuous version, Equation (47) is followed by

$$(\mathbf{x}, L, U) = S(A, \mathbf{b}) \tag{63}$$

and by Equation (49) in the forward section and Equation (50) precedes

$$\mathbf{b}_{(1)} = B(L^T, F(U^T, \mathbf{x}_{(1)})) \tag{64}$$

$$A_{(1)} = -\mathbf{b}_{(1)} \cdot \mathbf{x}^T \tag{65}$$

and Equation (52) in the reverse section. Refer to Appendix D for a corresponding implementation that has been validated against the discrete adjoint version from Appendix C. The unit rank of $A_{(1)}$ can be exploited by storing $\mathbf{b}_{(1)}$ and $\mathbf{x}$ instead of $A_{(1)}$ thus reducing the memory requirement locally from $O(n^2)$ to $O(n)$.

***QR*-Factorization** Equation (47) is followed by

$$(\mathbf{x}, Q, R) = S(A, \mathbf{b}) \tag{66}$$

and by Equation (49) in the forward section and Equation (50) precedes

$$\mathbf{b}_{(1)} = Q \cdot F(R^T, \mathbf{x}_{(1)}) \tag{67}$$

$$A_{(1)} = -\mathbf{b}_{(1)} \cdot \mathbf{x}^T \tag{68}$$

and Equation (52) in the reverse section. The computational complexity of the adjoint linear solver is reduced to $O(n^2)$ due to $\mathbf{x}_{(1)} = A^T \cdot \mathbf{b}_{(1)} = (Q \cdot R)^T \cdot \mathbf{b}_{(1)} = R^T \cdot Q^T \cdot \mathbf{b}_{(1)}$ and, hence, $\mathbf{b}^{(1)}$ can be obtained by multiplying the result of the solution of $R^T \cdot (Q^T \cdot \mathbf{b}^{(1)}) = \mathbf{x}_{(1)}$ by forward substitution with $Q^{-T} = Q$.

***LL^T*-Factorization** If A is symmetric positive definite within the range of P, then U can simply be replaced by L^T in Section 4.3 similar to Section 3.3.

5 Case Study

We consider the solution $u^* = u^*(x, z)$ of the one-dimensional linear differential equation

$$\nabla^2(z \cdot u^*) = 0 \quad \text{on} \quad \Omega = (0, 1) \tag{69}$$

$$u^* = 1 \quad \text{and} \quad z = 1 \quad \text{on} \quad \partial\Omega \tag{70}$$

$$\tag{71}$$

with parameter $z = z(x)$. For given measurements $u^m = u^m(x)$ we consider the following parameter estimation problem for z

$$z^* = \arg\min_{z \in I\!R} J(z) \tag{72}$$

with

$$J(z) = \|u^* - u^m\|_2^2 \ . \tag{73}$$

The measurements are generated by a given set of parameters (the wanted parameter distribution $z^*(x)$). An equidistant second-order central finite difference discretization results for a given $\mathbf{u}$ (as in the previous sections, discretized and, hence, vector-valued variables are written as bold letters) in the residual function

$$[\mathbf{r}]_i = \frac{1}{\Delta^2} \cdot ([\mathbf{z}]_{i-1} \cdot [\mathbf{u}]_{i-1} - 2 \cdot [\mathbf{z}]_i \cdot [\mathbf{u}]_i + [\mathbf{z}]_{i+1} \cdot [\mathbf{u}]_{i+1}) \tag{74}$$

with $\Delta = 1/n$ and n the number of discretization points yielding the linear system

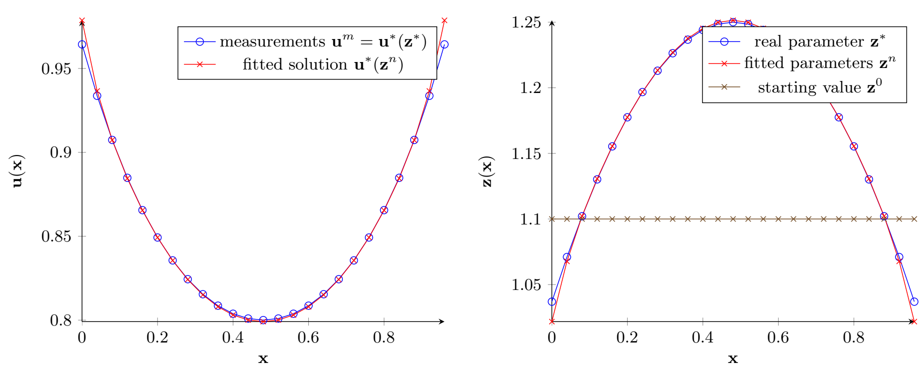
$$\nabla \mathbf{r}|_{\mathbf{u}\equiv 0} \cdot \mathbf{u}^* = -\mathbf{r}|_{\mathbf{u}\equiv 0} \ . \tag{75}$$

To ensure consistency with the notation used throughout the previous sections, we use subscripted square brackets to denote accesses to individual tensor, resp. vector, entries.

In order to solve the parameter estimation problem, we apply a steepest descent algorithm to the discrete objective $J(\mathbf{z})$ as follows

$$\mathbf{z}^{i+1} = \mathbf{z}^i - \nabla J(\mathbf{z}^i) \ , \tag{76}$$

where the computation of the gradient of J at the current iterate $\mathbf{z}^i$ includes the differentiation of the solution process for $\mathbf{u}^*$, i. e., the differentiation of solver for the linear system in Equation (75). Extension of the given example to the use of Quasi-Newton methods (for example, BFGS) is straight forward.



(a) The measured and fitted solution $\mathbf{u}(\mathbf{x})$ (b) The real and the fitted parameter $\mathbf{z}(\mathbf{x})$

Fig. 4. Visualization of the parameter fitting problem Equation (72)

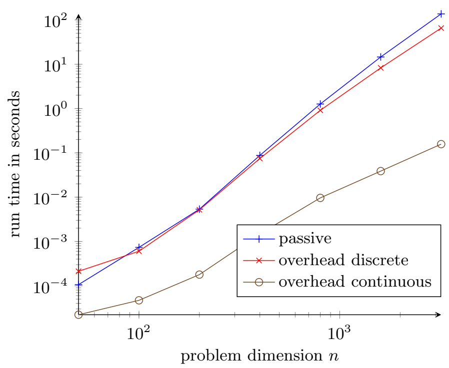
The preprocessor $P(\mathbf{z})$ computes the Jacobian matrix $\nabla \mathbf{r}$ as well as the residual $\mathbf{r}$. The postprocessor $p(\mathbf{u})$ computes the cost functional $J(\mathbf{z})$.

Fig. 4 shows the measured solution $\mathbf{u}^m$, the fitted solution $\mathbf{u}^*(\mathbf{z}^n)$ as well as the starting parameter set $\mathbf{z}^0$, the real (wanted) parameter $\mathbf{z}^*$ and the fitted parameter $\mathbf{z}^n$ after $n = 3$ steepest descent steps.

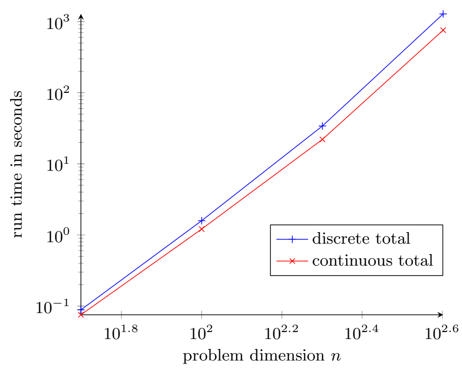
In the following we compare the run time and memory consumption of the linear solver for the discrete and continuous methods and we show the impact on the overall performance of the parameter fitting problem.

5.1 Tangent-Linear Mode

Fig. 5(a) shows in a double logarithmic scale the run time for the solution of the linear system and the overhead introduced by the discrete and continuous methods for the computation of the required derivatives. The overhead of the discrete



(a) Run time overhead – tangent-linear mode



(b) Overall performance – tangent-linear mode

Fig. 5. Performance of tangent-linear mode.

method is approximately identical to the passive execution, which is consistent with the fact that the discrete tangent-linear method roughly duplicates every floating point operation. Both, the passive solution of the linear system and the discrete overhead have a computational complexity of $O(n^3)$. The continuous overhead is merely $O(n^2)$.

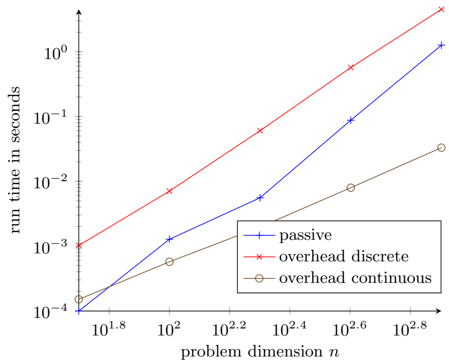
n	speed-up
50	1.17
100	1.31
200	1.54
400	1.69

Table 1. Overall speed-up of continuous versus discrete tangent-linear mode.

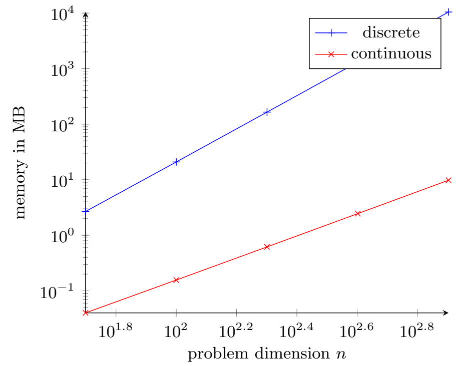
In Fig. 5(b) the overall performance for the steepest descent algorithm is shown. The difference between the continuous and the discrete approaches increases with growing problem dimension n, and even the speed-up increases for the continuous approach (see Table 1).

5.2 Adjoint Mode

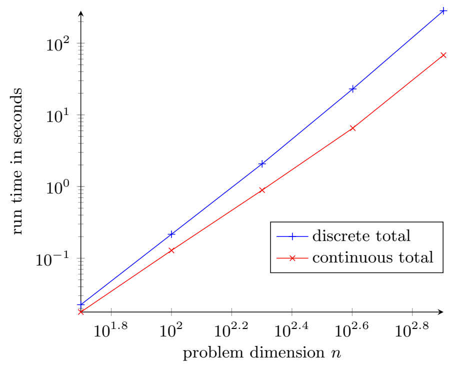
In Fig. 6 a comparison for the discrete and continuous adjoint approaches is shown including the run time overhead for the adjoint computation, the tape memory usage, and the overall performance of the steepest descent algorithm. The ratio of the discrete run time overhead and the passive run time in Fig. 6(a) is ≈ 8, while the ratio is only ≈ 0.03 for the continuous approach. For increasing problem size n, the discrete ratio stays constant, while the continuous ratio will decrease further. The amount of tape memory needed grows – as expected – with $O(n^3)$ and $O(n^2)$ for the discrete and the continuous approaches, respectively (see Fig. 6(b)). The overall speed-up of the steepest descent algorithm increases



(a) Run time overhead – adjoint mode



(b) Tape memory usage



(c) Overall performance – adjoint mode

Fig. 6. Performance of adjoint mode.

significantly faster than in the tangent-linear case as shown in Table 2.

n	speed-up
50	1.26
100	1.69
200	2.33
400	3.5

Table 2. Overall speed-up of continuous versus discrete adjoint mode.

6 Conclusion and Outlook

Simulation codes in Computational Science, Engineering, and Finance contain a hierarchy of calls to numerical algorithms ranging from (basic) linear algebra to optimization routines and including other special function such as interpolation or integration routines. Knowledge about the partial derivatives of the respective relevant outputs with respect to inputs enable us to treat such functions as intrinsic in the context of AD. Significant improvements can be made in terms of computational cost and memory requirement. Each algorithm needs to be analyzed individually for this purpose.

In this paper, we consider direct solvers for systems of n linear equations as potential intrinsics of AD tools. Our analysis shows that AD of such solvers should be avoided. The overhead in the computational cost induced by the propagation of directional derivatives or adjoints can thus be reduced from $O(n^3)$ to $O(n^2)$. The additional memory requirement of adjoint mode becomes $O(n^2)$ (potentially even $O(n)$) compared to $O(n^3)$.

Linear solvers are at the core of many numerical algorithms including nonlinear solvers, optimizers, and interpolation methods. Hence, they should be treated according to the results in this paper even if the enclosing algorithm cannot be made intrinsic in the context of AD. Many of them can (and should) be made AD intrinsics as illustrated by ongoing research.

At the algorithmic level, iterative linear solvers can be treated similar to their direct counterparts. For example, in adjoint mode, the original linear system is solved within the forward section of the enclosing adjoint code followed by the iterative solution of the adjoint linear system in the reverse section. As a proof of concept we have successfully reimplemented the case study from Section 5 using a basic GMRES [SS86] solver. The error in the solution of the adjoint system can be expected to depend on the error in the original solution. A detailed convergence analysis is the subject of ongoing work.

7 Acknowledgement

Johannes Lotz is supported by the German Science Foundation (DFG) under grant number NA487/4-1.

References

[Bau74] F. Bauer. Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11:87–96, 1974.

[BBH+08] C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, volume 64 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2008.

[BCH+06] M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, volume 50 of *Lecture Notes in Computational Science and Engineering*, Berlin, 2006. Springer.

[CG91] G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series. SIAM, 1991.

[DS88] N. Dunford and J. T. Schwartz. *Linear Operators, Part 1, General Theory.* Wiley, 1988.

[Gil08] M. Giles. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In *[BBH+08]*, pages 35–44. 2008.

[GR91] A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markovitz rule. In *[CG91]*, pages 126–135, 1991.

[GV03] A. Griewank and O. Vogel. Analysis and exploitation of Jacobian scarcity. In *Proceedings of HPSC Hanoi*. Springer, 2003.

[GW08] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[HNP05] L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded Analysis in Reverse Mode Automatic Differentiation. *Future Generation Computer Systems*, 21:1401–1417, 2005.

[HPUU09] M. Hinze, R. Pinnau, M. Ulbrich, and S. Ulbrich. *Optimization with PDE Constraints.* Number 23 in Mathematical Modelling: Theory and Applications. Springer, 2009.

[MN02] J. Magnus and H. Neudecker. *Matrix Differential Calculus with Applications in Statistics and Econometrics.* Wiley, 2002.

[Nau04] U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Mathematical Programming*, 99:399–421, 2004.

[Nau08] U. Naumann. Optimal Jacobian accumulation is NP-complete. *Mathematical Programming*, 112:427–441, 2008.

[Nau12a] U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic differentiation.* Number 24 in Software, Environments, and Tools. SIAM, Philadelphia, PA, 2012.

[Nau12b] U. Naumann. Getting started with dco 0.9 and dcc 0.9. http://www.ec-securehost.com/SIAM/SE24.html, 2012.

[SS86] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[TB97] L. N. Trefethen and D. Bau. *Numerical Linear Algebra.* SIAM, June 1997.

[TFK06] M. Tadjouddine, S. Forth, and A. Keane. Adjoint differentiation of a structural dynamics solver. In *[BCH+06]*, pages 309–319. 2006.

A Discrete Tangent-Linear Solver

Throughout this appendix we apply basic LU-factorization followed by forward and backward substitution to the linear system

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{3} \\ \frac{1}{5} & \frac{3}{4} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 9 \\ 11 \end{pmatrix} .$$

Discrete tangent-linear and adjoint versions are generated by version 0.9 of `dco` as discussed in detail in [Nau12a]. `dco` is available for downloading on

`www.siam.org/books/se24`.

There is also a user guide to `dco` and to version 0.9 of the derivative code compiler `dcc`. The following source code is available for downloading under

`www.stce.rwth-aachen.de/publications/Naumann2012DLS`.

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

// declaration of dco's
// tangent-linear 1st-order scalar type
#include "dco_t1s/dco_t1s_type.hpp"

// dimension of linear system
const int n=2;

// L*U
// Factorization in tangent-linear mode
void LU(dco_t1s_type** A) {
  for (int k=0;k<n;k++) {
    for (int i=k+1;i<n;i++) A[i][k]=A[i][k]/A[k][k];
```

```
    for (int j=k+1;j<n;j++)
      for (int i=k+1;i<n;i++)
        A[i][j]=A[i][j]-A[i][k]*A[k][j];
  }
}

// L*b
// Forward substitution in tangent-linear mode
void F(dco_t1s_type** A, dco_t1s_type* b) {
  for (int i=0;i<n;i++)
    for (int j=0;j<i;j++)
      b[i]=b[i]-A[i][j]*b[j];
}

// U*b
// Backward substitution in tangent-linear mode
void B(dco_t1s_type** A, dco_t1s_type* b) {
  for (int i=n-1;i>=0;i--) {
    for (int j=n-1;j>i;j--)
      b[i]=b[i]-A[i][j]*b[j];
    b[i]=b[i]/A[i][i];
  }
}

int main() {
// allocation of active data as of dco's
// tangent-linear 1st-order scalar type
  dco_t1s_type *b=new dco_t1s_type[n];
  dco_t1s_type **A=new dco_t1s_type*[n];
  for (int i=0;i<n;i++) A[i]=new dco_t1s_type[n];

// Jacobian in discrete tangent-linear mode
//
// directional derivatives of A range over
// the Cartesian basis vectors in R^(n x n)
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++) {
      A[0][0]=1./2; A[0][1]=1./3; A[1][0]=1./5; A[1][1]=3./4;
      b[0]=9.; b[1]=11.;
      A[i][j].t=1;
      LU(A); F(A,b); B(A,b);
      cout << "dx/dA[" << i << "][" << j << "]=( ";
      for (int ii=0;ii<n;ii++) cout << b[ii].t << " ";
      cout << ")" << endl;
    }
// directional derivatives of b range over
// the Cartesian basis vectors in R^n
  for (int i=0;i<n;i++) {
    A[0][0]=1./2; A[0][1]=1./3; A[1][0]=1./5; A[1][1]=3./4;
    b[0]=9.; b[1]=11.;
    b[i].t=1;
    LU(A); F(A,b); B(A,b);
    cout << "dx/db[" << i << "]=( ";
    for (int j=0;j<n;j++) cout << b[j].t << " ";
    cout << ")" << endl;
  }

// deallocation of active data
  for (int i=0;i<n;i++) delete [] A[i];
```

```
  delete [] A; delete [] b;
  return 0;
}
```

Compilation of the source file and linkage with the implementation of dco's tangent-linear 1st-order scalar type dco_t1s_type yields an executable that generates the following output:

```
dx/dA[0][0]=(-24.3243, 6.48649)
dx/dA[0][1]=(-29.1892, 7.78378)
dx/dA[1][0]=(10.8108, -16.2162)
dx/dA[1][1]=(12.973, -19.4595)
dx/db[0]=(2.43243, -0.648649)
dx/db[1]=(-1.08108, 1.62162)
```

B Continuous Tangent-Linear Solver

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

// dimension of linear system
const int n=2;

// L*U
// Result of factorization to be reused
// for computation of directional derivatives
// with respect to system matrix and right-hand side
void LU(double** A) {
  for (int k=0;k<n;k++) {
    for (int i=k+1;i<n;i++) A[i][k]=A[i][k]/A[k][k];
    for (int j=k+1;j<n;j++)
      for (int i=k+1;i<n;i++)
        A[i][j]=A[i][j]-A[i][k]*A[k][j];
  }
}

// L*b
// Forward substitution required for solution
// and for directional derivative with respect to
// right-hand side
void F(double** A, double* b) {
  for (int i=0;i<n;i++)
    for (int j=0;j<i;j++)
      b[i]=b[i]-A[i][j]*b[j];
}

// U*b
// Backward substitution required for solution
// and for directional derivative with respect to
// right-hand side
void B(double** A, double* b) {
  for (int i=n-1;i>=0;i--) {
    for (int j=n-1;j>i;j--)
      b[i]=b[i]-A[i][j]*b[j];
    b[i]=b[i]/A[i][i];
  }
}
```

```
int main() {
// duplication of active data segment
  double *b=new double[n];
  double *b_t1s=new double[n];
  double **A=new double*[n];
  for (int i=0;i<n;i++) A[i]=new double[n];
  double **A_t1s=new double*[n];
  for (int i=0;i<n;i++) A_t1s[i]=new double[n];

//  Jacobian in continuous tangent-linear mode
  A[0][0]=1./2; A[0][1]=1./3; A[1][0]=1./5; A[1][1]=3./4;
  b[0]=9.; b[1]=11.;
// solution of linear system
  LU(A); F(A,b); B(A,b);

// directional derivatives A_t1s of A range over
// the Cartesian basis vectors in R^(n x n)
// to compute right-hand sides (see Eq. (38))
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++) {
      for (int ii=0;ii<n;ii++)
        for (int jj=0;jj<n;jj++) A_t1s[ii][jj]=0;
      A_t1s[i][j]=1;
      for (int ii=0;ii<n;ii++) {
        b_t1s[ii]=0;
        for (int jj=0;jj<n;jj++) b_t1s[ii]-=A_t1s[ii][jj]*b[jj];
      }
// existing factorization of A is reused to solve the
// tangent-linear sytem
    F(A,b_t1s); B(A,b_t1s);
    cout << "dx/dA[" << j << "][" << i << "]=( ";
    for (int ii=0;ii<n;ii++) cout << b_t1s[ii] << " ";
    cout << ")" << endl;
  }

// directional derivatives of b range over
// the Cartesian basis vectors in R^n
// yielding right-hand sides (see Eq. (38))
  for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++) b_t1s[j]=0;
    b_t1s[i]=1;
// existing factorization of A is reused to solve the
// tangent-linear sytem
    F(A,b_t1s); B(A,b_t1s);
    cout << "dx/db[" << i << "]=( ";
    for (int j=0;j<n;j++) cout << b_t1s[j] << " ";
    cout << ")" << endl;
  }

// deallocation of activated data segment
  for (int i=0;i<n;i++) delete [] A[i];
  for (int i=0;i<n;i++) delete [] A_t1s[i];
  delete [] A; delete [] b;
  delete [] A_t1s; delete [] b_t1s;
  return 0;
}
```

Compilation of the source file yields an executable that generates the same output as shown in Appendix A.

C Discrete Adjoint Solver

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

// declaration of dco's
// adjoint 1st-order scalar type
#include "dco_a1s/dco_a1s_type.hpp"

// dimension of linear system
const int n=2;

// L*U
// Factorization is recorded on tape
void LU(dco_a1s_type** A) {
  for (int k=0;k<n;k++) {
    for (int i=k+1;i<n;i++) A[i][k]=A[i][k]/A[k][k];
    for (int j=k+1;j<n;j++)
      for (int i=k+1;i<n;i++)
        A[i][j]=A[i][j]-A[i][k]*A[k][j];
  }
}

// L*b
// Forward substitution is recorded on tape
void F(dco_a1s_type** A, dco_a1s_type* b) {
  for (int i=0;i<n;i++)
    for (int j=0;j<i;j++)
      b[i]=b[i]-A[i][j]*b[j];
}

// U*b
// Backward substitution is recorded on tape
void B(dco_a1s_type** A, dco_a1s_type* b) {
  for (int i=n-1;i>=0;i--) {
    for (int j=n-1;j>i;j--)
      b[i]=b[i]-A[i][j]*b[j];
    b[i]=b[i]/A[i][i];
  }
}

int main() {
// allocation of active data as of dco's
// adjoint 1st-order scalar type
  dco_a1s_type *b=new dco_a1s_type[n];
  dco_a1s_type **A=new dco_a1s_type*[n];
  for (int i=0;i<n;i++) A[i]=new dco_a1s_type[n];
// tape
  extern dco_a1s_tape_entry dco_a1s_tape[DCO_A1S_TAPE_SIZE];

// Jacobian in discrete adjoint mode
  for (int i=0;i<n;i++) {
    A[0][0]=1./2; A[0][1]=1./3; A[1][0]=1./5; A[1][1]=3./4;
    int va_A[n*n]={A[0][0].va,A[0][1].va,A[1][0].va,A[1][1].va};
    b[0]=9.; b[1]=11.;
```

```
    int va_b[n]={b[0].va,b[1].va};
// Solution procedure gets recorded on tape
    LU(A); F(A,b); B(A,b);
// adjoints of solution range over
// the Cartesian basis vectors in R^n
    dco_a1s_tape[b[i].va].a=1;
// tape gets interpreted in current adjoint direction
    dco_a1s_interpret_tape();
    cout << "dx[" << i << "]/db=( ";
    for (int j=0;j<n;j++) cout << dco_a1s_tape[va_b[j]].a << " ";
    cout << ")" << endl;
    cout << "dx[" << i << "]/dA=( ";
    for (int j=0;j<n;j++)
      for (int k=0;k<n;k++)
        cout << dco_a1s_tape[va_A[j*n+k]].a << " ";
    cout << ")" << endl;
    dco_a1s_reset_tape();
  }

// deallocation of active data
  for (int i=0;i<n;i++) delete [] A[i];
  delete [] A; delete [] b;
  return 0;
}
```

Compilation of the source file and linkage with the implementation of `dco`'s adjoint 1st-order scalar type dco_a1s_type yields an executable that generates the following output:

```
dx[0]/db=( 2.43243 -1.08108 )
dx[0]/dA=( -24.3243 -29.1892 10.8108 12.973 )
dx[1]/db=( -0.648649 1.62162 )
dx[1]/dA=( 6.48649 7.78378 -16.2162 -19.4595 )
```

D Continuous Adjoint Solver

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

// dimension of linear system
const int n=2;

// L*U
// Result of factorization to be reused
// for computation of adjoints
// with respect to right-hand side
void LU(double** A) {
  for (int k=0;k<n;k++) {
    for (int i=k+1;i<n;i++) A[i][k]=A[i][k]/A[k][k];
    for (int j=k+1;j<n;j++)
      for (int i=k+1;i<n;i++)
        A[i][j]=A[i][j]-A[i][k]*A[k][j];
  }
}

// L*b
// Forward substitution required for solution
// of linear system
```

```
void F(double** A, double* b) {
  for (int i=0;i<n;i++)
    for (int j=0;j<i;j++)
      b[i]=b[i]-A[i][j]*b[j];
}

// U*b
// Backward substitution required for solution
// of linear system
void B(double** A, double* b) {
  for (int i=n-1;i>=0;i--) {
    for (int j=n-1;j>i;j--)
      b[i]=b[i]-A[i][j]*b[j];
    b[i]=b[i]/A[i][i];
  }
}

// L*b
// Transposed forward substitution required for
// adjoints with respect to right-hand side
void F_a1s(double** A, double* b) {
  for (int i=0;i<n;i++) {
    for (int j=0;j<i;j++)
      b[i]=b[i]-A[i][j]*b[j];
    b[i]=b[i]/A[i][i];
  }
}

// U*b
// Transposed backward substitution required for
// adjoints with respect to right-hand side
void B_a1s(double** A, double* b) {
  for (int i=n-1;i>=0;i--)
    for (int j=n-1;j>i;j--)
      b[i]=b[i]-A[i][j]*b[j];
}

int main() {
// duplication of active data segment
  double *b=new double[n];
  double *b_a1s=new double[n];
  double **A=new double*[n];
  for (int i=0;i<n;i++) A[i]=new double[n];
  double **A_a1s=new double*[n];
  for (int i=0;i<n;i++) A_a1s[i]=new double[n];

//  Jacobian in continuous adjoint mode
  A[0][0]=1./2; A[0][1]=1./3; A[1][0]=1./5; A[1][1]=3./4;
  b[0]=9.; b[1]=11.;
// solution of linear system
  LU(A); F(A,b); B(A,b);

  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++) A_a1s[i][j]=A[j][i];

// adjoints b_a1s of solution range over
// the Cartesian basis vectors in R^n
// to compute right-hand sides (see Eq. (64))
  for (int i=0;i<n;i++) {
```

```
    for (int j=0;j<n;j++) b_a1s[j]=0;
    b_a1s[i]=1;
    F_a1s(A_a1s,b_a1s); B_a1s(A_a1s,b_a1s);
    cout << "dx[" << i << "]/db=( ";
    for (int j=0;j<n;j++) cout << b_a1s[j] << " ";
    cout << ")" << endl;
// rank-1 adjoint with respect to system matrix
// (see Eq. (65))
    cout << "dx[" << i << "]/dA=( ";
    for (int ii=0;ii<n;ii++)
      for (int jj=0;jj<n;jj++)
        cout << -b_a1s[ii]*b[jj] << " ";
    cout << ")" << endl;
  }

// deallocation of activated data segment
  for (int i=0;i<n;i++) delete [] A[i];
  for (int i=0;i<n;i++) delete [] A_a1s[i];
  delete [] A; delete [] b;
  delete [] A_a1s; delete [] b_a1s;
  return 0;
}
```

Compilation of the source file yields an executable that generates the same output as shown in Appendix C.

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`

2009-01 * Fachgruppe Informatik: Jahresbericht 2009
2009-02 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications
2009-03 Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices
2009-05 George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs
2009-06 George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete
2009-07 Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I
2009-08 Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs
2009-09 Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem
2009-10 Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm
2009-11 Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs
2009-12 Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes
2009-13 Martin Zimmermann: Time-optimal Winning Strategies for Poset Games
2009-14 Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)
2009-15 Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems
2009-16 George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs
2009-17 Carsten Kern: Learning Communicating and Nondeterministic Automata
2009-18 Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies
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 domänenspezifischen 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 Taolue 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 Armoush: 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
2010-16 Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles
2010-17 Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten
2010-18 Ibrahim Armaç: 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

2011-11 Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains

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

2011-18 Kamal Barakat: Introducing Timers to pi-Calculus

2011-19 Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode

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

2012-05 Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting

2012-06 Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data

2012-08 Hongfei Fu: Computing Game Metrics on Markov Decision Processes

* These reports are only available as a printed version.

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