Computational Optimization and Applications (2020) 77:597–616
https://doi.org/10.1007/s10589-020-00214-x

Consistent treatment of incompletely converged iterative linear solvers in reverse-mode algorithmic differentiation

Siamak Akbarzadeh[1] · Jan Hückelheim[1] · Jens-Dominik Müller[1]

Received: 29 July 2019 / Published online: 3 August 2020

Abstract
Algorithmic differentiation (AD) is a widely-used approach to compute derivatives of numerical models. Many numerical models include an iterative process to solve non-linear systems of equations. To improve efficiency and numerical stability, AD is typically not applied to the linear solvers. Instead, the differentiated linear solver call is replaced with hand-produced derivative code that exploits the linearity of the original call. In practice, the iterative linear solvers are often stopped prematurely to recompute the linearisation of the non-linear outer loop. We show that in the reverse-mode of AD, the derivatives obtained with partial convergence become inconsistent with the original and the tangent-linear models, resulting in inaccurate adjoints. We present a correction term that restores consistency between adjoint and tangent-linear gradients if linear systems are only partially converged. We prove the consistency of this correction term and show in numerical experiments that the accuracy of adjoint gradients of an incompressible flow solver applied to an industrial test case is restored when the correction term is used.

Keywords Algorithmic differentiation · Reverse-mode · Iterative linear solvers · Differentiated solver replacement

1 Introduction

The computation of gradients is required for numerous applications, such as shape and topology optimisation, error estimation, goal-based mesh adaptation and uncertainty quantification. Algorithmic differentiation (AD) to automatically produce accurate derivatives for numerical codes [13, 23] is a commonly used

✉ Jens-Dominik Müller
j.mueller@qmul.ac.uk

Siamak Akbarzadeh
siamak.akbarzadehLalkami@qmul.ac.uk

[1] Queen Mary University of London (SEMS), London, UK

technique [3, 5, 8, 16, 32]. In typical numerical models this involves the solution of large linear systems of the form

$$\mathbf{A}x = b$$

which often represents the most expensive part of a computation. We assume here that $\mathbf{A} \in \mathbb{R}^{n\times n}$ is a known non-singular matrix and $x, b \in \mathbb{R}^n$ are the unknown and right hand side (RHS) vectors respectively. Historically, linear solver methods have been categorised into two main groups, namely, direct solvers and iterative solvers, even though this classification has become increasingly blurred by developments that combine solvers from either category. Direct solvers are typically robust and widely used in scientific computing packages, but scale poorly with the problem size. Because of this, applications that require the solution to large linear systems such as CFD flow solvers often use efficient iterative linear solvers [29], commonly used methods include CG, BiCG, GMRES, and algebraic or geometric multi-grid methods.

When AD is applied to an algorithm that uses a linear solver, the linear solver itself is typically differentiated by manually produced replacement derivative code rather than applying AD to the solver. This is often the only practical option, for example if the linear solver is part of an external library, or if an AD-differentiated solver would be computationally inefficient [6, 9] or numerically unstable [21]. A manual differentiation can take into account high-level mathematical properties of a given function, which may not be exploited by an automated AD process.

The most common way of manually differentiating calls to linear solvers (direct or iterative) is presented in [9] and hereafter referred to as *Differentiated Solver Replacement* (DSR). The approach assumes that a linear solver call, `x=solve(A,b)`, is equivalent to the expression $x = \mathbf{A}^{-1}b$, which is valid if the solver computes the solution to machine precision. In this case, the derivative computation can be performed using another call to the same linear solver for a modified system, as shown in Sect. 2.

Often a numerical algorithm solves a non-linear problem and converges to a steady-state solution within a fixed-point iteration (FPI) loop. An example is the typical iterative approach to solving non-linear systems, consisting of a number of outer, non-linear iterations, each of which performs linearisation and contains calls to linear system solves. In the early phase of convergence to the non-linear solution, it is not efficient to exactly solve the linear system for a linearisation based on a poor approximation. An example in Computational Fluid Dynamics (CFD) is the typical segregated approach to solve the incompressible Navier–Stokes equations through a sequence of linear problems for the momentum and pressure correction equations [7]. In a straightforward application of AD to such algorithms, the gradients are accumulated from a zero initial solution, hence are not in FPI form. Different techniques [31] have been presented to have an FPI discrete adjoint of such algorithms. For instance, the *reverse accumulation* method [4]. More recently some AD tools (e.g. Tapenade [15]) even offer this capability as an option for reverse differentiation. However, except for fully coupled systems, implementation of fixed-point adjoints for algorithmically differentiated codes is complex and accumulation of gradients is most often used.

It is to be noted that in the original problem, the incomplete linear solves do not affect the accuracy of the final solution, the *primal* solution, provided enough outer iterations are conducted. Contrary to what one might expect, incomplete linear solves of an accumulated adjoint that uses DSR leads to inaccurate sensitivities, as the analysis and the numerical experiments in this paper show.

In this paper, we show that this is caused by neglecting the influence of the initial guess on the linear system solution, which can be significant if the system is not fully converged. The proposed C-DSR correction achieves consistency between primal and adjoint gradient computation by correctly modelling the adjoint derivatives of an algorithm that uses truncated iterative solvers with the same convergence threshold used for the primal linear systems.

A number of studies have investigated the correction of objective functionals using estimated errors and weighting with adjoint sensitivities, e.g. [12] and [33]. The approaches perform post-processing and consider error estimates derived from the converged steady-state flow solution and weight this with the converged adjoint field. This produces a correction to the objective functional computed from the converged primal. The algorithm proposed in this paper is different, in that it corrects the errors arising from incomplete linear solves in each accumulation step during the computation of the adjoint solution.

The structure of the paper is as follows. In Sect. 2, a brief introduction of AD and DSR is presented. The shortcomings of DSR in the context of reverse-mode AD of algorithms with incompletely converged linear solvers, as well as the proposed correction method C-DSR, are presented in Sect. 3. In Sect. 4 we show numerical experiments that demonstrate the effectiveness of C-DSR. Finally, a summary and conclusions are presented in Sect. 5.

2 Background

In this section, a brief background of AD is provided. Following this, the DSR in both forward and reverse-mode AD is presented.

2.1 Algorithmic differentiation

AD is a technique that evaluates the derivative of the output of a computer program with respect to its inputs. AD differentiates a given primal computer program by applying the chain rule of calculus to the program's sequence of elementary operations (e.g. additions, subtractions, transcendental functions) [13].

AD has two basic modes of operation, namely the forward-mode (resulting in a *tangent-linear* model of the primal), and the reverse-mode (producing an *adjoint* model of the primal). The tangent-linear model computes the product of the Jacobian matrix of the primal program with a given seed vector that has the same number of dimensions as the program input. In contrast, the adjoint model computes the product of the transpose Jacobian with a seed vector that has the size of the primal output.

In the application of AD to numerical codes, the derivative of a given scalar objective function with respect to a scalar primal input variable can be computed at almost equal cost in both tangent-linear and adjoint models. However, in many applications such as gradient-based shape optimisation with CFD, the number of design parameters is much larger than the number of objective functions that are to be computed. As a consequence, the use of adjoint models is essential to compute the gradients at a computational cost that is independent of the number of control variables [10, 11, 19, 24].

A variety of AD tools have been developed in the past, which vary in the supported languages and the used techniques. Examples include Tapenade [15], ADIFOR [1], ADOL-C [14], dco/c++ [18], CoDiPack [30] and ADiMat [2]. The discussion in this study is valid to all types of AD tool.

2.2 Model problem

Consider a non-linear system of the form

$$\mathbf{A}(x,\alpha)x(\alpha) = b(x,\alpha)$$

with α as input and x the solution to the system. The problem can be re-formulated as

$$\mathbf{R}(x(\alpha),\alpha) = \mathbf{A}(x,\alpha)x(\alpha) - b(x,\alpha)\,.$$

Applying a linearisation technique, the numerical solution to such a system can be gained by an iterative algorithm

$$x_m = x_{m-1} + \mathbb{P}_m \mathbf{R}_m \tag{1}$$

where $\mathbb{P}$ is the algorithm operator and the system is considered to be fully solved when $\mathbf{R}_m$ is almost zero. In each iteration of this algorithm a linear system needs to be solved:

$$\mathbf{A}_m x_m - b_m = 0\,, \tag{2}$$

which itself is often solved by an iterative linear solver.

In many numerical models, the objective functional, J, that is going to be differentiated is implicitly dependent on the design variable α through the solution $x(\alpha)$ of a non-linear system of equations similar to (1). Assuming $J = J(x(\alpha),\alpha)$, the general form of such an algorithm is shown in Algorithm 1.

```
$x_0 \leftarrow$ evaluate_x_0 $(\alpha)$
DO m=1:M
IF m==1 THEN
    $x_m^0 = x_0$
ELSE
    $x_m^0 = x_{m-1}^N$
END IF
    $\mathbf{A}_m \leftarrow$ evaluate_A $(x_m^0)$
    $b_m \leftarrow$ evaluate_b $(x_m^0)$
    $x_m^N \leftarrow$ solve $(\mathbf{A}_m, x_m^0, b_m, N)$
End DO
$J \leftarrow$ evaluate_J $(x_M^N, \alpha)$
```

Algorithm 1: Algorithm to solve the non-linear system $\mathbf{A}(\,x(\alpha),\,\alpha\,)\,x(\alpha) = b(x,\alpha)$.

In Algorithm 1, the subscripts (m, M) denote non-linear (outer) iterations while the superscript N denotes the solution after N linear (inner) solver iterations. The arrows denote output. $\mathbf{A}$ and b are being updated in the non-linear loop and x_m^N is the approximate solution to the linear system after N inner iterations at outer iteration m. For each linear solve, `solve`, x is an input (as the initial guess, x_m^0) and an output (as the solution, x_m^N). The objective functional J is dependent on the final solution of the algorithm, x_M^N.

In the following, we consider the case that the number of inner iterations N is not sufficient to fully converge the linear, inner solver to machine accuracy, $x_m^N \approx \mathbf{A}_m^{-1} b_m$. However, we assume that a sufficient number of M outer, non-linear iterations is conducted, each containing N inner, linear iterations. In this way, in the final outer iterations, the linear system is sufficiently close to the non-linear system, and the error in both non-linear and linear system solutions is close to machine precision.

2.3 Differentiated solver replacement: tangent-linear

The forward differentiation of the gradient of the objective functional J w.r.t. α requires the differentiation of the non-linear algorithm which at iteration 'm' reads

$$\dot{x}_m = \dot{x}_{m-1} + \dot{\mathbb{P}}_m \mathbf{R}_m + \mathbb{P}_m \dot{\mathbf{R}}_m$$

Knowing that the number of primal outer iteration is enough to drive $\mathbf{R}_m$ to zero, the differentiated system can be simplified as

$$\dot{x}_m = \dot{x}_{m-1} + \mathbb{P}_m \dot{\mathbf{R}}_m \,.$$

This requires to compute a solution to the differentiated linear system (2) in each differentiated outer iteration as

$$\mathbf{A}_m \dot{x}_m = \dot{b}_m - \dot{\mathbf{A}}_m x_m \quad \text{or} \quad \dot{x}_m = \mathbf{A}_m^{-1}(\dot{b}_m - \dot{\mathbf{A}}_m x_m)\,. \tag{3}$$

The forward differentiation of the Algorithm 1 is shown in Algorithm 2. The function appended with the suffix '_d' represents the tangent-linear derivative of that function.

```
$x_0, \dot{x}_0 \leftarrow$ evaluate_x_0_d $(\alpha, \dot{\alpha})$
DO m=1:M
IF m==1 THEN
    $x_m^0 = x_0$
    $\dot{x}_m^0 = \dot{x}_0$
ELSE
    $x_m^0 = x_{m-1}^N$
    $\dot{x}_m^0 = \dot{x}_{m-1}^N$
END IF
    $\mathbf{A}_m, \dot{\mathbf{A}}_m \leftarrow$ evaluate_A_d $(x_m^0, \dot{x}_m^0)$
    $b_m, \dot{b}_m \leftarrow$ evaluate_b_d $(x_m^0, \dot{x}_m^0)$
    $x_m^N, \dot{x}_m^N \leftarrow$ solve_d $(\mathbf{A}_m, \dot{\mathbf{A}}_m, x_m^0, \dot{x}_m^0, b_m, \dot{b}_m, N)$
End DO
$J, \dot{J} \leftarrow$ evaluate_J_d $(x_M^N, \dot{x}_M^N, \alpha, \dot{\alpha})$
```

Algorithm 2: Forward-mode appl. of AD to Algorithm 1.

The forward-differentiation in Algorithm 2 naturally inherits the fixed-point form of the primal, hence the resulting tangent-linear solution and the gradients computed with it are also impervious to incomplete inner solves, as long as the number of outer iterations is sufficient. If the problem to solve is steady, then linearisation around the converged solution to (2) is sufficient, making the entire problem linear which means that inner and outer iterations solve the same problem.

As mentioned in the Introduction, the differentiation of linear solvers is in practice often performed using an approach that we refer to as *differentiated linear solver replacement* or DSR. A sample pseudo code of DSR in forward-mode for the linear solver in Algorithm 1 is illustrated in Algorithm 3.

```
solve_d ($\mathbf{A}_m, \dot{\mathbf{A}}_m, x_m^0, \dot{x}_m^0, b_m, \dot{b}_m, N$)
{
    $x_m^N \leftarrow$ solve ($\mathbf{A}_m, x_m^0, b_m, N$)
    $\dot{b}'_m = \dot{b}_m - \dot{\mathbf{A}}_m x_m^N$
    $\dot{x}_m^N \leftarrow$ solve ($\mathbf{A}_m, \dot{x}_m^0, \dot{b}'_m, N$)
}
```

Algorithm 3: DSR in forward-mode AD.

2.4 Differentiated solver replacement: adjoint

The objective J is assumed here to depend on the control α and the state x: $J = J(\alpha, x)$. Its derivative is hence

$$\dot{J} = \frac{\partial J}{\partial \alpha} + \frac{\partial J}{\partial x}\dot{x}$$

In many applications, the function $J(\alpha)$ can be computed explicitly, without requiring linear solvers. We therefore focus in this work on the term $\frac{\partial J}{\partial x}\dot{x}$ which is implicit, that is, it involves a linear solve for $\dot{x}$. Since x is a function of A and b, the function J also depends on A and b, or formally, $x = x(\mathbf{A}, b)$ and $J = J(\mathbf{A}, b)$, and we can expand and transpose as

$$\frac{\partial J}{\partial x}\dot{x} = \frac{\partial J}{\partial \mathbf{A}}\frac{\partial \mathbf{A}}{\partial x}\dot{x} + \frac{\partial J}{\partial b}\frac{\partial b}{\partial x}\dot{x} \tag{4}$$

$$\dot{x}^T\left(\frac{\partial J}{\partial x}\right)^T = \dot{x}^T\left(\frac{\partial \mathbf{A}}{\partial x}\right)^T\left(\frac{\partial J}{\partial \mathbf{A}}\right)^T + \dot{x}^T\left(\frac{\partial b}{\partial x}\right)^T\left(\frac{\partial J}{\partial b}\right)^T \tag{5}$$

By definition, the adjoint of the reverse-differentiated variables is

$$\bar{x} = \left(\frac{\partial J}{\partial x}\right)^T, \ \bar{\mathbf{A}} = \left(\frac{\partial J}{\partial \mathbf{A}}\right)^T, \ \bar{b} = \left(\frac{\partial J}{\partial b}\right)^T,$$

which simplifies (5) to

$$\dot{x}^T\bar{x} = \dot{\mathbf{A}}^T\bar{\mathbf{A}} + \dot{b}^T\bar{b} \quad \text{or} \quad \langle \dot{x}, \bar{x}\rangle = \langle \dot{\mathbf{A}}, \bar{\mathbf{A}}\rangle + \langle \dot{b}, \bar{b}\rangle, \tag{6}$$

where the column vectors are expressed as row matrices and $\langle\ ,\ \rangle$ denotes an inner product between matrices. Substituting $\dot{x}$ with $\mathbf{A}^{-1}(\dot{b} - \dot{\mathbf{A}}x)$, one can rearrange (6) into

$$\begin{aligned}\langle \dot{\mathbf{A}}, \bar{\mathbf{A}}\rangle + \langle \dot{b}, \bar{b}\rangle &= \langle \mathbf{A}^{-1}(\dot{b} - \dot{\mathbf{A}}x), \bar{x}\rangle \\ &= \langle \mathbf{A}^{-1}\dot{b}, \bar{x}\rangle - \langle \mathbf{A}^{-1}\dot{\mathbf{A}}x, \bar{x}\rangle\,.\end{aligned} \tag{7}$$

Then recalling from linear algebra [20], the inner product between two matrices, $\mathbf{M}_1$ and $\mathbf{M}_2$, reads

$$\langle \mathbf{M}_1, \mathbf{M}_2 \rangle = \mathrm{Tr}\big(\mathbf{M}_2^T \mathbf{M}_1\big)\,.$$

Here 'Tr' stands for trace of a matrix, i.e., the sum of its diagonal elements. Now we can expand and rewrite the inner products in RHS of (7),

$$\begin{aligned}\langle \mathbf{A}^{-1}\dot{b}, \bar{x}\rangle &= \mathrm{Tr}\big(\bar{x}^T\mathbf{A}^{-1}\dot{b}\big)\\ &= \mathrm{Tr}\big((\mathbf{A}^{-T}\bar{x})^T\dot{b}\big)\\ &= \langle \dot{b}, \mathbf{A}^{-T}\bar{x}\rangle\,,\end{aligned} \tag{8}$$

$$\begin{aligned}-\langle \mathbf{A}^{-1}\dot{\mathbf{A}}x, \bar{x}\rangle &= -\ \mathrm{Tr}\big(\bar{x}^T\mathbf{A}^{-1}\dot{\mathbf{A}}x\big)\\ &= -\ \mathrm{Tr}\big((\mathbf{A}^{-T}\bar{x})^T(x^T\dot{\mathbf{A}}^T)^T\big)\\ &= -\ \mathrm{Tr}\big((\mathbf{A}^{-T}\bar{x}x^T)^T\dot{\mathbf{A}}\big)\\ &= -\ \langle \dot{\mathbf{A}}, \mathbf{A}^{-T}\bar{x}x^T\rangle\,.\end{aligned} \tag{9}$$

Finally, (8) and (9) can be replaced into (7)

$$\langle \dot{\mathbf{A}}, \bar{\mathbf{A}}\rangle + \langle \dot{b}, \bar{b}\rangle = \langle \dot{\mathbf{A}}, -\mathbf{A}^{-T}\bar{x}x^T\rangle + \langle \dot{b}, \mathbf{A}^{-T}\bar{x}\rangle\,. \tag{10}$$

Therefore, $\bar{b}$ and $\bar{\mathbf{A}}$ at iteration 'm' can be expressed as follows [9]

$$\bar{b}_m = \mathbf{A}_m^{-T}\bar{x}_m, \tag{11a}$$

$$\bar{\mathbf{A}}_m = -\mathbf{A}_m^{-T}\bar{x}_m x_m^T = -\bar{b}_m x_m^T\,. \tag{11b}$$

In practice, the adjoints are incremented because they may already contain previously computed sensitivities from elsewhere in the program (see Algorithm 4). One can derive the adjoint of $\mathbf{A}$ and b and obtain the reverse-DSR as,

$$\bar{b}'_m = \mathbf{A}_m^{-T}\bar{x}_m, \tag{12a}$$

$$\bar{b}_m = \bar{b}_m + \bar{b}'_m, \tag{12b}$$

$$\bar{\mathbf{A}}_m = \bar{\mathbf{A}}_m - \bar{b}'_m x_m^T, \tag{12c}$$

$$\bar{x}_m = 0\,. \tag{12d}$$

The reverse-mode application of AD to the Algorithm 1 and the hand assembled reverse DSR are illustrated in Algorithms 4 and 5. A function appended with the suffix '_b' represents the reverse derivative of that function. For brevity, brackets are used to show the accumulation of sensitivities for matrices and vectors via reverse-differentiated functions.

```
$\bar{J} = 1$
$\{\bar{x}_M^N, \bar{\alpha}\} = \{\bar{x}_M^N, \bar{\alpha}\} + \{\texttt{evaluate\_J\_b}\,(x_M^N, \bar{x}_M^N, \alpha, \bar{\alpha}, \bar{J})\}$
$\bar{\mathbf{A}}_M = 0$
$\bar{b}_M = 0$
DO  m=M:1
    $\bar{x}_m^0, \bar{\mathbf{A}}_m, \bar{b}_m \leftarrow \texttt{solve\_b}\,(\mathbf{A}_m, \bar{\mathbf{A}}_m, x_m^0, \bar{x}_m^N, b_m, \bar{b}_m, N)$
    $\{\bar{x}_m^0, \bar{b}_{m-1}\} = \{\bar{x}_m^0, \bar{b}_m\} + \{\texttt{evaluate\_b\_b}\,(x_m^0, \bar{x}_m^0, \bar{b}_m)\}$
    $\{\bar{x}_m^0, \bar{\mathbf{A}}_{m-1}\} = \{\bar{x}_m^0, \bar{\mathbf{A}}_m\} + \{\texttt{evaluate\_A\_b}\,(x_m^0, \bar{x}_m^0, \bar{\mathbf{A}}_m)\}$
IF  m==1  THEN
        $\bar{x}_0 = \bar{x}_m^0$
ELSE
        $\bar{x}_{m-1}^N = \bar{x}_m^0$
END  IF
End  DO
$\{\bar{\alpha}\} = \{\bar{\alpha}\} + \{\texttt{evaluate\_x\_0\_b}\,(\alpha, \bar{\alpha}, \bar{x}_0)\}$
```

Algorithm 4: Reverse-mode appl. of AD to Alg. 1 in which the adjoint sensitivities are accumulated.

```
solve_b $(\mathbf{A}_m, \bar{\mathbf{A}}_m, x_m^0, \bar{x}_m^N, b_m, \bar{b}_m, N)$
  {
     $x_m^N \leftarrow \texttt{solve}\,(\mathbf{A}_m, x_m^0, b_m, N)$
     $\mathbf{A}_m^T \leftarrow \texttt{transpose\_A}\,(\mathbf{A}_m)$
     $\bar{b}_m'^N \leftarrow \texttt{solve}\,(\mathbf{A}_m^T, \bar{b}_m', \bar{x}_m^N, N)$
     $\bar{b}_m = \bar{b}_m + \bar{b}_m'^N$
     $\bar{\mathbf{A}}_m = \bar{\mathbf{A}}_m - \bar{b}_m'^N (x_m^N)^T$
     $\bar{x}_m^0 = 0$
  }
```

Algorithm 5: DSR in reverse-mode AD.

As we will show in Sect. 3, in contrast to the primal and its forward differentiation, if the adjoint linear systems are not fully solved, $\bar{b}'_m \neq \mathbf{A}_m^{-T}\bar{x}_m$, they introduce an error to the system that would not vanish even after a large number of outer iterations. In the next section, this error and its correction will be discussed.

3 Corrected differentiated solver replacement in reverse-mode

In this section, the forward and reverse differentiation of the Jacobi solver within an outer, non-linear iterative solver is discussed. Furthermore, we discuss the effect that the initial guess has on the solution, when the differentiated inner linear solver is only partially converged. The C-DSR correction method is then developed, which includes a correction term for this error. Finally, we demonstrate the benefit of this correction method. We choose the Jacobi solver because it is easy to prove properties of its differentiation and convergence. However, our C-DSR method also benefits other solvers, as we will show later in this paper.

3.1 Error correction for reverse differentiation of Jacobi solver

The system matrix $\mathbf{A}_m$ can be decomposed as $\mathbf{A}_m = \mathbf{D}_m + \mathbf{Q}_m$, where $\mathbf{D}_m$ and $\mathbf{Q}_m$ hold the diagonal and off-diagonals entries of $\mathbf{A}_m$, respectively. The iterative relaxation scheme can be written as

$$x_m^{n+1} = \mathbf{D}^{-1}(b - \mathbf{Q}x_m^n) \,. \tag{13}$$

The error due to incomplete Jacobi convergence can be expressed as

$$\begin{aligned} x_m^{n+1} - x_m = \delta x_m^{n+1} &= \mathbf{D}^{-1}(b - \mathbf{Q}x_m^n) - x_m \\ &= \mathbf{D}^{-1}b - \mathbf{D}^{-1}\mathbf{Q}x_m^n - x_m \\ &= \mathbf{D}^{-1}\mathbf{A}x_m - \mathbf{D}^{-1}\mathbf{Q}x_m^n - x_m \\ &= (\mathbf{D}^{-1}\mathbf{A} - \mathbf{I})x_m - \mathbf{D}^{-1}\mathbf{Q}x_m^n \\ &= (\mathbf{D}^{-1}\mathbf{A} - \mathbf{I})x_m - (\mathbf{D}^{-1}\mathbf{A} - \mathbf{I})x_m^n \\ &= (\mathbf{I} - \mathbf{D}^{-1}\mathbf{A})\delta x_m^n \,, \end{aligned} \tag{14}$$

where x_m is the exact solution to the linear system at the (m)th outer iteration. Therefore, after N iterations starting from an initial guess x_m^0, the approximated solution obtained from the linear solver can be written explicitly as

$$x_m^N = x_m + (\mathbf{I} - \mathbf{D}^{-1}\mathbf{A})^{(N)}(x_m^0 - x_m) \,. \tag{15}$$

The initial guess of the system is actually the solution to the linear system in the previous outer iteration. In this context, one can write

$$x_m^N = x_m + (\mathbf{I} - \mathbf{D}_m^{-1}\mathbf{A}_m)^{(N)}(x_{m-1}^N - x_m); \quad m = 1, \dots, M \,. \tag{16}$$

in which after a sufficient number iterations

$$(\mathbf{I} - \mathbf{D}_m^{-1}\mathbf{A}_m)^{(N)} = 0 \,. \tag{17}$$

3.1.1 Forward differentiation

The tangent-linear model has the same behaviour as its primal, that is, the initial guess for the differentiated solver replacement (DSR) is the solution to the system in the previous outer iteration (see Algorithm 2):

$$x_m^0 = x_{m-1}^N,$$
$$\dot{x}_m^0 = \dot{x}_{m-1}^N\,.$$

The tangent-linear model of (16) is given by

$$\dot{x}_m^N = \dot{x}_m + (\epsilon_1)_m + (\epsilon_2)_m, \tag{18}$$

where $\dot{x}_m$ is the exact solution to the tangent-linear problem , ϵ_1 and ϵ_2 are the errors due to incomplete solve of the primal and tangent-linear problems, respectively

$$\begin{aligned}(\epsilon_1)_m =&(N)(-\dot{\mathbf{D}}_m^{-1}\mathbf{A}_m - \mathbf{D}_m^{-1}\dot{\mathbf{A}}_m)\\ &\left[(\mathbf{I} - \mathbf{D}_m^{-1}\mathbf{A}_m)^{(N-1)}(x_{m-1}^N - x_m)\right],\\ (\epsilon_2)_m =&(\mathbf{I} - \mathbf{D}_m^{-1}\mathbf{A}_m)^{(N)}(\dot{x}_{m-1}^N - \dot{x}_m)\,.\end{aligned}$$

Even though the linear systems are not solved to machine precision in each outer iteration, the errors vanish when the outer loop is iterated sufficiently. Please be aware that the number of outer iteration, M, is considered large enough such that $\mathbf{A}_m = \mathbf{A}_{m-1}$ and $b_m = b_{m-1}$. As a result, the initial guess and final result of the linear solver are identical to machine precision in the final outer iteration, or formally,

$$x_M^N = x_M^0 = x_{M-1}^N$$
$$\dot{x}_M^N = \dot{x}_M^0 = \dot{x}_{M-1}^N$$

3.1.2 Reverse differentiation

The reverse-mode application of AD to the model is shown in Algorithm 4, where the sensitivities are accumulated over the reverse loop and for better clarity, except in the DSR, the primal expressions are not depicted. The incomplete convergence of the adjoint linear system means

$$\bar{b}_m'^N \neq \mathbf{A}_m^{-T}\bar{x}_m^N \tag{19}$$

with the residual $(\epsilon_{\bar{b}})_m$ of the system

$$(\epsilon_{\bar{b}})_m = \mathbf{A}_m^{-T}\bar{x}_m^N - \bar{b}_m'^N \tag{20}$$

As a result, the computation of terms $\bar{\mathbf{A}}$ and $\bar{b}$ are affected in each DSR call such that

$$\bar{b}_m = \bar{b}_m + \bar{b}'^N_m + (\epsilon_{\bar{b}})_m, \tag{21}$$

$$\begin{aligned}\bar{\mathbf{A}}_m &= \bar{\mathbf{A}}_m - (\bar{b}'^N_m + (\epsilon_{\bar{b}})_m)(x^N_m)^T, \\ &= \bar{\mathbf{A}}_m - \bar{b}'^N_m (x^N_m)^T - (\epsilon_{\bar{b}})_m (x^N_m)^T .\end{aligned} \tag{22}$$

It is not difficult to derive the derivative of J w.r.t. α in the reverse-mode from (4, 5),

$$\bar{\alpha} = \dot{x}^T \left(\frac{\partial \mathbf{A}}{\partial x}\right)^T \sum_{M}^{1} \bar{\mathbf{A}}_m + \dot{x}^T \left(\frac{\partial b}{\partial x}\right)^T \sum_{M}^{1} \bar{b}_m, \tag{23}$$

which leads to an accumulated error given by

$$\epsilon_{\bar{\alpha}} = \dot{x}^T \left(-\left(\frac{\partial \mathbf{A}}{\partial x}\right)^T \sum_{M}^{1} \left((\epsilon_{\bar{b}})_m (x^N_m)^T\right) + \left(\frac{\partial b}{\partial x}\right)^T \sum_{M}^{1} (\epsilon_{\bar{b}})_m \right) \tag{24}$$

The source of the error is the residual of the adjoint systems, and this error is accumulated over the outer iterations. It is important to realise that running more outer iterations does not remove the error, contrary to what might be extrapolated from the behaviour of the primal. Due to the accumulative nature of the adjoint differentiation, with standard DSR any incomplete convergence of the adjoint systems imparts an error on the gradients, which remains even if the number of outer iterations is enough for the primal algorithm to converge. To correct this error, the state of the art is to converge the inner adjoint system solves to machine precision, which makes the adjoint computation significantly more expensive than the primal. This paper proposes an alternative approach, namely an effective way to compute a correction for this error.

3.2 Reverse-DSR correction

Equation (18) can be rewritten as

$$\begin{aligned}\dot{x}^N_m =& \left[\mathbf{I} - (\mathbf{I} - \mathbf{D}^{-1}_m \mathbf{A}_m)^{(N)}\right] \dot{x}_m \\ &+ \underbrace{(\mathbf{I} - \mathbf{D}^{-1}_m \mathbf{A}_m)^{(N)} \dot{x}^N_{m-1}}_{(\gamma)_m} \\ &+ (\epsilon_1)_m ,\end{aligned} \tag{25}$$

where the term γ is the influence of the initial guess on the approximated tangent-linear derivative in the (m)th outer iteration of the algorithm after N Jacobi steps (linear solver iterations),

$$\gamma_m = (\mathbf{I} - \mathbf{D}^{-1}_m \mathbf{A}_m)^{(N)} \dot{x}^N_{m-1} = (\mathbf{I} - \mathbf{D}^{-1}_m \mathbf{A}_m)^{(N)} \dot{x}^0_m . \tag{26}$$

To derive the reverse differentiation of expression (25) we first rewrite it as addition of three vectors:

$$\dot{x}_m^N = \dot{l}_1 + \dot{l}_2 + \dot{l}_3\,. \tag{27}$$

As shown in the section 2.2.1 of [9], for such an equation the following expression holds in the reverse-mode:

$$\bar{l}_1 = \bar{l}_2 = \bar{l}_3 = \bar{x}_m^N\,. \tag{28}$$

Moreover, from section 2.2.2 of [9], the adjoint of a multiplication expression, $\dot{l}_2 = (\mathbf{I} - \mathbf{D}_m^{-1}\mathbf{A}_m)^{(N)}\dot{x}_{m-1}^N$, gives

$$\bar{x}_m^N = \left[(\mathbf{I} - \mathbf{D}_m^{-1}\mathbf{A}_m)^{(N)}\right]^T\bar{l}_2\,. \tag{29}$$

From (28, 29) the influence of initial guess in the reverse-mode can be shown to be

$$\bar{x}_m^0 = \bar{x}_{m-1}^N = (\mathbf{I} - \mathbf{A}_m^T\mathbf{D}_m^{-1})^{(N)}\bar{x}_m^N\,. \tag{30}$$

The vector $\bar{x}_m^0$ in (30) is one of the outputs of the differentiated solver (see Algorithm 4). On the other hand, DSR is based on the assumption that the linear systems are fully converged; meaning N is large enough such that

$$\bar{x}_m^0 = \cancelto{0\quad [\textit{if } N \textit{ is large enough}]}{(\mathbf{I} - \mathbf{A}_m^T\mathbf{D}_m^{-1})^{(N)}}\bar{x}_m^N = 0\,. \tag{31}$$

However, the incomplete convergence causes this assumption to be violated. If the adjoint linear system is not fully solved the term $\bar{x}_m^0$ is not zero. Consequently, the sensitivity computation is inaccurate by the error shown in (24).

N matrix-vector products are required to compute (30), which is essentially as expensive as the primal linear solver. However, it can be computed much cheaper as a by-product of a computation that is already part of the DSR.

In order to solve $\mathbf{A}_m^T\bar{b}'_m = \bar{x}_m^N$ in DSR, one Jacobi iteration is performed as

$$\bar{b}'^{n+1}_m = \mathbf{D}_m^{-1}(\bar{x}_m^N - \mathbf{R}_m^T\bar{b}'^n_m). \tag{32}$$

If the same number of iterations N are used for (32) as for the primal system and using an initial guess of zero, from (15) one obtains

$$\begin{aligned}\bar{b}'^N_m &= \bar{b}'_m + (\mathbf{I} - \mathbf{D}_m^{-1}\mathbf{A}_m^T)^{(N)}(0 - \bar{b}'_m)\\ &= (\mathbf{I} - (\mathbf{I} - \mathbf{D}_m^{-1}\mathbf{A}_m^T)^{(N)})\bar{b}'_m\,.\end{aligned} \tag{33}$$

Using (33) it can be shown that computing the residual after N iterations can be done with a single matrix-vector product which yields exactly the same result as (30):

$$
\begin{aligned}
\boldsymbol{r}_m &= \bar{x}_m^N - \mathbf{A}_m^T \bar{b}_m'^N \\
&= \bar{x}_m^N - \mathbf{A}_m^T \left[(\mathbf{I} - (\mathbf{I} - \mathbf{D}_m^{-1}\mathbf{A}_m^T)^{(N)}) \bar{b}_m' \right] \\
&= \cancelto{0}{\bar{x}_m^N - \mathbf{A}_m^T \bar{b}_m'} + \mathbf{A}_m^T \left[(\mathbf{I} - \mathbf{D}_m^{-1}\mathbf{A}_m^T)^{(N)} \bar{b}_m' \right] \\
&= (\mathbf{I} - \mathbf{A}_m^T \mathbf{D}_m^{-1})^{(N)} \mathbf{A}_m^T \bar{b}_m' \\
&= (\mathbf{I} - \mathbf{A}_m^T \mathbf{D}_m^{-1})^{(N)} \bar{x}_m^N \\
&= \bar{x}_m^0 \, .
\end{aligned}
\tag{34}
$$

Hence, if the adjoint system of DSR in reverse-mode is not fully solved, the output variable $\bar{x}_m^0$ can be defined as the residual of the system. We call this C-DSR, as in *corrected DSR*. The DSR and C-DSR approaches are compared in Algorithms 6, 7 and 8.

$$
\begin{cases}
\mathbf{A}_m x_m^N \approx b_m \\
\dot{b}_m' = \dot{b}_m - \dot{\mathbf{A}}_m x_m^N \\
\mathbf{A}_m \dot{x}_m^N \approx \dot{b}_m'
\end{cases} ;
$$

Algorithm 6: Forward DSR.

$$
\begin{cases}
\mathbf{A}_m x_m^N \approx b_m \\
\mathbf{A}^T \bar{b}_m'^N \approx \bar{x}_m^N \\
\bar{b}_m = \bar{b}_m + \bar{b}_m'^N \\
\bar{\mathbf{A}}_m = \bar{\mathbf{A}}_m - \bar{b}_m' (x_m^N)^T \\
\boxed{\bar{x}_m^0 = 0}
\end{cases} ;
$$

Algorithm 7: Reverse DSR.

$$
\begin{cases}
\mathbf{A}_m x_m^N \approx b_m \\
\mathbf{A}^T \bar{b}_m'^N \approx \bar{x}_m^N \\
\bar{b}_m = \bar{b}_m + \bar{b}_m'^N \\
\bar{\mathbf{A}}_m = \bar{\mathbf{A}}_m - \bar{b}_m' (x_m^N)^T \\
\boxed{\bar{x}_m^0 = \bar{x}_m^N - \mathbf{A}_m^T \bar{b}_m'^N}
\end{cases} ;
$$

Algorithm 8: Reverse C-DSR.

3.3 Application of C-DSR to other solvers

In the previous section we presented a correctness proof for C-DSR with Jacobi solvers. A similar proof can be established for any other linear solver using linear operators. Linear solvers with non-linear operators,

such as GMRES or CG, do not yield to this type of analysis. However, the test cases shown in the remainder of this paper demonstrates that C-DSR also leads to improved consistency for other solvers, when incomplete convergence is set at levels typical for the primal algorithm.

4 Test cases

In this section we first demonstrate the effectiveness of C-DSR using a one-dimensional heat equation solver that uses Jacobi iterations to solve the linear systems. Then with a three-dimensional CFD solver we show that the application of C-DSR to Krylov-type linear solvers also improves the gradient accuracy.

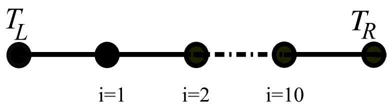

Fig. 1 1D non-linear steady-state heat transfer

Table 1 1D heat transfer: Jacobi solver set up

	Maximum iterations	Convergence (absolute tolerance)
Settings 1	1000	1e−14
Settings 2	1000	1e−4

Table 2 1D heat transfer: comparison of derivative calculation accuracy

	AD-forward	AD-adjoint
Settings 1		
Differentiation of solver	0.14308617311652	0.14308617311652
DSR	0.14308617311652	0.14308617311652
C-DSR	–	0.14308617311652
FD: 0.14308617551322		
Settings 2		
Differentiation of solver	0.14308617311652	0.14308617311652
DSR	0.14308617311652	0.13289602339787
C-DSR	–	0.14308617311652
FD: 0.14308617551322		

4.1 One-dimensional (1D) non-linear heat equation

The first validation study is the finite-difference (central differences in space, backward Euler in time) solution to a non-linear 1D steady-state heat conduction problem in a uniform rod lying on the x-axis from $x_L = 0$ to $x_R = 1$

$$\frac{\partial}{\partial x}(k(T)\frac{\partial T}{\partial x}) = 0, \tag{35}$$

where the heat conduction coefficient k is a simple linear function of temperature T, $k = c_1 + c_2 T$, where $c_1 = 1.1$ and $c_2 = 0.2$.

The domain (see Fig. 1) has 12 nodes and is discretised by central finite difference in space and backward Euler in time. Dirichlet boundary conditions are imposed on both ends. The temperature at the right boundary T_R is defined as the control variable and the objective function is evaluated as a function of the temperature at one of the internal nodes, $J = 100 \times (T_{(i=1)})^2$. The primal outer loop is iterated enough that in the final outer iterations the error of the linear system is close to machine zero. The Tapenade source-transformation AD tool[15] is used to differentiate the code with checkpointing of all outer iterations in the reverse-mode.

It is worth noting that this is a steady-state problem that does not require time marching; hence the adjoint solution can be computed by linearising only around the final steady state solution, without checkpointing. We solve the primal and its adjoint in this way so that it can serve as a model problem that can be extended to more complex problems such as unsteady or segregated (decoupled) solvers later in the paper.

Two different settings are considered for Jacobi solver (see Table 1). The results are compared in Table 2. The results confirm that when the Jacobi solver is solved to machine precision, the sensitivity ($\frac{dJ}{dT_R}$) obtained by DSR (in both AD-forward and adjoint) and the second order finite difference computation are in good agreement. However, when the solver is not fully solved, the computed sensitivity with DSR in reverse-mode shows a relative error of 6%. C-DSR improves the accuracy of gradient and reduces the error to machine precision.

4.2 Three-dimensional (3D) S-Bend Duct

The second validation study is an adjoint CFD computation of a VW Golf air climate duct [34], a benchmark case of the About Flow project [26] provided by Volkswagen AG. The flow is steady, laminar and incompressible with a Reynolds number of 300 at the inlet relative to the height of the duct, the domain is discretised with 40,000 hexahedral mesh cells.

The objective function is mass averaged pressure drop between inlet and outlet. To solve the flow, the in-house incompressible flow solver *gpde* [17] is used, which is based on the finite volume segregated SIMPLE pressure-correction method [25]. The arising linear systems for momentum and pressure correction are solved using bi-conjugate gradient stabilised (Bi-CGSTAB) and conjugate

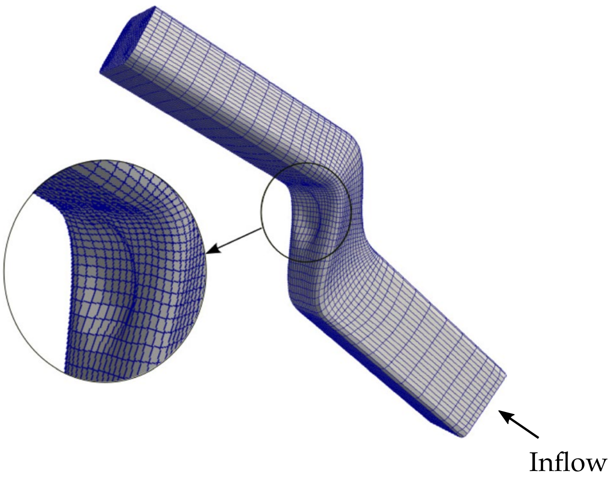

Fig. 2 3D duct flow with perturbed inward bump

Table 3 Linear solvers set up for the primal, adjoint and forward AD

	Momentum eqn.	Pressure eqn.
Linear solver	Bi-CGSTAB	CG
Maximum number of iterations	1000	1000
Convergence (relative tolerance)	1e−3	1e−5

Table 4 3D duct flow: comparison of derivative calculation accuracy

Method	AD-forward	AD-adjoint
DSR	1.364812242779	1.365269308207
C-DSR	–	1.364812240069
FD: 1.364812241599		

Table 5 The effect of iterative linear solver accuracy on DSR in reverse-mode

	Bi-CGSTAB	CG	Max. iter.	DSR in AD-adjoint
Settings 1	1e−3	1e−5	1000	1.365269308207
Settings 2	1e−6	1e−8	2000	1.364813964852
Settings 3	1e−8	1e−10	2000	1.364812307219
Settings 4	1e−12	1e−12	10,000	1.364812297209

gradient (CG) linear solvers, respectively, from the SPARSKIT library [28]. The spring analogy method [27] is implemented in *gpde* to deform the volume mesh following a design change. The *gpde* solver is written in FORTRAN 90 and differentiated by the AD tool Tapenade [15] and without checkpointing all outer iterations.

To compare sensitivities, the surface mesh coordinates of the middle S-section of the duct, $\mathbf{x}_i$, are perturbed by a cosine function,

$$\mathbf{x}_i = \mathbf{x}_i - \alpha \cos(\frac{\pi d_i}{2})\, \mathbf{n}_0; \quad d_i = \min(\|\mathbf{x}_i - \mathbf{x}_0\|, 1.0) \tag{36}$$

where $\mathbf{x}_0$ and $\mathbf{n}_0$ are the bump centre and the surface normal, respectively. The perturbation is designed to create an inward bump in the duct (see Fig. 2) and the bump height is controlled by the variable α.

The differentiated code computes the derivative of the objective function at fully converged flow state w.r.t. the design variable, in this case the height of the perturbed bump.

In practice, the convergence criteria of linear solvers in non-linear numerical methods such as CFD solvers are determined from experience [7, 22]. The solver settings for this duct flow using the gpde solver is shown in Table 3.

In addition, using several convergence criteria, different accuracies of iterative linear solver are tested for DSR in reverse-mode to determine when the precision of gradients, $\dot{J}$, computed with DSR matches that of C-DSR. The settings and the results are shown in Table 5.

The gradient computation comparison in Table 4 demonstrates the validity and significance of the correction for a practical application using Krylov solvers. Table 5 shows that tightening the convergence level improves the accuracy of gradients with DSR, but C-DSR still achieves a higher accuracy at a much smaller computational effort.

5 Summary and conclusions

The correct treatment of iterative linear solvers in forward and reverse-mode AD has been studied. The most commonly used previous method to differentiate linear solvers is based on the assumption that linear systems are fully converged, which in practice is often not the case. The analysis presented in our paper identifies the exact source of errors arising from incompletely converged linear systems used in inner iterations of the solution of non-linear unsteady or segregated problems. We show how this error is linked to the initial guess provided to the linear solver, and how the error accumulates to severely affect adjoint gradients of non-linear solvers. This is also demonstrated in two test cases.

The C-DSR correction term proposed in this paper is shown in our work to be exact for relaxation-type solvers such as Jacobi iterations and other iterative linear solvers. A test case with Jacobi solvers demonstrates the validity of the approach. The C-DSR correction is then applied to a test case from Computational Fluid Dynamics which uses Krylov type solvers for the inner systems. Comparing to DSR, the proposed correction shows significant improvement in the gradient accuracy with much smaller computational cost.

Because the correction formula consists of only a single matrix-vector product and a vector subtraction, the computational cost of computing the correction is small, which makes our method affordable and beneficial for widespread practical application.

Acknowledgements This work is conducted within the About Flow project, which has received funding from the European Unions Seventh Framework Programme for research, technological development and demonstration under Grant Agreement No. 317006. http://aboutflow.sems.qmul.ac.uk

References

1. Bischof, C., Khademi, P., Mauer, A., Carle, A.: ADIFOR 2.0: automatic differentiation of Fortran 77 programs. IEEE Comput. Sci. Eng. **3**(3), 18–32 (1996). https://doi.org/10.1109/99.537089
2. Bischof, C.H., Bücker, H., Lang, B., Rasch, A., Vehreschild, A.: Combining source transformation and operator overloading techniques to compute derivatives for Matlab programs. In: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation, Montreal, QC, Canada, pp. 65–72. IEEE (2002). https://doi.org/10.1109/SCAM.2002.1134106
3. Capriotti, L., Giles, M.B.: Fast correlation Greeks by adjoint algorithmic differentiation. arXiv.org, Quantitative Finance Papers (2010). https://doi.org/10.2139/ssrn.1587822
4. Christianson, B.: Reverse accumulation and attractive fixed points. Optim. Methods Softw. **3**(4), 311–326 (1994). https://doi.org/10.1080/10556789408805572
5. Courty, F., Dervieux, A., Koobus, B., Hascoët, L.: Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation. Optim. Methods Softw. **18**(5), 615–627 (2003). https://doi.org/10.1080/10556780310001610501
6. Davies, A.J., Christianson, D.B., Dixon, L.C.W., Roy, R., van der Zee, P.: Reverse differentiation and the inverse diffusion problem. Adv. Eng. Softw. **28**(4), 217–221 (1997). https://doi.org/10.1016/S0965-9978(97)00005-7
7. Ferziger, J., Perić, M.: Computational Methods for Fluid Dynamics. Springer, New York (2002)
8. Giering, R., Kaminski, T., Slawig, T.: Generating efficient derivative code with TAF: adjoint and tangent linear Euler flow around an airfoil. Fut. Gen. Comput. Syst. **21**(8), 1345–1355 (2005). https://doi.org/10.1016/j.future.2004.11.003
9. Giles, M.B.: Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In: Bischof, C., Bücker, H., Hovland, P., Naumann, U., Utke, J. (eds.) Advances in Automatic Differentiation. Lecture Notes in Computational Science and Engineering, vol. 64, pp. 35–44. Springer, Berlin (2008)
10. Giles, M.B., Duta, M.C., Müller, J.D., Pierce, N.A.: Algorithm developments for discrete adjoint methods. AIAA J. **41**(2), 198–205 (2003). https://doi.org/10.2514/2.1961
11. Giles, M.B., Pierce, N.A.: An introduction to the adjoint approach to design. Flow Turbul. Combust. **65**(3), 393–415 (2000). https://doi.org/10.1023/A:1011430410075
12. Giles, M.B., Süli, E.: Adjoint methods for PDEs: a posteriori error analysis and postprocessing by duality. Acta Numer. **11**, 145–236 (2002). https://doi.org/10.1017/S096249290200003X
13. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, 2nd edn. SIAM, Philadelphia (2008)
14. Griewank, A., Juedes, D., Utke, J.: Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. ACM Trans. Math. Softw. **22**(2), 131–167 (1996). https://doi.org/10.1145/229473.229474

15. Hascoët, L., Pascual, V.: The Tapenade automatic differentiation tool: principles, model, and specification. ACM Trans. Math. Softw. (2013). https://doi.org/10.1145/2450153.2450158
16. Heimbach, P., Bugnion, V.: Greenland ice-sheet volume sensitivity to basal, surface and initial conditions derived from an adjoint model. Ann. Glaciol. **50**(52), 67–80 (2009). https://doi.org/10.3189/172756409789624256
17. Jones, D., Müller, J.D., Christakopoulos, F.: Preparation and assembly of discrete adjoint CFD codes. Comput. Fluids **46**(1), 282–286 (2011). https://doi.org/10.1016/j.compfluid.2011.01.042
18. Lotz, J., Leppkes, K., Naumann, U.: DCO/C++ : Derivative Code by Overloading in C++, Introduction and Summary of Features. Tech. rep., Department of Computer Science, RWTH Aachen University, Aachen, Germany (2016). Report No.: AIB-2016-08
19. Mavriplis, D.J.: Discrete adjoint-based approach for optimization problems on three-dimensional unstructured meshes. AIAA J. **45**(4), 740 (2007). https://doi.org/10.2514/1.22743
20. Meyer, C.D.: Matrix Analysis and Applied Linear Algebra, chap. 5. SIAM, Philadelphia (2000)
21. Moré, J.J., Wild, S.M.: Do you trust derivatives or differences? Comput. Phys. **273**, 268–277 (2014). https://doi.org/10.1016/j.jcp.2014.04.056
22. Müller, J.: Essentials of Computational Fluid Dynamics. CRC Press, Boca Raton (2015)
23. Naumann, U.: The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation. SIAM, Philadelphia (2012)
24. Nielsen, E.J., Diskin, B., Yamaleev, N.K.: Discrete adjoint-based design optimization of unsteady turbulent flows on dynamic unstructured grids. AIAA J. **48**(6), 1195 (2010). https://doi.org/10.2514/1.J050035
25. Patankar, S.V., Spalding, D.B.: A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. Heat Mass Transf. **15**(10), 1787–1806 (1972). https://doi.org/10.1016/B978-0-08-030937-8.50013-1
26. Queen Mary University of London: AboutFlow, an EU-Funded Project: Adjoint-based Optimisation of Industrial and Unsteady Flows. https://aboutflow.sems.qmul.ac.uk/. Accessed 30 Oct 2019
27. Rausch, R.D., Batina, J.T., Yang, H.T.: Three-dimensional time-marching aeroelastic analyses using an unstructured-grid Euler method. AIAA J. **31**(9), 1626–1633 (1993). https://doi.org/10.2514/3.11824
28. Saad, Y.: Sparskit: A Basic Tool Kit for Sparse Matrix Computations—Version 2 User Manual (1994)
29. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia (2003)
30. Sagebaum, M., Albring, T., Gauger, N.R.: High-performance derivative computations using CoDiPack. CoRR **abs/1709.07229** (2017). arXiv:1709.07229
31. Taftaf, A.: Extensions of Algorithmic Differentiation by Source Transformation Inspired by Modern Scientific Computing. Ph.D. thesis, General Mathematics, Université Côte d'Azur, France (2017)
32. Towara, M., Naumann, U.: Simple adjoint message passing. Optim. Methods Softw. **33**(4–6), 1232–1249 (2018). https://doi.org/10.1080/10556788.2018.1435653
33. Venditti, D.A., Darmofal, D.L.: Adjoint error estimation and grid adaptation for functional outputs: application to quasi-one-dimensional flow. Comput. Phys. **164**(1), 204–227 (2000). https://doi.org/10.1006/jcph.2000.6600
34. Xu, S., Jahn, W., Müller, J.D.: CAD-based shape optimisation with CFD using a discrete adjoint. Numer. Methods Fluids **74**(3), 153–68 (2013). https://doi.org/10.1002/fld.3844