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

UWE NAUMANN, JOHANNES LOTZ, KLAUS LEPPKES, and MARKUS TOWARA,
RWTH Aachen University

We discuss software tool support for the algorithmic differentiation (AD), also known as automatic differentiation, of numerical simulation programs that contain calls to solvers for parameterized systems of n nonlinear equations. The local computational overhead and 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 (and symbolic) 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 present an implementation based on the AD software `dco/c++`. A representative case study supports the theoretically obtained computational complexity results with practical runtime measurements.

Categories and Subject Descriptors: G.1.4 [**Quadrature and Numerical Differentiation**]: Automatic Differentiation; G.4 [**Mathematical Software**]: Algorithm Design and Analysis

General Terms: Algorithms

Additional Key Words and Phrases: Tangent/adjoint nonlinear solver, algorithmic differentiation

ACM Reference Format:
Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara. 2015. Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations. ACM Trans. Math. Softw. 41, 4, Article 26 (October 2015), 21 pages.
DOI: http://dx.doi.org/10.1145/2700820

1. INTRODUCTION, TERMINOLOGY, AND SUMMARY OF RESULTS

We consider the computation of first 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 $F(\mathbf{x}, \boldsymbol{\lambda}) = 0$. To provide a context for the differentiation of nonlinear solvers and without loss of generality, the nonlinear solver is assumed to be embedded (see Section 2) into the unconstrained convex nonlinear programming problem (NLP)

$$\min_{\mathbf{z} \in I\!R^q} f(\mathbf{z}, \mathbf{x}^0) \tag{2}$$

J. Lotz and K. Leppkes were supported by DFG grant NA 487/4-1 ("A Hybrid Approach to the Generation of Adjoint C++ Code").
Authors' address: U. Naumann, J. Lotz, K. Leppkes, and M. Towara, LuFG Informatik 12, RWTH Aachen, D-52062 Aachen, Germany; emails: {naumann, lotz, leppkes, towara}@stce.rwth-aachen.de.

DOI: http://dx.doi.org/10.1145/2700820

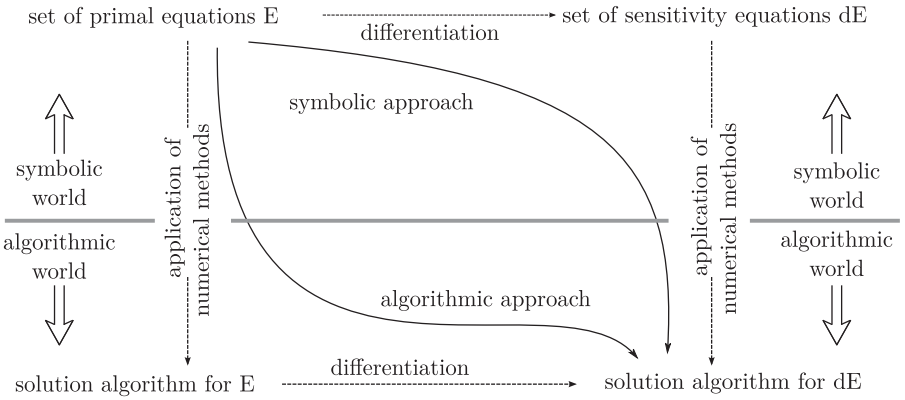


Fig. 1. Illustration of symbolic and algorithmic approaches to differentiation.

for a given objective function $f : \mathbb{R}^q \times \mathbb{R}^n \to \mathbb{R}$ and a start value $\mathbf{x}^0 \in \mathbb{R}^n$ for the nonlinear solver. In the context of first-order derivative-based methods (e.g., steepest descent or quasi-Newton methods such as BFGS [Broyden 1970; Nocedal and Wright 2006] if q exceeds n considerably) for the solution of the NLP, the gradient of $y = f(\mathbf{z}, \mathbf{x}^0) \in \mathbb{R}$ with respect to $\mathbf{z} \in \mathbb{R}^q$ needs to be computed, which involves the differentiation of the nonlinear solver itself.

Algorithmic differentiation (AD) [Griewank and Walther 2008; Naumann 2012] 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 q (in Equation (2)) of free parameters. AD tools for compile- (source code transformation) and runtime (operator and function overloading potentially combined with C++ metaprogramming techniques) 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 six international conferences on the subject (e.g., see Bücker et al. [2006], Bischof et al. [2008], and Forth et al. [2012]). Computational finance has seen a particular rise of interest in AD over the past decade. Related works include Giles and Glasserman [2006], Capriotti [2011], and Henrard [2014].

Traditionally, AD tools take a fully *algorithmic* approach to differentiation by transforming the given source code at the level of arithmetic operators and built-in[1] functions. Potentially complex numerical kernels, such as matrix products or the solvers for systems of linear and nonlinear equations to be discussed in this article, typically are not considered as intrinsic functions, often resulting in suboptimal computational performance. Ideally, one would like to reuse 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 $O(n^2)$ [Davies et al. 1997; Giles 2008]. These results are built upon in this article in the context of symbolic differentiation methods applied to different levels of (Newton-type) numerical solution algorithms for systems of nonlinear equations.

Figure 1 illustrates our use of the term *symbolic*. In general, the primal problem is assumed to be given as a set E of (potentially nonlinear [[partial] differential])

[1]... into the given programming language.

symbolic 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* $x \in \mathbb{R}$. Symbolic derivation of the corresponding (tangent or adjoint) sensitivity equations yields a new set dE of equations (upper right corner of Figure 1). Their numerical solution delivers (approximations of) tangent or adjoint sensitivities (lower right corner of Figure 1). For example, the (symbolic) total differentiation of $x^2 - \lambda = 0$ with respect to λ yields

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

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

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

for some $x_{(1)} \in \mathbb{R}$ in adjoint mode. Note that the primal solution x enters the definitions of both the tangent and adjoint sensitivity equations. The numerical approximation $\tilde{x}$ of the solution yields approximations of the tangent and adjoint sensitivities—that is,

$$\frac{\lambda^{(1)}}{2 \cdot x} = \frac{dx}{d\lambda} \cdot \lambda^{(1)} \approx \frac{d\tilde{x}}{d\lambda} \cdot \lambda^{(1)} \approx \frac{\lambda^{(1)}}{2 \cdot \tilde{x}} \tag{3}$$

and

$$\frac{x_{(1)}}{2 \cdot x} = x_{(1)} \cdot \frac{dx}{d\lambda} \approx x_{(1)} \cdot \frac{d\tilde{x}}{d\lambda} \approx \frac{x_{(1)}}{2 \cdot \tilde{x}}, \tag{4}$$

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

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

Consistency of the symbolic and algorithmic approaches to the differentiation of nonlinear solvers is shown in Griewank and Faure [2002] and discussed further in Griewank and Walther [2008]. Refer also to Christianson [1994] for a related discussion in the context of attractive fixed-point solvers.

In this article, 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 (Equation (3)) or the adjoint (Equation (4)) are computed. In finite precision arithmetic, an approximate solution of the primal equations E yields approximate sensitivities in algorithmic differentiation mode. The primal solution enters the symbolic sensitivity equations dE. The numerical solution of dE will produce approximate sensitivities that will generally not match those obtained in algorithmic mode exactly. 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 symbolic and algorithmic 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 and the linear system to be solved in each Newton step can be treated in either way. Table I 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

Table I. Computational Complexities of Algorithmic and Symbolic Tangent and Adjoint Modes of Differentiation for ν Newton Iterations Applied to Systems of n Nonlinear Equations

		algorithmic NLS	symbolic NLS	symbolic LS
tangent mode (runtime)		$\nu \cdot O(n^3)$	$O(n^3)$	$\nu \cdot O(n^2)$
adjoint mode	(memory)	$\nu \cdot O(n^3)$	$O(n^2)$	$\nu \cdot O(n^2)$
	(runtime)	$\nu \cdot O(n^3)$	$O(n^3)$	$\nu \cdot O(n^2)$

Note: Symbolic differentiation can be applied at the level of the nonlinear (symbolic NLS mode) and (here dense) linear systems. Fully algorithmic treatment of the nonlinear solver (algorithmic NLS mode) implies the algorithmic differentiation of the linear solver. Alternatively, a symbolically differentiated linear solver can be embedded into an algorithmically differentiated nonlinear solver (symbolic LS mode).

of the residual. The performance of the different approaches depends on the number of Newton iterations ν and on the problem size n. Algorithmic 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. Correct derivatives of the actually performed Newton iterations are computed independent of whether convergence has been achieved or not. Symbolic differentiation delivers approximations of the derivatives, the accuracy of which also depends on the quality of the primal solution. Symbolic differentiation of the embedded linear solver yields an improvement over both the algorithmic and symbolic 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 symbolic method by a factor of ν.

2. FOUNDATIONS

For further illustration, f is decomposed as

$$y = f(\mathbf{z}, \mathbf{x}^0) = p(S(\mathbf{x}^0, \boldsymbol{\lambda})) = p(S(\mathbf{x}^0, P(\mathbf{z}))), \tag{5}$$

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, such as parameter estimation problems for mathematical models that involve the solution of nonlinear partial differential equations. Refer to Section 6 for a case study.

The discussion in this article will be based on the following algorithmic description of Equation (5):

$$\boldsymbol{\lambda} := P(\mathbf{z}); \;\; \tilde{\mathbf{x}} := S(\mathbf{x}^0, \boldsymbol{\lambda}); \;\; y := p(\tilde{\mathbf{x}}). \tag{6}$$

The parameters $\boldsymbol{\lambda} \in \mathbb{R}^m$ are computed as a function of $\mathbf{z} \in \mathbb{R}^q$ by the given implementation of P. They enter the nonlinear solver S as arguments alongside with the given initial estimate $\mathbf{x}^0 \in \mathbb{R}^n$ of the solution $\mathbf{x} \in \mathbb{R}^n$. 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 Griewank and Walther [2008] and Naumann [2012]. Without loss of generality, the following discussion will be based on the residual function in Equation (1). Let

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

and $h = n + m$. AD yields semantical transformations of the given implementation of $F : \mathbb{R}^h \to \mathbb{R}^n$ 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 Naumann [2012].

2.1.1. First Derivative Models. The Jacobian $\nabla F = \nabla F(\mathbf{u})$ of $\mathbf{r} = F(\mathbf{u})$ induces a linear mapping defined by the directional derivative $\mathbf{u}^{(1)} \mapsto <\nabla F, \mathbf{u}^{(1)}>$. The function $F^{(1)}$: $\mathbb{R}^h \times \mathbb{R}^h \to \mathbb{R}^n$, defined as

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

is referred to as the *tangent model* of F. It can be generated by forward (also tangent) mode AD.

The adjoint of a linear operator is its transpose [Dunford and Schwartz 1988]. Consequently, the transposed Jacobian $\nabla F^T = \nabla F(\mathbf{u})^T$ induces a linear mapping defined by $\mathbf{r}_{(1)} \mapsto <\mathbf{r}_{(1)}, \nabla F>$. The function $F_{(1)} : \mathbb{R}^h \times \mathbb{R}^n \to \mathbb{R}^h$, defined as

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

is referred to as the *adjoint model* of F. It can be generated by reverse (also adjoint) mode AD.

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 (e.g., see Giering and Kaminski [1998] for details). Moreover, partial derivatives of the built-in functions or even local gradients (of assignments) or Jacobians (of basic blocks) can be stored. This approach is taken by `dco/c++`. The minimization of the additional persistent memory requirement is one of the great challenges of algorithmic adjoint code generation [Hascoët et al. 2005]. The associated DAG Reversal and Call Tree Reversal problems are known to be NP-complete [Naumann 2008, 2009].

2.1.2. 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 and adjoint versions of Newton's algorithm will require second-order tangent and adjoint versions of the given implementation of F, respectively. Again, we use the notation from Naumann [2012] for the resulting projections of the Hessian tensor.

The Hessian $\nabla^2 F = \nabla^2 F(\mathbf{u})$ of $F = [F]_i$, $i = 0, \ldots, n-1$, induces a bi-linear mapping 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 $<\nabla^2 F, \mathbf{u}^{(1)}, \mathbf{u}^{(2)}> \in \mathbb{R}^n$ is given as

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

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

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

We use the ∂-notation for partial derivatives. Individual entries of an l-tensor T are denoted by $[T]_{i_1 \ldots i_l}$ for $l = 1, 2, \ldots$. The function $F^{(1,2)} : \mathbb{R}^h \times \mathbb{R}^h \times \mathbb{R}^h \to \mathbb{R}^n$,

defined as

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

is referred to as the *second-order tangent 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 (7) 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 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. For example, in Naumann [2012], the mathematical equivalence of the various incarnations of second-order adjoint mode (i.e., forward-over-reverse, reverse-over-forward, and reverse-over-reverse modes) due to symmetry within the Hessian of twice symbolically 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 as $(\mathbf{u}^{(1)}, \mathbf{r}^{(1)}_{(2)}) \mapsto <\mathbf{r}^{(1)}_{(2)}, \nabla^2 F, \mathbf{u}^{(1)}>$. The function $F^{(1)}_{(2)} : I\!R^h \times I\!R^h \times I\!R^n \to I\!R^h$, defined as

$$\mathbf{u}_{(2)} = F^{(1)}_{(2)}\left(\mathbf{u}, \mathbf{u}^{(1)}, \mathbf{r}^{(1)}_{(2)}\right) = <\mathbf{r}^{(1)}_{(2)}, \nabla^2 F, \mathbf{u}^{(1)}>, \tag{8}$$

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

$$\left[<\mathbf{r}^{(1)}_{(2)}, \nabla^2 F, \mathbf{u}^{(1)}>\right]_j = \sum_{i=0}^{n-1}\sum_{k=0}^{h-1} [\nabla^2 F]_{ijk} \cdot \left[\mathbf{r}^{(1)}_{(2)}\right]_i \cdot \left[\mathbf{u}^{(1)}\right]_k \tag{9}$$

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 $\mathbf{r}^{(1)}_{(2)} \in I\!R^n$ 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 (8) becomes

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

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 $\mathbf{b} = -F(\mathbf{x}^i, \boldsymbol{\lambda})$. According to Davies et al. [1997] and Giles [2008], the tangent 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 computed by a direct primal solver can be reused for the solution of the linear tangent and adjoint sensitivity equations. The computational cost of a directional derivative can be reduced significantly—for example, from $O(n^3)$

to $O(n^2)$ for a dense system. A similar statement applies to the adjoint computation; compare with Table I.

2.3. Nonlinear Solvers

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

For the latter, we consider a basic version of Newton's algorithm without local line search defined by

$$\begin{aligned}
&\textbf{for } i = 0, \ldots, \nu - 1 \\
&\qquad A := F'(\mathbf{x}^i, \lambda) \equiv \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}^i, \lambda) \qquad (10)\\
&\qquad \mathbf{b} := -F(\mathbf{x}^i, \lambda) \\
&\qquad \mathbf{s} := L(A, \mathbf{b}) \quad (\Rightarrow A \cdot \mathbf{s} = \mathbf{b}) \\
&\qquad \mathbf{x}^{i+1} := \mathbf{x}^i + \mathbf{s}, \qquad (11)
\end{aligned}$$

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 article. 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.

The Jacobian

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

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 function

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

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, Gebremedhin et al. [2005].

3. TANGENT SOLVER

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

In algorithmic 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 Table I). Directional derivatives of the approximation of the solution that is actually computed by the algorithm are obtained.

In symbolic NLS mode, directional derivatives of the solution are computed by a tangent 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 Table I).

Potential discrepancies in the results computed by the algorithmic and the symbolic tangent nonlinear solvers depend on the given problem, as well as on the accuracy of the approximation $\tilde{\mathbf{x}}$ of the primal solution. In both cases, a more accurate primal solution is required to achieve the desired accuracy in the tangent (or adjoint; see Section 4) solution.

For Newton-type solvers, a combination of the algorithmic and symbolic modes yields symbolic LS mode, where a symbolic approach is taken for the differentiation of the solver of the linear Newton system as part of an otherwise algorithmic approach to the differentiation of the nonlinear solver. The use of a direct linear solver makes this approach mathematically equivalent to the algorithmic NLS method, as both the Newton system and its tangent 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 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 Table I). Numerical consistency of the algorithmic NLS and the symbolic LS modes is not guaranteed if iterative linear solvers are employed [Griewank and Walther 2008, p. 367]. In the following, only a direct solution of the linear system is considered. Further issues with algorithmic differentiation of iterative processes and with Newton's method in particular are discussed in Gilbert [1992].

3.1. Algorithmic NLS Mode

An algorithmic tangent version of the given objective with an embedded solution of a parameterized systems of nonlinear equations results from the straight application of tangent mode AD to the given implementation of the nonlinear solver. For example, 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 (10) and (11) is considered. Hence, n evaluations of the second-order tangent function are required. The linear solver is augmented at the statement level with local tangent models, thus roughly duplicating the required memory (MEM) and the number of operations (OPS) performed ($MEM(L^{(1)}) \sim MEM(L) \sim O(n^2)$, $OPS(L^{(1)}) \sim OPS(L) \sim O(n^3)$).

3.2. Symbolic 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

$$\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \mathbf{x}^{(1)} = -\frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \boldsymbol{\lambda}^{(1)}, \tag{12}$$

the right-hand side of which can be obtained by a single evaluation of the tangent routine for given $\mathbf{x}, \boldsymbol{\lambda}$, and $\boldsymbol{\lambda}^{(1)}$. The direct solution of Equation (12) requires the $n \times n$ Jacobian $\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})$, which is preferably accumulated using tangent mode AD while exploiting potential sparsity. Refer to Griewank and Faure [2002] for an analysis of the consistency with the exact tangent projection $\mathbf{x}^{*(1)}$ of the exact solution $\mathbf{x}^*$.

3.3. Symbolic LS Mode

The quality of a symbolic tangent 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. Symbolic differentiation of the linear solver aims for a reduction of the computational cost of the tangent solver while preserving the accuracy of the algorithmic tangent solver if an accurate solution of the linear Newton system is available.

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, corresponding superscripts in the notation ($A^{(1)}$, $\mathbf{b}^{(1)}$, $\mathbf{s}^{(1)}$, $\mathbf{x}^{i(2)}$, and $\boldsymbol{\lambda}^{(2)}$) are used in the following. Building on Section 2.2, we get

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

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

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

$$\mathbf{b} := -F(\mathbf{x}^i, \boldsymbol{\lambda}) \tag{15}$$

$$\mathbf{b}^{(1)} := -<\frac{\partial F}{\partial (\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda}), \begin{pmatrix} \mathbf{x}^{i(2)} \\ \boldsymbol{\lambda}^{(2)} \end{pmatrix}> \tag{16}$$

$$A \cdot \mathbf{s} = \mathbf{b} \quad (\Rightarrow \mathbf{s}) \tag{17}$$

$$A \cdot \mathbf{s}^{(1)} = \mathbf{b}^{(1)} - A^{(1)} \cdot \mathbf{s} \quad \left(\Rightarrow \mathbf{s}^{(1)}\right) \tag{18}$$

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

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

If a direct linear solver is used, then a factorization of A needs to be computed once in Equation (17) followed by simple (e.g., forward and backward if LU decomposition is used) substitutions in Equation (18). 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 (15) and (16) can be evaluated simultaneously by calling the first-order tangent routine. A maximum of n calls of the second-order tangent routine is required to evaluate Equations (13) and (14).

Consistency with the algorithmic approach is given naturally, as we compute the algorithmic tangent projection with a direct linear solver.

4. ADJOINT SOLVER

As in Section 3, we distinguish between algorithmic NLS, symbolic LS, and symbolic NLS modes when deriving adjoint solvers for parameterized systems of nonlinear equations.

4.1. Algorithmic NLS Mode

For solutions of the primal nonlinear system computed at very low accuracy, we may be interested in the exact sensitivities of the (e.g., norm of the) given solution with respect to the potentially very large number of free parameters. The algorithmic NLS approach can be beneficial if the Jacobian at the computed solution turns out to be rank deficient or ill conditioned [Nemili et al. 2013].

In algorithmic adjoint mode, the dataflow through the nonlinear solver needs to be reversed. Depending on the AD approach (overloading, source transformation, or combinations thereof; see Griewank and Walther [2008] or Naumann [2012] 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) 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 [Griewank 1992].

Algorithmic adjoint mode AD can be applied in a straightforward way to the given implementation of the nonlinear solver. Data required within the reverse section is recorded on a tape in the augmented forward section. 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 (6), it is initially equal to zero as adjoints of intermediate (neither input nor output) variables should be (e.g., see Griewank and Walther [2008]). The memory requirement becomes proportional to the number of operations performed by the primal nonlinear solver.

Checkpointing yields a potentially optimal trade-off between operations count and memory requirement [Griewank and Walther 2000] for an a priori known number of Newton iterations. Unfortunately, this value is typically not available. Effective heuristics for online checkpointing need to be used instead [Stumm and Walther 2010]. Concurrent checkpointing schemes allow for algorithmic adjoint code to be ported to parallel high-performance computer architectures [Lehmann and Walther 2002]. If the nonlinear system is solved exactly, then only a single iteration at the converged solution needs to be recorded [Christianson 1994].

4.2. Symbolic 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 symbolic adjoint solver needs to solve the linear system

$$\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \mathbf{z} = -\mathbf{x}_{(1)} \tag{19}$$

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

$$\boldsymbol{\lambda}_{(1)} = \boldsymbol{\lambda}_{(1)} + \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \mathbf{z} \tag{20}$$

for given $\mathbf{x} = \mathbf{x}^*$, $\boldsymbol{\lambda}$, and $\mathbf{x}_{(1)} = \tilde{\mathbf{x}}_{(1)}$ (see Equation (6)). The direct solution of Equation (19) requires the transpose of the $n \times n$ Jacobian $\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})$, which is preferably accumulated using tangent mode AD while exploiting sparsity. Matrix-free iterative solvers require a single call of the adjoint routine (Equation (20)) per iteration.

Consistency with the exact adjoint projection is also shown in Griewank and Faure [2002], similar to the tangent case in Section 3.2.

4.3. Symbolic LS Mode

Comments similar to those made in Section 3.3 apply. The quality of the result of an 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. Symbolic differentiation of the linear solver as part of an algorithmic differentiation approach to the enclosing nonlinear solver aims for a reduction of the computational cost of the adjoint. Exact sensitivities of what is computed (i.e., an approximation of the solution of the primal nonlinear system) are obtained if the primal Newton steps are computed exactly.

Building on Section 2.2, we get for Newton's method

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

$$
\begin{aligned}
&\textbf{for } i = \nu - 1, \ldots, 0 \\
&\qquad \mathbf{x}^i_{(1)} := \mathbf{s}^i_{(1)} = \mathbf{x}^{i+1}_{(1)} \\
&\qquad A^{i\,T} \cdot \mathbf{b}^i_{(1)} = \mathbf{s}^i_{(1)} \quad (\Rightarrow \mathbf{b}^i_{(1)}) \qquad (21)\\
&\qquad A^i_{(1)} := -\mathbf{b}^i_{(1)} \cdot \mathbf{s}^{i\,T} \\
&\qquad \begin{pmatrix} \mathbf{x}^i_{(1)} \\ \boldsymbol{\lambda}_{(1)} \end{pmatrix} := \begin{pmatrix} \mathbf{x}^i_{(1)} \\ \boldsymbol{\lambda}_{(1)} \end{pmatrix} - <\mathbf{b}^i_{(1)}, \frac{\partial F}{\partial(\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda})> \\
&\qquad\qquad + <A^i_{(1)}, \frac{\partial^2 F}{\partial \mathbf{x} \partial(\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda})>, \qquad (22)
\end{aligned}
$$

where

$$
\begin{aligned}
<A^i_{(1)}, \frac{\partial^2 F}{\partial \mathbf{x} \partial(\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda})> &= <-\mathbf{b}^i_{(1)} \cdot \mathbf{s}^{i\,T}, \frac{\partial^2 F}{\partial \mathbf{x} \partial(\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda})> \qquad (23)\\
&= <-\mathbf{s}^i, \mathbf{b}^i_{(1)}, \frac{\partial^2 F}{\partial \mathbf{x} \partial(\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda})> \qquad (24)\\
&= <\mathbf{b}^i_{(1)}, \frac{\partial^2 F}{\partial(\mathbf{x}, \boldsymbol{\lambda}) \partial \mathbf{x}}(\mathbf{x}^i, \boldsymbol{\lambda}), -\mathbf{s}^i> . \qquad (25)
\end{aligned}
$$

The step from Equation (23) to Equation (24) is shown in Lemma 4.1. The expression $<A^i_{(1)}, \frac{\partial^2 F}{\partial \mathbf{x} \partial(\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda})>$ in Equation (22) 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 Equation (9)). Equation (24) suggests an evaluation of the third term in Equation (22) 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 (25). The overhead in computational cost of reverse-over-reverse mode is due to the repeated dataflow reversal, which yields a substantially less efficient data access pattern.

LEMMA 4.1. *With the previously introduced notation, we get*

$$
<-\mathbf{b}_{(1)} \cdot \mathbf{s}^{i\,T}, \frac{\partial^2 F}{\partial \mathbf{x} \partial(\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda})> = <-\mathbf{s}^i, \mathbf{b}_{(1)}, \frac{\partial^2 F}{\partial \mathbf{x} \partial(\mathbf{x}, \boldsymbol{\lambda})}(\mathbf{x}^i, \boldsymbol{\lambda})>. \qquad (26)
$$

PROOF. We consider the k-th entry of Equation (26) ($0 \leq k \leq n + m - 1$):

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

The primal factorization of A can be reused to solve Equation (21) at the computational cost of $O(n^2)$ as discussed previously.

5. IMPLEMENTATION (*MIND THE GAP*)

Although the symbolic 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 typically is not straightforward. Users of such tools deserve an intuitive generic application programming interface, which facilitates the exploitation of mathematical and structural knowledge inside of often highly complex tangent 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=λ and output x=x^ν. More generically, the proposed approach allows users of AD tools to treat arbitrary parts of the primal code as *external*. 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 *external 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` [Naumann et al. 2014]. 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 [Rauser et al. 2010; Sagebaum et al. 2013; Towara and Naumann 2013; Ungermann et al. 2011; Naumann and du Toit 2014].

In the following, we focus on the external adjoint interface of `dco/c++` in the context of first-order adjoint mode. Whereas similar remarks apply to tangent mode, the preferred method of implementation of external tangents 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 dataflow of tangent or adjoint numerical simulations. There always are 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 [Griewank et al. 1996] features a similar but less generic external adjoint concept.

5.1. Algorithmic Approach (No Gap)

The primal function is made generic with respect to the floating-point type FT, yielding

```
1 template <class FT>
2 void S(vector <FT> &x, const vector<FT> &lbd);
```

Thus, it can be instantiated with the `dco/c++` type dco::ga1s<**double**>::type, which implements first-order scalar adjoint mode. A tape of the entire computation is generated and interpreted as discussed in Section 4.1.

5.2. Symbolic Approach (Gap)

A specialization of the generic primal solver S for `dco`'s scalar first-order adjoint type dco::ga1s<**double**>::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 2). For the following listings, we use the **typedef**s from Listing 1.

```
1 typedef dco::ga1s<double> DCO_MODE;
2 typedef DCO_MODE::type    DCO_TYPE;
```

Listing 1. C++ **typedef**s used throughout the code listings.

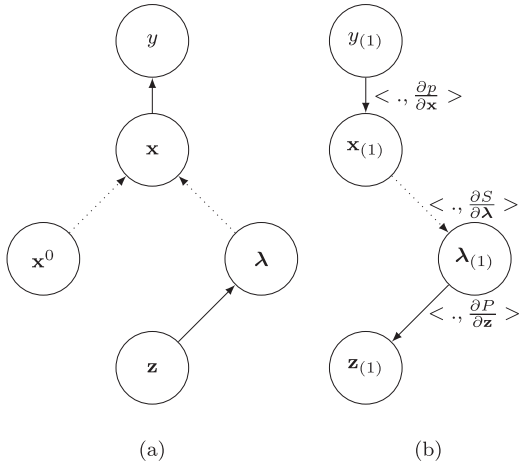


Fig. 2. *Mind the gap* in the tape. Implementation of symbolic NLS mode: solid lines represent the generation (in the forward section of the adjoint code shown in (a)) and interpretation (in the reverse section of the adjoint code shown in (b)) of the tape. Dotted lines denote gaps in the tape to be filled by a corresponding user-defined adjoint function. In the given example, $\boldsymbol{\lambda}_{(1)}$ is computed in symbolic 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.

The tape interpreter fills the gap between the tapes of P and p by calling the function adjoint_S, which implements the adjoint mapping $<\mathbf{x}_{(1)}, \frac{\partial S}{\partial \boldsymbol{\lambda}}>$ (see Listing 3). Refer to Figure 2 for graphical illustration.

The benefit of this approach is twofold. 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 adjoint_S more efficiently than by interpretation of a corresponding tape.

As a general approach, the external adjoint 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 runtime library with user-defined intrinsics for (domain-specific) numerical kernels, such as turbulence models in computational fluid dynamics or pay off functions in mathematical finance.

The external adjoint 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 [Hascoet and Pascual 2013] 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 adjoint interface typically is rather straightforward. 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

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

```
1  // forward declaration of adjoint_S;
2  void adjoint_S(external_adjoint_object_t *data);
3
4  /*
5   * passive evaluation of primal solver augmented
6   * with recording of data required by adjoint_S
7   */
8  template<>
9      void S<DCO_TYPE>(vector<DCO_TYPE> &x,
10                      const vector<DCO_TYPE> &lbd) {
11     // request new adjoint object required to fill gap
12     external_adjoint_object_t *data =
13       tape->create_callback_object<external_adjoint_object_t>();
14     // passive values of parameters
15     vector<double> plbd(lbd.size());
16     // active outputs of P = active inputs of S
17     data->register_input(lbd,plbd);
18     // passive values of inputs
19     vector<double> px(x.size());
20     px = dco::value(x);
21     // adjoint_S requires values plbd
22     data->write_data(plbd);
23     // passive nonlinear solver
24     S(px,plbd);
25     // adjoint_S requires primal solution
26     data->write_data(px);
27     // active outputs of S = active inputs of p
28     data->register_output(px, x);
29
30     // insert filled adjoint object connected to adjoint_S
31     // called by the interpreter in order to fill the gap
32     tape->insert_callback(adjoint_S, data);
33 }
```

Listing 2. Specialization of the implementation of S(n,x,lbd) making the gap.

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

$$z^* = \arg\min_{z \in \mathbb{R}} J(z), \tag{27}$$

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 in bold letters) the residual function

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

```
/*
 * adjoint_S fills the gap in the tape during interpretation;
 * evaluates adjoint of x with respect to lbd
 */
void adjoint_S(external_adjoint_object_t *data) {
    // recover parameters
    const vector<double> &lbd =
      data->read_data<vector<double> >();
    // recover primal solution
    const vector<double> &x =
      data->read_data<vector<double> >();
    int n = x.size();
    // required temporary variables
    vector<double> xb(n), z(n);
    vector<DCO_TYPE> al, residual(n);
    // copy passive values into active variables
    copy(lbd.begin(), lbd.end(), back_inserter(al));

    // compute Jacobian of F with respect to x
    vector<vector<double> > J(n, vector<double>(n));
    compute_jacobian(lbd,x,J);
    // extract adjoints of p from tape
    data->get_output_adjoint(xb);
    // solve Equation (21)
    solve_transposed_system(J,xb,z);
    // store end of tape of P
    DCO_MODE::tape_t::position_t pos=tape->get_position();
    // generate tape of F
    tape->register_variable(al); F(al,x,residual);
    // interpret tape of F (evaluate Equation (22))
    dco::derivative(residual) = z;
    tape->interpret_adjoint_to(pos);
    // transfer adjoints into tape of P
    dco::derivative(al) = xb;
    data->increment_input_adjoint(xb);
    // remove tape of F
    tape->reset_to(pos);
}
```

Listing 3. Implementation of the adjoint function adjoint_S filling the gap.

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

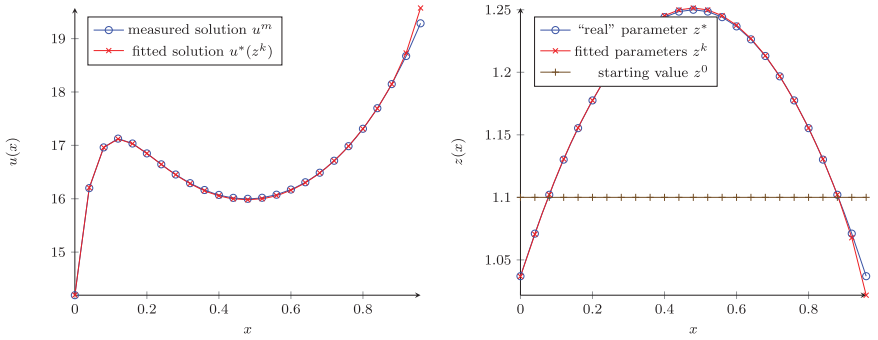
$$\mathbf{r}(\mathbf{u}, \mathbf{z}) = 0, \quad \mathbf{u} \in I\!R^n, \mathbf{z} \in I\!R^n, \tag{28}$$

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 $j = 1, \ldots, \nu$.

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



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

Fig. 3. Visualization of the parameter fitting problem in Equation (27).

the gradient of J at the current iterate $\mathbf{z}^k$ implies the differentiation of the solution process for $\mathbf{u}^*$—that is, differentiation of the solver for Equation (28). Extension of the given example to the use of quasi-Newton methods (e.g., BFGS) is straightforward. 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 article [Safiran et al. 2014].

The algorithmic part of the derivative computation is done by `dco/c++`. The implementation of symbolic tangent and adjoint methods is supported by the previously described external adjoint interface.

The preprocessor $\boldsymbol{\lambda} = P(\mathbf{z})$ is the identity, whereas the postprocessor $p(\mathbf{u})$ computes the cost functional $J(\mathbf{z})$. Figure 3 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 runtime 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 Jacobian inside of Newton's method is performed in tangent mode.

6.1. Tangent Mode

We first compare the computation of a single tangent 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 algorithmic and the symbolic approaches.

The implementation of algorithmic tangent mode is based on the `dco/c++` tangent type dco::gt1s<double>::type. Overloading yields second-order projections during the Jacobian accumulation inside of Newton's method. Theoretically, the overhead in computational cost of algorithmic tangent mode over a passive function evaluation is expected to be of the same order as the passive computation itself.

The implementation of the symbolic NLS and LS modes combines the use of the type dco::gt1s<double>::type with manual implementation effort based on explicit template specialization techniques. In the specialized function, body value and tangent 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 symbolic version of the solver consists of a first-order tangent evaluation of the

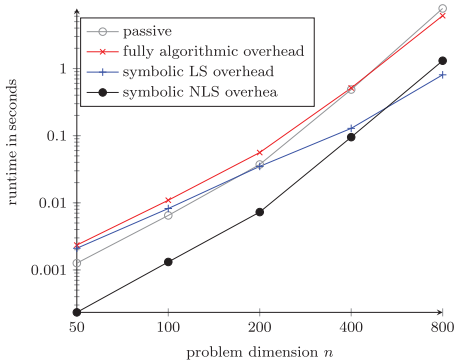


Fig. 4. Runtime to passive runtime overhead comparison for algorithmic and symbolic tangent modes; double logarithmic scale. Order of complexities (slopes): passive $O(n^{3.1})$, fully algorithmic overhead $O(n^{2.8})$, symbolic LS overhead $O(n^{2.1})$, and symbolic NLS overhead $O(n^{3.1})$.

nonlinear function $F(\mathbf{x}, \boldsymbol{\lambda})$—that is,

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

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 the `dco/c++` tangent type. The overhead becomes $O(n^3)$ due to the direct solution of an additional linear $n \times n$ system (see Table I).

Symbolic differentiation of the linear system in symbolic LS mode yields a call to the second-order tangent version of the nonlinear function $F(\mathbf{x}, \boldsymbol{\lambda})$—that is,

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

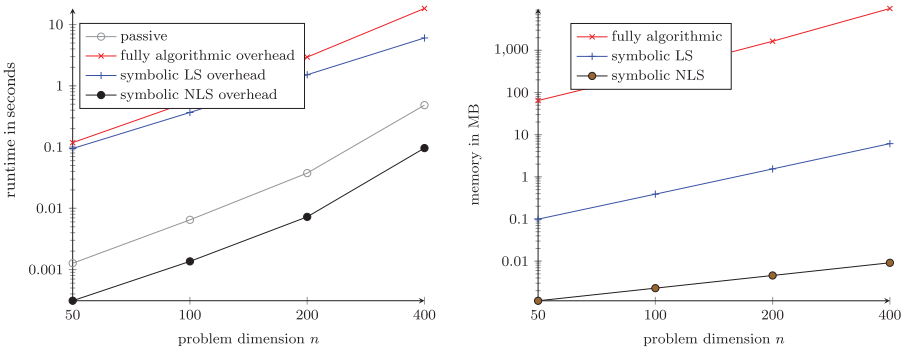
which can be evaluated by overloading based on nested first-order `dco/c++` types. Additionally, a linear $n \times n$ system needs to be solved with the same system matrix, which is already used for the computation of the Newton step (see Equations (17) through (18)). The previously computed factorization 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 Table I).

In Figure 4, we observe the expected behavior for the computational overhead induced by the different approaches. The overhead in algorithmic NLS mode is roughly the same as the passive runtime. Both symbolic NLS and symbolic LS modes yield less overhead, especially for larger problem sizes. Also as expected, the complexity order of the overhead of symbolic LS mode becomes $\nu \cdot O(n^2)$. It outperforms symbolic NLS mode—the complexity order 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 algorithmic and symbolic modes are investigated.

The implementation of algorithmic adjoint mode uses the `dco/c++` adjoint type dco::ga1s<double>::type. Overloading yields second-order adjoint projections during the Jacobian accumulation inside of Newton's method. Automatic C++ template



(a) Runtime (b) Tape Memory

Fig. 5. Runtime to passive runtime overhead comparison and additional memory consumption for algorithmic and symbolic adjoint modes; double logarithmic scale. Order of complexities (slopes): (a) passive $O(n^{2.9})$, fully algorithmic overhead $O(n^{2.4})$, symbolic LS overhead $O(n^{2.0})$, and symbolic NLS overhead $O(n^{2.8})$; (b) fully algorithmic $O(n^{2.4})$, symbolic LS $O(n^{2.0})$, and symbolic NLS $O(n^{1.0})$.

nesting yields the so-called reverse-over-forward mode (e.g., see Naumann [2012]). The overhead of the algorithmic version is expected to range between factors of 4 to 8 relative to a passive evaluation and depending on the given primal code.

The implementation of the symbolic 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 symbolic mode). Moreover, the external adjoint interface of `dco/c++` is used to embed the adjoint function into the reverse section (tape interpretation) to fill the gap left in the tape by the passive solution of the nonlinear (in symbolic NLS mode) or linear (in symbolic LS mode) systems (see Section 5).

In symbolic 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—that is,

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

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 Table I).

In symbolic LS mode, the factorization of the Jacobian and 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 (21)) 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 (25)). The checkpoint memory and the runtime overhead are hence expected to be of order $\nu \cdot O(n^2)$ (see Table I).

Figure 5(a) illustrates the cubic runtime complexities of passive mode and symbolic adjoint mode scaled with different constant factors. The runtime overhead and the tape memory complexity (see Figure 5(b)) of the fully algorithmic version seem to be dominated by the quadratic part of the nonlinear solver (i.e., forward and backward

substitutions). This is expected to converge to cubic behavior for larger problem sizes. Quadratic growth in runtime can be observed in symbolic LS mode. The amount of memory needed for checkpoints (included in the tape memory) in symbolic LS mode is $O(n^2)$. In symbolic NLS mode, it is reduced to $O(n)$ as supported by Figure 5(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 exact Newton case, exact sensitivities of the approximate primal solution with respect to the system parameters can be obtained by differentiating the embedded linear system symbolically as part of a tangent 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, e.g., 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

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

REFERENCES

C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke (Eds.). 2008. *Advances in Automatic Differentiation*. Lecture Notes in Computational Science and Engineering, Vol. 64. Springer, Berlin.

C. Broyden. 1970. The convergence of a class of double-rank minimization algorithms. *Journal of the Institute of Mathematics and Its Applications* 6, 76–90.

M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris (Eds.). 2006. *Automatic Differentiation: Applications, Theory, and Tools*. Lecture Notes in Computational Science and Engineering, Vol. 50. Springer, Berlin.

L. Capriotti. 2011. Fast Greeks by algorithmic differentiation. *Journal of Computational Finance* 14, 3.

B. Christianson. 1994. Reverse accumulation and attractive fixed points. *Optimization Methods and Software* 3, 4, 311–326.

A. J. Davies, D. B. Christianson, L. C. W. Dixon, R. Roy, and P. van der Zee. 1997. Reverse differentiation and the inverse diffusion problem. *Advances in Engineering Software* 28, 4, 217–221. DOI:http://dx.doi.org/10.1016/S0965-9978(97)00005-7

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

S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther (Eds.). 2012. *Recent Advances in Algorithmic Differentiation*. Lecture Notes in Computational Science and Engineering, Vol. 87. Springer, Berlin.

A. Gebremedhin, F. Manne, and A. Pothen. 2005. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review* 47, 4, 629–705.

R. Giering and T. Kaminski. 1998. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software* 24, 437–474.

J. C. Gilbert. 1992. Automatic differentiation and iterative processes. *Optimization Methods and Software* 1, 13–21.

M. Giles. 2008. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In *Advances in Automatic Differentiation*. Lecture Notes in Computational Science and Engineering, Vol. 64. Springer, 35–44.

M. Giles and P. Glasserman. 2006. Smoking adjoints: Fast Monte Carlo Greeks. *Risk* 19, 88–92.

A. Griewank. 1992. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software* 1, 35–54.

A. Griewank and C. Faure. 2002. Reduced functions, gradients and Hessians from fixed-point iterations for state equations. *Numerical Algorithms* 30, 2, 113–139.

A. Griewank, D. Juedes, and J. Utke. 1996. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software* 22, 131–167.

A. Griewank and A. Walther. 2000. 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. DOI:http://dx.doi.org/10.1145/347837.347846

A. Griewank and A. Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (2nd ed.). Other Titles in Applied Mathematics, No. 105. SIAM, Philadelphia, PA.

L. Hascoët, U. Naumann, and V. Pascual. 2005. To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems* 21, 1401–1417.

L. Hascoet and V. Pascual. 2013. The tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software* 39, 3, 20.

M. Henrard. 2014. Adjoint algorithmic differentiation: Calibration and implicit function theorem. *Journal of Computational Finance* 17, 4.

U. Lehmann and A. Walther. 2002. The implementation and testing of time-minimal and resource-optimal parallel reversal schedules. In *Computational Science—ICCS 2002*. Lecture Notes in Computer Science, Vol. 2330. Springer, 1049–1058.

U. Naumann. 2008. Call tree reversal is NP-complete. In *Advances in Automatic Differentiation*. Lecture Notes in Computational Science and Engineering, Vol. 64. Springer, 13–22.

U. Naumann. 2009. DAG reversal is NP-complete. *Journal of Discrete Algorithms* 7, 402–410.

U. Naumann. 2012. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation*. Software, Environments, and Tools, No. 24. SIAM, Philadelphia, PA.

U. Naumann and J. du Toit. 2014. *Adjoint Algorithmic Differentiation Tool Support for Typical Numerical Patterns in Computational Finance*. Technical Report AIB2014-13. RWTH Aachen University, Aachen, Germany. http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2014/2014-13.pdf.

U. Naumann, K. Leppkes, and J. Lotz. 2014. dco/c++ User Guide. Technical Report AIB-2014-03. RWTH Aachen University, Aachen, Germany. http://aib.informatik.rwth-aachen.de/2014/2014-03.pdf.gz.

A. Nemili, E. Özkaya, N. Gauger, F. Kramer, A. Carnarius, and F. Thiele. 2013. Discrete adjoint based sensitivity analysis for optimal flow control of a 3D high-lift configuration. In *Proceedings of the 21st AIAA Computational Fluid Dynamics Conference.* 1–14.

J. Nocedal and S. J. Wright. 2006. *Numerical Optimization* (2nd ed.). Springer-Verlag, New York, NY.

F. Rauser, J. Riehme, K. Leppkes, P. Korn, and U. Naumann. 2010. On the use of discrete adjoints in goal error estimation for shallow water equations. *Procedia Computer Science* 1, 1, 107–115. DOI:http://dx.doi.org/DOI: 10.1016/j.procs.2010.04.013

N. Safiran, J. Lotz, and U. Naumann. 2014. *Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Nonlinear Equations*. Technical Report AIB-2014-07. RWTH Aachen University, Aachen, Germany. http://aib.informatik.rwth-aachen.de/2014/2014-07.pdf.gz.

M. Sagebaum, N. Gauger, J. Lotz, and U. Naumann. 2013. Algorithmic differentiation of a large simulation code including libraries. *Procedia Computer Science* 18, 208–217.

P. Stumm and A. Walther. 2010. New algorithms for optimal online checkpointing. *SIAM Journal on Scientific Computing* 32, 2, 836–854.

M. Towara and U. Naumann. 2013. Toward discrete adjoint OpenFOAM. *Procedia Computer Science* 18, 429–438.

J. Ungermann, J. Blank, J. Lotz, K. Leppkes, L. Hoffmann, T. Guggenmoser, M. Kaufmann, P. Preusse, U. Naumann, and M. Riese. 2011. A 3-D tomographic retrieval approach with advection compensation for the air-borne limb-imager GLORIA. *Atmospheric Measurement Techniques* 4, 11, 2509–2529. DOI:http://dx.doi.org/10.5194/amt-4-2509-2011

Received September 2012; revised December 2014; accepted December 2014