RWTH Aachen

Department of Computer Science
Technical Report

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

Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2012-15

RWTH Aachen · Department of Computer Science · Jan. 2013 (revised version)

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

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

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

Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara

Lehr- und Forschungsgebiet Informatik 12:
Software and Tools for Computational Engineering
RWTH Aachen University, Aachen, Germany
Email: {naumann,lotz,leppkes,towara}@stce.rwth-aachen.de

Abstract. We discuss software tool support for the Algorithmic Differentiation (also known as Automatic Differentiation; AD) of numerical simulation programs that contain calls to solvers for parameterized systems of n nonlinear equations. The local computational overhead as well as the additional memory requirement for the computation of directional derivatives or adjoints of the solution of the nonlinear system with respect to the parameters can quickly become prohibitive for large values of n. Both are reduced drastically by analytical (in the following also: continuous) approaches to differentiation of the underlying numerical methods. Following the discussion of the proposed terminology we develop the algorithmic formalism building on prior work by other colleagues and we present an implementation based on the AD software `dco/c++`. A representative case study supports the theoretically obtained computational complexity results with practical run time measurements.

1 Introduction, Terminology, and Summary of Results

We consider the computation of first-order directional derivatives $\mathbf{x}^{(1)} \in I\!R^n$ (also: tangents) and adjoints $\boldsymbol{\lambda}_{(1)} \in I\!R^m$ for solvers of parameterized systems of nonlinear equations described by the residual

$$\mathbf{r} = F(\mathbf{x}, \boldsymbol{\lambda}) : I\!R^n \times I\!R^m \to I\!R^n. \tag{1}$$

For $\boldsymbol{\lambda} \in I\!R^m$, a vector $\mathbf{x} = \mathbf{x}(\boldsymbol{\lambda}) \in I\!R^n$ is sought such that $F(\mathbf{x}, \boldsymbol{\lambda}) = 0$. In order to provide a context for the differentiation of nonlinear solvers and without loss of generality, the nonlinear solver is assumed to be embedded (see also Section 2) into the unconstrained convex nonlinear programming problem (NLP)

$$\min_{\mathbf{z} \in I\!R^q} f(\mathbf{z})$$

for a given objective function $f : I\!R^q \to I\!R$. In the context of first-order derivative-based methods (e.g., Steepest Descent or Quasi-Newton methods such as BFGS [3, 25] if the number of parameters q exceeds the number of state variables n considerably) for the solution of the NLP the gradient of $y = f(\mathbf{z}) \in I\!R$ with respect to $\mathbf{z} \in I\!R^q$ needs to be computed, which involves the differentiation of the nonlinear solver itself.

Algorithmic Differentiation (AD) [15, 23] 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`. Numerous successful

applications of AD are described in the proceedings of so far six international conferences on the subject; see, for example, [4, 2, 8].

Traditionally, AD tools take a fully *discrete* approach to differentiation by transforming the given source code at the level of arithmetic operators and built-in[1] functions. Potentially complex numerical kernels, for example, matrix products or the solvers for systems of linear and nonlinear 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 (also: *primal*) kernel for the evaluation of directional derivatives and/or of adjoints, thus, potentially reducing the computational overhead induced by differentiation. For direct solvers for dense systems of n linear equations mathematical insight yields a reduction of the overhead from $O(n^3)$ to $O(n^2)$ [6, 11]. These results are built upon in this paper in the context of continuous differentiation methods applied to different levels of (Newton-type) numerical solution algorithms for systems of nonlinear equations.

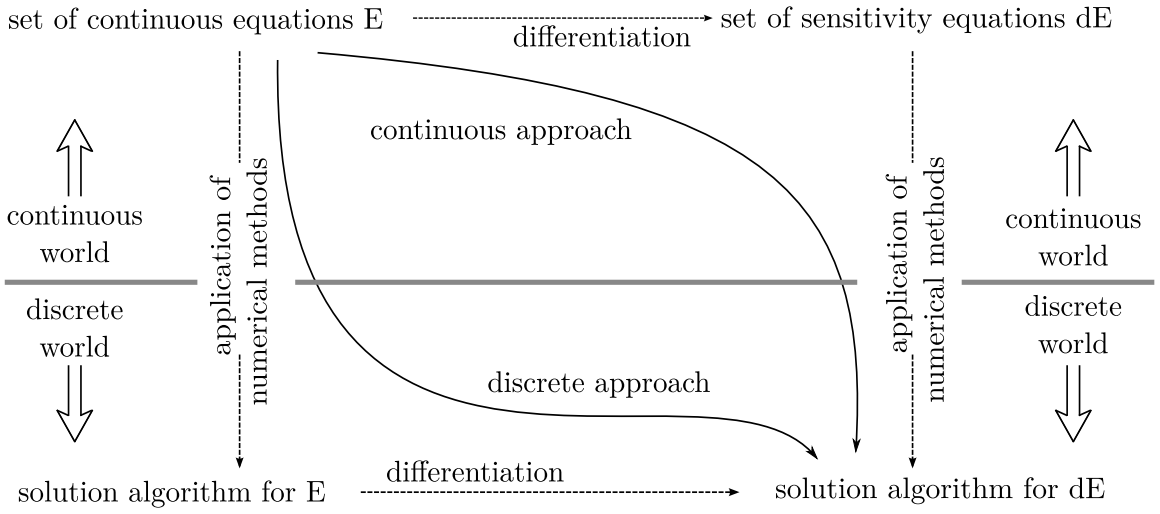


Fig. 1. Illustration of continuous and discrete approaches to differentiation

Fig. 1 illustrates our use of the term *continuous*. In general, the primal problem is assumed to be given as a set E of (potentially nonlinear [[partial] differential]) continuous equations. As a very simple example we consider the parameterized nonlinear equation $x^2 - \lambda = 0$ with *free parameter* $\lambda \in \mathbb{R}$ and *state variable* $x \in \mathbb{R}$; the scenario is illustrated by Fig. 2. Analytic derivation of the corresponding continuous (tangent-linear or adjoint) sensitivity equations yields a new set dE of continuous equations (upper right corner of Fig. 1). Their numerical solution delivers (approximations of) tangent-linear or adjoint sensitivities (lower right corner of Fig. 1). For example, the continuous differentiation of $x^2 - \lambda = 0$ with respect to λ yields

$$\frac{\partial(x^2 - \lambda)}{\partial \lambda} \cdot \lambda^{(1)} = \left(2 \cdot x \cdot \frac{\partial x}{\partial \lambda} - 1\right) \cdot \lambda^{(1)} = 0$$

[1] ... into the given programming language

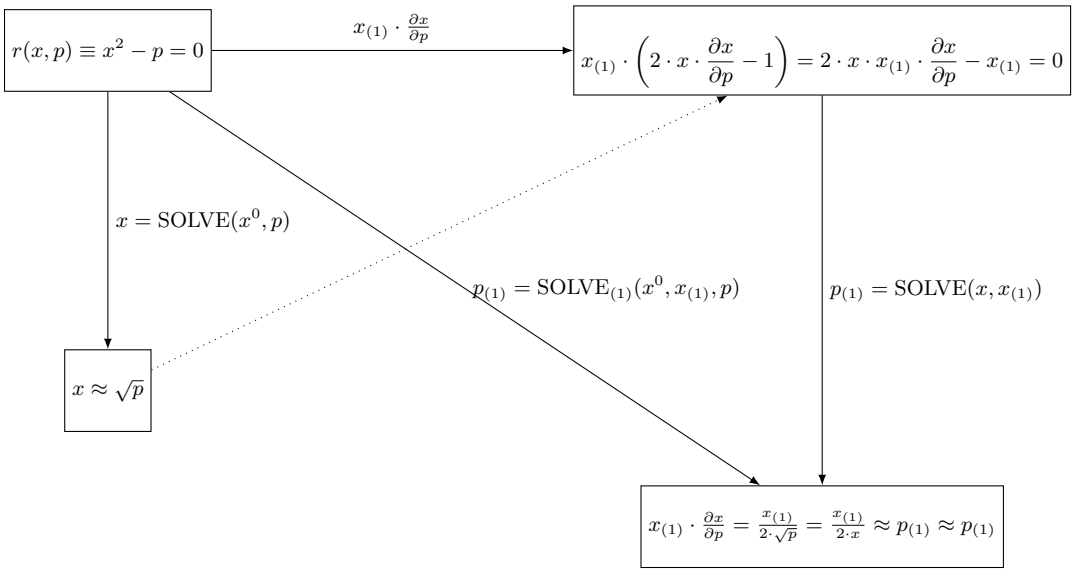


Fig. 2. Example

for some $\lambda^{(1)} \in \mathbb{R}$ (see Section 2.1 for details on the notation) in tangent-linear mode and

$$x_{(1)} \cdot \frac{\partial(x^2 - \lambda)}{\partial \lambda} = x_{(1)} \cdot \left(2 \cdot x \cdot \frac{\partial x}{\partial \lambda} - 1\right) = 0$$

for some $x_{(1)} \in \mathbb{R}$ in adjoint mode. Note that the primal solution x enters the definitions of both the tangent-linear and adjoint sensitivity equations. Numerical solutions of the latter yield approximations of the tangent-linear and adjoint sensitivities, that is,

$$\frac{\partial x}{\partial \lambda} \cdot \lambda^{(1)} = \frac{\lambda^{(1)}}{2 \cdot \sqrt{\lambda}}$$

and

$$x_{(1)} \cdot \frac{\partial x}{\partial \lambda} = \frac{x_{(1)}}{2 \cdot \sqrt{\lambda}},$$

respectively. The solution of the sensitivity equations turns out to be trivial for scalar nonlinear equations. In general, it will involve potentially sophisticated numerical methods, for example, the solution of a linear system when considering systems of nonlinear equations.

In *discrete* differentiation mode the solution method for the primal problem is differentiated (transition from lower left to lower right corner in Fig. 1). For example, the ν Newton iterations performed for the computation of $x \approx \sqrt{\lambda}$ by solving the primal equation $f(x, \lambda) = x^2 - \lambda = 0$ numerically are transformed according to the principles of AD. In tangent-linear mode the primal Newton iterations

$$x_{i+1} = x_i - \frac{f(x_i, \lambda)}{f_x(x_i, \lambda)} = x_i - \frac{x_i^2 - \lambda}{2 \cdot x_i}$$

are transformed into

$$\begin{aligned} x_{i+1} &= x_i - \frac{f}{f_x} \\ x_{i+1}^{(1)} &= x_i^{(1)} - \left(\frac{f_\lambda + f_x \cdot x_\lambda}{f_x} - \frac{(f_{x,\lambda} + f_{x,x} \cdot x_\lambda) \cdot f}{f_x^2}\right) \cdot \lambda^{(1)} \end{aligned}$$

$$
\begin{aligned}
&= x_i^{(1)} - \frac{f_\lambda}{f_x} \cdot \lambda^{(1)} - \frac{f_x \cdot x_\lambda}{f_x} \cdot \lambda^{(1)} + \frac{f_{x,\lambda} \cdot f}{f_x^2} \cdot \lambda^{(1)} + \frac{f_{x,x} \cdot x_\lambda \cdot f}{f_x^2} \cdot \lambda^{(1)} \\
&= \left(\frac{f_{x,\lambda} \cdot f}{f_x^2} - \frac{f_\lambda}{f_x} \right) \cdot \lambda^{(1)} + \frac{f_{x,x} \cdot f}{f_x^2} \cdot x_i^{(1)} \\
&= \frac{1}{2 \cdot x_i} \cdot \lambda^{(1)} + \frac{x_i^2 - \lambda}{2 \cdot x_i^2} \cdot x_i^{(1)}
\end{aligned}
$$

for $i = 0, \ldots, \nu - 1$ and for given $x_0, \lambda, x_0^{(1)}, \lambda^{(1)} \in I\!R$. The following notation is used:

$$
x_\lambda = \frac{\partial x_i}{\partial \lambda},\ f_x = \frac{\partial f}{\partial x}(x_i, \lambda),\ f_\lambda = \frac{\partial f}{\partial \lambda}(x_i, \lambda),\ f_{x,x} = \frac{\partial^2 f}{\partial x^2}(x_i, \lambda),\ f_{x,\lambda} = \frac{\partial^2 f}{\partial x \partial \lambda}(x_i, \lambda).
$$

Initialization of $\lambda^{(1)}$ with 1 and of $x_0^{(1)}$ with 0 yields the partial derivative of the primal solution x_ν with respect to λ in $x_{\nu^{(1)}}$.

Adjoint mode AD applied to the primal Newton iterations yields the *forward section*

$$
x_{i+1} = x_i - \frac{f(x_i, \lambda)}{f_x(x_i, \lambda)}, \quad i = 0, \ldots, \nu - 1
$$

followed by the *reverse section*

$$
\begin{aligned}
x_{i(1)} &= x_{i+1(1)} \cdot \left(1 - \frac{f_x}{f_x} + \frac{f_{x,x} \cdot f}{f_x^2} \right) = x_{i+1(1)} \cdot \frac{f_{x,x} \cdot f}{f_x^2} = \frac{x_{i+1(1)}}{2} \cdot \left(1 - \frac{\lambda}{x_i^2} \right) \\
\lambda_{(1)} &= \lambda_{(1)} + x_{i+1(1)} \cdot \left(x_\lambda - \frac{f_\lambda + f_x \cdot x_\lambda}{f_x} + \frac{(f_{x,\lambda} + f_{x,x} \cdot x_\lambda) \cdot f}{f_x^2} \right) \\
&= \lambda_{(1)} \cdot \left(1 + \frac{f_{x,x} \cdot f}{f_x^2} \right) - x_{i+1(1)} \cdot \left(\frac{f_\lambda}{f_x} - \frac{f_{x,\lambda} \cdot f}{f_x^2} \right) \\
&= \lambda_{(1)} \cdot \left(1 + \frac{x^2 - \lambda}{2 \cdot x^2} \right) + \frac{x_{i+1(1)}}{2 \cdot x}
\end{aligned}
$$

for $i = \nu - 1, \ldots, 0$ and for given $x_0, \lambda, \lambda_{(1)}, x_{\nu(1)} \in I\!R$. Initialization of $\lambda_{(1)}$ with 0 and of $x_{\nu(1)}$ with 1 yields the partial derivative of the primal solution x_ν with respect to λ in $\lambda_{(1)}$.

Consistency of the continuous and discrete approaches to the differentiation of nonlinear solvers is shown in [16] and discussed further in [15]; see Section 3.2. Refer also to [5] for a related discussion in the context of attractive fixed point solvers.

In this paper we aim for further algorithmic formalization of the treatment of nonlinear solvers from the perspective of AD tool development. For our example, the directional derivative

$$
\frac{\partial x}{\partial \lambda} \cdot \lambda^{(1)} \approx \frac{\lambda^{(1)}}{2 \cdot \sqrt{\lambda}}
$$

or the adjoint

$$
x_{(1)} \cdot \frac{\partial x}{\partial \lambda} \approx \frac{x_{(1)}}{2 \cdot \sqrt{\lambda}}
$$

are computed. In finite precision arithmetic an approximate solution of the primal equations E yields approximate sensitivities in discrete differentiation mode. The primal solution enters the continuous sensitivity equations dE. The numerical solution of dE will produce approximate sensitivities that will generally not match those obtained in discrete mode exactly. This effect

ϵ	discrete adjoint	continuous adjoint	$\frac{1}{2\cdot\sqrt{\lambda}}$
10^{-1}	0.35291073699211	0.35347538461395	0.353553390593274
10^{-2}	0.35355324021214	0.35355338198598	0.353553390593274
10^{-3}	0.35355324021214	0.35355338198598	0.353553390593274
10^{-4}	0.35355339059327_	0.353553390593274	0.353553390593274
10^{-5}	0.35355339059327_	0.353553390593274	0.353553390593274
10^{-6}	0.35355339059327_	0.353553390593274	0.353553390593274
10^{-7}	0.35355339059327_	0.353553390593274	0.353553390593274
10^{-8}	0.353553390593274	0.353553390593274	0.353553390593274

Table 1. Consistency of differentiation of the numerical solution x of the equation $x^2 - \lambda = 0$ with respect to λ: We list 15 significant digits (using double precision IEEE 754 floating-point arithmetic) of the results obtained in discrete and continuous modes for increasing accuracy of the primal Newton iteration according to the termination criterion $|x_i^2 - \lambda| < \epsilon$ and we compare them with the exact derivative $\frac{\partial x}{\partial \lambda} = \frac{1}{2\cdot\sqrt{\lambda}}$ for $\lambda = 2$. Discrepancies are underlined.

is illustrated in Table 1. Potential discrepancies are due to approximate primal solutions as well as possibly different solution methods for primal and sensitivity equations (including different discretization schemes for nonlinear [[partial] differential] equations).

The choice between continuous and discrete differentiation methods can be made at various levels of a given numerical method. For example, in the present context of a multidimensional Newton method the nonlinear system itself as well as the linear system to be solved in each Newton step can be treated in either way. Table 2 summarizes the computational complexities of the various approaches to differentiation of Newton's algorithm for the solution of systems of n nonlinear equations assuming a dense Jacobian of the residual. The performance of the different approaches depends on the number of Newton-iterations ν and on the problem size n. Discrete differentiation of the nonlinear solver corresponds to a straight application of AD without taking any mathematical or structural properties of the numerical method into account. It turns out to be the worst approach in terms of computational efficiency. Its main advantage is that correct derivatives of the actually performed Newton-iterations are computed independent of whether convergence has been achieved or not. Continuous differentiation delivers approximations of the derivatives the accuracy of which also depends on the quality of the primal solution. Continuous differentiation of the embedded linear solver yields an improvement over both the discrete and continuous approaches to the differentiation of the nonlinear solver. The only downside is that the associated additional persistent memory requirement (only applicable in adjoint mode; see Section 4) exceeds that of the continuous method by a factor of ν.

2 Foundations

For further illustration, f is decomposed as

$$y = f(\mathbf{z}) = p(S(\mathbf{x}^0, \boldsymbol{\lambda})) = p(S(\mathbf{x}^0, P(\mathbf{z}))), \tag{2}$$

where $P : \mathbb{R}^q \to \mathbb{R}^m$ denotes the part of the computation that precedes the nonlinear solver $S : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ and where $p : \mathbb{R}^n \to \mathbb{R}$ maps the result $\tilde{\mathbf{x}}$ onto the scalar objective y. Conceptually, many real-world problems fit into this category, for example, parameter estimation problems for mathematical models that involve the solution of nonlinear partial differential equations. Refer to Section 6 for a case study.

		discrete NLS	continuous NLS	continuous LS
tangent-linear mode (run time)		$\nu \cdot O(n^3)$	$O(n^3)$	$\nu \cdot O(n^2)$
adjoint mode	(memory)	$\nu \cdot O(n^3)$	$O(n^2)$	$\nu \cdot O(n^2)$
	(run time)	$\nu \cdot O(n^3)$	$O(n^3)$	$\nu \cdot O(n^2)$

Table 2. Computational complexities of discrete and continuous tangent-linear and adjoint modes of differentiation for ν Newton-iterations applied to systems of n nonlinear equations: Continuous differentiation can be applied at the level of the nonlinear (continuous NLS mode) and (here dense) linear systems. Fully discrete treatment of the nonlinear solver (discrete NLS mode) implies the discrete differentiation of the linear solver. Alternatively, a continuously differentiated linear solver can be embedded into a discrete nonlinear solver (continuous LS mode).

The discussion in this paper will be based on the following algorithmic description of Equation (2):

$$\boldsymbol{\lambda} := P(\mathbf{z}); \quad \tilde{\mathbf{x}} := S(\mathbf{x}^0, \boldsymbol{\lambda}); \quad y := p(\tilde{\mathbf{x}}). \tag{3}$$

The parameters $\boldsymbol{\lambda} \in I\!R^m$ are computed as a function of $\mathbf{z} \in I\!R^q$ by the given implementation of P. They enter the nonlinear solver S as arguments alongside with the given initial estimate $\mathbf{x}^0 \in I\!R^n$ of the solution $\mathbf{x} \in I\!R^n$. Finally, the computed approximation $\tilde{\mathbf{x}}$ of the solution $\mathbf{x}$ is reduced to a scalar objective value y by the given implementation of p.

2.1 Algorithmic Differentiation

We recall some crucial elements of AD described in further detail in [15, 23]. Without loss of generality, the following discussion will be based on the residual function in Equation (1). Let therefore

$$\mathbf{u} \equiv \begin{pmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{pmatrix} \in I\!R^h$$

and $h = n + m$. AD yields semantical transformations of the given implementation of $F : I\!R^h \to I\!R^n$ as a computer program into first and potentially also higher (k-th order) derivative code. For this purpose F is assumed to be k times continuously differentiable for $k = 1, 2, \ldots$. In the following we use the notation from [23].

The given implementation of F is assumed to decompose into a *single assignment code* (SAC)

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

where $i \prec j$ denotes a direct dependence of v_j on v_i. The result of each *intrinsic function*[2] φ_j is assigned to a unique auxiliary variable v_j. The h *independent inputs* $u_i = v_i$, for $i =$

[2] 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 reasonably straight forward. Support of higher-level intrinsics by AD tools is very desirable. For a complex algorithm to become an intrinsic function all we require is the existence of and knowledge about the partial derivatives of its results with respect to its arguments.

$0, \ldots, h-1$, are mapped onto n *dependent outputs* $r_j = v_{h+q+j}$, for $j = 0, \ldots, n-1$. The values of q *intermediate variables* v_k are computed for $k = h, \ldots, h+q-1$.

The SAC induces a directed acyclic graph (DAG) $G = (V, E)$ with integer vertices $V = \{0, \ldots, h+q+n-1\}$ and edges $E = \{(i,j) | i \prec j\}$. The vertices are sorted topologically with respect to variable dependence inducing a partial order according to $\forall i, j \in V : (i,j) \in E \Rightarrow i < j$.

The 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 a *linearized DAG*. The linearized DAG of our reference objective is shown in Fig. 3 (a) with (high-level) intrinsic functions P, S, and p.

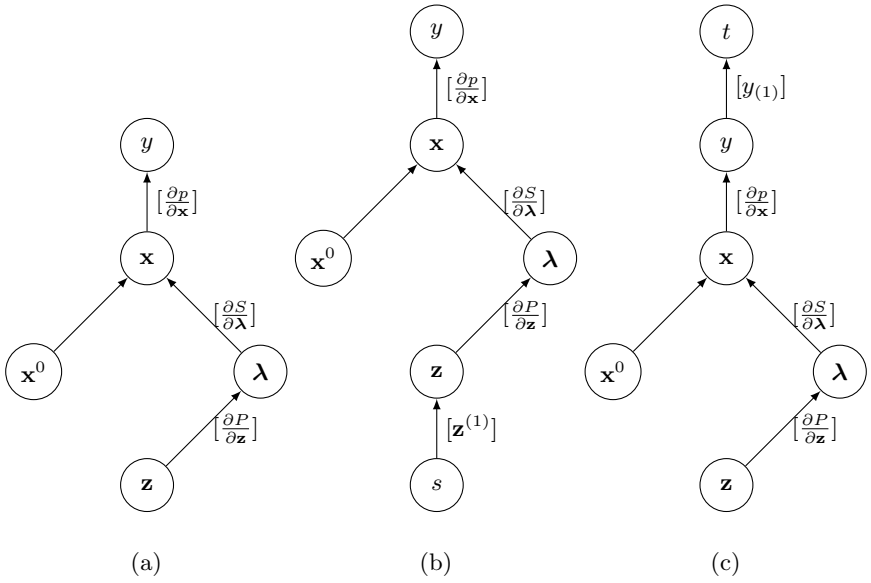


Fig. 3. Reference Problem: (a) Linearized DAG; (b) Tangent-Linear Extension; (c) Adjoint Extension

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

$$a_{i,j} = \sum_{\pi \in [i \to h+q+j]} \prod_{(k,l) \in \pi} c_{l,k}, \tag{4}$$

where

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

and where $[i \to h+q+j]$ denotes the set of all paths that connect the independent vertex i with the dependent vertex $h+q+j$ [1]. For example, according to Fig. 3 (a)

$$\frac{\partial f}{\partial \mathbf{z}} \equiv \frac{\partial y}{\partial \mathbf{z}} = \frac{\partial p}{\partial \mathbf{x}} \cdot \frac{\partial S}{\partial \boldsymbol{\lambda}} \cdot \frac{\partial P}{\partial \mathbf{z}}. \tag{5}$$

First Derivative Models The Jacobian $\nabla F = \nabla F(\mathbf{u})$ of the residual $\mathbf{r} = F(\mathbf{u})$ induces a linear mapping $\nabla F : I\!R^h \to I\!R^n$ defined by the directional derivative

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

The function

$$F^{(1)} : I\!R^h \times I\!R^h \to I\!R^n,$$

defined as

$$\mathbf{r}^{(1)} = F^{(1)}(\mathbf{u}, \mathbf{u}^{(1)}) =< \nabla F, \mathbf{u}^{(1)} >\equiv \nabla F \cdot \mathbf{u}^{(1)},$$

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

$$\mathbf{u}^{(1)} \equiv \frac{\partial \mathbf{u}}{\partial s}.$$

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

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

For example, the tangent-linear extension of the linearized DAG of our reference objective is shown in Fig. 3 (b). Equation (4) applied to Fig. 3 (b) yields with Equation (5) $y^{(1)} = \frac{\partial y}{\partial \mathbf{z}} \cdot \mathbf{z}^{(1)} =< \frac{\partial y}{\partial \mathbf{z}}, \mathbf{z}^{(1)} >$. Note that $\frac{\partial y}{\partial \mathbf{z}} \in I\!R^{1 \times h}$.

The adjoint of a linear operator is its transpose [7]. Consequently, the transposed Jacobian $\nabla F^T = \nabla F(\mathbf{u})^T$ induces a linear mapping $I\!R^n \to I\!R^h$ defined by

$$\mathbf{r}_{(1)} \mapsto< \mathbf{r}_{(1)}, \nabla F > .$$

The function

$$F_{(1)} : I\!R^h \times I\!R^n \to I\!R^h$$

defined as

$$\mathbf{u}_{(1)} = F_{(1)}(\mathbf{u}, \mathbf{r}_{(1)}) =< \mathbf{r}_{(1)}, \nabla F >\equiv \nabla F^T \cdot \mathbf{r}_{(1)}$$

is referred to as the *adjoint model* of F. It can be generated by reverse (also adjoint) mode AD. Adjoints can be regarded as partial derivatives of an auxiliary scalar variable t with respect to $\mathbf{r}$ and $\mathbf{u}$ where

$$\mathbf{r}_{(1)} \equiv \left(\frac{\partial t}{\partial \mathbf{r}}\right)^T \quad \text{and} \quad \mathbf{u}_{(1)} \equiv \left(\frac{\partial t}{\partial \mathbf{u}}\right)^T .$$

By the chain rule, we get

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

For example, the adjoint extension of the linearized DAG of our reference objective is shown in Fig. 3 (c). Equation (4) applied to Fig. 3 (c) yields with Equation (5) $\mathbf{z}_{(1)} = \frac{\partial y}{\partial \mathbf{z}}^T \cdot y_{(1)} =< y_{(1)}, \frac{\partial y}{\partial \mathbf{z}} >$.

The reverse order of evaluation of the chain rule in adjoint mode yields an additional persistent memory requirement. Values of variables that are required (used/read) by the adjoint code need to be made available by evaluation of an appropriately augmented primal code (executed within the forward section of the adjoint code). Required values need to be

stored if they are overwritten during the primal computation and they need to be restored for the propagation of adjoints within the reverse section of the adjoint code. Alternatively, required values can be recomputed from known values; see, for example, [10] for details. The minimization of the additional persistent memory requirement is one of the great challenges of discrete adjoint code generation [17]. The associated DAG REVERSAL and CALL TREE REVERSAL problems are known to be NP-complete [21, 22].

Second Derivative Models Newton's algorithm uses the Jacobian of Equation (1) at the current iterate $\mathbf{x}^i$ to determine the next Newton step. Consequently, tangent-linear and adjoint versions of Newton's algorithm will require second-order tangent-linear and adjoint versions of the given implementation of F, respectively. Again, we use the notation from [23] for the resulting projections of the Hessian tensor.

The Hessian $\nabla^2 F = \nabla^2 F(\mathbf{u})$ of the vector function $F = [F]_i$, $i = 0, \ldots, n-1$, induces a bi-linear mapping $\nabla^2 F : I\!R^h \times I\!R^h \to I\!R^n$ defined by the second directional derivative

$$(\mathbf{u}^{(1)}, \mathbf{u}^{(2)}) \mapsto < \nabla^2 F, \mathbf{u}^{(1)}, \mathbf{u}^{(2)} > ,$$

where the i-th entry of the result $< \nabla^2 F, \mathbf{u}^{(1)}, \mathbf{u}^{(2)} > \in I\!R^n$ is given as

$$[< \nabla^2 F, \mathbf{u}^{(1)}, \mathbf{u}^{(2)} >]_i = \sum_{j=0}^{h-1} \sum_{k=0}^{h-1} \left[\nabla^2 F\right]_{ijk} \cdot [\mathbf{u}^{(1)}]_j \cdot [\mathbf{u}^{(2)}]_k$$

for $i = 0, \ldots, n-1$ and

$$\left[\nabla^2 F\right]_{ijk} = \frac{\partial^2 [F]_i}{\partial [\mathbf{u}]_j \partial [\mathbf{u}]_k} .$$

We denote individual entries of an l-tensor T by $[T]_{i_1 \ldots i_l}$ for $l = 1, 2, \ldots$. The function

$$F^{(1,2)} : I\!R^h \times I\!R^h \times I\!R^h \to I\!R^n,$$

defined as

$$\mathbf{r}^{(1,2)} = F^{(1,2)}(\mathbf{u}, \mathbf{u}^{(1)}, \mathbf{u}^{(2)}) = < \nabla^2 F, \mathbf{u}^{(1)}, \mathbf{u}^{(2)} > \tag{6}$$

is referred to as the *second-order tangent-linear model* of F. The Hessian tensor is projected along its two domain dimensions (of size h) in directions $\mathbf{u}^{(1)}$ and $\mathbf{u}^{(2)}$. For scalar multivariate functions $r = F(\mathbf{u})$ Equation (6) becomes

$$r^{(1,2)} = < \nabla^2 F, \mathbf{u}^{(1)}, \mathbf{u}^{(2)} > \equiv {\mathbf{u}^{(1)}}^T \cdot \nabla^2 F \cdot \mathbf{u}^{(2)}.$$

With tangent-linear and adjoint as the two basic modes of AD there are three combinations remaining, each of them involving at least one application of adjoint mode. In [23] the mathematical equivalence of the various incarnations of second-order adjoint mode (that is, forward-over-reverse, reverse-over-forward, and reverse-over-reverse modes) due to symmetry within the Hessian of twice continuously differentiable multivariate vector functions is shown. All three variants compute projections of the Hessian tensor in the image dimension (of size n) and one of the two equivalent domain dimensions (of size h). For example, in reverse-over-forward mode a bi-linear mapping $\nabla^2 F : I\!R^h \times I\!R^n \to I\!R^h$ is evaluated defined by

$$(\mathbf{u}^{(1)}, \mathbf{r}^{(1)}_{(2)}) \mapsto < \mathbf{r}^{(1)}_{(2)}, \nabla^2 F, \mathbf{u}^{(1)} > .$$

The function

$$F_{(2)}^{(1)} : I\!R^h \times I\!R^h \times I\!R^n \to I\!R^h,$$

defined as

$$\mathbf{u}_{(2)} = F_{(2)}^{(1)}(\mathbf{u}, \mathbf{u}^{(1)}, \mathbf{r}_{(2)}^{(1)}) =< \mathbf{r}_{(2)}^{(1)}, \nabla^2 F, \mathbf{u}^{(1)} > , \tag{7}$$

where the j-th entry of the result $< \mathbf{r}_{(2)}^{(1)}, \nabla^2 F, \mathbf{u}^{(1)} > \in I\!R^h$ is given as

$$[< \mathbf{r}_{(2)}^{(1)}, \nabla^2 F, \mathbf{u}^{(1)} >]_j = \sum_{i=0}^{n-1} \sum_{k=0}^{h-1} \left[\nabla^2 F\right]_{ijk} \cdot [\mathbf{r}_{(2)}^{(1)}]_i \cdot [\mathbf{u}^{(1)}]_k \tag{8}$$

for $j = 0, \ldots, h-1$, is referred to as a *second-order adjoint model* of F. The Hessian tensor is projected in its leading image dimension in the adjoint direction $\mathbf{r}_{(2)}^{(1)} \in I\!R^n$ and in one of the two equivalent trailing domain dimensions in direction $\mathbf{u}^{(1)} \in I\!R^h$. For scalar multivariate functions $r = F(\mathbf{u})$ Equation (7) becomes

$$\mathbf{u}_{(2)} =< r_{(2)}^{(1)}, \nabla^2 F, \mathbf{u}^{(1)} > \equiv r_{(2)}^{(1)} \cdot \nabla^2 F \cdot \mathbf{u}^{(1)}.$$

2.2 Linear Solvers

During the solution of the nonlinear system by Newton's method a linear system $A \cdot \mathbf{s} = \mathbf{b}$ is solved for the Newton step $\mathbf{s}$ with Jacobian matrix

$$A := F'(\mathbf{x}^i, \boldsymbol{\lambda}) \equiv \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}^i, \boldsymbol{\lambda}) \ ,$$

and right-hand side $\mathbf{b} = -F(\mathbf{x}^i, \lambda)$. According to [6, 11] the tangent-linear projection $\mathbf{s}^{(1)}$ of the solution $\mathbf{s} = L(A, \mathbf{b})$ in directions $A^{(1)}$ and $\mathbf{b}^{(1)}$ is implicitly given as the solution of the linear system

$$A \cdot \mathbf{s}^{(1)} = \mathbf{b}^{(1)} - A^{(1)} \cdot \mathbf{s} \ .$$

The adjoint projections $A_{(1)}$ and $\mathbf{b}_{(1)}$ for given adjoints $\mathbf{s}_{(1)}$ can be computed as

$$\begin{aligned} A^T \cdot \mathbf{b}_{(1)} &= \mathbf{s}_{(1)} \\ A_{(1)} &= -\mathbf{b}_{(1)} \cdot \mathbf{s}^T \ . \end{aligned}$$

A given factorization of A, for example, as $A = L \cdot U$, computed by the primal solver can be reused for the solution of the linear tangent-linear and adjoint sensitivity equations. The computational cost of a directional derivative can be reduced significantly, e.g. from $O(n^3)$ to $O(n^2)$ for a dense system. A similar statement applies to the adjoint computation; compare with Table 2, where this observation is stated in the context of ν Newton iterations.

2.3 Nonlinear Solvers

As before, we consider three modes of differentiation of the nonlinear solver. The first approach, continuous NLS mode, does not rely on a specific method for the solution of $F(\mathbf{x}, \boldsymbol{\lambda}) = 0$. In the second approach, discrete NLS mode, AD is applied to the individual algorithmic steps performed by the nonlinear solver. Finally, the linear solver is differentiated continuously as part of an overall discrete approach to the differentiation of the enclosing nonlinear

solver in the third approach, continuous LS mode. For the latter we consider a basic version of Newton's algorithm without local line search defined by

$$\begin{aligned}
&\textbf{for } i = 0, \ldots, \nu - 1 \\
&\qquad A := F'(\mathbf{x}^i, \boldsymbol{\lambda}) \equiv \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}^i, \boldsymbol{\lambda}) \qquad (11)\\
&\qquad \mathbf{b} := -F(\mathbf{x}^i, \boldsymbol{\lambda}) \\
&\qquad \mathbf{s} := L(A, \mathbf{b}) \quad (\Rightarrow A \cdot \mathbf{s} = \mathbf{b}) \\
&\qquad \mathbf{x}^{i+1} := \mathbf{x}^i + \mathbf{s}\ , \qquad (12)
\end{aligned}$$

where the linear system is assumed to be solved directly. The investigation of iterative methods in the context of inexact Newton methods is beyond the scope of this paper. Algorithmically, their treatment turns out to be similar to the direct case. Ongoing work is focused on the formalization of the impact of the error in the primal Newton step on the directional and adjoint derivatives of the enclosing Newton solver.

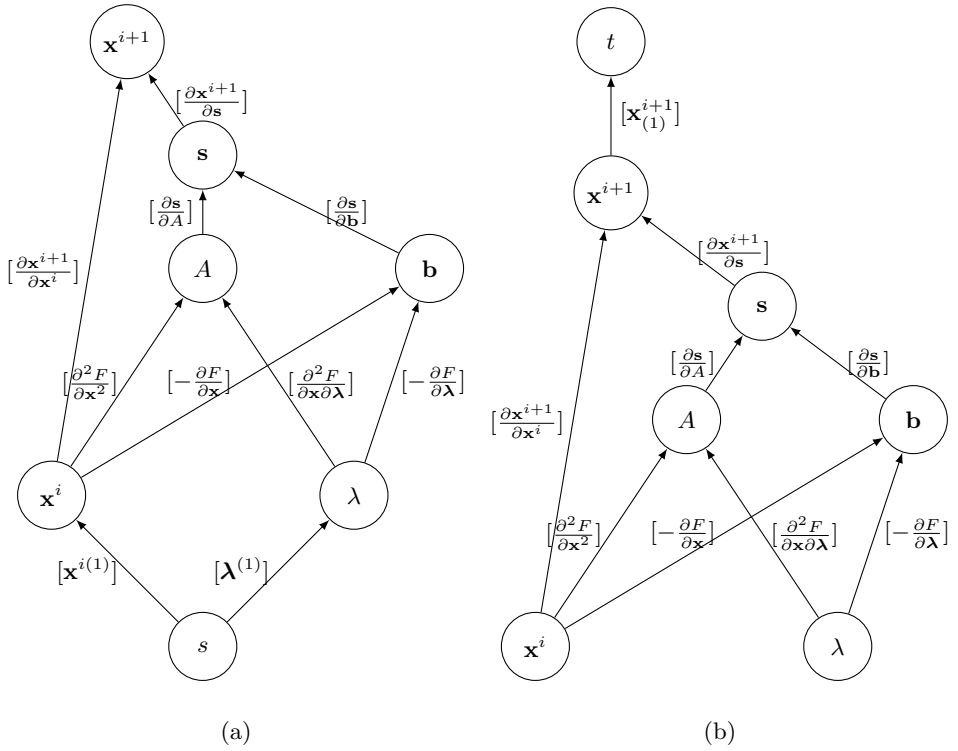


Fig. 4. DAG of Single Newton Step: (a) Tangent-Linear Extension; (b) Adjoint Extension

Fig. 4 shows the tangent-linear and adjoint extensions of the linearized DAG of a single Newton step. They provide a useful perspective on the differentiation of Newton's algorithm. A corresponding graphical illustration of several consecutive Newton steps follows trivially. The Jacobian

$$A =< \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}^i, \boldsymbol{\lambda}), I_n >=< \frac{\partial F}{\partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}), \underbrace{\begin{pmatrix} I_n \\ 0 \end{pmatrix}}_{(\in \mathbb{R}^{(n+m)\times n})} >$$

is assumed to be evaluated as its product with the identity $I_n \in \mathbb{R}^{n \times n}$ padded with m zero rows by a maximum of n calls of the tangent-linear function

$$\mathbf{r}^{(1)} =< \nabla F(\mathbf{x}^i, \lambda), \mathbf{x}^{(1)i} >$$

and with $\mathbf{x}^{(1)i}$ ranging over the Cartesian basis vectors in $\mathbb{R}^n$. The columns of A are returned in $\mathbf{r}^{(1)}$. Potential sparsity of the Jacobian of F can and should be exploited yielding a possible decrease in the number of directional derivatives required for its accumulation. See, for example, [9].

3 Tangent-Linear Solver

We distinguish between three alternative approaches to the generation of tangent-linear solvers for systems of nonlinear equations.

In discrete NLS mode AD is applied to the individual statements of the given implementation yielding roughly a duplication of the memory requirement as well as the operations count; see also Table 2. Directional derivatives of the approximation of the solution that is actually computed by the algorithm are obtained.

In continuous NLS mode directional derivatives of the solution are computed by a tangent-linear version of the solver under the assumption that the exact primal solution $\mathbf{x}^*$ has been reached. $F(\mathbf{x}, \boldsymbol{\lambda}) = 0$ can be differentiated symbolically in this case. Consequently, the computation of the directional derivative amounts to the solution of a linear system based on the Jacobian of F with respect to $\mathbf{x}$, which results in a significant reduction of the computational overhead; see also Table 2.

Potential discrepancies in the results computed by the discrete and the continuous tangent-linear nonlinear solvers depend on the given problem as well as on the accuracy of the approximation $\tilde{\mathbf{x}}$ of the primal solution. A more accurate primal solution is required in order to achieve the desired accuracy in the continuous tangent-linear (or adjoint; see Section 4) solution.

A combination of the discrete and continuous modes yields continuous LS mode, where a continuous approach is taken for the differentiation of the solver of the linear Newton system as part of an otherwise discrete approach to the differentiation of the nonlinear solver. Use of a direct linear solver makes this approach numerically equivalent to the discrete NLS method as both the Newton system and its tangent-linear versions are solved with machine accuracy. The computational complexity of the evaluation of local directional derivatives of the Newton step with respect to a dense the system matrix (the Jacobian of the residual with respect to the current iterate) and the right-hand side (the negative residual at the current iterate) can be reduced from cubic to quadratic through the reuse of the factorization of the system matrix as described in Section 2.2; see also Table 2. Numerical consistency of the discrete NLS and the continuous LS modes is not guaranteed if iterative linear solvers are employed [15, p. 367]. In the following, only a direct solution of the linear system is considered. Again, the primal Newton step must be computed with sufficiently high accuracy in order to obtain comparable accuracy in the tangent-linear (or adjoint) solution. Consistency of the continuous tangent-linear and adjoint solutions is shown in [16] and revisited here. For the continuous LS approach consistency is guaranteed naturally.

3.1 Discrete NLS Mode

The following discrete tangent-linear version of the given objective with Newton's algorithm used for the solution of the embedded parameterized systems of nonlinear equations results from the straight application of tangent-linear mode AD to Equations (11)–(12):

for $i = 0, \ldots, \nu - 1$

$$A := F'(\mathbf{x}^i, \boldsymbol{\lambda}) \equiv \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}^i, \boldsymbol{\lambda}) \tag{13}$$

$$A^{(1)} :=< \frac{\partial F'}{\partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}), \begin{pmatrix} \mathbf{x}^{i(2)} \\ \boldsymbol{\lambda}^{(2)} \end{pmatrix} >$$

$$\mathbf{b} := -F(\mathbf{x}^i, \boldsymbol{\lambda})$$

$$\mathbf{b}^{(1)} := - < \frac{\partial F}{\partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}), \begin{pmatrix} \mathbf{x}^{i(2)} \\ \boldsymbol{\lambda}^{(2)} \end{pmatrix} > \tag{14}$$

$$\mathbf{s} := L(A, \mathbf{b})$$

$$\mathbf{s}^{(1)} :=< \frac{\partial L}{\partial (A, \mathbf{b})}(A, \mathbf{b}), \begin{pmatrix} A^{(1)} \\ \mathbf{b}^{(1)} \end{pmatrix} >$$

$$\mathbf{x}^{i+1} := \mathbf{x}^i + \mathbf{s} \tag{15}$$

$$\mathbf{x}^{i+1(2)} := \mathbf{x}^{i(2)} + \mathbf{s}^{(1)}. \tag{16}$$

The computation of first directional derivatives of A, $\mathbf{b}$, and $\mathbf{s}$ involves the evaluation of second derivatives of F with respect to $\mathbf{x}$ and $\boldsymbol{\lambda}$. Hence, we use corresponding superscripts in the notation ($A^{(1)}$, $\mathbf{b}^{(1)}$, $\mathbf{s}^{(1)}$, $\mathbf{x}^{i(2)}$, and $\boldsymbol{\lambda}^{(2)}$). From

$$A =< \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}^i, \boldsymbol{\lambda}), I_n >=< \frac{\partial F}{\partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}), \begin{pmatrix} I_n \\ 0 \end{pmatrix} >$$

follows

$$\begin{aligned} A^{(1)} &=< \frac{\partial^2 F}{\partial \mathbf{x} \partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}), I_n, \begin{pmatrix} \mathbf{x}^{i(2)} \\ \boldsymbol{\lambda}^{(2)} \end{pmatrix} > \\ &=< \frac{\partial^2 F}{\partial (\mathbf{x}, \boldsymbol{\lambda})^2}(\mathbf{x}^i, \boldsymbol{\lambda}), \begin{pmatrix} I_n \\ 0 \end{pmatrix}, \begin{pmatrix} \mathbf{x}^{i(2)} \\ \boldsymbol{\lambda}^{(2)} \end{pmatrix} > \end{aligned}$$

if a Newton solver as in Equations (11)–(12) is considered. Hence, n evaluations of the second-order tangent-linear function are required to evaluate Equations (13)–(14).

The linear solver is augmented at the statement-level with local tangent-linear models, thus roughly duplicating the required memory (MEM) as well as the number of operations (OPS) performed ($MEM(L^{(1)}) \sim MEM(L) \sim O(n^2)$, $OPS(L^{(1)}) \sim OPS(L) \sim O(n^3)$ if a direct solver is used).

The tangent-linear Newton step in Equation (16) follows trivially from Equation (15).

3.2 Continuous NLS Mode

Differentiation of $F(\mathbf{x}, \boldsymbol{\lambda}) = 0$ at the solution $\mathbf{x} = \mathbf{x}^*$ with respect to $\boldsymbol{\lambda}$ yields

$$\frac{dF}{d\boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) = \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) + \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}} = 0$$

and hence

$$\frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}} = -\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^{-1} \cdot \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}).$$

The computation of the directional derivative

$$\mathbf{x}^{(1)} =< \frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}}, \boldsymbol{\lambda}^{(1)} >= \frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}} \cdot \boldsymbol{\lambda}^{(1)} = -\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^{-1} \cdot \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \boldsymbol{\lambda}^{(1)}$$

amounts to the solution of the linear system

$$\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \mathbf{x}^{(1)} = -\frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \boldsymbol{\lambda}^{(1)} \tag{17}$$

the right-hand side of which can be obtained by a single evaluation of the tangent-linear routine. The direct solution of Equation (17) requires the $n \times n$ Jacobian $\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})$, which is preferably accumulated using tangent-linear mode AD while exploiting potential sparsity.

Consistency with the exact tangent-linear projection $\mathbf{x}^{*(1)}$ of the exact solution $\mathbf{x}^*$ is defined by [16] as

$$\left\| \mathbf{x}^{(1)} - \mathbf{x}^{*(1)} \right\| \leq \Gamma \left(\|F(\mathbf{x}, \lambda)\| + \left\| F^{(1)}(\mathbf{x}, \lambda, \mathbf{x}^{(1)}, \lambda^{(1)}) \right\| \right) . \tag{18}$$

For Equation (18) to hold, we inherit the following assumptions from Newton's method:

$$\left\| \left(\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \lambda) \right)^{-1} \right\| \leq \beta \text{ and } \left\| \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \lambda) - \frac{\partial F}{\partial \mathbf{x}}(y, \lambda) \right\| \leq \gamma \|\mathbf{x} - y\|$$

in some neighborhood. $\Gamma = \Gamma(\gamma, \beta)$ is a function of the bound of the inverse Jacobian of F and the Lipschitz constant γ and

$$F^{(1)}(\mathbf{x}, \lambda, \mathbf{x}^{(1)}, \lambda^{(1)}) = \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \lambda^{(1)} + \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \mathbf{x}^{(1)} .$$

3.3 Continuous LS Mode

The quality of a continuous tangent-linear nonlinear solver depends on the accuracy of the primal solution. Moreover, in the Newton case, the computational effort is dominated by the solution of the linear system in each iteration. Continuous differentiation of the linear solver aims for a reduction of the computational cost of the tangent-linear solver while eliminating the dependence of its accuracy on the error in the primal solution. An accurate solution of the linear Newton system is required instead.

Building on Section 2.2 we get

for $i = 0, \ldots, \nu - 1$

$$A := F'(\mathbf{x}^i, \boldsymbol{\lambda}) \equiv \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}^i, \boldsymbol{\lambda}) \tag{19}$$

$$A^{(1)} :=< \frac{\partial F'}{\partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}), \begin{pmatrix} \mathbf{x}^{i(2)} \\ \boldsymbol{\lambda}^{(2)} \end{pmatrix} > \tag{20}$$

$$\mathbf{b} := -F(\mathbf{x}^i, \boldsymbol{\lambda}) \tag{21}$$

$$\mathbf{b}^{(1)} := - < \frac{\partial F}{\partial(\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}), \begin{pmatrix} \mathbf{x}^{i(2)} \\ \boldsymbol{\lambda}^{(2)} \end{pmatrix} > \tag{22}$$

$$A \cdot \mathbf{s} = \mathbf{b} \quad (\Rightarrow \mathbf{s}) \tag{23}$$

$$A \cdot \mathbf{s}^{(1)} = \mathbf{b}^{(1)} - A^{(1)} \cdot \mathbf{s} \quad (\Rightarrow \mathbf{s}^{(1)}) \tag{24}$$

$$\mathbf{x}^{i+1} := \mathbf{x}^i + \mathbf{s}$$

$$\mathbf{x}^{i+1(2)} := \mathbf{x}^{i(2)} + \mathbf{s}^{(1)}.$$

Refer to Section 2.2 for a derivation of the continuous tangent-linear linear solver. If a direct linear solver is used, then a factorization of A needs to be computed once in Equation (23) followed by simple (for example, forward and backward if LU decomposition is used) substitutions in Equation (24). The computational complexity of evaluating the directional derivative $\mathbf{s}^{(1)}$ can thus be reduced from $O(n^3)$ to $O(n^2)$ in each Newton iteration if A is dense.

Equations (21) and (22) can be evaluated simultaneously by calling the first-order tangent-linear routine. A maximum of n calls of the second-order tangent-linear routine is required to evaluate Equations (19) and (20). The vector $\mathbf{x}^{(1)}$ needs to range over the Cartesian basis vectors in $\mathbb{R}^n$ as it does during the computation of A as the Jacobian of F using the first-order tangent-linear function $F^{(1)}$. Potential sparsity of A can and should be exploited.

Consistency with the discrete approach is given naturally, as we compute the discrete tangent-linear projection with machine accuracy.

4 Adjoint Solver

As in Section 3 we distinguish between discrete NLS, continuous LS, and continuous NLS modes when deriving adjoint solvers for systems of nonlinear equations. Similar remarks as in Section 3 apply regarding numerical consistency between the primal and the adjoint solutions.

4.1 Discrete NLS Mode

In adjoint mode the data flow induced by the loop over the individual Newton steps needs to be reversed. Depending on the AD approach (overloading, source transformation, or combinations thereof; see [15] or [23] for details) certain data needs to be stored persistently in a separate data structure (often referred to as the *tape*) during an augmented forward evaluation of the solver (the augmented forward section of the adjoint code) in order to be recovered for use by the propagation of adjoints in the reverse section. For practically relevant problems the size of the tape may easily exceed the available memory resources. Checkpointing techniques have been proposed to overcome this problem by trading memory for additional operations due to re-evaluations of intermediate steps from stored intermediate states [12].

In the following we consider discrete adjoint versions of Newton's algorithm without (Section 4.1) and with (Section 4.1) checkpointing. The potential need for discrete adjoints of nonlinear solvers follows immediately from the numerical results in Table 1. For solutions computed at very low accuracy we may be interested in the exact sensitivities of the (for example, norm of the) given solution with respect to the potentially very large number of free parameters. The discrete NLS approach can be beneficial, if the Jacobian at the computed solution turns out to be rank deficient or ill-conditioned.

Discrete Adjoint without Checkpointing Straight application of (incremental; see [15] and/or [23] for details) adjoint mode AD to Equations (11)–(12) yields

for $i = 0, \ldots, \nu - 1$

$$(A, \tau) := F'(\mathbf{x}^i, \boldsymbol{\lambda}) \equiv \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}^i, \boldsymbol{\lambda}) \tag{25}$$

$$(\mathbf{b}, \tau) := -F(\mathbf{x}^i, \boldsymbol{\lambda})$$

$$(\mathbf{s}, \tau) := L(A, \mathbf{b}) \tag{26}$$

$$\mathbf{x}^{i+1} := \mathbf{x}^i + \mathbf{s} \tag{27}$$

for $i = \nu - 1, \ldots, 0$

$$\mathbf{x}^i_{(1)} := \mathbf{s}_{(1)} := \mathbf{x}^{i+1}_{(1)} \tag{28}$$

$$\begin{pmatrix} A_{(1)} \\ \mathbf{b}_{(1)} \end{pmatrix} := L_{(1)}(\mathbf{s}_{(1)}, \tau)$$

$$\begin{pmatrix} \mathbf{x}^i_{(1)} \\ \boldsymbol{\lambda}_{(1)} \end{pmatrix} := \begin{pmatrix} \mathbf{x}^i_{(1)} \\ \boldsymbol{\lambda}_{(1)} \end{pmatrix} + F'_{(1)}(A_{(1)}, \tau) - F_{(1)}(\mathbf{b}_{(1)}, \tau). \tag{29}$$

Data required within the reverse section (Equations (28)–(29)) is recorded on the tape τ in the augmented forward section (Equations (25)–(27)). The input value of $\boldsymbol{\lambda}_{(1)}$ depends on the context in which the nonlinear solver is called. In the specific scenario given by Equation (3) it is initially equal to zero as adjoints of intermediate (neither input nor output) variables should be; see, for example, [15]. The memory requirement becomes proportional to the number of operations performed by the primal nonlinear solver.

Both the Jacobian accumulation in Equation (25) and the linear solver in Equation (26) are treated in a straight forward fashion through application of AD software. Their mathematical properties will be exploited in continuous LS mode described in Section 4.3 resulting in a more targeted use of AD.

Discrete Adjoint with Checkpointing Assuming that the amount of memory required to store the tape of a single Newton iteration exceeds by far the size in memory of $\mathbf{x}^i$ we apply a basic equidistant checkpointing scheme using a matrix $C \in \mathbb{R}^{\nu \times n}$ the rows C_i of which represent the individual checkpoints at the beginning of each Newton iteration. A single execution of the solver is performed in Equations (30)–(31) to populate C followed by the adjoint Newton iterations in Equations (32)–(36). Each iteration recovers the corresponding checkpointed state $\mathbf{x}^i$ in Equation (32), builds the local tape τ of a single Newton iteration in Equations (33)–(34) and evaluates its adjoint in Equations (35)–(36).

for $i = 0, \ldots, \nu - 1$

$$C_i := \mathbf{x}^i \tag{30}$$

$$A := F'(\mathbf{x}^i, \boldsymbol{\lambda}) \equiv \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}^i, \boldsymbol{\lambda})$$

$$\mathbf{b} := -F(\mathbf{x}^i, \boldsymbol{\lambda})$$

$$\mathbf{s} := L(A, \mathbf{b})$$

$$\mathbf{x}^{i+1} := \mathbf{x}^i + \mathbf{s} \tag{31}$$

for $i = \nu - 1, \ldots, 0$

$$\mathbf{x}^i := C_i \tag{32}$$

$$(A, \tau) := F'(\mathbf{x}^i, \boldsymbol{\lambda}) \tag{33}$$

$$(\mathbf{b}, \tau) := -F(\mathbf{x}^i, \boldsymbol{\lambda})$$

$$(\mathbf{s}, \tau) := L(A, \mathbf{b})$$

$$\mathbf{x}^{i+1} := \mathbf{x}^i + \mathbf{s} \tag{34}$$

$$\mathbf{x}^i_{(1)} := \mathbf{s}_{(1)} := \mathbf{x}^{i+1}_{(1)} \tag{35}$$

$$\begin{pmatrix} A_{(1)} \\ \mathbf{b}_{(1)} \end{pmatrix} := L_{(1)}(\mathbf{s}_{(1)}, \tau)$$

$$\begin{pmatrix} \mathbf{x}^i_{(1)} \\ \boldsymbol{\lambda}_{(1)} \end{pmatrix} := \begin{pmatrix} \mathbf{x}^i_{(1)} \\ \boldsymbol{\lambda}_{(1)} \end{pmatrix} + F'_{(1)}(A_{(1)}, \tau) - F_{(1)}(\mathbf{b}_{(1)}, \tau) \tag{36}$$

The maximum size of the tape is equal to that of a single Newton iteration. The additional overall memory requirement is reduced to the size in memory of C.

Checkpointing yields a potentially optimal tradeoff between operations count and memory requirement [14] for an a priori known number of Newton iterations. Concurrent checkpointing schemes allow for discrete adjoint code to be ported to parallel high-performance computer architectures [19].

4.2 Continuous NLS Mode

Differentiation of $F(\mathbf{x}, \boldsymbol{\lambda}) = 0$ at the solution $\mathbf{x} = \mathbf{x}^*$ with respect to $\boldsymbol{\lambda}$ yields

$$\frac{dF}{d\boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) = \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) + \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}} = 0$$

and hence

$$\frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}} = -\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^{-1} \cdot \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda})$$

the transposal of which results in

$$\left(\frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}}\right)^T = -\frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^{-T}$$

and hence

$$\begin{aligned} \boldsymbol{\lambda}_{(1)} :&= \boldsymbol{\lambda}_{(1)} + < \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}} >= \boldsymbol{\lambda}_{(1)} + \left(\frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}}\right)^T \cdot \mathbf{x}_{(1)} \\ &= \boldsymbol{\lambda}_{(1)} - \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^{-T} \cdot \mathbf{x}_{(1)}. \end{aligned}$$

Consequently, the continuous adjoint solver needs to solve the linear system

$$\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \mathbf{z} = -\mathbf{x}_{(1)} \tag{37}$$

followed by a single call of the adjoint model of F to obtain

$$\boldsymbol{\lambda}_{(1)} = \boldsymbol{\lambda}_{(1)} + \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \mathbf{z}. \tag{38}$$

The direct solution of Equation (37) requires the transpose of the $n \times n$ Jacobian $\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})$ that is preferably accumulated using tangent-linear mode AD while exploiting sparsity. Matrix-free iterative solvers require a single call of the adjoint routine (Equation (38)) per iteration.

Consistency with the exact adjoint projection is also shown in [16] similar to the tangent-linear case in Section 3.2.

4.3 Continuous LS Mode

Comments similar to those made in Section 3.3 apply. The quality of the result of a continuous adjoint nonlinear solver depends on the accuracy of the primal solution. Moreover, in the Newton case, the computational effort is dominated by the solution of the linear system in each iteration. Continuous differentiation of the linear solver as part of a discrete differentiation approach to the enclosing nonlinear solver aims for a reduction of the computational cost of the adjoint while eliminating the dependence of its accuracy on the error in the primal solution. An accurate solution of the linear system is required instead.

Building on Section 2.2 we get for Newton's method

for $i = 0, \ldots, \nu - 1$

$$\begin{aligned} A^i &:= \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}^i, \boldsymbol{\lambda}) \\ \mathbf{b}^i &:= -F(\mathbf{x}^i, \boldsymbol{\lambda}) \\ A^i \cdot \mathbf{s}^i &= \mathbf{b}^i \quad (\Rightarrow \mathbf{s}^i) \\ \mathbf{x}^{i+1} &:= \mathbf{x}^i + \mathbf{s}^i \end{aligned} \tag{39}$$

for $i = \nu - 1, \ldots, 0$

$$\begin{aligned} \mathbf{x}^i_{(1)} &:= \mathbf{s}^i_{(1)} = \mathbf{x}^{i+1}_{(1)} \\ {A^i}^T \cdot \mathbf{b}^i_{(1)} &= \mathbf{s}^i_{(1)} \quad (\Rightarrow \mathbf{b}^i_{(1)}) \\ A^i_{(1)} &:= -\mathbf{b}^i_{(1)} \cdot {\mathbf{s}^i}^T \\ \begin{pmatrix} \mathbf{x}^i_{(1)} \\ \boldsymbol{\lambda}_{(1)} \end{pmatrix} &:= \begin{pmatrix} \mathbf{x}^i_{(1)} \\ \boldsymbol{\lambda}_{(1)} \end{pmatrix} + < \mathbf{b}^i_{(1)}, \frac{\partial F}{\partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}) > \\ &\quad + < A^i_{(1)}, \frac{\partial^2 F}{\partial \mathbf{x} \partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}) > , \end{aligned} \tag{40, 41}$$

where

$$< A^i_{(1)}, \frac{\partial^2 F}{\partial \mathbf{x} \partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}) > = < -\mathbf{b}^i_{(1)} \cdot {\mathbf{s}^i}^T, \frac{\partial^2 F}{\partial \mathbf{x} \partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}) > \tag{42}$$

$$= < -\mathbf{s}^i, \mathbf{b}^i_{(1)}, \frac{\partial^2 F}{\partial \mathbf{x} \partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}) > \tag{43}$$

$$= < \mathbf{b}^i_{(1)}, \frac{\partial^2 F}{\partial (\mathbf{x}, \boldsymbol{\lambda}) \partial \mathbf{x}}(\mathbf{x}^i, \boldsymbol{\lambda}), -\mathbf{s}^i > \ . \tag{44}$$

The step from Equation (42) to Equation (43) is shown in Lemma 1. The expression $< A^i_{(1)}, \frac{\partial^2 F}{\partial \mathbf{x} \partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}) >$ in Equation (41) denotes a projection of the serialized image dimension of length $n^2 (\Leftarrow n \times n)$ of the first derivative of the Jacobian $\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}^i, \boldsymbol{\lambda})$ with respect

to $\mathbf{x}$ and $\boldsymbol{\lambda}$ (the Hessian $\frac{\partial^2 F}{\partial \mathbf{x} \partial(\mathbf{x},\boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda})$) in the direction obtained by a corresponding serialization of $A^i_{(1)}$; see also Equation (8). Equation (43) suggests an evaluation of the third term in Equation (41) in reverse-over-reverse mode of AD. Symmetry of the Hessian tensor in its two domain dimensions yields the equivalence of this approach with the computationally less expensive and hence preferred reverse-over-forward or forward-over-reverse modes in Equation (44).

Lemma 1. *With the previously introduced notation we get*

$$< -\mathbf{b}_{(1)} \cdot \mathbf{s}^{iT}, \frac{\partial^2 F}{\partial \mathbf{x} \partial(\mathbf{x},\boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}) >=< -\mathbf{s}^i, \mathbf{b}_{(1)}, \frac{\partial^2 F}{\partial \mathbf{x} \partial(\mathbf{x},\boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}) > . \tag{45}$$

Proof. We consider the k-th entry of Equation (45) ($0 \leq k \leq n+m-1$):

$$\begin{aligned}
\left[< -\mathbf{b}_{(1)} \cdot \mathbf{s}^{iT}, \frac{\partial \frac{\partial F(\mathbf{x}^i,\boldsymbol{\lambda})}{\partial \mathbf{x}}}{\partial(\mathbf{x},\boldsymbol{\lambda})} >\right]_k &= \sum_{j=0}^{n-1}\sum_{l=0}^{n-1} -[\mathbf{b}_{(1)}]_j \cdot [\mathbf{s}^i]_l \cdot \frac{\partial \left[\frac{\partial F(\mathbf{x}^i,\boldsymbol{\lambda})}{\partial \mathbf{x}}\right]_{j,l}}{\partial[(\mathbf{x},\boldsymbol{\lambda})]_k} \\
&= \sum_{l=0}^{n-1} -[\mathbf{s}^i]_l \cdot \sum_{j=0}^{n-1} [\mathbf{b}_{(1)}]_j \cdot \frac{\partial^2 [F]_j(\mathbf{x}^i,\boldsymbol{\lambda})}{\partial[\mathbf{x}]_l \partial[(\mathbf{x},\boldsymbol{\lambda})]_k} \\
&= \left[< -\mathbf{s}^i, < \mathbf{b}_{(1)}, \frac{\partial^2 F}{\partial \mathbf{x} \partial(\mathbf{x},\boldsymbol{\lambda})}(\mathbf{x}^i,\boldsymbol{\lambda}) >>\right]_k \\
&= \left[< -\mathbf{s}^i, \mathbf{b}_{(1)}, \frac{\partial^2 F}{\partial \mathbf{x} \partial(\mathbf{x},\boldsymbol{\lambda})}(\mathbf{x}^i,\boldsymbol{\lambda}) >\right]_k .
\end{aligned}$$

If the linear system in Equation (39) is solved by a direct method, then the computed factorization of a dense A can be reused to solve Equation (40) at the computational cost of $O(n^2)$ as discussed previously.

5 Implementation (*Mind The Gap*)

While the continuous differentiation of solvers for systems of linear and nonlinear equations has been discussed in the literature before, the seamless integration of the theoretical results into existing AD software tools is typically not straight-forward. Users of such tools deserve an intuitive generic API,[3] which facilitates the exploitation of mathematical and structural knowledge inside of often highly complex tangent-linear and adjoint numerical simulations.

As a representative case study for the implementation of higher-level (user-defined) intrinsics in the context of overloading AD tools we consider the solver S(n,x,lbd) for systems of n nonlinear equations with inputs x=x^0 and lbd=$\boldsymbol{\lambda}$ and output x=x^*. More generically, the proposed approach allows users of AD tools to treat arbitrary parts of the primal code as *external functions*. The latter yield *gaps* in the tape due to their passive evaluation within the forward section of the adjoint code. These gaps need to be filled by corresponding user-defined adjoint functions to be called by the tape interpreter within the reverse section of the adjoint code. This concept is part of the overloading AD tool `dco` [24]. It supports both C/C++ and Fortran and has been applied successfully to a growing number of practically relevant problems in Computational Science, Engineering, and Finance [26–29].

[3] Application Programming Interface

In the following we focus on the external function interface of `dco/c++` in the context of first-order adjoint mode. While similar remarks apply to tangent-linear mode the preferred method of implementation of tangent-linear external functions is through replacement of the overloaded primal function with a user-defined version. One should not expect to be presented with *the* method for filling gaps in the data flow of tangent-linear or adjoint numerical simulations. There are always several alternatives that implement mathematically equivalent functions. The particular choice made for `dco/c++` is meant to be both intuitive and easy to maintain. The overloading AD tool ADOL-C [13] features a similar, but less generic external function concept.

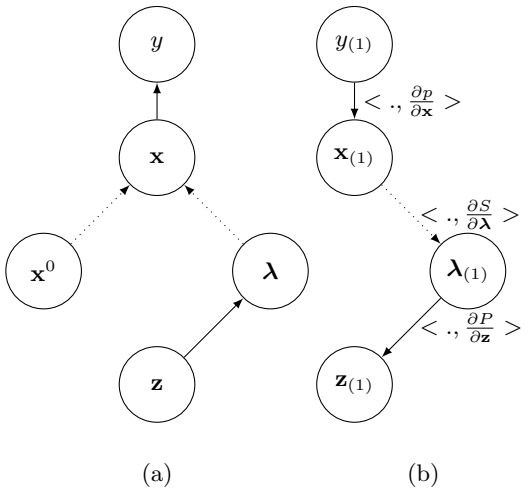


Fig. 5. *Mind the gap* in the tape – Implementation of continuous NLS mode: Solid lines represent the generation (in the forward section of the adjoint code shown in (a)) and interpretation (in the reverse section of the adjoint code shown in (b)) of the tape. Dotted lines denote gaps in the tape to be filled by a corresponding user-defined adjoint function. In the given example, $\boldsymbol{\lambda}_{(1)}$ is computed in continuous NLS mode as $< \mathbf{x}_{(1)}, \frac{\partial S}{\partial \boldsymbol{\lambda}} > \equiv \frac{\partial S}{\partial \boldsymbol{\lambda}}^T \cdot \mathbf{x}_{(1)}$ without generation of a tape for the nonlinear solver S.

5.1 Discrete Approach (No Gap)

The primal function

```
1 void S(int n, double *x, double *l);
```

is made generic with respect to the floating-point data type FT yielding

```
1 template <class FT>
2 void S(int n, FT *x, FT *lbd);
```

Thus it can be instantiated with the `dco/c++` data type dco::a1s::type, which implements first-order scalar adjoint mode. A tape of the entire computation is generated and interpreted as discussed in Section 4.1.

In C++ one would replace dynamic array parameters acting as both inputs and outputs such as FT *x by, for example, std::vector<FT> &x in order to avoid memory leaks due to potentially missing deallocation. We decided to use the C-style declaration for better readability by a larger audience.

5.2 Continuous Approach (Gap)

A specialization of the generic primal solver S for `dco`'s scalar first-order adjoint type dco::a1s::type marks the gap in the tape, records data that is required for filling the gap during interpretation, and runs the primal solver passively (without taping); see Listing 1.1.

```
/*
 * forward declaration of a1s_S;
 * closes the gap in the tape during interpretation
 */
void a1s_S(external_function_data *data);

/*
 * passive evaluation of primal solver augmented
 * with recording of data required by a1s_S
 */
template<>
void S<dco::a1s::type>(int n, dco::a1s::type *x,
                               dco::a1s::type *lbd ) {
  external_function_data *data = new external_function_data();

  // active outputs of P = active inputs of S
  double *plbd=new double[n];
  data->register_input(lbd,plbd,n);
  // passive values of inputs
  double *px=new double[n];
  for (int i=0;i<n;i++) dco::a1s::get(x[i],px[i]);
  // a1s_S requires values of n and plbd
  data->write_to_checkpoint(n);
  data->write_to_checkpoint(plbd,n);
  // passive nonlinear solver
  S(n,px,plbd);
  // a1s_S requires primal solution
  data->write_to_checkpoint(px,n);
  // active outputs of S = active inputs of p
  data->register_output(x,px,n);
  // tape interpreter to call a1s_S in order to fill gap
  tape->register_external_function(data, &a1s_S);
  // clear heap
  delete [] plbd, delete [] px;
}
```

Listing 1.1. External function for S(n,x,l).

The tape interpreter fills the gap between the tapes of P and p by calling the function a1s_S which implements the adjoint mapping $< \mathbf{x}_{(1)}, \frac{\partial S}{\partial \boldsymbol{\lambda}} >$; see Listing 1.2. Refer to Fig. 5 for graphical illustration.

```
/*
 * a1s_S fills the gap in the tape during interpretation;
 * evaluates adjoint of x with respect to lbd
 */
void a1s_S(external_function_data *data) {
  // recover n; required for correct memory allocation
  int n; data->read_from_checkpoint(n);
  double *lbd=new double[n], *x=new double[n];
  double *xb=new double[n], *z=new double[n];
  dco::a1s::type *al=new dco::a1s::type[n];
```

```
11    dco::a1s::type *residual=new dco::a1s::type[n];
12    // recover parameters
13    data->read_from_checkpoint(lbd,n);
14    // recover primal solution
15    data->read_from_checkpoint(x,n);
16    // compute Jacobian of F with respect to x
17    double **J=new double*[n];
18    for (int i=0;i<n;i++) J[i]=new double[n];
19    compute_jacobian(n,lbd,x,J);
20    // extract adjoints from tape of p
21    data->get_output_adjoint(xb,n);
22    // solve Equation (20)
23    solve_transposed_system(J,xb,z,n);
24    // store end of tape of P
25    dco::a1s::tape::iterator pos=tape->get_position();
26    // generate tape of F
27    tape->register_variable(al,lbd,n); F(al,x,residual,n);
28    // interpret tape of F (evaluate Equation 21)
29    for (int i=0;i<n;i++)
30      dco::a1s::set(residual[i],z[i],-1);
31    tape->interpret_and_reset_adjoint_to(pos);
32
33    // transfer adjoints into tape of P
34    for (int i=0;i<n;i++) {
35      double a; dco::a1s::get(al[i],a,-1);
36      data->increment_input_adjoint(a);
37    }
38    // clear heap ...
39  }
```

Listing 1.2. Adjoint function a1s_S.

The benefit of this approach is two-fold. First, taping of the nonlinear solver is avoided yielding a substantial reduction in memory requirement of the overloading-based adjoint. Second, the actual adjoint mapping can be implemented in a1s_S more efficiently than by interpretation of a corresponding tape.

As a general approach the external function feature can/should be applied whenever a similar reduction in memory requirement / computational cost can be expected. Users of `dco/c++` are encouraged to extend the run time library with user-defined intrinsics for (domain-specific) numerical kernels such as, for example, turbulence models in computational fluid dynamics or pay off functions in mathematical finance.

The external function interface of `dco/c++` facilitates a hybrid overloading / source transformation approach to AD. Currently, none of the available source transformation tools covers the entire latest C++ or Fortran standards. Moreover, these tools can be expected to struggle keeping up with the evolution of the programming languages for the foreseeable future. Mature source transformation AD tools such as Tapenade [18] can handle (considerable subsets of) C and/or Fortran 95. They can (and should) be applied to suitable selected parts of the given C++XX or Fortran 20XX code. Integration into an enclosing overloading AD solution via the external function interface is typically rather straight forward. The hybrid approach to AD promises further improvements in terms of robustness and computational efficiency.

6 Case Study

As a case study we consider the one dimensional nonlinear differential equation

$$\begin{aligned} \nabla^2(z \cdot u^*) + u^* \cdot \nabla(z \cdot u^*) = 0 \quad & \text{on} \quad \Omega = (0,1) \\ u^* = 10 \quad \text{and} \quad z = 1 \quad & \text{for} \quad x = 0 \\ u^* = 20 \quad \text{and} \quad z = 1 \quad & \text{for} \quad x = 1 \end{aligned}$$

with parameters $z(x)$. For given measurements $u^m(x)$ we aim to solve the following parameter fitting problem for z

$$z^* = \arg\min_{z \in I\!R} J(z) \tag{53}$$

with $J(z) = \|u(x,z) - u^m(x)\|_2^2$. Measurements $u^m(x)$ are generated by a given set of parameters (the "real" parameter distribution $z^*(x)$). Building on an equidistant central finite difference discretization we get for a given $\mathbf{u}$ (as in the previous sections, discretized and, hence, vector-valued variables are written as bold letters) the residual function

$$\begin{aligned} [\mathbf{r}]_i = \frac{1}{h^2} \cdot ([\mathbf{z}]_{i-1} \cdot [\mathbf{u}]_{i-1} - 2 \cdot [\mathbf{z}]_i \cdot [\mathbf{u}]_i + [\mathbf{z}]_{i+1} \cdot [\mathbf{u}]_{i+1}) \\ + [\mathbf{u}]_i \cdot \frac{1}{2 \cdot h} \cdot ([\mathbf{z}]_{i+1} \cdot [\mathbf{u}]_{i+1} - [\mathbf{z}]_{i-1} \cdot [\mathbf{u}]_{i-1}) \end{aligned}$$

with $h = 1/n$ and n the number of discretization points, that is, $i = 1, \ldots, n-2$. Discretization yields a system of n nonlinear equations

$$\mathbf{r}(\mathbf{u}, \mathbf{z}) = 0, \quad \mathbf{u} \in I\!R^n, \ \mathbf{z} \in I\!R^n \ , \tag{56}$$

which is solved by Newton's method yielding in the j-th Newton iteration the linear system

$$\frac{\partial \mathbf{r}}{\partial \mathbf{u}}(\mathbf{u}^j) \cdot \mathbf{s} = -\mathbf{r}(\mathbf{u}^j) \ .$$

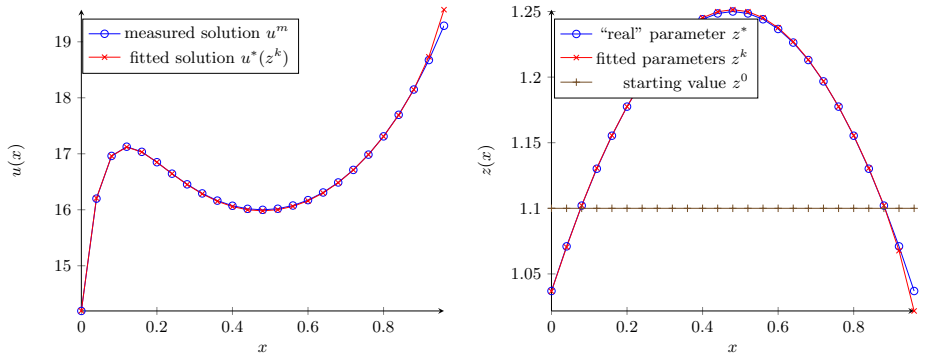
The vector $\mathbf{u}^j$ is updated with the Newton step $\mathbf{u}^{j+1} = \mathbf{u}^j + \mathbf{s}$ for $j = 1, \ldots, \nu$.

In order to solve the parameter fitting problem, we apply a simple steepest descent algorithm to the discrete objective $J(\mathbf{z})$ as follows $\mathbf{z}^{k+1} = \mathbf{z}^k - \nabla J(\mathbf{z}^k)$, where the computation of the gradient of J at the current iterate $\mathbf{z}^k$ implies the differentiation of the solution process for $\mathbf{u}^*$, i.e., differentiation of the solver for Equation (56). Extension of the given example to the use of Quasi-Newton methods (for example, BFGS) is straight forward. Second-order methods rely on the efficient evaluation of second derivative information, which turns out to be a logical extension of the framework described in this paper. A corresponding report is under development.

The discrete part of the derivative computation is done by `dco/c++`. The implementation of continuous tangent-linear and adjoint methods is supported by the previously described external function interface.

The preprocessor $\boldsymbol{\lambda} = P(\mathbf{z})$ is the identity, while the postprocessor $p(\mathbf{u})$ computes the cost functional $J(\mathbf{z})$. Fig. 6 shows the measured solution $\mathbf{u}^m$, the fitted solution $\mathbf{u}^*(\mathbf{z}^k)$ as well as the starting parameter set $\mathbf{z}^0$, the "real" (wanted) parameter $\mathbf{z}^*$ and the fitted parameter $\mathbf{z}^k$ after convergence.

In the following we compare run time and memory consumption of the various differentiated versions of the nonlinear solver. The titles of the following sections refer to the outermost derivative computation (computation of the gradient of the objective of the parameter fitting problem). The accumulation of the dense Jacobian inside of Newton's method is performed in tangent-linear mode.



(a) Measured and fitted solutions (b) "Real" and fitted parameters

Fig. 6. Visualization of the parameter fitting problem in Equation (53)

6.1 Tangent-Linear Mode

We first compare the computation of a single tangent-linear projection of the objective function $y = J(\mathbf{z})$ into a direction $\mathbf{z}^{(1)}$, i.e., $y^{(1)} = \nabla J(\mathbf{z}) \cdot \mathbf{z}^{(1)}$, for the discrete and the continuous approaches.

The implementation of discrete tangent-linear mode is based on the `dco/c++` tangent-linear datatype `dco::t1s::type`. Overloading yields second-order projections during the Jacobian accumulation inside of Newton's method. Theoretically, the overhead in computational cost of discrete tangent-linear mode over a passive function evaluation is expected to be of the same order as the passive computation itself. The use of nested C++ templates inside of `dco/c++` yields lower overheads in reality (typically, factors between 1.3 and 1.5).

The implementation of the continuous NLS and LS modes combines the use of the data type `dco::t1s::type` with manual implementation effort based on explicit template specialization techniques. In the specialized function body value and tangent-linear components of incoming data are separated. Results and their directional derivatives are computed explicitly and are stored in the corresponding components of the output data. The continuous version of the solver consists of a first-order tangent-linear evaluation of the nonlinear function $F(\mathbf{x}, \boldsymbol{\lambda})$, i.e.,

$$\mathbf{b} = -\frac{\partial F(\mathbf{x}, \boldsymbol{\lambda})}{\partial \boldsymbol{\lambda}} \cdot \boldsymbol{\lambda}^{(1)} \ ,$$

followed by the direct solution of the linear system

$$\frac{\partial F(\mathbf{x}, \boldsymbol{\lambda})}{\partial \mathbf{x}} \cdot \mathbf{x}^{(1)} = \mathbf{b} \ ,$$

where the occurring derivatives are again implemented using `dco::t1s::type`. The overhead becomes $O(n^3)$ due to the direct solution of an additional linear $n \times n$ system; see also Table 2.

Continuous differentiation of the linear system in continuous LS mode yields a call to the second-order tangent-linear version of the nonlinear function $F(\mathbf{x}, \boldsymbol{\lambda})$, i.e.,

$$A^{(1)} =< \frac{\partial F'}{\partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}), \begin{pmatrix} \mathbf{x}^{i(2)} \\ \boldsymbol{\lambda}^{(2)} \end{pmatrix} > \ ,$$

which can be evaluated by overloading based on nested first-order `dco/c++` types. Additionally, a linear $n \times n$ system needs to be solved with the same system matrix, which is already

used for the computation of the Newton step (see Equations (23)–(24)). The previously computed factorization of the system matrix can be reused yielding an expected overhead that is proportional to $\nu \cdot O(n^2)$, where ν denotes the number of Newton iterations; see also Table 2.

In Fig. 7 we observe the expected behavior for the computational overhead induced by the different approaches. The overhead in discrete NLS mode is roughly the same as the passive run time. Both continuous NLS and continuous LS modes yield far less overhead. Also as expected, for large problem dimensions n the overhead of continuous LS mode becomes $\nu \cdot O(n^2)$. It outperforms continuous NLS mode the overhead of which amounts to $O(n^3)$.

6.2 Adjoint Mode

We consider the computation of the gradient of the objective function $y = J(\mathbf{z})$ by setting $y_{(1)} = 1$ in

$$\mathbf{z}_{(1)} = y_{(1)} \cdot \nabla J(\mathbf{z}) \ .$$

Both discrete and continuous modes are investigated.

The implementation of discrete adjoint mode uses the `dco/c++` adjoint datatype `dco::a1s::type`. Overloading yields second-order adjoint projections during the Jacobian accumulation inside of Newton's method. Automatic C++ template nesting yields the so called reverse-over-forward mode; see, for example, [23]. The overhead of the discrete version is expected to range between factors of 4–8 (see [20]) relative to a passive evaluation and depending on the given primal code.

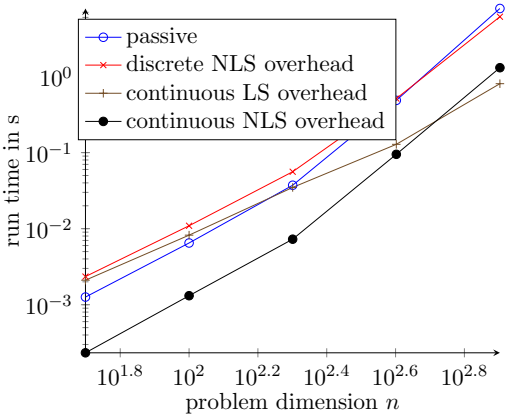


Fig. 7. Passive run time to run time overhead comparison for discrete and continuous tangent-linear modes; double logarithmic scale.

The implementation of the continuous NLS and LS modes requires changes in the forward and reverse sections of the adjoint code. In the forward section overloading or template specialization techniques can be used to save the required data (e. g. the solution $\mathbf{x}^\nu$ in continuous mode). Moreover, the external function interface of `dco/c++` is used to embed the adjoint function into the reverse section (tape interpretation) in order to fill the gap left in the tape by the passive solution of the nonlinear (in continuous NLS mode) or linear (in continuous LS mode) systems; see Section 5.

In continuous NLS mode the last Newton iterate $\mathbf{x}^\nu$ is checkpointed at the end of the forward section. A linear $n \times n$ system with the transposed Jacobian of the nonlinear function is solved in the reverse section, i.e.,

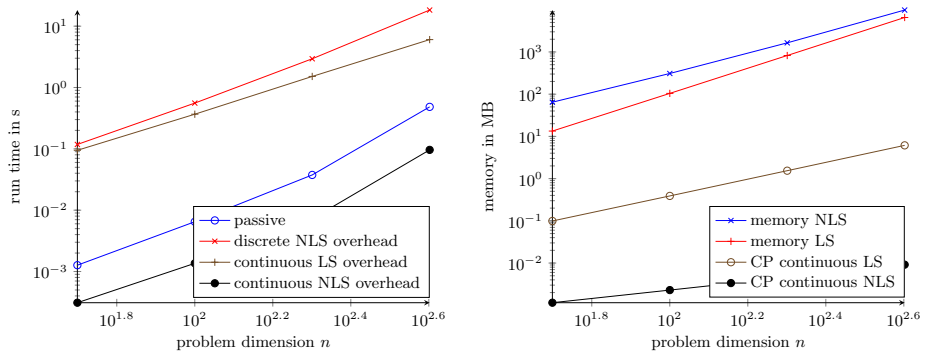
$$\frac{\partial F(\mathbf{x}^\nu, \boldsymbol{\lambda})}{\partial \mathbf{x}}^T \cdot \mathbf{z} = -\mathbf{x}_{(1)}$$

followed by a single evaluation of the adjoint

$$\boldsymbol{\lambda}_{(1)} = \frac{\partial F(\mathbf{x}^\nu, \boldsymbol{\lambda})}{\partial \boldsymbol{\lambda}}^T \cdot \mathbf{z} \ .$$

A direct approach to the solution of the linear system results in a computational overhead of $O(n^3)$ while limiting the overhead in memory requirement to $O(n^2)$; see also Table 2.

In continuous LS mode the factorization of the Jacobian as well as the current iterate for each Newton step need to be stored in the forward section. A single linear $n \times n$ system with the transposed Jacobian as the system matrix (already decomposed – see Equation (40)) is solved in the reverse section. This step is followed by a second-order adjoint model evaluation using nested first-order `dco/c++` types (see Equation (44)). The checkpoint memory (denoted by CP in Fig. 8) as well as the run time overhead is hence expected to be of order $\nu \cdot O(n^2)$; see also Table 2.



(a) Run Time Overhead (b) Memory Overhead

Fig. 8. Passive run time to run time overhead comparison and additional memory consumption for discrete and continuous adjoint modes; double logarithmic scale.

Fig. 8(a) illustrates the cubic run time complexities of passive, discrete, and continuous modes scaled with different constant factors. Quadratic growth in run time can be observed in continuous LS mode. The memory consumption in discrete NLS mode is dominated by the direct linear solver. The amount of memory needed for checkpoints in continuous LS mode is $O(n^2)$. In continuous NLS mode it is reduced to $O(n)$ as supported by Fig. 8(b).

7 Summary, Conclusion, and Outlook

The exploitation of mathematical and algorithmic insight into solvers for systems of nonlinear equations yields considerable reductions in the computational complexity of the corresponding differentiated solvers. Symbolic differentiation of the nonlinear system yields derivative code the accuracy of which depends on the error in the primal solution. In the Newton case, this dependence can be eliminated by differentiating the embedded linear system symbolically as part of a tangent-linear or adjoint nonlinear solver. The various approaches can be implemented elegantly by software tools for AD as illustrated by `dco/c++`. User-friendly applicability to practically relevant large-scale numerical simulation and optimization problems can thus be facilitated.

Ongoing work targets the scalability of the proposed methods in the context of high-performance scientific computing on modern parallel architectures. Highly optimized parallel linear solvers (potentially running on modern accelerators, such as GPUs) need to be combined with scalable differentiated versions of the primal residual. We aim for a seamless inclusion of these components into future AD software.

Acknowledgments

Lotz and Leppkes were supported by DFG grant NA 487/4-1 ("A hybrid approach to the generation of adjoint C++ code").

The authors would like to thank the anonymous referees for a number of very helpful comments on the manuscript.

References

1. F. Bauer. Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11:87–96, 1974.
2. 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.
3. C. Broyden. The convergence of a class of double-rank minimization algorithms. *Journal of the Institute of Mathematics and Its Applications*, (6):76–90, 1970.
4. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, number 50 in Lecture Notes in Computational Science and Engineering, Berlin, 2006. Springer.
5. Bruce Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3(4):311–326, 1994.
6. A. J. Davies, D. B. Christianson, L. C. W. Dixon, R. Roy, and P. van der Zee. Reverse differentiation and the inverse diffusion problem. *Adv. Eng. Softw.*, 28(4):217–221, June 1997.
7. N. Dunford and J. Schwartz. *Linear Operators, Part 1, General Theory*. Wiley, 1988.
8. S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, editors. *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2012.
9. A. Gebremedhin, F. Manne, and A. Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005.
10. R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software*, 24:437–474, 1998.
11. M. Giles. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In Bischof et al. [2], pages 35–44.
12. A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.
13. A. Griewank, D. Juedes, and J. Utke. ADOL-C, a package for the Automatic Differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22:131–167, 1996.
14. A. Griewank and A. Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45, 2000.
15. 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.
16. Andreas Griewank and Christèle Faure. Reduced functions, gradients and hessians from fixed-point iterations for state equations. *Numerical Algorithms*, 30(2):113–139, 2002.
17. L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems*, 21:1401–1417, 2005.
18. L. Hascoet and V. Pascual. The Tapenade automatic differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software*, 39(3):20, 2013.
19. U. Lehmann and A. Walther. The implementation and testing of time-minimal and resource-optimal parallel reversal schedules. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, volume 2330 of *Lecture Notes in Computer Science*, pages 1049–1058, Berlin, 2002. Springer.
20. K. Leppkes, J. Lotz, and U. Naumann. `dco/c++` – derivative code by overloading in C++. Technical Report AIB-2011-05, RWTH Aachen, 2011.
21. U. Naumann. Call Tree Reversal is NP-complete. In *[2]*, pages 13–22. Springer, 2008.

22. U. Naumann. DAG Reversal is NP-complete. *Journal of Discrete Algorithms*, 7:402–410, 2009.
23. U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation*. Number 24 in Software, Environments, and Tools. SIAM, Philadelphia, PA, 2012.
24. U. Naumann, K. Leppkes, and J. Lotz. `dco/c++` user guide. Technical Report AIB-2014-03, RWTH Aachen University, January 2014.
25. J. Nocedal and S. J. Wright. *Numerical Optimization, 2nd Edition*. Springer Series in Operations Research. Springer-Verlag, New York, NY, 2006.
26. F. Rauser, J. Riehme, K. Leppkes, P. Korn, and U. Naumann. On the use of discrete adjoints in goal error estimation for shallow water equations. *Procedia Computer Science*, 1(1):107 – 115, 2010.
27. M. Sagebaum, N. Gauger, J. Lotz, and U. Naumann. Algorithmic differentiation of a large simulation code including libraries. *Procedia Computer Science (ICCS 2013)*, 18:208–217, 2013.
28. M. Towara and U. Naumann. Toward discrete adjoint OpenFOAM. *Procedia Computer Science (ICCS 2013)*, 18:429–438, 2013.
29. J. Ungermann, J. Blank, J. Lotz, K. Leppkes, Lars Hoffmann, T. Guggenmoser, M. Kaufmann, P. Preusse, U. Naumann, and M. Riese. A 3-d tomographic retrieval approach with advection compensation for the air-borne limb-imager GLORIA. *Atmospheric Measurement Techniques*, 4(11):2509–2529, 2011.

Aachener Informatik-Berichte

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

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

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

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

2010-01 * Fachgruppe Informatik: Jahresbericht 2010
2010-02 Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03 Holger Krahn: MontiCore: Agile Entwicklung von 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

2011-17 Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis

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

2012-07 André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms

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

2012-09 Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains

2012-10 Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations

2012-12 Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs

2012-15 Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations

2012-16 Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets

2012-17 Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods

2013-01 * Fachgruppe Informatik: Annual Report 2013

2013-02 Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen

2013-03 Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM

2013-04 Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries

2013-05 Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013

2013-06 Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation

2013-07 André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung

2013-08 Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers

2013-10 Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata

2013-12 Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs

2013-13 Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators

2013-19 Florian Schmidt, David Orlea, and Klaus Wehrle: Support for error tolerance in the Real-Time Transport Protocol

* These reports are only available as a printed version.

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