Progress in Aerospace Sciences 110 (2019) 100542

Contents lists available at ScienceDirect

Progress in Aerospace Sciences

journal homepage: www.elsevier.com/locate/paerosci

Effective adjoint approaches for computational fluid dynamics

Gaetan K.W. Kenway, Charles A. Mader, Ping He, Joaquim R.R.A. Martins*

University of Michigan, Department of Aerospace Engineering, USA

ARTICLE INFO

Keywords:
Adjoint methods
Computational fluid dynamics
Sensitivity analysis
Algorithmic differentiation
Discrete adjoint benchmarks
Aerodynamic shape optimization

ABSTRACT

The adjoint method is used for high-fidelity aerodynamic shape optimization and is an efficient approach for computing the derivatives of a function of interest with respect to a large number of design variables. Over the past few decades, various approaches have been used to implement the adjoint method in computational fluid dynamics solvers. However, further advances in the field are hindered by the lack of performance assessments that compare the various adjoint implementations. Therefore, we propose open benchmarks and report a comprehensive evaluation of the various approaches to adjoint implementation. We also make recommendations on effective approaches, that is, approaches that are efficient, accurate, and have a low implementation cost. We focus on the discrete adjoint method and describe adjoint implementations for two computational fluid dynamics solvers by using various methods for computing the partial derivatives in the adjoint equations and for solving those equations. Both source code transformation and operator-overloading algorithmic differentiation tools are used to compute the partial derivatives, along with finite differencing. We also examine the use of explicit Jacobian and Jacobian-free solution methods. We quantitatively evaluate the speed, scalability, memory usage, and accuracy of the various implementations by running cases that cover a wide range of Mach numbers, Reynolds numbers, mesh topologies, mesh sizes, and number of CPU cores. We conclude that the Jacobian-free method using source code transformation algorithmic differentiation to compute the partial derivatives is the best option because it computes exact derivatives with the lowest CPU time and the lowest memory requirements, and it also scales well up to 10 million cells and over one thousand CPU cores. The superior performance of this approach is primarily due to its Jacobian-free adjoint strategy. The cases presented herein are publicly available and represent platform-independent benchmarks for comparing other current and future adjoint implementations. Our results and discussion provide a guide for discrete adjoint implementations, not only for computational fluid dynamics but also for a wide range of other partial differential equation solvers.

1. Introduction

The formulation and solving of partial differential equations (PDEs) has led to tremendous progress in the numerical simulation of engineering systems. In most cases, these simulations are eventually used to aid the design of the corresponding engineering systems, which often requires decisions involving large numbers of design variables. Changing these variables manually by trial and error is costly and is not likely to achieve the best possible design. Numerical optimization can facilitate design exploration by automatically finding the set of design variables that minimizes a given objective function subject to the design constraints. Currently, gradient-based optimization algorithms are the only viable way to handle optimization problems with large numbers of variables [1,2]. In addition to being efficient, they allow designers to rigorously search for the optimal design.

To take full advantage of gradient-based optimization algorithms, it is desirable to compute the derivatives of the objective and constraint functions with respect to the design variables in an accurate and efficient way. Various options are available for computing derivatives, which we review in more detail in Sec. 4.2. Most gradient-based optimization software automatically estimates the required gradients by using finite differences. However, finite-difference approximations are subject to numerical errors and are inefficient because their cost is proportional to the number of design variables. Instead, users are recommended to provide their own derivative estimates so that they can use more efficient methods or at least verify the derivatives and control the errors.

Adjoint methods compute accurate derivatives at a cost that is independent of the number of variables. Instead, the computational cost of adjoint methods scales with the number of functions of interest that need to be differentiated. Since many design optimization problems involve large numbers of design variables with only a few objectives

* Corresponding author.
E-mail address: jrram@umich.edu (J.R.R.A. Martins).

https://doi.org/10.1016/j.paerosci.2019.05.002
Received 24 January 2019; Received in revised form 18 May 2019; Accepted 23 May 2019
Available online 06 September 2019
0376-0421/

Nomenclature

C_D	Drag coefficient
C_L	Lift coefficient
f	Function of interest (objective or constraints)
M	Mach number
$\boldsymbol{R}$	Vector of flow residuals
Re	Reynolds number
$\boldsymbol{w}$	Vector of state variables
$\boldsymbol{x}$	Vector of design variables
c	Chord length of a wing section
b	Wing span
γ	Wing twist angle

and constraints, the adjoint method and gradient-based optimization form a powerful combination.

One of the prime applications for the adjoint method has been aerodynamic shape optimization based on computational fluid dynamics (CFD). The large number of variables in these problems stems from the detailed parametrization of three-dimensional (3D) shapes that are required to optimize the design [3]. A typical problem is to minimize the drag of a wing or aircraft configuration subject to lift and moment constraints [3–6].

Two fundamental approaches exist to formulate the adjoint for a set of PDEs: the continuous approach and the discrete approach [7,8]. The continuous approach consists in differentiating the PDEs analytically to get the continuous adjoint equations, which are then discretized so that they can be solved numerically. In contrast, the discrete approach starts with the discretized form of the PDEs, which is then linearized to obtain the discrete adjoint equations. For reasons that we explain in the next section, we adopt exclusively the discrete approach.

Given the power of the adjoint method, its lack of widespread use in the design optimization of engineering systems governed by PDEs is curious. One of the main reasons is that the development of adjoint methods requires a large effort and specialized knowledge [7,8]. Initially, discrete adjoint implementations required hand differentiation of discretized PDEs [9–16], which is time-consuming and error-prone [7].

One way to address this large development effort is to use *algorithmic differentiation* (AD) [17] (also known as *automatic differentiation*), which consists in using software that automatically differentiates source code to produce code that computes derivatives. One way to perform this differentiation, called the *reverse mode*, is analogous to the adjoint method in that the cost of computing derivatives is independent of the number of design variables [18]. However, the use of AD to differentiate the entire CFD code (full-code AD) [19–22] usually results in inefficiencies and, in the worst case, the computational and memory cost is prohibitive [8].

To improve efficiency, researchers have used AD to selectively differentiate the relevant parts of a CFD code. For example, Christianson [23] reported a reverse accumulation adjoint method where the total derivatives are iteratively incremented. Griewank and Faure [24] proposed a piggyback iterative method and Giles et al. [25] developed an iterative primal-dual accurate approach to solve the adjoint equations. Xu et al. [26] reported a Jacobian-trained Krylov-implicit Runge–Kutta method to stabilize the flow and adjoint solutions and implemented it in an open source solver STAMPS [27]. Albring et al. [28] developed a duality-preserving adjoint method that perfectly inherits the convergence rate of the primal solver and implemented it in an open source solver SU2 [29]. Mader et al. [30] used AD to selectively differentiate the code that computes residuals and objective functions and solved the adjoint equations by using the generalized minimal residual (GMRES) method. In that work, we achieved fully consistent total derivatives verified against the complex-step method [31] to be accurate to machine precision. We have since improved and generalized the implementation in the open-source adjoint solver ADflow,[1] which we detail in Sec. 4 and Appendix A. The idea of using AD to selectively differentiate parts of the code has been used in other discrete adjoint implementations [32–34].

Although numerous other efforts have reported adjoint methods for CFD [10,11,13,14,16,35–38], they focus on a single adjoint implementation for a single flow solver. A shortage exists of detailed *quantitative* comparisons of the accuracy and efficiency of various derivative computation and adjoint equation solution options that use well-defined benchmark problems. Quantifying the efficiency and other performance metrics is important because it affects the capability to optimize aerodynamic design in practice. Although existing adjoint studies evaluate their performance, they use different geometries, meshes, flow conditions, and computing platforms. A need also exists for a universal, platform-independent adjoint benchmark that allows us to compare various adjoint implementations.

Our main goal herein is thus to seek *effective* approaches for adjoint development; that is, approaches that are not only efficient and accurate but also incur low implementation and maintenance effort. We achieve this goal by first discussing the pros and cons of the partial derivative computation and adjoint equation solution options. We then implement three combinations of these adjoint options in ADflow and in an incompressible solver (DAFoam) and quantitatively evaluate their performance in terms of six factors: speed, accuracy, memory usage, scalability, development effort, and extensibility. We also use an adjoint solver that implements a full-code AD approach for comparison.

To facilitate future performance comparisons, we propose a suite of platform-independent adjoint benchmarks based on three cases from the AIAA Aircraft Design Optimization Discussion Group (ADODG): the low-speed rectangular wing (Case 3), the transonic Common Research Model (CRM) wing (Case 4), and the wing-body-tail aircraft configuration (Case 5). The meshes and configurations for the benchmarks are publicly available [39], which allows for direct comparisons with other solvers and adjoint implementations.

In this work, we restrict our focus to discrete adjoint implementations for the steady-state Navier–Stokes (NS) equations. However, much of the discussion and ideas can be extended to other governing equations and discretization methods, such as the finite-element structural analysis.

This paper can be read in different ways, depending on the level and goals of the reader. For readers wishing to get a quick overview of the general recipe for adjoint implementation and the pros and cons of the different adjoint implementation options, reading Sec. 4 (especially Secs. 4.5 and 4.6), and the conclusions (Sec. 6) will suffice. Readers interested in the solver-specific adjoint implementation details can refer to Appendix A. The remainder of this paper provides a historical perspective on adjoint methods (Sec. 2), a detailed description of the two flow solvers (Sec. 3), and a quantitative comparison of the adjoint performance for various implementation options (Sec. 5).

2. Historical perspective

Adjoint methods originated in the optimal control community in the 1960s [40,41]. The structural design community later adopted adjoint methods to help solve structural optimization problems [42,43]. Pironneau [44] first introduced adjoint methods in fluid mechanics, where he handled drag minimization problems for Stokes flow. This approach was then extended by Jameson [45] to the aerodynamic optimization of

[1] https://github.com/mdolab/adflow.

Euler flow in the late 1980s.

Since then, the adjoint method has been implemented in more complex PDE models, such as the NS equations [10,46–48], the lattice Boltzmann equation [49,50,52], NS with various turbulence models [53–57], and NS with transitional turbulence models [57–59]. The adjoint method has also been implemented for time-spectral unsteady solvers [60–64] and RANS-based stability derivative computation [65–67]. The other aspect that evolved was the number of flow conditions—initially, optimization considering a single flow condition was the norm [10,45,68], but the need to consider multiple flow conditions to obtain robust designs quickly became evident [69–75].

In applications, the adjoint method has enabled the aerodynamic shape optimization of wings [3–5, 16, 74], aircraft configurations [6,68,76,77], wind turbines [78,79], gas turbines [26,62,80,81], hydrofoils [82], ships [83,84], cars [85–87], and duct flows [88–90]. The adjoint method has also been used in topology optimization for flows [33,91–93] and structures [94,95], as well as for error estimation [96–98], uncertainty quantification [96,99,100], and inverse designs [48,101,102].

In addition to single-discipline PDEs, the adjoint method has been generalized to multidisciplinary systems, which started with the development of a coupled-adjoint approach for an aerostructural model that coupled CFD to finite-element structural analysis [103] and its application to the design of a supersonic business jet [104], where aerodynamic and structural sizing were optimized simultaneously. In the last several years, further developments greatly improved the efficiency of this approach [105] and enabled the aerostructural optimization of aircraft wings [106–108]. In addition to aerostructural optimization, the coupled-adjoint approach has been used in aerothermal problems [90,109–111], aeroacoustic designs [112,113], and flutter-constrained optimization [64,114–117].

The coupled adjoint has since been further generalized. In an effort to unify the theory for the various methods for computing derivatives, Martins and Hwang [18] derived the *unified derivatives equation*, which also applies to coupled systems. This theoretical development resulted in the development of an algorithmic framework for the solution and derivative computation of coupled systems called modular analysis and unified derivatives [118], which has since been implemented in the OpenMDAO framework [119]. This greatly facilitates the implementation of adjoint methods for coupled systems and enabled the optimization in multidisciplinary design optimization applications [120–125]. These included an propulsion-airframe integration effort that coupled CFD with an engine cycle model to study boundary-layer ingestion effects [126] and the simultaneous design of aerodynamic shape and propulsor sizing for boundary-layer ingestion configurations [127].

As mentioned in the introduction, two fundamentally different approaches are available for formulating the adjoint of a PDE solver: the continuous and the discrete approaches. The early work on adjoint methods used the continuous approach. This includes the work of Jameson et al. [46], Anderson and Venkatakrishnan [128], and the initial adjoint implementations in SU2 [29] and OpenFOAM [91]. The advantage of the continuous adjoint approach is that it generates a linearized PDE that is typically solved by using the same numerical iteration technique as the primal flow solution. This eliminates the need to explicitly form the Jacobian and leads to fast and low-memory adjoint implementations [7,8].

The main disadvantages of the continuous approach are a low accuracy for coarser meshes and a challenging implementation. The fact that the PDEs are first linearized and then discretized means that the discretized form of these equations is guaranteed to result in a fully consistent gradient only at the limit of an infinitely fine mesh [129]. Therefore, the continuous adjoint produces inaccurate gradients for cases where the mesh or the numerical methods have an effect on the solution accuracy [130].

Additionally, the continuous adjoint system requires hand differentiation of the PDEs, which is challenging for some terms. For instance, for the turbulent RANS equations, adjoint developers typically simplify some of the terms in the turbulence model, which creates an additional source of error [8]. From an implementation perspective, the fact that the differentiation of the primal PDE is done by hand also restricts the implementation of this method to a few research groups with specialized knowledge of both optimization and CFD. In addition, the boundary conditions of the linearized PDEs are not always obvious and require careful consideration. For example, the adjoint boundary condition for the shear stress term in viscous flow is ill-posed, which eventually impedes us from evaluating the impact of viscous drag [8,128].

Much of the recent work on adjoints has focused on the discrete adjoint method. The advantage of this approach is that, because it is formulated from the discretized flow equations, it provides numerically consistent derivatives regardless of the coarseness of the grid or the assumptions made in the analysis of the primal flow [7,8]. Another advantage is that the partial derivatives that appear in the discretized adjoint equations do not necessarily need to be differentiated by hand and can be computed by using several numerical methods (finite differences, complex step, and AD) [131]. These techniques make it possible to generate numerically exact derivatives, even through complex routines such as the various turbulence models used in RANS analysis [7,55]. The main disadvantage of the discrete approach is the very high cost, in both computational and memory terms, of computing and storing the exact Jacobian, which is necessary in at least some form. Therefore, efficiently implementing these computations is vital to ensure a good overall adjoint performance. The discrete approach is the natural choice because our final goal is to provide a recipe for adjoint implementation and to partially automate the implementation of adjoint methods.

The aforementioned adjoint studies focus on steady- or periodic-flow problems. Developing an efficient adjoint method to handle general nonperiodic unsteady flow is challenging because an unsteady adjoint implementation requires storing and recomputing linearizations at each time step instead of using information from the final converged states. As a result, the computational cost for the unsteady adjoint is at least one order of magnitude greater than that for the corresponding steady adjoint. Moreover, the chaotic nature of turbulent flow poses the fundamental challenge that a small perturbation in the initial conditions may be amplified exponentially at some finite steps and eventually degrade the adjoint derivatives [132]. This problem is especially severe when using high-fidelity, eddy-resolving turbulence-modeling techniques, such as large-eddy simulation and direct numerical simulation. Thus, despite recent efforts [133,134], the development of efficient adjoint methods to handle chaotic turbulent flows for practical engineering design problems remains largely unsolved.

3. Flow modeling and solution

This section introduces the 3D NS equations and describes the two CFD solvers, ADflow and DAFoam, used in this work.

3.1. Governing equations

In the derivations below, we focus on the 3D, steady-state turbulent flow equations, commonly known as the Navier-Stokes (NS) equations:

$$\nabla\cdot(\rho\boldsymbol{U}) = 0, \tag{1}$$

$$\nabla\cdot(\rho\boldsymbol{U}\boldsymbol{U}) + \nabla p - \nabla\cdot\boldsymbol{\tau} = 0, \tag{2}$$

$$\nabla\cdot(\rho E\boldsymbol{U}) + \nabla\cdot(p\boldsymbol{U}) - \nabla\cdot(\boldsymbol{\tau}\cdot\boldsymbol{U}) + \nabla\cdot\boldsymbol{q} = 0, \tag{3}$$

where ρ is the density, p is the pressure, E is the total energy, and $\boldsymbol{U} = [u, v, w]$ is the velocity vector, with u, v, and w being the velocities in the x, y, and z directions, respectively. The quantity $\boldsymbol{q}$ is the heat flux

and $\boldsymbol{\tau}$ is the viscous stress tensor, which is a function of $\boldsymbol{U}$, molecular viscosity μ, and turbulent viscosity μ_t. We ignore all body forces and internal heat sources.

To connect the turbulent viscosity to the mean flow variables, we use the Spalart–Allmaras (SA) turbulence model:

$$\nabla\cdot(\boldsymbol{U}\tilde{\nu}) - \frac{1}{\sigma}\{\nabla\cdot[(\nu+\tilde{\nu})\nabla\tilde{\nu}] + C_{b2}|\nabla\tilde{\nu}|^2\} - C_{b1}\tilde{S}\tilde{\nu} + C_{w1}f_w\left(\frac{\tilde{\nu}}{d}\right)^2 = 0, \tag{4}$$

where $\tilde{\nu}$ is related to the turbulent viscosity $\nu_t = \mu_t/\rho$ via

$$\nu_t = \tilde{\nu}\frac{\chi^3}{\chi^3 + C_{v1}^3}, \chi = \frac{\tilde{\nu}}{\nu}. \tag{5}$$

The four terms in Eq. (4) represent the convective, diffusion, production, and near-wall destruction for the turbulent flow, respectively. Spalart and Allmaras [135] defines these terms and their parameters in more detail.

To discretize the above governing equations, we divide a bounded domain into a mesh of finite-volume cells (finite-volume method). The specific numerical formulation depends on the discretization schemes, flow solution techniques, and mesh topology. In this work, we focus on two flow solvers, ADflow and DAFoam. A brief summary of their implementations is included below.

3.2. ADflow: a structured overset compressible solver

ADflow is a finite-volume structured multiblock and overset mesh solver that is available under an open-source license.[2] The core code is based on SUmb [136], but the code has been completely refactored with a discrete adjoint, overset mesh capability, and Newton-type solvers. It also features a Python API that operates as a Python module for multidisciplinary analysis and design optimization. The code is written in Fortran 90 to maximize computational efficiency. In keeping with the desire for computational efficiency, the initial implementation of the code was based on structured multiblock meshes. Although this type of implementation allows for very efficient computations, there are well-known challenges to making multiblock meshes for complex geometric configurations. As a result, the code was recently extended to handle overset meshes by using an implicit hole cutting technique developed by Kenway et al. [137].

In ADflow, the inviscid fluxes are discretized by using three different numerical schemes: the scalar Jameson–Schmidt–Turkel [138] (JST) artificial dissipation scheme, a matrix dissipation scheme based on the work of Turkel and Vatsa [139], and a monotone upstream-centered scheme for conservation laws (MUSCL) based on the work of van Leer [140] and Roe [141]. The viscous flux gradients are calculated by using the Green–Gauss approach. ADflow solves the compressible flow equations, so the mean flow equations—defined by the continuity of mass, momentum and energy—are solved simultaneously. For turbulent RANS solutions, a turbulence model is used to close the equations. ADflow has several turbulence models implemented, including the one-equation SA model [135] and the two-equation Menter's shear stress transport (SST) model [142]. These turbulence models can be solved either as a segregated system of equations or as a fully coupled system of equations with the mean flow.

Four numerical algorithms are implemented to converge the residual equations including a Runge–Kutta (RK) algorithm, a diagonalized-diagonally-dominant alternating direction implicit (D3ADI) algorithm [143], an approximate Newton–Krylov (ANK) algorithm [144], and a full Newton–Krylov (NK) algorithm [35]. The RK and D3ADI schemes are multigrid compatible, so they can be used with any of the meshes compatible with ADflow. However, although they both act as useful globalization methods for the NK scheme, without a significant number of multigrid levels, these methods converge relatively slowly and so are mostly useful for non-overset meshes. The ANK solver is suitable for both multiblock and overset meshes, although it is not compatible with the multigrid scheme in ADflow. It is sufficiently robust to act as a globalization scheme for the full NK solver and converges well even without multigrid. Therefore, this has become the default solver for overset cases in ADflow. The robustness of the ANK solver has enabled the solution of challenging problems, such as the optimization of an airfoil starting from a circle [145]. The NK solver provides efficient terminal convergence for all types of meshes once the solution is within the Newton basin of attraction.

3.3. DAFoam: an unstructured incompressible solver

DAFoam uses the built-in simpleFoam solver in OpenFOAM to solve the incompressible flow equations. This form of the equations assumes that the density is constant, allowing the energy equation to be ignored for isothermal problems. simpleFoam is a finite-volume solver for both structured and unstructured meshes. It uses the semi-implicit method for pressure-linked equations (SIMPLE) [146] algorithm along with the Rhie–Chow interpolation [147] to solve the flow equations. simpleFoam solves the NS equations in a segregated manner; it first solves the momentum equations based on an old pressure field, then updates the pressure field by solving a Poisson equation. The inviscid terms are discretized by using a second-order upwind scheme, whereas the viscous terms are discretized by using central differencing. simpleFoam solves the pressure equation by using the Gauss–Seidel iteration with the algebraic multigrid technique, whereas the velocity and turbulence equations are solved by using the regular Gauss–Seidel method. DAFoam is written in C++ and is available under an open-source license[3].

4. Adjoint implementation approaches

The final goal of an adjoint solver is to compute the total derivative ($\mathrm{d}f/\mathrm{d}\boldsymbol{x}$), where f is a function of interest and $\boldsymbol{x}$ is the vector of design variables. As previously mentioned, the main advantage of the adjoint method is that it efficiently computes the derivative vector $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$ for any length of the vector $\boldsymbol{x}$. In aerodynamic design optimization, the functions of interest are the objective (e.g., drag coefficient) and constraint functions (e.g., lift coefficient, pitching moment coefficient). The design variables can be divided into aerodynamic design variables (e.g., angle of attack) and geometric variables (e.g., surface shape).

In this section, we describe the various possible approaches to implement the steady-state discrete adjoint method. We start by deriving the discrete adjoint equations (Sec. 4.1) and then describe the options for computing the derivatives in the adjoint equations (Sec. 4.2), along with techniques that accelerate the derivative computation (Sec. 4.3). Section 4.4 discusses the method we selected to efficiently solve the adjoint equations, and Sec. 4.5 reviews existing implementations of discrete adjoint methods in CFD solvers. Finally, we outline our proposed adjoint implementations by using three adjoint approaches that use combinations of the options given above for computing derivatives and solving adjoint equations (Sec. 4.6). The material in this section is meant to be general and thus applicable to any CFD solver. The solver-specific adjoint implementations for our two CFD solvers (ADflow and DAFoam) are detailed in Appendix A.

4.1. Adjoint equations

The discrete adjoint method assumes that a discretized form of the NS (1) and (2), energy (3), and turbulence (4) equations is available through the flow solver and that, for a given design variable vector $\boldsymbol{x} \in \mathbb{R}^{n_x}$, the discretized equations are solved for the state variable

[2] https://github.com/mdolab/adflow.

[3] https://github.com/mdolab/dafoam.

vector $\boldsymbol{w} \in \mathbb{R}^{n_w}$ such that

$$\boldsymbol{R}(\boldsymbol{x}, \boldsymbol{w}) = 0, \tag{6}$$

where $\boldsymbol{R} \in \mathbb{R}^{n_w}$ is the residual vector. The $\boldsymbol{R}$ and $\boldsymbol{w}$ vectors contain the residuals and state variables for all mesh cells, respectively. Therefore, Eq. (6) contains nonlinear equations that involve millions of state variables and require specialized iterative solvers. To ensure fully consistent adjoint derivatives, $\boldsymbol{R}$ and $\boldsymbol{w}$ contain all the residuals and state variables (including the turbulent variables); we do not assume frozen turbulence, as is commonly done in continuous adjoint implementations. We elaborate on the choice of $\boldsymbol{R}$, $\boldsymbol{w}$, and $\boldsymbol{x}$ in Appendix A.1.

The functions of interest are then functions of both the design variables and the state variables, i.e.,

$$f = f(\boldsymbol{x}, \boldsymbol{w}). \tag{7}$$

This computation does not require iteration and is much cheaper than the nonlinear (primal) solution.

In general, there are multiple functions of interest (the objective and multiple design constraints), but in the following derivations, we consider f to be a scalar without loss of generality. As shown later, each additional function requires the solution of another adjoint system.

To obtain the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$, we apply the chain rule:

$$\underbrace{\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}}}_{1\times n_x} = \underbrace{\frac{\partial f}{\partial \boldsymbol{x}}}_{1\times n_x} + \underbrace{\frac{\partial f}{\partial \boldsymbol{w}}}_{1\times n_w} \underbrace{\frac{\mathrm{d}\boldsymbol{w}}{\mathrm{d}\boldsymbol{x}}}_{n_w\times n_x}, \tag{8}$$

where the partial derivatives $\partial f/\partial \boldsymbol{x}$ and $\partial f/\partial \boldsymbol{w}$ are relatively cheap to evaluate because they only involve explicit computations. The total derivative $\mathrm{d}\boldsymbol{w}/\mathrm{d}\boldsymbol{x}$ matrix, on the other hand, is expensive because $\boldsymbol{w}$ is implicitly determined by the residual equations $\boldsymbol{R}(\boldsymbol{w}, \boldsymbol{x}) = 0$.

To solve for $\mathrm{d}\boldsymbol{w}/\mathrm{d}\boldsymbol{x}$, we apply the chain rule for $\boldsymbol{R}$ and use the fact that the total derivative $\mathrm{d}\boldsymbol{R}/\mathrm{d}\boldsymbol{x}$ must be zero for the governing equations to remain feasible with respect to variations in the design variables. The result is

$$\frac{\mathrm{d}\boldsymbol{R}}{\mathrm{d}\boldsymbol{x}} = \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} + \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}} \frac{\mathrm{d}\boldsymbol{w}}{\mathrm{d}\boldsymbol{x}} = 0, \tag{9}$$

which can be rearranged to get the linear system

$$\underbrace{\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}}}_{n_w\times n_w} \underbrace{\frac{\mathrm{d}\boldsymbol{w}}{\mathrm{d}\boldsymbol{x}}}_{n_w\times n_x} = -\underbrace{\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}}_{n_w\times n_x}. \tag{10}$$

We can solve a column of $\mathrm{d}\boldsymbol{w}/\mathrm{d}\boldsymbol{x}$ by using the same column of $\mathrm{d}\boldsymbol{R}/\mathrm{d}\boldsymbol{x}$ as the right-hand side. We then repeatedly solve the linear equation n_x times until all the columns in the $\mathrm{d}\boldsymbol{w}/\mathrm{d}\boldsymbol{x}$ matrix are computed. Finally, we substitute $\mathrm{d}\boldsymbol{w}/\mathrm{d}\boldsymbol{x}$ into Eq. (8) to compute the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$ for any objective function f. This approach is known as the *direct* method, and its computational cost is proportional to the number of design variables.

Alternatively, we substitute the solution for $\mathrm{d}\boldsymbol{w}/\mathrm{d}\boldsymbol{x}$ from Eq. (10) into Eq. (8) to get

$$\underbrace{\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}}}_{1\times n_x} = \underbrace{\frac{\partial f}{\partial \boldsymbol{x}}}_{1\times n_x} - \overbrace{\underbrace{\frac{\partial f}{\partial \boldsymbol{w}}}_{1\times n_w} \underbrace{\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}}^{-1}}_{n_w\times n_w}}^{\boldsymbol{\psi}^T} \underbrace{\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}}_{n_w\times n_x}. \tag{11}$$

Typically, we do not actually find the inverse of the Jacobian $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ explicitly. Instead, we solve the corresponding linear system with the appropriate right-hand-side vector. The direct method consists in solving with $-\partial \boldsymbol{R}/\partial \boldsymbol{x}$ as the right-hand side. Alternatively, we transpose the Jacobian and solve with $[\partial f/\partial \boldsymbol{w}]^T$ as the right-hand side, which yields the *adjoint equations*

$$\underbrace{\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}}^T}_{n_w\times n_w} \underbrace{\boldsymbol{\psi}}_{n_w\times 1} = \underbrace{\frac{\partial f}{\partial \boldsymbol{w}}^T}_{n_w\times 1}, \tag{12}$$

where $\boldsymbol{\psi}$ is the *adjoint vector*. Next, we substitute the adjoint vector into Eq. (11) to compute the total derivative:

$$\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} = \frac{\partial f}{\partial \boldsymbol{x}} - \boldsymbol{\psi}^T \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}. \tag{13}$$

Since the design variable $\boldsymbol{x}$ does not explicitly appear in Eq. (12), we need to solve this equation only once for each function of interest, so the computational cost is independent of the number of design variables. This approach is also known as the *adjoint* method, and its computational cost is proportional to the number of objective functions.

In the context of large-scale engineering optimization problems such as aircraft design, the *adjoint* method is advantageous because the functions of interest typically number fewer than 10, but the design variables can number several hundred.

To summarize, a discrete adjoint consists of the following four major steps: First, we compute the partial derivatives $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$ and $[\partial f/\partial \boldsymbol{w}]^T$. Next, we solve the linear equation (12) to obtain the adjoint vector $\boldsymbol{\psi}$. Finally, we compute the partial derivatives $\partial \boldsymbol{R}/\partial \boldsymbol{x}$ and $\partial f/\partial \boldsymbol{x}$ and use Eq. (13) to compute the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$. Specifically, the steps are as follows:

1. Compute $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$ and $[\partial f/\partial \boldsymbol{w}]^T$.
2. Solve the adjoint equation $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T \boldsymbol{\psi} = [\partial f/\partial \boldsymbol{w}]^T$ to obtain $\boldsymbol{\psi}$.
3. Compute $\partial \boldsymbol{R}/\partial \boldsymbol{x}$ and $\partial f/\partial \boldsymbol{x}$.
4. Compute the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x} = \partial f/\partial \boldsymbol{x} - \boldsymbol{\psi}^T [\partial \boldsymbol{R}/\partial \boldsymbol{x}]$.

These four steps indicate that an effective adjoint implementation requires efficient partial derivative computation, as well as adjoint equation solutions. Multiple options are available for these two tasks. The following sections outline the methods available and highlight their advantages and disadvantages.

4.2. Partial derivative computations

Five primary options are available for computing the derivatives in a discrete adjoint implementation [8,148]: analytic methods, finite-difference methods, the complex-step method [31], AD [17], and symbolic differentiation. More details on these methods, including their unification, can be found elsewhere [18]; here we just discuss what is needed for the computation of the partial derivatives in the adjoint equations.

4.2.1. Analytic method

The analytic method, which consists in hand differentiation, was used in early discrete adjoint implementations [9–16]. However, analytically differentiating the adjoint partial derivative terms requires significant expertise in the particular flow-solver implementation and is prone to human error [7]. Furthermore, the complex nature of the turbulent NS equations is often handled by ignoring some terms (e.g., frozen turbulence [8]). This simplifies the analytic differentiation of the residuals but introduces inaccuracies in the gradients. Although this approach can produce computationally efficient derivatives, it also typically requires a significant amount of development and implementation time. As a result, this approach has largely fallen out of favor for recently developed adjoint solvers.

4.2.2. Finite-difference approximations

Finite-difference methods compute partial derivatives by perturbing the corresponding variables and recomputing the outputs. Consider a general vector function with the input vector $\boldsymbol{x}$ and output vector $\boldsymbol{y} = \boldsymbol{y}(\boldsymbol{x})$. The corresponding Jacobian matrix $\partial \boldsymbol{y}/\partial \boldsymbol{x}$ can be computed by using first-order forward differencing as follows:

$$\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)_{n,m} = \frac{y_n(\boldsymbol{x} + \varepsilon \boldsymbol{e}_m) - y_n(\boldsymbol{x})}{\varepsilon} + O(\varepsilon), \tag{14}$$

where the subscripts n and m are the row and column indices of the

matrix, respectively, ε is the finite-difference step, and $\boldsymbol{e}_m$ is a unit vector with a single unity in row m and zeros in all other rows. By looping over the input index m and applying perturbations for each input in sequence, the entire Jacobian may be populated with $M + 1$ function evaluations, with M being the number of columns. The finite-difference method is much easier to implement than analytic methods, and it allows the Jacobian to be computed even when the underlying implementation of the function computation is not accessible (such as in black-box computations). However, the finite-difference method is subject to truncation and subtractive cancellation errors, making the derivative value sensitive to step size, especially for functions with strong nonlinearities. This requires a careful step-size study to keep the error at an acceptable level.

4.2.3. Complex-step method

The complex-step method computes the derivative of a real function by perturbing it with a complex step instead of a real one and taking the imaginary part of the result, yielding the approximation [31,149]

$$\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)_{n,m} = \frac{\mathrm{Im}[y_n(\boldsymbol{x} + \varepsilon \boldsymbol{e}_m i)]}{\varepsilon} + O(\varepsilon^2), \tag{15}$$

where εi is a pure imaginary step. Since there is no subtraction in the above equation, there is no subtractive cancellation error. Therefore, we can choose an arbitrarily small step size (e.g., 10^{-40}) to reduce the truncation error to machine precision. Given this advantage, the complex-step method is useful for computing the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$ for verification purposes. However, only a handful of adjoint implementations use the complex-step method for partial derivative computation [37]. This is primarily because, to implement the complex-step method, all the floating point data types in the CFD code need to be replaced with complex number data types. This modification usually makes the code between two and four times slower than the equivalent real code, which is partially because we need more arithmetic operations for complex numbers than real ones. Moreover, the operations for the overloaded complex data type may not be fully optimized by compilers. In addition to the slower speed, we need twice as much memory when replacing all real numbers with complex numbers in the code. In addition, we need to carefully complexify certain real-value operations such as the relational logic operators and absolute value function [31].

4.2.4. Algorithmic differentiation

AD is another option for accurately computing derivatives [17]. This approach focuses directly on the computer code and uses the fact that any code, no matter how complex, consists of a sequence of elementary arithmetic operations. These operations can then be differentiated to create differentiated versions of the code that compute derivatives by using the chain rule.

There are two derivative computation modes for AD: forward and reverse. Both use the same chain rule but accumulate the derivatives in different directions. In forward-mode AD, the input variables of interest are specified and the derivatives with respect to those variables are accumulated in the forward direction, together with the execution of the original code. In reverse-mode AD, on the other hand, the outputs of interest are specified and the derivatives of those outputs are accumulated backward. Before this reverse accumulation is performed, however, the original code must be run (forward) and all intermediate variables must be stored for use in the reverse accumulation.

Consider the vector function $\boldsymbol{y} = \boldsymbol{y}(\boldsymbol{x})$. In the forward mode, the input and output for its differentiated function $\dot{\boldsymbol{y}}(\boldsymbol{x}, \dot{\boldsymbol{x}})$ can be expressed as

$$\dot{\boldsymbol{y}} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\dot{\boldsymbol{x}} \Rightarrow \begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \vdots \\ \dot{y}_N \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_M} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_M} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_N}{\partial x_1} & \frac{\partial y_N}{\partial x_2} & \cdots & \frac{\partial y_N}{\partial x_M} \end{bmatrix} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_N \end{bmatrix}, \tag{16}$$

where $\dot{\boldsymbol{x}}$ is the input seed and $\dot{\boldsymbol{y}}$ contains the output of a forward-mode AD computation. Note that Eq. (16) illustrates only the input and output structures of the differentiated $\boldsymbol{y}$ function in the forward mode. The details of how the differentiated function is constructed by using the chain rule can be found in AD textbooks [17,150]. We use the differentiated $\boldsymbol{y}$ function to compute individual partial derivatives by providing an input seed vector such that the n^{th} entry is unity and all other entries are zero:

$$\dot{x} = [\dot{x}_1, \dot{x}_2, ..., \dot{x}_n, ..., \dot{x}_N]^T = [0, 0, ..., 1, ..., 0]^T \tag{17}$$

Equation (16) then yields the partial derivative of all components of $\boldsymbol{y}$ with respect to $\boldsymbol{x_n}$:

$$\dot{y} = \left[\frac{\partial y_1}{\partial x_n}, \frac{\partial y_2}{\partial x_n}, ..., \frac{\partial y_N}{\partial x_n}\right]^T \tag{18}$$

The differentiated function in the reverse-mode $\bar{\boldsymbol{y}}(\boldsymbol{x}, \bar{\boldsymbol{y}})$ has the following input and output structure:

$$\bar{\boldsymbol{x}} = \left[\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right]^T \bar{\boldsymbol{y}} \Rightarrow \begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_N \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_N}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_N}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_M} & \frac{\partial y_2}{\partial x_M} & \cdots & \frac{\partial y_N}{\partial x_M} \end{bmatrix} \begin{bmatrix} \bar{y}_1 \\ \bar{y}_2 \\ \vdots \\ \bar{y}_N \end{bmatrix}. \tag{19}$$

Now $\bar{\boldsymbol{y}}$ is the input seed and $\bar{\boldsymbol{x}}$ contains the output of a reverse-mode AD computation. To find the desired derivatives, we set the input seed vector

$$\bar{y} = [\bar{y}_1, \bar{y}_2, ..., \bar{y}_n, ..., \bar{y}_N]^T = [0, 0, ..., 1, ..., 0]^T \tag{20}$$

The output of reverse-mode AD then contains the derivatives of y_n with respect to all $\boldsymbol{x}$:

$$\bar{x} = \left[\frac{\partial y_n}{\partial x_1}, \frac{\partial y_n}{\partial x_2}, ..., \frac{\partial y_n}{\partial x_M}\right]^T \tag{21}$$

In addition to explicitly computing partial derivatives, we can use AD to efficiently compute matrix-vector products with the Jacobian $\partial \boldsymbol{y}/\partial \boldsymbol{x}$. This can be done by multiplying the Jacobian by a full vector $\boldsymbol{\psi}$ that we set to be the seed vector ($\dot{\boldsymbol{x}} = \boldsymbol{\psi}$ or $\bar{\boldsymbol{y}} = \boldsymbol{\psi}$). Specifically, the forward mode allows the computation of regular matrix-vector products $[\partial \boldsymbol{y}/\partial \boldsymbol{x}]\boldsymbol{\psi}$, whereas the reverse mode allows the computation of transpose matrix-vector products $[\partial \boldsymbol{y}/\partial \boldsymbol{x}]^T\boldsymbol{\psi}$. We use this unique feature of reverse-mode AD to implement an efficient Jacobian-free adjoint solver using Krylov subspace methods, as detailed in Sec. 4.4.

There are two main ways of implementing AD: *source code transformation* and *operator overloading* [150]. Tools using source code transformation add new statements to the original source code that compute the derivatives of the original statements. The operator-overloading approach consists in using a new user-defined type instead of real numbers. This new type includes not only the value of the original variable but also its derivative. All intrinsic operations and functions have to be redefined (i.e., *overloaded*) to compute the derivatives together with the original computations. Both source code transformation and operator overloading differentiate the original functions in a semi-automatic manner. The operator-overloading approach results in fewer changes to the original code but is less efficient because overloading user-defined data types results in code that is two to four times slower, similar to the complex-step approach mentioned in Sec. 4.2.3. Moreover, this approach uses at least twice as much memory as the original code because the overloaded date type stores both the value and derivative, and because the reverse-mode AD requires storing all intermediate variables on tapes

[151,152]. Conversely, the source code transformation creates new codes to compute derivatives on the fly while keeping the same data type. With careful implementation, the source code transformation AD can be as efficient as analytic methods in terms of speed and memory usage [153]. Section 5 compares the performance of source code transformation and operator-overloading AD.

4.2.5. Symbolic differentiation

Symbolic differentiation [154] is another option to compute derivatives. In contrast to AD, symbolic differentiation directly manipulates the mathematical expression of a function. The advantage of symbolic differentiation is that we obtain the mathematical form of derivatives, and the process can be automated by using software such as Maple, Mathematica, or Matlab. However, symbolic differentiation requires a function to be expressed in closed form, which is not always feasible for modern CFD codes that use iterative solution procedures. Moreover, as the function becomes complex, the mathematical expression generated by symbolic differentiation can be prohibitively long and thus computationally inefficient. Given these limitations, symbolic-differentiation-based derivative computation has not been widely used in adjoint development. However, in the finite-element-method (FEM) community, a semi-symbolic-differentiation method was proposed to compute derivatives, which is elaborated on in Sec. 4.5.

4.3. Accelerating Jacobian computation

Being able to compute partial derivatives in both forward and reverse mode allows flexibility when computing partial derivatives. However, the Jacobian required for the adjoint system is square, which limits the benefits of using a particular computational approach for the derivatives. Furthermore, computing the state Jacobian matrix $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ by naively computing all the elements in the matrix requires $O(N^2)$ operations, where N is the number of rows in the Jacobian. This is prohibitive for a 3D problem because N can exceed 100 million. Therefore, it is critical to accelerate the computation of the Jacobian matrix by exploiting its sparsity.

Even if we use the Jacobian-free Krylov methods enabled by reverse-mode AD, we still need a preconditioner matrix, which must also be computed efficiently. We thus want to reduce the number of residual evaluations required to compute these matrices to a small fixed number. We now discuss two approaches to exploit the sparsity pattern and accelerate Jacobian matrix computations.

4.3.1. Single-cell residual

In this approach, we first identify the computational stencil to compute a single-cell residual. Fig. 1 shows examples of stencils for a single-cell residual. Here, the residual cell is black, and the level-one and -two connected cells are red and blue, respectively. We then recode a reduced residual routine that computes the residual for only a single cell. Next, we compute the derivatives of its residual with respect to the states and design variables associated with this cell. This process is repeated for every cell in the mesh until all the nonzero Jacobian elements are computed. The single-cell residual computation is relatively intuitive to implement and is effective in minimizing the number of computations required to form the Jacobian. This approach was used in our original adjoint implementation [30] and also in other recent work [33].

However, one big disadvantage of the single-cell residual approach is that we need to re-code a reduced residual routine because the original residual computation typically has an input vector of all the design variables and state variables [$\boldsymbol{x}$, $\boldsymbol{w}$] and an output vector of all residuals [$\boldsymbol{R}$], instead of an output for a single-cell residual. This code duplication goes against best programming practices and makes it more difficult to maintain and extend the code. In particular, this approach is susceptible to discrepancies between the original residual computation and the single-cell residual computation. Furthermore, for an object-oriented, industrial-scale code such as OpenFOAM, even isolating the functions needed to create a single-cell residual computation is challenging. Therefore, this approach is not widely used in the finite-volume CFD community.

4.3.2. Graph coloring

Similar to the single-cell residual approach, graph coloring [155] exploits the sparsity of Jacobian matrices and requires the identification of the stencil of each residual computation. In this case, the residual stencil is used to define an adjacency matrix that serves to compute a unique set of groupings that covers the entire domain with non-overlapping residual computations. This is done by partitioning all columns of a matrix into different structurally orthogonal subgroups—the *colors*—such that, in one structurally orthogonal subgroup, no two columns have a nonzero entry in a common row. Coloring allows simultaneous perturbation of multiple columns of the Jacobian by using any of the forward-mode methods (i.e., finite difference, complex step, and forward-mode AD) because we know that no residuals in a given color overlap, thereby reducing the quantity of residual evaluations required to a small fixed number n_{color}. The coloring approach also allows us to directly use the original residual computation functions because we are able to account for the sparsity by using the coloring. This avoids code duplication, thereby ensuring consistency between the original and derivative computations.

As an example, we consider a five-point-stencil Jacobian matrix obtained from a 5×5 two-dimensional mesh (see Fig. 2). If we perturb

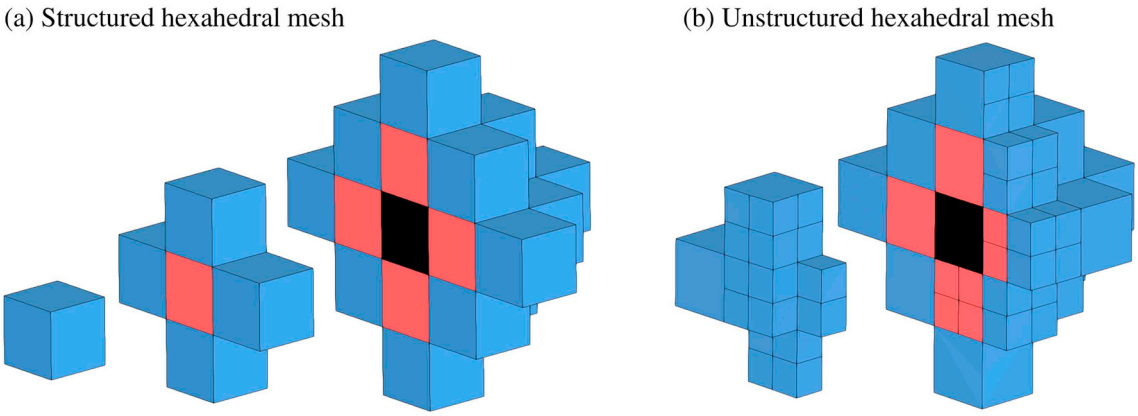

Fig. 1. Examples of stencils for a single-cell residual. The residual cell is black, and the level-one and -two connected cells are red and blue, respectively. The structured hexahedral mesh has a regular stencil pattern, whereas the unstructured hexahedral mesh splits its cell faces and has an irregular stencil. (For interpretation of the references to color in this figure legend, the reader is referred to the Web version of this article.)

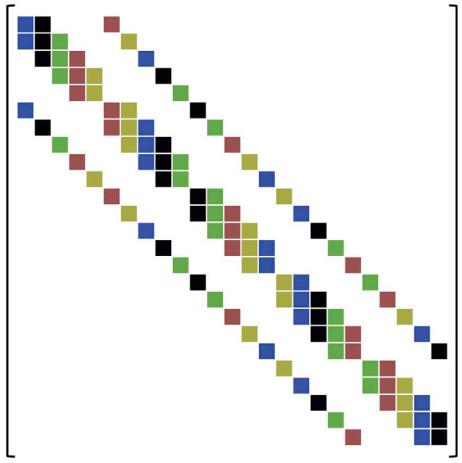

Fig. 2. Example of graph coloring for a five-point-stencil Jacobian matrix obtained from a 5 × 5 two-dimensional mesh. Naively computing the Jacobian by using finite differencing (column by column) requires 26 residual function evaluations. With graph coloring, we simultaneously perturb columns that have the same color, which reduces to six the number of function evaluations. (For interpretation of the references to color in this figure legend, the reader is referred to the Web version of this article.)

each column of the matrix and use the first-order finite-difference method to compute the Jacobian, we need 26 residual function evaluations. With graph coloring, we simultaneously perturb columns that have the same color, which reduces to six the number of function evaluations. The actual Jacobian matrices for the 3D NS equations are much more complex than the five-point-stencil matrix. The cost and complexity of computing a coloring can be high, depending on the mesh topology and the residual stencil of the primal solver. This requires us to selectively adopt effective coloring algorithms for ADflow and DAFoam; we discuss the details of these algorithms in Appendix A.2.

4.4. Methods for solving the adjoint equations

As mentioned in Sec. 4.1, in addition to computing the partial derivatives, we need to solve the adjoint Eqn. (12). A system of linear equations may be solved by either direct or iterative methods [156]. The benefit of using direct methods, such as lower and upper factorization, is that we obtain exact solutions with a fixed number of operations. However, the application of direct methods is relatively limited. To be more specific, large-size linear systems incur prohibitive computational cost for exact lower and upper factorization, especially for linear systems that are not tightly banded [156]. Although direct methods have been used to solve the adjoint equations for problems in computational structural dynamics [95], they have not been widely adopted in CFD. Iterative methods, on the other hand, have no restriction on the size or banded structure of linear systems and is the natural choice to solve the adjoint equations for large-scale CFD problems.

One option to solve the adjoint equations is the linear fixed-point iteration method, also known as the stationary iteration method [156,157]. In this approach, we first reformulate Eq. (12) as a residual function,

$$\boldsymbol{R}_{\text{adj}}(\boldsymbol{\psi}) = \left[\frac{\partial f}{\partial \boldsymbol{w}}\right]^T - \left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}}\right]^T \boldsymbol{\psi} = 0, \tag{22}$$

and then solve for $\boldsymbol{\psi}$ by using

$$\boldsymbol{\psi}^{n+1} = \boldsymbol{\psi}^n + \boldsymbol{M}\boldsymbol{R}_{\text{adj}}(\boldsymbol{\psi}^n), \tag{23}$$

where n is the iteration number, and $\boldsymbol{M}$ is an operator that depends on the particular iterative method (e.g., Euler, Gauss–Seidel, Runge–-Kutta). One advantage of reformulating the adjoint equation in residual form is that we do not need to factorize and solve the exact Jacobian matrix $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$. Another advantage is that we can use the same iterative solver as the flow solver, achieving same rate of convergence for the linear adjoint and the nonlinear flow equations—a feature that is known as *duality preserving* [8,158–160]. Given these two advantages, the fixed-point iteration methods have been used in a number of adjoint implementations [9,12–14,25,27,28,32,34,88,161,162]. However, one drawback of linear fixed-point iteration methods is that the current solution depends on information only from the previous iteration, so information accumulated throughout the iterations is not fully used. Therefore, the convergence of fixed-point iteration methods is *always* linear [156].

Given the pros and cons cited above, we opt to use the GMRES iterative method to solve the linear equations. The GMRES method conducts iterations by using the Krylov subspace,

$$K_n = \text{span}\{\boldsymbol{r}_0, A\boldsymbol{r}_0, A^2\boldsymbol{r}_0, ..., A^{n-1}\boldsymbol{r}_0\}, \tag{24}$$

where A is the transpose of the Jacobian $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$, and the initial residual $\boldsymbol{r}_0$ is

$$\boldsymbol{r}_0 = \left[\frac{\partial f}{\partial \boldsymbol{w}}\right]^T - \left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}}\right]^T \boldsymbol{\psi}_0. \tag{25}$$

Compared with the fixed-point iteration method, the GMRES method uses information gathered throughout the iterations and therefore converges faster [156,157]. Furthermore, by using the NK method to solve the primal flow equations, the GMRES method provides a similar duality preserving behavior as the fixed-point iteration approach because the eigenvalues for Jacobians (primal) and transposed-Jacobians (adjoint) are the same, which ensures consistent convergence between the flow and adjoint solutions [8,15].

In addition to the above advantages, we can directly compute the matrix-vector products to construct the Krylov subspace without forming $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$; a method known as *Jacobian-free* GMRES. The benefit of Jacobian-free GMRES is that it saves the computational time and memory involved in computing and storing $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$.

To perform Jacobian-free GMRES iterations, we use the reverse-mode AD to efficiently compute the matrix-vector products by using Eq. (19), which yields

$$\bar{\boldsymbol{w}} = \left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}}\right]^T \bar{\boldsymbol{R}}. \tag{26}$$

We set the input vector $\bar{\boldsymbol{R}} = \boldsymbol{r}_0$, and the output $\bar{\boldsymbol{w}}$ is the matrix-vector product $\bar{\boldsymbol{w}} = [\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T \boldsymbol{r}_0 = A\boldsymbol{r}_0$. Note that, although we call it the Jacobian-free GMRES method, we still need to explicitly form and store the preconditioner matrix $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$ [see Eq. (27)]. We find that this approach is more efficient because a fully-Jacobian-free method requires extra computation for $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$, which makes the overall adjoint solution slower. We compare the performance of the Jacobian-free adjoint solution method with other approaches in Sec. 5.

In addition to reducing memory usage, another advantage of using the Jacobian-free GMRES method is that the sparsity information for $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$ need not be computed. This important feature allows us to easily extend our adjoint implementation to any computational mesh. We demonstrate this flexibility by applying our adjoint implementation to a wing-body-tail aircraft configuration with overset meshes (Sec. 5).

One disadvantage of GMRES methods is that they solve the exact Jacobian system $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$, which is typically more difficult to converge because the state Jacobian matrix that results from a 3D viscous turbulent flow solution is typically ill-conditioned, especially for realistic geometries with complex flow [6,8,28,163–165]. Therefore, we need a strong preconditioner to improve the eigenvalues clustering. The right-preconditioned adjoint equations reads

$$\left(\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}}^T \left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}}_{PC}^T\right]^{-1}\right)\left(\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}}_{PC}^T \boldsymbol{\psi}\right) = \frac{\partial f}{\partial \boldsymbol{w}}^T, \tag{27}$$

where $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$ is the preconditioner matrix and $([\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC})^{-1}$ is the approximated inverse of $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$. We require that the preconditioner matrix is an approximation of $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$ but easily invertible. Unfortunately, no general-purpose preconditioner matrix is guaranteed to be effective, so we must carefully construct $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$ for each specific problem [156]. Appendix A.3 elaborates on the solver-specific computation of the preconditioner matrix.

In this paper, we solve the adjoint equations by using the Portable, Extensible Toolkit for Scientific Computation (PETSc) software library [166–168]. PETSc provides a wide range of parallel linear and nonlinear equation solvers with various preconditioning options. We select the GMRES as the top-level iterative solver and adopt a nested preconditioning strategy (see Fig. 3). We use the additive Schwartz method (ASM)—with one or two levels of overlap—as the global preconditioner. The ASM divides the linear system into sub-blocks and then solves them in parallel. For the local preconditioner in each sub-block, we use the incomplete lower and upper (ILU) factorization approach. The effectiveness of the ILU approach is largely impacted by the level of fill-in: the use of a higher fill-in level converges faster but increases memory usage. To reduce memory usage, we apply the ILU preconditioning multiple times in each sub-block by using the Richardson method (inner Richardson iteration), where the Richardson damping factor is set to 1.0. Similarly, we perform outer Richardson iterations when applying the ASM global preconditioner. Performing the inner and outer Richardson iterations has the same effect as increasing the ILU fill-in level. Therefore, this strategy allows us to use a relatively low fill-in level (one or two) to save memory. In terms of speed, we find that the improvement in convergence rate outweighs the extra computational cost of Richardson iterations, which is attributed to $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$ being much sparser than the exact state Jacobian $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$, so the cost of extra matrix-vector products in the inner and outer Richardson iterations is much cheaper than that in the GMRES iterations. In addition, we observe that the outer Richardson iteration is more effective than the inner one, which we attribute to the fact that the overlapped ASM preconditioner gathers information across the blocks, whereas the ILU approach only affects local blocks. In this paper, we perform three and two iterations for outer and inner Richardson iterations, respectively. The above nested preconditioning strategy is effective for solving the adjoint equation, as perviously reported [55,105].

4.5. Overview of approaches used in existing discrete adjoint solvers

Existing discrete adjoint implementations for CFD codes use different combinations of the options described above to compute the partial derivatives and solve the adjoint equations (see Table 1).

Several codes have used hand differentiation to compute the partial derivatives (Cart3D, *els*A, FUN3D, Jetstream, NSU3D, and Tau-Code). As mentioned in Sec. 4.2, this approach is potentially the most efficient from the computational point of view. However, it requires expertise in the primal code, and deriving derivative routines by hand requires significant effort [7,37]. Moreover, the hand-derived routines are not straightforward to extend (e.g., adding new turbulence models and discretization schemes).

More recent adjoint implementations (ADflow, DAFoam, HYDRA, piggySimpleFoam, reverseAccSimpleFoam, STAMPS, and SU2) use AD to efficiently compute partial derivatives and matrix-vector products. For solvers written in Fortran (ADflow, HYDRA, and STAMPS), the source code transformation AD is the best choice because it results in faster code and incurs a lower memory overhead compared with operator-overloading AD. The source code transformation tool Tapenade [174] was used to compute the partial derivatives for these three codes. For C++ solvers that use object-oriented code structures (DAFoam, piggySimpleFoam, reverseAccSimpleFoam, and SU2), the operator-overloading AD becomes the only option. The benefit of using the operator-overloading AD is that it is relatively easy to implement and extend. SU2 uses the CoDiPack [152] as the operator-overloading AD tool, whereas DAFoam, piggySimpleFoam, and reverseAccSimpleFoam use dco/c++ [151].

The best strategy for choosing between the Krylov and fixed-point iteration approaches is to use the same iterative approach for solving the flow and adjoint equations, which is because the codes from the flow solvers can be reused to reduce the effort required to implement the adjoint method. More importantly, by carefully formulating the iterative scheme, we achieve the same convergence behavior for the flow solution and adjoint computation, which helps to ensure robust optimization (see Sec. 4.4). The benefit of using the Krylov approach is that it converges faster than the fixed-point iteration, as mentioned in Sec. 4.4.

As mentioned in the introduction, an adjoint implementation is available that uses reverse-mode AD to differentiate the entire CFD code (full-code AD). The cost of full-code AD is independent of the number of design variables and shares this advantage with the discrete adjoint method derived in Sec. 4.1; however, it tends to have a high overall cost both in terms of memory and computation time. A naive implementation of this approach requires all of the intermediate flow variables to be stored in memory in the forward computational pass, which is not feasible for large 3D PDE solvers. Modern algorithmic differentiation techniques, such as checkpointing and local pre-accumulation, can reduce the computational cost and memory usage [17,150]. Yet despite these improvements, the size of problems that can be solved remains limited. An example of current state-of-the-art full-code AD is the adjointSimpleShapeCheckpointingFoam adjoint solver developed by Towara and Naumann [22], which we compare with our adjoint implementations in Sec. 5.

As mentioned in Sec. 4.2.5, there is an adjoint implementation in the FEM community (dolfin-adjoint [175]) that uses a semi-symbolic-differentiation approach. dolfin-adjoint exploits the special code structure of FEniCS [176], which is a finite-element method framework that supports the automated solution of PDEs starting from high-level mathematical expressions. Semi-symbolic-differentiation is similar to full-code AD, except that it stores a symbolic tape in the forward execution, as opposed to floating-point operations in AD. Once finished, dolfin-adjoint analyzes the symbolic tape and derives mathematical expressions for the adjoint, which are then automatically converted to low-level computer codes to compute derivatives. The main advantage of dolfin-adjoint is its high flexibility and automation when creating a new adjoint solver. In addition, the dolfin-adjoint code is well optimized and achieves a better adjoint-to-flow runtime ratio than the full-code AD approach [175]. However, no detailed comparisons are available for the performance of the finite-element dolfin-adjoint solver versus that of the finite-volume adjoint solvers. Moreover, the semi-symbolic-differentiation approach is applicable only within the FEniCS framework; it is not designed for general adjoint solvers.

In addition to the research community, commercial software

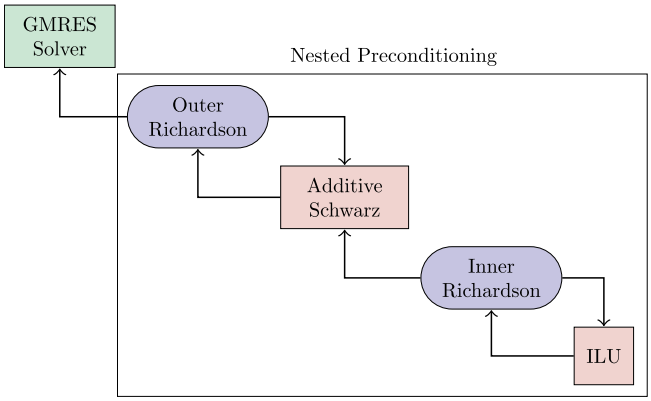

Fig. 3. The GMRES method along with a nested preconditioning strategy for solving the adjoint equations.

Table 1
Overview of implementation options used in existing discrete adjoint solvers.

Solver	Organization	Programming language	Partial derivative computation	Adjoint solution	References
ADflow	Univ. of Michigan	Fortran	Transformation AD	Krylov	This work
Cart3D[a]	NASA	Fortran	Hand derived	Fixed point	[12]
DAFoam	Univ. of Michigan	C++	Overloading AD	Krylov	This work
elsA	ONERA	C++	Hand derived	Fixed point	[169]
FUN3D	NASA	Fortran	Hand derived	Fixed point	[10,161]
HYDRA	Rolls–Royce[b]	Fortran	Transformation AD	Fixed point	[25,170]
Jetstream	Univ. of Toronto	Fortran	Hand derived	Krylov	[16,171,172]
NSU3D	Univ. of Wyoming	Fortran	Hand derived	Krylov	[13,38,173]
SimpleFoam[c]	Aachen Univ.	C++	Overloading AD	Fixed point	[34]
SimpleFoam[d]	Aachen Univ.	C++	Overloading AD	Full-code AD	[22,34]
STAMPS	Queen Mary Univ.	Fortran	Transformation AD	Fixed point	[27]
SU2	Stanford Univ.[e]	C++	Overloading AD	Fixed point	[28,29]
TAU-Code	DLR	C	Hand derived	Krylov	[15]

[a] Inviscid solver.
[b] Initially developed at University of Oxford.
[c] piggySimpleFoam and reverseAccSimpleFoam.
[d] adjointSimpleShapeCheckpointingFoam.
[e] Discrete adjoint developed by TU Kaiserslautern.

companies have implemented the discrete adjoint method for their CFD solvers (e.g., CFD-ACE+ [177] from ESI Group, Fluent [178] from ANSYS, and STAR-CCM+ [179] from Siemens). We do not elaborate on these because only limited information is currently available, but we hope that they will be benchmarked by using the cases presented herein.

To summarize, we have reviewed various options used in existing adjoint solvers. We implement the best combination of these options in ADflow (see the Jacobian-free method in Sec. 4.6 and the appendix). To be more specific, we use a structured mesh solver written in Fortran, which is faster than unstructured solvers and solvers written in C++. In addition, we compute the partial derivatives by using source code transformation AD, which is faster and requires less memory than the operator-overloading AD. Furthermore, we use the Jacobian-free GMRES method to solve the adjoint equations, which converges faster than the fixed-point iteration approach while requiring moderate memory overhead. Finally, we achieve consistent convergence between the flow and adjoint solutions because we use the NK method to solve the primal nonlinear equations.

One limitation of the Jacobian-free adjoint method in ADflow is that it requires greater effort for the initial development using source code transformation AD than does operator-overloading AD. However, once implemented, extending adjoint codes to include new terms and objectives is straightforward (see Appendix A.4). Another drawback is that mesh generation is generally less flexible than for unstructured solvers, although this limitation has been alleviated by the overset mesh capability in ADflow. Overall, the advantage in performance outweighs the above limitations.

Next, we elaborate on the Jacobian-free adjoint implementation along with two variations. Section 5 compares their performance and demonstrates the efficiency of the proposed Jacobian-free method.

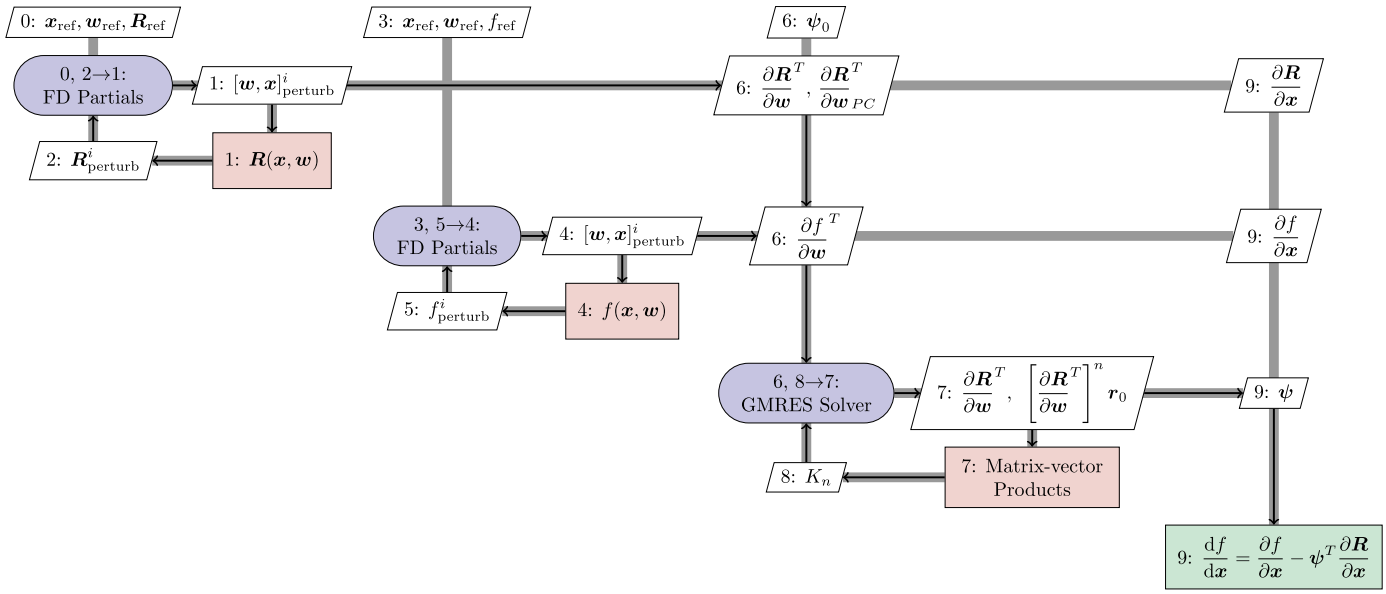

Fig. 4. Steps for the FD Jacobian adjoint option: (1) We compute $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$, $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$, and $\partial \boldsymbol{R}/\partial \boldsymbol{x}$ by looping $0 \rightarrow 2 \rightarrow 0$. The notation "ref" and "perturb" represents the reference and perturbed values, respectively. $\boldsymbol{w}^i_{\text{perturb}}$ denotes the state variable vector whose rows associated with the ith color are perturbed. (2) We compute $[\partial f/\partial \boldsymbol{w}]^T$ and $\partial f/\partial \boldsymbol{x}$ by looping $3 \rightarrow 5 \rightarrow 3$. (3) We solve the adjoint equations to obtain $\boldsymbol{\psi}$ by looping $6 \rightarrow 8 \rightarrow 6$. n is the iteration number. (4) We compute the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$ in process 9. (For interpretation of the references to color in this figure legend, the reader is referred to the Web version of this article.)

4.6. Solver-agnostic implementation of proposed adjoint approach

This section elaborates on our proposed adjoint implementation (Jacobian free) along with two variations (FD Jacobian and AD Jacobian), which are summarized as follows:

FD Jacobian: Uses a coloring-accelerated finite-difference method to compute all partial derivatives $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$, $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$, $\partial \boldsymbol{R}/\partial \boldsymbol{x}$, $[\partial f/\partial \boldsymbol{w}]^T$, and $\partial f/\partial \boldsymbol{x}$, which are all stored. $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$ is explicitly provided in the GMRES iteration to compute the matrix-vector product.

AD Jacobian: This is similar to the FD Jacobian except that all partial derivatives are computed by coloring-accelerated forward-mode AD.

Jacobian free: Uses coloring-accelerated forward-mode AD to compute $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$. Also uses reverse-mode AD to compute $[\partial f/\partial \boldsymbol{w}]^T$ and $\partial f/\partial \boldsymbol{x}$, the matrix-vector product $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T \boldsymbol{r}_0$ for the GMRES iteration, and $[\partial \boldsymbol{R}/\partial \boldsymbol{x}]^T \boldsymbol{\psi}$ for the total derivative.

The above adjoint implementation variations are in the most general form; for each implementation, the actual combination of options used for a specific solver to compute the partial derivatives and solve the adjoint equations may differ. We summarize these modifications in Appendix A.4. The present section keeps the explanation general by omitting solver-specific implementation details (described in the appendix).

The FD Jacobian is the most straightforward adjoint option because it does not require the construction of differentiated versions of the $\boldsymbol{R}(\boldsymbol{x}, \boldsymbol{w})$ and $f(\boldsymbol{x}, \boldsymbol{w})$ functions by using AD or the complex-step method. Moreover, it is much easier to implement compared with the hand-coded analytical derivative method, as discussed in Sec. 4.2. We illustrate the operations and data flow for a generic FD Jacobian approach in Fig. 4. We use the extended design structure matrix standard developed by Lambe and Martins [180]. The diagonal entries are the modules in the optimization process, whereas the off-diagonal entries are the data. Each module takes data input from the vertical direction and outputs data in the horizontal direction. The thick gray lines and the thin black lines denote the data and process flow, respectively. The numeric label for each process denotes their execution order. Taking $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$ as an example, the coloring-accelerated FD Jacobian is computed as follows: We first compute the reference residual $\boldsymbol{R}_{\text{ref}}$ by using the unperturbed reference variables $\boldsymbol{x}_{\text{ref}}$ and $\boldsymbol{w}_{\text{ref}}$. Next, for each color i, we add perturbation ε to $\boldsymbol{w}_{\text{ref}}$ at the rows associated with this color and compute the perturbed residual $\boldsymbol{R}_{\text{perturb}}$. Based on the reference and perturbed $\boldsymbol{R}$, we compute $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ at the columns associated with color i by using the finite-difference method (loop $0 \rightarrow 2 \rightarrow 0$). Once done, we store the computed columns in $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ to its transposed matrix $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$ in memory. As mentioned in Sec. 4.3, the coloring ensures that the residual stencils do not overlap for these computed columns, which allows us to compute multiple columns for $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ by calling the $\boldsymbol{R}$ function only once. We then loop over all colors until all columns in $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ are computed. A similar strategy is used for $\partial \boldsymbol{R}/\partial \boldsymbol{x}$. We need special treatment for $[\partial f/\partial \boldsymbol{w}]^T$ and $\partial f/\partial \boldsymbol{x}$ because they are typically dense vectors (e.g., drag depends on state variables integrated over all discrete cell faces on the design surface). Taking $[\partial f/\partial \boldsymbol{w}]^T$ as an example, we divide f into N_D discrete cell faces, i.e., $[\partial f/\partial \boldsymbol{w}]^T = \sum_{i=1}^{N_D} [\partial f_i/\partial \boldsymbol{w}]^T$. We then compute each $[f_i/\partial \boldsymbol{w}]^T$ by using coloring and sum their values to $[\partial f/\partial \boldsymbol{w}]^T$. Note that we always use the graph coloring method instead of the single-cell residual method to accelerate Jacobian computation, which avoids code duplication and ensures consistency in residual computation between the primal and adjoint solvers, as discussed in Sec. 4.3. When solving the adjoint equation, we pass a fully assembled $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$ into PETSc (processes 6) to compute the matrix-vector product $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T \boldsymbol{r}_0$ for the GMRES iteration (loop $6 \rightarrow 8 \rightarrow 6$) and solve for $\boldsymbol{\psi}$. Finally, we compute the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$ in process 9.

The implementation of the AD Jacobian approach is similar to that of the FD Jacobian approach with two exceptions. First, instead of using the original $\boldsymbol{R}(\boldsymbol{x}, \boldsymbol{w})$ and $f(\boldsymbol{x}, \boldsymbol{w})$ functions, we construct their differentiated versions by using forward-mode AD. Although the complex-step method also offers accurate derivative computation, it has a memory and computational cost that is similar to the operator-overloading AD but is much higher than the source code transformation AD. Moreover, it does not support the transpose matrix-vector product needed for the Jacobian-free GMRES method (see details in Sec. 4.4). Therefore, in the present work, we do not consider the complex-step partial derivative computation. An example of original and differentiated $\boldsymbol{R}$ and f functions is shown in Fig. 5. Note that we show only the inputs and outputs for these functions (we elaborate in Appendix A.4 on how to use different AD to construct them). The second difference is that, instead of using the finite-difference method, we use the forward-mode AD to compute the partial derivatives. Taking $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$ as an example, we first assign the coloring vector $\boldsymbol{c}_i$ to the input vector $\dot{\boldsymbol{w}}$ for the forward-differentiated residual function $\dot{\boldsymbol{R}}$, where $\boldsymbol{c}_i$ has ones for the rows associated with color i and zeros for the other rows. We then call $\dot{\boldsymbol{R}}$ to compute $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ at the columns associated with color i (refer to loop $0 \rightarrow 2 \rightarrow 0$ in Fig. 6). Finally, we assign the computed columns in $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ to its transposed matrix $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$. As in the FD Jacobian approach, we need to loop over all the colors and we store all the partial derivative matrices in the memory.

The Jacobian-free approach is a hybrid-AD option, as shown in Fig. 6. We differentiate the $\boldsymbol{R}(\boldsymbol{x}, \boldsymbol{w})$ and $f(\boldsymbol{x}, \boldsymbol{w})$ functions in both forward and reverse modes to enable further performance improvement. Similar to the AD Jacobian approach, we use the coloring-accelerated forward-mode AD (loop $0 \rightarrow 2 \rightarrow 0$) to compute $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$ and store it in memory. One major difference compared with the AD Jacobian approach is that we do not compute or store $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$. Instead, the matrix-vector product $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T \boldsymbol{r}_0$ that is needed for the GMRES iteration is directly computed by using the reverse-mode AD (process 5). This is done by assigning $\boldsymbol{r}_0$ to the input vector $\bar{\boldsymbol{R}}$ and calling the reverse-mode AD to compute $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T \boldsymbol{r}_0$. Similar to what we did for $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$, we do not compute or store $\partial \boldsymbol{R}/\partial \boldsymbol{x}$; instead, we use the reverse-mode AD (process 7) to directly compute $[\partial \boldsymbol{R}/\partial \boldsymbol{x}]^T \boldsymbol{\psi}$. Moreover, we use reverse-mode AD (process 3) to compute $[\partial f/\partial \boldsymbol{w}]^T$ and $\partial f/\partial \boldsymbol{x}$ instead of coloring-accelerated forward-mode AD. As mentioned in Sec. 4.4, the use of the Jacobian-free GMRES strategy saves the computation time and memory involved in computing and storing $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$. Section 5 quantitatively evaluates the performance of the above adjoint implementation variations.

5. Results

In this section, we use three cases from the ADODG to evaluate the

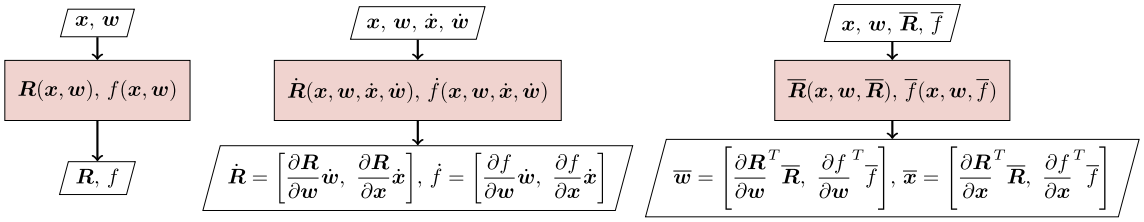

Fig. 5. General inputs and outputs for the original $\boldsymbol{R}$ and f functions and their differentiated versions using forward- and reverse-mode AD.

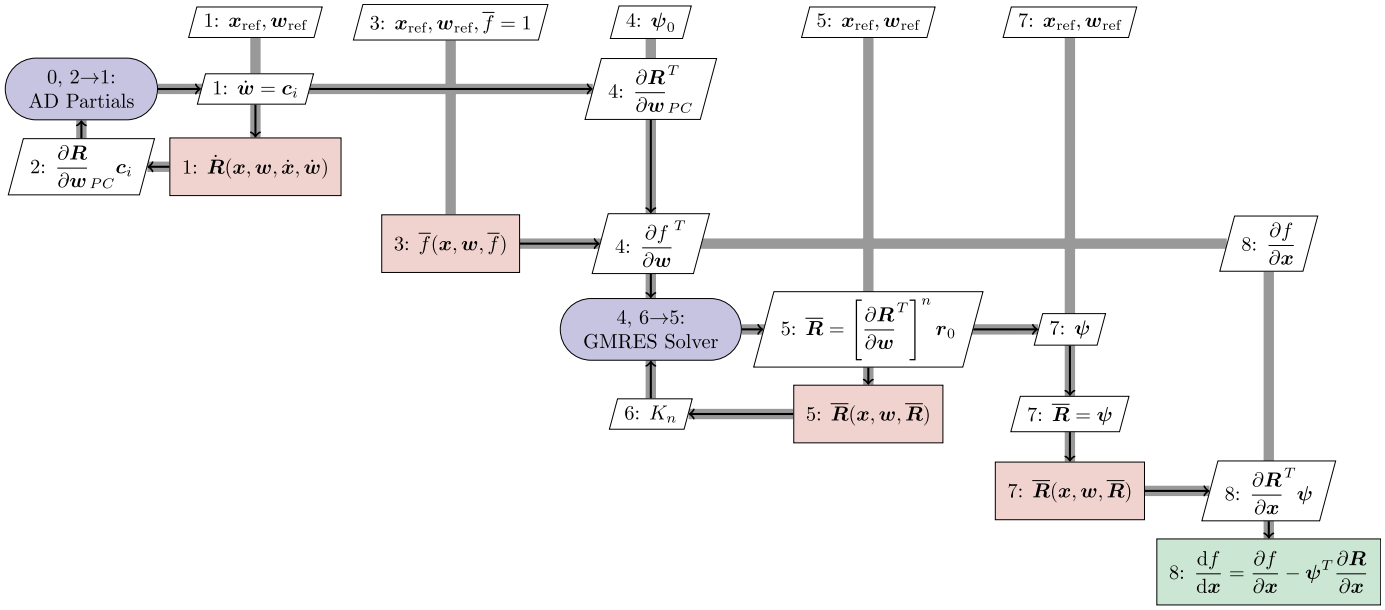

Fig. 6. Steps for the Jacobian-free adjoint option: (1) We compute $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$ by looping $0 \rightarrow 2 \rightarrow 0$. "ref" represents the unperturbed reference values, $\boldsymbol{c} \in \mathbb{R}^{n_w}$ is the coloring vector. $\boldsymbol{c}_i$ has ones at the rows associated with the ith color and zeros at other rows. (2) We compute $[\partial f/\partial \boldsymbol{w}]^T$ using reverse-mode AD in process 3. (3) We solve the adjoint equations to obtain $\boldsymbol{\psi}$ by looping $4 \rightarrow 6 \rightarrow 4$, while the required matrix-vector products are computed using reverse-mode AD in process 5. n denotes the iteration number. (4) We compute $[\partial \boldsymbol{R}/\partial \boldsymbol{x}]^T \boldsymbol{\psi}$ using reverse-mode AD in process 7. (5) We compute the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$ in process 8. (For interpretation of the references to color in this figure legend, the reader is referred to the Web version of this article.)

speed, scalability, memory usage, and accuracy of the various adjoint implementation variations (AD Jacobian, FD Jacobian, and Jacobian free; see Sec. 4.6) in the two CFD solvers.

The first case consists of a low-speed rectangular wing (ADODG Case 3) [181] that we use to compare the two CFD solvers and their adjoint implementations. More specifically, we compare the performance of adjoint implementations for ADflow, DAFoam, and adjointSimpleShapeCheckpointingFoam [22]. The second case consists of the transonic CRM wing (ADODG Case 4) [3], which we use to benchmark several variations of the adjoint implementation in ADflow and test the parallel efficiency. The third and final case is based on the wing-body-tail configuration (ADODG Case 5) [6] with an overset mesh, which we use to demonstrate the ability of our ADflow adjoint implementation to handle complex geometry configurations. To facilitate future performance comparisons, the files required to setup benchmarks are publicly available [39].

5.1. Low-speed rectangular wing

ADODG Case 3 involves an unswept low-speed rectangular wing with an aspect ratio of 6.12 and a NACA 0012 airfoil. This case allows us to evaluate the adjoint calculations by using ADflow, DAFoam, and full-code AD using the same geometry and flow conditions. We use a coarse structured mesh with 102 912 cells, providing an average $y^+ = 1.2$ with a domain that extends 20 chords from the surface.

The simulation is run at $Re = 10^6$ and $M = 0.15$ by using the SA turbulence model. The objective function is the drag coefficient, and the target lift coefficient is 0.375. The design variables are the sectional twists γ at eight spanwise locations.

To evaluate the computational speed, we set the relative residual tolerance to 10^{-10} and 10^{-8} for the flow and adjoint solutions, respectively. For ADflow, the flow and adjoint converge in 1011 and 310 iterations, respectively, whereas for DAFoam, the flow and adjoint converge in 720 and 143 iterations, respectively. When verifying the accuracy of the adjoint derivative, we run the flow until the residuals no longer change significantly. The flow residuals for ADflow and DAFoam decrease by 14 and 12 orders of magnitude, respectively. We use the same convergence criteria for all other flow and adjoint computations in this paper.

In this case, we also run the adjoint solver adjointSimpleShapeCheckpointingFoam of Towara and Naumann [22] to obtain values for the reference derivatives for DAFoam. We use the same problem setup and convergence criteria as described above, except that we use the algebraic multigrid solver with diagonal incomplete Cholesky for the solution to the pressure equation and the symmetric Gauss–Seidel iteration solver for the other linear equations. In addition, to save memory, we switch on the checkpointing and use symbolic differentiation for the embedded linear solvers [182,183]. We apply the equidistant checkpointing option with an interval of two time steps, which ensures that the memory usage is similar to that of DAFoam (Jacobian free) for a consistent comparison of speed.

For this case, we run all the simulations on Stampede 2 by using one CPU core on one Skylake node. The Skylake nodes are equipped with an Intel Xeon Platinum 8160 CPU running at 2.1 GHz, and each node has 48 CPU cores and 196 GB of memory. The Skylake nodes are interconnected through a 100 GB/s Intel Omni-Path network. The reason for using a coarse mesh is that both ADflow and DAFoam can converge the flow tightly. This prerequisite is critical when evaluating the accuracy of the adjoint derivative. Moreover, running the cases using one CPU core allows us to isolate the impact of parallel communication on the performance.

To create a platform-independent benchmark, the runtime is reported as TauBench work units (T_{WU}), which is computed as[4]

$$T_{\text{WU}} = \frac{nT}{T_{\text{ref}}}, \tag{28}$$

where T is the wall-clock runtime of a solver in seconds (s), n is the number of CPU cores that are used to run the solver, and T_{ref} is the reference wall-clock runtime of the Tau solver on the same computing platform. We follow the instructions described in the third international workshop on high-order CFD methods guidelines[5] and run the Tau code on a Skylake node by using the command: mpirun -np 1 ./TauBench -n 250000 -s 10. We obtain $T_{\text{ref}} = 2.972$ s for Skylake nodes. In this paper,

[4] http://www.ipacs-benchmark.org/index.php.
[5] https://www.grc.nasa.gov/hiocfd/guidelines.

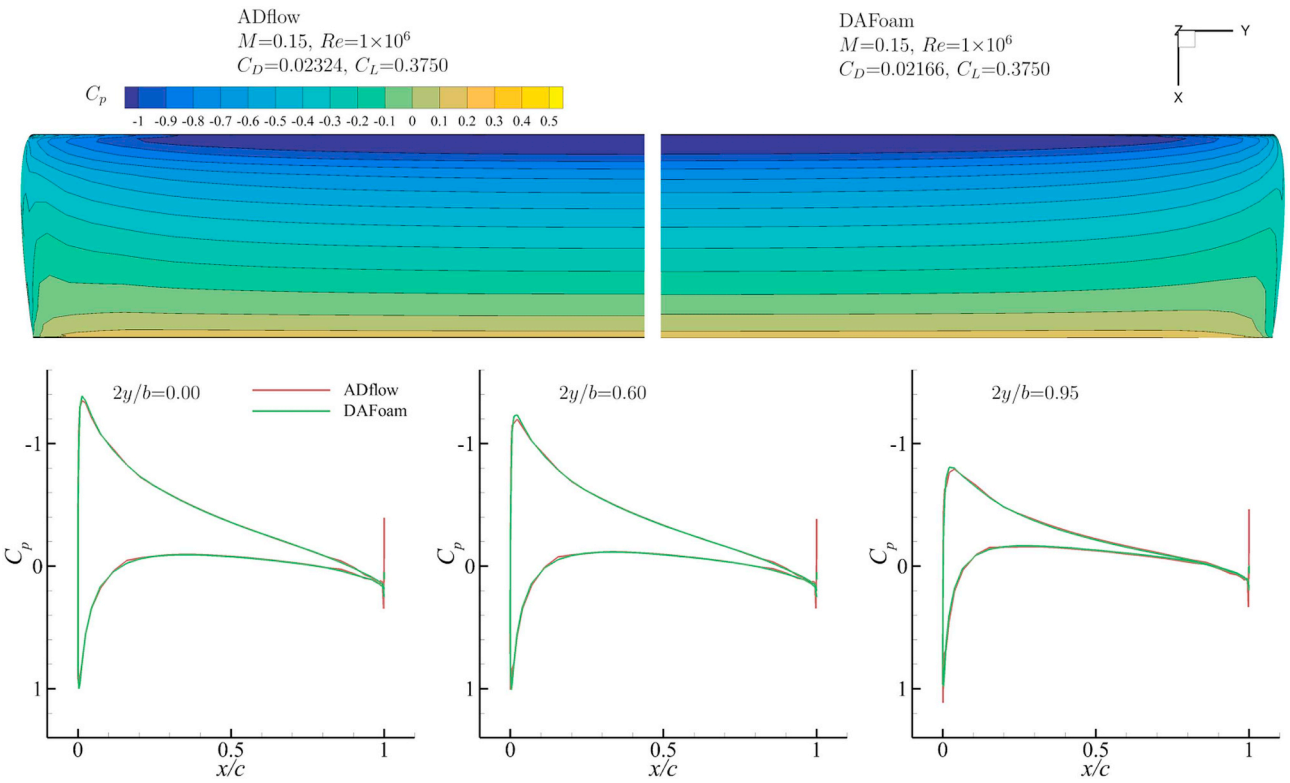

Fig. 7. ADflow and DAFoam predict identical pressure distributions; however, the C_D predicted by DAFoam is 6.8% lower than ADflow.

Table 2
Performance of flow and adjoint computation for the ADODG3 case. The runtime is reported as TauBench work units (T_{WU}) whereas, for reference, the wall-clock runtime for the flow simulation (ADflow) is 148.6 s. All the cases are run on a single CPU core. The mesh contains 102 912 cells. For ADflow, Jacobian free is the most efficient approach in terms of both runtime and memory usage. For DAFoam, Jacobian free is the fastest method; however, it requires more memory than FD Jacobian because it uses operator-overloading AD.

Solvers	ADflow			DAFoam		adjointSimpleShapeCheckpointingFoam
Adjoint options	Jacobian free (transformation)	AD Jacobian (transformation)	FD Jacobian	Jacobian free (overloading)	FD Jacobian	Full-code AD (overloading)
Runtime (T_{WU})						
Flow	50.0	50.0	50.0	39.7	39.7	165.2
Adjoint	41.1	56.4	45.3	187.4	245.6	791.4
Jacobian assembly	2.4	19.8	9.3	100.3	193.5	–
Adjoint solution	38.7	36.6	36.0	87.1	52.1	–
Adjoint-flow ratio	0.8	1.2	0.9	4.7	6.2	4.8
Peak memory (GB)						
Flow	1.7	1.7	1.7	0.3	0.3	7.6
Adjoint	2.1	3.0	3.0	17.2	7.1	22.1
Adjoint-flow ratio	1.2	1.8	1.8	57.3	23.7	2.9

the memory usage of a solver is reported as the peak resident set size in gigabytes (GB).

We first compare the flow solutions obtained by ADflow and DAFoam. The pressure contours on the wing surface and the pressure distributions at the root, midspan, and tip sections are almost identical between the two solvers, as shown in Fig. 7. However, we observe that the drag coefficient predicted by DAFoam is 6.8% lower than ADflow. We suspect that the difference is primarily due to the coarse mesh, for which the discretization error in both solvers is significant.

We now compare the speed and memory usage of the flow and adjoint solutions for ADflow and DAFoam (Table 2). To help assess the detailed performance, we break the adjoint total runtime into a Jacobian assembly and an adjoint equation solution. The Jacobian assembly consists of the runtime required to compute all the required partial derivatives for solving the adjoint equations (12)—$[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$, $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$, and $[\partial f/\partial \boldsymbol{w}]^T$—and for computing the total derivative (13)—$\partial \boldsymbol{R}/\partial \boldsymbol{x}$ and $\partial f/\partial \boldsymbol{x}$. Note that the adjoint solver from Towara and Naumann [22] uses the full-code AD approach, so the breakdown is not available.

For ADflow, the runtime for the adjoint equation solution using the Jacobian-free approach is 6% greater than for the AD Jacobian and FD Jacobian approaches (Table 2). This is expected because Jacobian free performs a reverse-mode AD residual computation for each GMRES iteration, which is slightly more expensive than a matrix-vector product that uses a stored Jacobian. For the Jacobian-free case, the Jacobian assembly time is only the time required for computing the preconditioner and adjoint right-hand side and is thus much lower than for AD Jacobian and FD Jacobian. Overall, when considering both the assembly time and solution time, the total runtime for the Jacobian free is 27% and 9% faster than AD Jacobian and FD Jacobian, respectively. Moreover, the memory usage of Jacobian free is 30% lower than that of AD Jacobian and FD Jacobian. Therefore, for this simple case, Jacobian

free is the best option, with an adjoint-flow runtime ratio of 0.8 and an adjoint-flow memory usage ratio of 1.2. Note that the runtime benefit shown here skews in favor of the explicitly stored Jacobian as the number of functions of interest increases, since the cost of computing and storing the exact Jacobian is increasingly amortized as the number of adjoint solutions increases.

The DAFoam adjoint total runtime for Jacobian free is 25% faster than FD Jacobian. Again, this is primarily because Jacobian free does not compute $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$. In terms of memory usage, however, Jacobian free needs 140% more memory than FD Jacobian, which is not consistent with what we observed in ADflow where Jacobian free used less memory. This result is primarily attributed to the fact that the complex object-oriented code structure in OpenFOAM introduces numerous intermediate variables into the residual computation routine, which contains tens of levels of subroutines. When performing reverse-mode operator-overloading AD computation in Jacobian free, all these intermediate variables need to be stored in tapes and thus requires more memory than FD Jacobian. One way to reduce the memory usage is to carefully examine all relevant subroutines and set the unused variables to "passive" in reverse-mode AD. By doing this, we store only variables that are actually used in the residual computation. However, we have not done this code optimization in DAFoam.

The ADflow adjoint derivative computation is 4.6 times faster than that of DAFoam (using Jacobian free). The largest difference is in the cost of the Jacobian assembly: 2.4 for ADflow versus 100.3 for DAFoam), as shown in the runtime breakdown in Table 2. This is primarily due to the different coloring techniques. As elaborated on in Appendix A.2, ADflow uses a local block coloring on a structured mesh, allowing for a compact analytic coloring scheme with only 162 colors to compute the preconditioner $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$ [55]. In contrast, for the unstructured mesh solver DAFoam, we require 945 colors. Since the runtime of computing $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$ is proportional to the number of colors, the Jacobian assembling for DAFoam is much slower. We have more colors in DAFoam because the SIMPLE algorithm with Rhie–Chow interpolation results in a state Jacobian matrix that is much denser than that in ADflow (Fig. A.10). Moreover, we use a heuristic global coloring scheme, so the number of colors is about three times the maximum number of nonzero entries in a row of the Jacobian matrix. However, we showed in our previous work [86] that the number of colors in DAFoam does not depend on the mesh size and CPU cores when running in parallel, which alleviates the problem.

In addition to runtime, DAFoam requires 8.2 times more memory than ADflow. This high memory cost is case specific for the DAFoam implementation and has two causes: First, OpenFOAM has complex object-oriented code structure and requires large tape memory when performing reverse-mode AD, as mentioned above. Second, the residual stencil in DAFoam is much larger than that in ADflow (Fig. A.10), which eventually increases the memory cost for reverse AD computation and for storing the preconditioner matrix. Despite these disadvantages, the Jacobian free (operator-overloading AD) option in DAFoam is 4.2 times faster than the adjointSimpleShapeCheckpointingFoam solver, while requiring 22.1% less memory, which confirms the efficiency of the Jacobian-free adjoint approach. Note that we test only the full-code AD solver adjointSimpleShapeCheckpointingFoam from Towara and Naumann [22] because it provides machine-precision values for derivatives. We have not tested their other adjoint implementations (e.g., reverseAccSimpleFoam and piggySimpleFoam), which were shown to perform better [34].

In addition to computational time, we also evaluate the accuracy of the derivative computations in DAFoam and ADflow in Tables 3 and 4, respectively. The accuracy of derivative computations is critical to ensure a robust optimization process because inaccurate gradients can cause an optimization to terminate early, which results in sub-optimal (optimality not tightly converged) or infeasible (design constraints not strictly satisfied) designs. Accurate derivatives are even more crucial when using advanced optimization algorithms such as the Broyden–Fletcher–Goldfarb–Shanno algorithm, as discussed by Peter and Dwight [8].

It is common practice to compare the adjoint derivatives with reference derivatives computed from finite differences. However, as mentioned in Sec. 4.2, the finite-difference method is subject to subtractive and cancellation errors, which degrades the accuracy of reference derivatives. Therefore, for ADflow, we use as reference the derivatives computed by the complex-step method. For DAFoam, the reference values are computed by using the full-code AD solver adjointSimpleShapeCheckpointingFoam [22].

Both the complex-step and the full-code AD methods provide machine-precision-level reference derivatives, as shown in Tables 3 and 4 The $\mathrm{d}C_D/\mathrm{d}\gamma$ values computed by using the Jacobian-free and AD Jacobian options match the reference values to 12 digits for ADflow and to 10 digits for DAFoam. Given that the flow converges by 14 and 12 orders of magnitude, respectively, the adjoint derivatives have an accuracy that is fully consistent with the flow solutions. We achieve machine-precision accurate adjoint derivatives because we do not make any approximations (e.g., ignore and simplify terms or use the frozen turbulence assumption) in our adjoint formulations. Moreover, we use AD to accurately compute all the partial derivatives. Finally, our adjoint code is bug-free. In contrast, the FD Jacobian option matches the references to only three to four digits and the average error is around 0.1% because of the errors introduced by using the finite-difference

Table 4
DAFoam derivatives ($\mathrm{d}C_D/\mathrm{d}\gamma, \times 10^{-4}$) at eight spanwise locations. The Jacobian-free approach matches the reference to 10 significant digits.

$2y/b$	Reference (Full-code AD)	DAFoam (Jacobian free)	DAFoam (FD Jacobian)
0.00	−4.882 503 197 6	−4.882 503 197 5	−4.883 286 244 8
0.20	−6.955 486 570 2	−6.955 486 570 1	−6.955 367 327 4
0.40	−7.518 312 658 1	−7.518 312 657 9	−7.519 901 569 7
0.60	−7.984 926 631 0	−7.984 926 630 6	−7.986 436 953 9
0.80	−5.597 739 570 2	−5.597 739 569 7	−5.603 676 158 6
0.90	−2.888 106 152 0	−2.888 106 151 4	−2.887 749 885 4
0.95	−1.365 193 880 0	−1.365 193 879 5	−1.355 452 150 5
1.00	−0.827 241 516 3	−0.827 241 515 5	−0.836 433 805 4

Table 3
ADflow derivatives ($\mathrm{d}C_D/\mathrm{d}\gamma, \times 10^{-4}$) at eight spanwise locations. The Jacobian-free and AD Jacobian approaches match the complex-step reference to 12 significant digits.

$2y/b$	Reference (complex step)	ADflow (Jacobian free)	ADflow (AD Jacobian)	ADflow (FD Jacobian)
0.00	−4.564 591 147 572	−4.564 591 147 573	−4.564 591 147 573	−4.564 801 684 647
0.20	−6.509 297 734 175	−6.509 297 734 172	−6.509 297 734 172	−6.509 597 548 593
0.40	−7.056 747 972 049	−7.056 747 972 043	−7.056 747 972 043	−7.057 061 656 265
0.60	−7.519 182 294 474	−7.519 182 294 469	−7.519 182 294 469	−7.519 470 654 527
0.80	−5.249 850 721 884	−5.249 850 721 882	−5.249 850 721 882	−5.249 998 051 928
0.90	−2.774 098 324 145	−2.774 098 324 146	−2.774 098 324 146	−2.774 132 680 066
0.95	−1.414 324 615 362	−1.414 324 615 362	−1.414 324 615 362	−1.414 316 189 209
1.00	−0.857 741 790 567	−0.857 741 790 567	−0.857 741 790 567	−0.857 698 448 112

method to compute the partial derivatives. Although this derivative error is still acceptable, our experience is that the exact derivative obtained from Jacobian free and AD Jacobian is crucial to ensure robust and tightly converged optimizations, as mentioned above.

5.2. Transonic wing

We now evaluate the performance of the ADflow adjoint for a transonic wing configuration—the CRM wing geometry from ADODG Case 4. In previous work, this benchmark was the subject of single- and multi-point aerodynamic shape optimization [3,74]. Here, we focus on benchmarking the parallel efficiency of the adjoint derivative computation.

The CRM is representative of a modern transonic commercial transport aircraft, with a size similar to that of a Boeing 777. The fuselage and tail are removed from the original CRM, and the root of the remaining wing is moved to the symmetry plane. The nominal flight condition is $C_L = 0.5$ at $M = 0.85$ and $Re = 5 \times 10^6$. The design variables are twists at eight spanwise locations. We use the L1 mesh described by Lyu et al. [3], which consists of 3 604 480 cells. We run the computation on 48 CPU cores with two Skylake nodes on Stampede 2.

Fig. 8 shows the surface sensitivity and C_p distributions at three spanwise sections. As shown in the sectional C_p distributions, a shock is present on the upper surface of the wing. Similarly to what Lyu et al. [3] observed, this shock causes a large negative-sensitivity region near the middle chord of the upper surface. In addition, we observe a region with large sensitivity near the leading edge, especially at the root.

Similarly to the subsonic wing case, Jacobian free is the fastest method and requires the least memory, as shown in Table 5. To be more specific, Jacobian free is 19.5% and 4.9% faster than AD Jacobian and FD Jacobian, respectively, and its memory usage is 37% lower. In addition, the derivatives computed by using Jacobian free and AD Jacobian match the complex-step reference values to 11 significant figures, as shown in Table 6. Given that the flow converges 13 orders of magnitude, we conclude that the Jacobian-free adjoint accuracy is fully consistent with the flow solutions.

Table 7 shows the flow and adjoint runtime in TauBench work units for an increasing number of CPU cores. The values in parentheses are the parallel efficiency, defined as $\eta = T^{24}_{\text{WU}}/T^{n}_{\text{WU}}$, where the superscript is

Table 5
ADflow performance for the CRM wing case. The runtime is reported as TauBench work units T_{WU} whereas, for reference, the wall-clock runtime for the flow simulation is 375.47 s. All the cases are run on 48 CPU cores on two nodes. The mesh contains 3 604 480 cells. The Jacobian-free option is the best.

ADflow	Jacobian free	AD Jacobian	FD Jacobian
Runtime (T_{WU})			
Flow	6064.1	6064.1	6064.1
Adjoint	4781.3	5719.1	5014.9
Jacobian assembly	110.4	1176.0	446.4
Adjoint equation solution	4670.9	4543.1	4568.5
Adjoint-flow ratio	0.8	0.9	0.8
Peak memory (GB)			
Flow	44.6	44.6	44.6
Adjoint	87.2	119.5	119.5
Adjoint-flow ratio	2.0	2.7	2.7

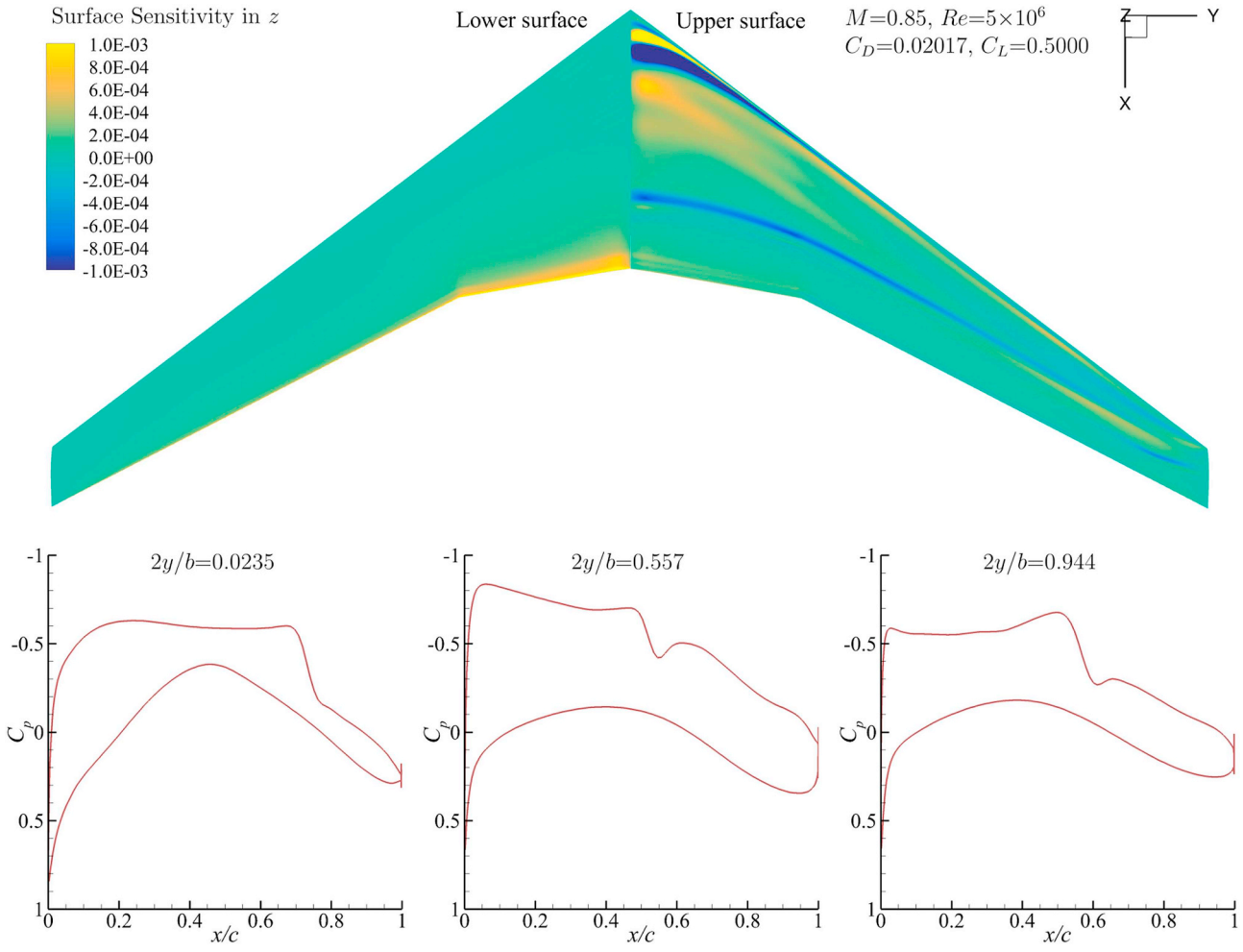

Fig. 8. Surface sensitivity ($\mathrm{d}C_D/\mathrm{d}z$) and pressure profiles for CRM under transonic flow conditions.

Table 6
ADflow derivatives ($dC_D/d\gamma$, $\times 10^{-3}$) for the CRM wing case. The Jacobian-free and AD Jacobian match the complex-step reference to 11 significant digits.

$2y/b$	Reference (complex step)	ADflow (Jacobian free)	ADflow (AD Jacobian)	ADflow (FD Jacobian)
0.000	1.467 830 420 22	1.467 830 420 24	1.467 830 420 24	1.467 913 755 59
0.149	1.827 776 113 24	1.827 776 113 27	1.827 776 113 27	1.828 063 048 53
0.296	1.539 421 192 63	1.539 421 192 65	1.539 421 192 65	1.539 699 968 60
0.437	1.199 899 961 45	1.199 899 961 48	1.199 899 961 48	1.200 078 068 00
0.579	0.719 895 630 07	0.719 895 630 08	0.719 895 630 08	0.719 919 787 75
0.721	0.404 497 687 88	0.404 497 687 89	0.404 497 687 89	0.404 401 010 12
0.861	0.240 308 245 18	0.240 308 245 20	0.240 308 245 20	0.240 184 238 91
1.000	0.093 333 672 78	0.093 333 672 79	0.093 333 672 79	0.093 296 748 82

the number of CPU cores. Note that Skylake processors have many more cores compared with traditional CPU processors: as a trade-off, the Skylake processors run at a relatively low frequency (2.1 GHz). Therefore, when evaluating the performance for the Skylake nodes, it is more appropriate to compare node-to-node as opposed to core-to-core performance. Both the flow and adjoint scale up to 768 CPU cores. The parallel efficiency for the adjoint is larger than 90% if each processor owns no less than 75 000 cells. Note that we use only 24 CPU cores per node in Table 7. Alternatively, we can use the maximum 48 CPU cores per node. We observe that the use of more CPU cores per node generally decreases the wall-clock runtime; however, it also decreases the parallel efficiency. For example, when using 1536 CPU cores with 32 nodes, the adjoint parallel efficiency for Jacobian free is 30.1%, which is less than the efficiency (36.7%; see Table 7) when using the same number of nodes. This behavior is due to limitations in the memory bandwidth on the nodes, which causes the computations to be limited by memory rather than by CPU speed.

5.3. Wing-body-tail configuration

The final case we solve is the wing-body-tail configuration from ADODG Case 5, which we have used extensively in previous work for both aerodynamic shape optimization [6,165] and aerostructural optimization [108,184–186]. Here, we use this case to demonstrate the ability of the ADflow adjoint to handle complex geometric configurations.

The ADODG Case 5 uses the same CRM wing geometry from Case 4 but includes the fuselage and horizontal tail of the original CRM, which is also known as the Drag Prediction Workshop 4 geometry [187]. The flow conditions are the same as for Case 4 except that the Reynolds number is $Re = 4.3 \times 10^7$ to match a realistic flight condition and scale, as opposed to the wind tunnel testing model of Case 4. We generate multiblock structured overset meshes for each component, resulting in a mesh with a total of 10 358 373 cells. The other problem settings are similar to those of the CRM wing-only case with the exception that we now compute sensitivities with respect to the wing twists at nine spanwise locations. We run the computation using 96 CPU cores and four Skylake nodes on Stampede 2.

Fig. 9 shows the surface sensitivity and pressure distribution. Similarly to what we observed in the CRM wing-only case, a shock wave appears on the upper surface of the wing, resulting in high local sensitivity of the drag coefficient with respect to shape change. However, this high sensitivity becomes weaker near the root, owing to the interaction between wing and fuselage. Table 8 lists the computational cost and memory usage for this case, where we run only Jacobian free. Again, the adjoint solution is faster than the flow solution, with an adjoint-flow runtime ratio of 0.7. In terms of the accuracy, the adjoint matches the complex-step reference values to six digits, as shown in Table 9. This good agreement highlights the effectiveness of Jacobian free in handling complex geometries and mesh topologies.

6. Conclusions

In this paper, we discuss and compare various approaches for implementing a discrete adjoint solver. Specifically, we compare three options for implementing a discrete adjoint based on an exact solution of the linear system; namely, Jacobian free (Jacobian-free adjoint solved by using GMRES with transpose matrix-vector products computed by using reverse-mode AD), AD Jacobian (explicit Jacobian computation using the forward-mode AD), and FD Jacobian (explicit Jacobian computation using finite differences). We implement these three options for two different flow solvers—a compressible flow Newton–Krylov solver for overset structured meshes (ADflow) and an incompressible solver that uses the SIMPLE algorithm for unstructured meshes (DAFoam). We then compare the adjoint implementations by using three ADODG benchmark cases covering a wide range of flight conditions and configurations. These include Reynolds number ranging from 1 to 43 million, incompressible and transonic flow conditions, as well as multiblock and overset structured meshes with up to 10 million cells. The simulations comprise both serial and parallel runs with up to 1536 CPU cores.

We evaluate the performance of Jacobian free, AD Jacobian, and FD Jacobian by comparing their speed, scalability, memory usage, accuracy, development effort, and extensibility. The conclusions of this study are summarized in Table 10. We do not have scalability data for the Jacobian-free (operator overloading) and the full-code AD (operator overloading) options because we run the adjoint computation only in serial in the ADODG Case 3 (Sec. 5.1).

Overall, the Jacobian-free approach with source code transformation AD is the best option. More specifically, the Jacobian-free approach is up to 27% and 9% faster than the AD Jacobian and FD Jacobian approaches, respectively. Moreover, Jacobian free uses up to 30% less

Table 7
Flow and adjoint runtime using 24 CPU cores per node in TauBench work units. The values in parentheses are the parallel efficiency, which is defined as $\eta = T^{24}_{\text{WU}}/T^{n}_{\text{WU}}$, where the superscript is the number of CPU cores.

Nodes	Cores	Flow	Jacobian free	AD Jacobian	FD Jacobian
1	24	5628.3 (100.0%)	4525.0 (100.0%)	5455.6 (100.0%)	4765.9 (100.0%)
2	48	6064.1 (92.8%)	4781.3 (94.6%)	5719.1 (95.4%)	5014.9 (95.0%)
4	96	7166.4 (78.5%)	6828.0 (66.3%)	7885.9 (69.2%)	6998.3 (68.1%)
8	192	10 237.7 (55.0%)	10 147.6 (44.6%)	10 712.9 (50.9%)	10 012.9 (47.6%)
16	384	11 581.0 (48.6%)	9208.2 (49.1%)	10 006.1 (54.5%)	9407.2 (50.7%)
32	768	14 180.6 (39.7%)	12 334.0 (36.7%)	13 168.2 (41.4%)	12 574.1 (37.9%)

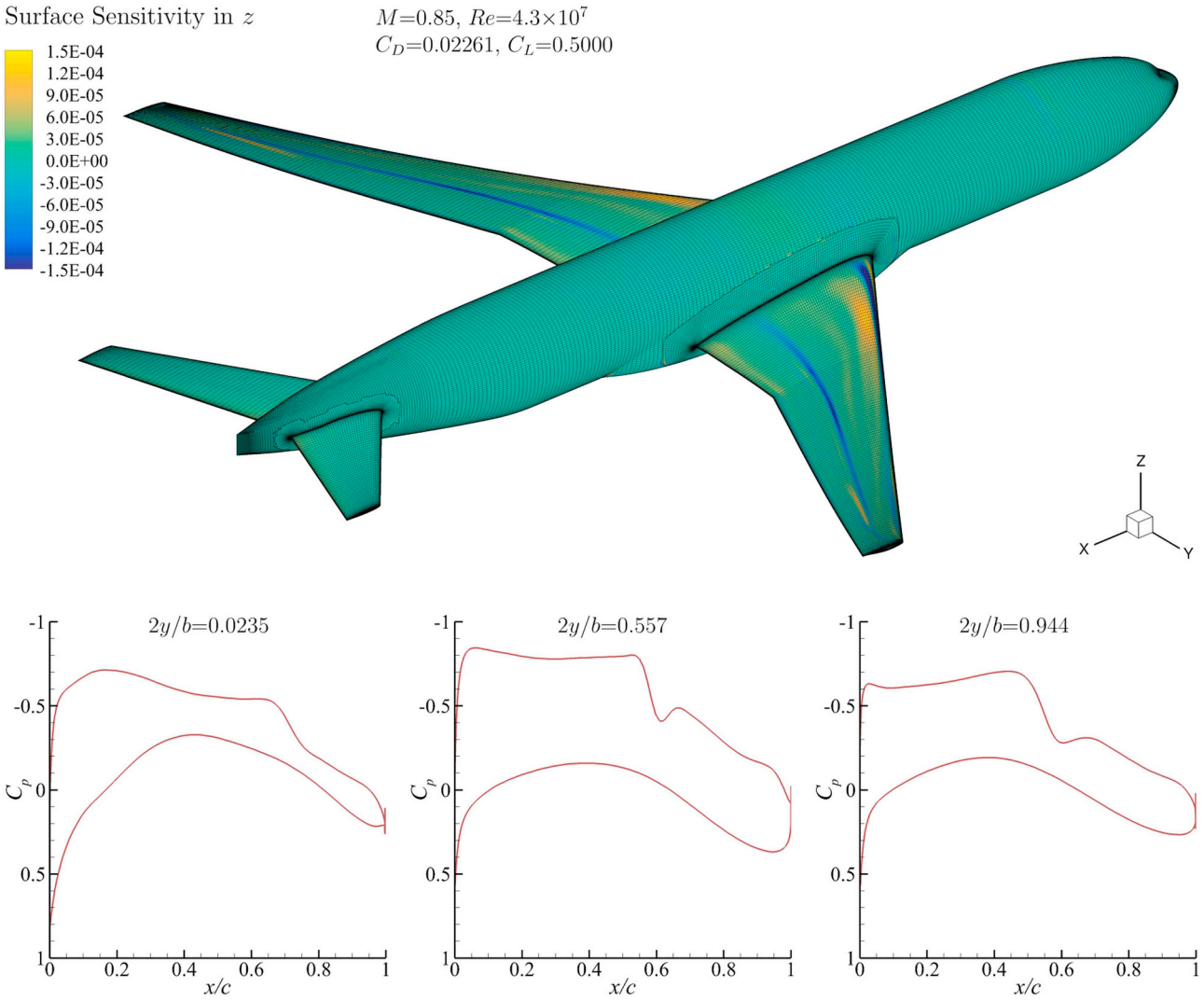

Fig. 9. Surface sensitivity (dC_D/dz) and pressure profiles for the CRM wing-body-tail configuration with overset mesh.

Table 8
ADflow performance for the CRM wing-body-tail configuration. The runtime is reported as TauBench work units T_{WU} whereas, for reference, the wall-clock runtime for the flow simulation is 1046.14 s. The case is run on 96 CPU cores on four nodes. The mesh contains 10 358 373 cells. Adjoint computation is faster than flow.

ADflow	Jacobian free
Runtime (T_{WU})	
Flow	33 792.0
Adjoint	23 577.6
Jacobian assembly	460.8
Adjoint solution	23 116.8
Adjoint-flow ratio	0.7
Peak memory (GB)	
Flow	123.7
Adjoint	252.9
Adjoint-flow ratio	2.0

Table 9
ADflow derivatives ($dC_D/d\gamma$, $\times 10^{-3}$) for the CRM wing-body-tail configuration. The adjoint derivative matches the complex-step reference to six significant digits.

$2y/b$	Reference (complex step)	ADflow (Jacobian free)
0.000	1.285 351 33	1.285 350 96
0.125	1.447 412 06	1.447 411 92
0.250	1.263 087 08	1.263 086 79
0.375	1.031 381 90	1.031 381 70
0.500	0.677 939 91	0.677 939 77
0.625	0.486 188 41	0.486 188 36
0.750	0.305 559 03	0.305 559 06
0.875	0.195 216 84	0.195 216 85
1.000	0.079 003 40	0.079 003 39

memory than AD Jacobian and FD Jacobian. This is primarily because Jacobian free computes matrix-vector products for the GMRES solver by using reverse-mode AD in a Jacobian-free manner. This strategy effectively avoids computing and storing the state Jacobian matrix and therefore results in superior speed and memory performance. Another advantage of the Jacobian-free approach is that we do not need to compute the connectivity information and sparsity structure of the Jacobian matrix. As a result, Jacobian free is easier to extend to complex mesh topologies, such as those that show up in overset meshes. These advantages enable parallel computation with up to 10 million cells and 1536 CPU cores. Partial derivative computation using AD allows the Jacobian-free approach to compute accurate derivatives with up to machine precision. In contrast, the FD Jacobian has an average error of around 0.1%.

In terms of AD tools, source code transformation AD is preferred

Table 10
Performance summary for discrete adjoint options implemented in ADflow, DAFoam, and adjointSimpleShapeCheckpointingFoam. Jacobian free with source code transformation is the best. Key: ○ poor, ◔ acceptable, ◕ good, ● excellent.

	Speed	Scalability	Memory	Accuracy	Effort	Extensibility
Jacobian free (transformation)	●	●	●	●	○	◕
Jacobian free (overloading)	◔↓ 460%	N/A	◔↑ 820%	●	◕	◕
AD Jacobian (transformation)	◕↓ 27%	●	◕↑ 30%	●	○	◕
FD Jacobian	●↓ 9%	●	◕↑ 30%	◔↓~0.1%	○	◕
Full-code AD (overloading)	○↓ 1930%	N/A	○↑ 1050%	●	●	●

because it is faster and requires less memory compared with operator-overloading AD. The Jacobian-free implementation using source code transformation is 4.6 times faster and requires 8.2 times less memory than the operator-overloading version. Furthermore, Jacobian free using source code transformation is 19.3 times faster and 10.5 times less memory intensive than the adjoint solver that uses operator overloading to differentiate the entire code.

In addition to the fact that source code transformation is naturally faster than operator overloading, two solver-specific factors impact the performance comparison between ADflow and DAFoam with the Jacobian-free option: First, ADflow uses structured meshes, which enables an analytical coloring scheme with only 162 colors for the preconditioner matrix. In contrast, DAFoam uses unstructured meshes and a heuristic coloring scheme with 945 colors, which results in an almost fivefold increase in computational cost for computing the preconditioner. Moreover, DAFoam uses a complex object-oriented code structure such that a single residual call could involve tens of subroutine levels and numerous intermediate variables. The residual stencil for the SIMPLE solver with Rhie–Chow interpolation is also much denser than that in ADflow. These two factors result in a large tape size and higher memory cost when performing reverse-mode AD.

The drawback of the source code transformation implementation is that it requires a higher development effort than operator overloading. Instead of directly applying operator overloading to every function, source code transformation requires manual construction of the routines to differentiate the residual and objective computation. To ensure optimal performance, these differentiated routines require careful implementation. However, once these routines are constructed by using source code transformation, extending the implementation to include new terms and objectives is straightforward.

The cases solved in this work are available as platform-independent benchmarks for comparing existing and future adjoint CFD solvers, and the ADflow solver is also available under an open-source license. The techniques presented herein provide a guide for future discrete adjoint development, not only for CFD but also for a wide range of other PDE solvers.

Acknowledgments

The computations were done in the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation Grant number ACI-1548562. The authors would also like to thank Markus Towara for sharing their adjoint code and Richard Dwight, Nicolas Gauger, Mike Giles, Jason Hicken, Dimitri Mavriplis, Jens-Dominik Müller, Jacques Peter, and Markus Towara for their suggestions.

Appendix B. Supplementary data

Supplementary data to this article can be found online at https://doi.org/10.1016/j.paerosci.2019.05.002.

Appendix A. Solver-specific implementation details of proposed adjoint approach

In Sec. 4.6, we define three adjoint approaches (FD Jacobian, AD Jacobian, and Jacobian free) in solver-agnostic terms for the sake generality. However, there are various solver-specific implementation details that have important repercussions in the overall numerical performance. This section elaborates on the implementation details for ADflow and DAFoam. These details concern the choice of elements and ordering for the $\boldsymbol{x}$, $\boldsymbol{w}$, and $\boldsymbol{R}$ vectors, the coloring implementations, the GMRES preconditioning strategy, and, most importantly, how to construct the differentiated $\boldsymbol{R}$ and f functions.

A.1. Choice of residuals, states, and design variables

When implementing an adjoint in a specific primal solver, we need to define exactly the residuals ($\boldsymbol{R}$), states ($\boldsymbol{w}$), and design variables ($\boldsymbol{x}$). The choice of states depends on the flow discretization and solution techniques. For ADflow, the states $[\rho, u, v, w, \rho E, \tilde{\nu}]$ are the primitive variables in the governing equations (1)–(4). It is critical to include the turbulence variable $\tilde{\nu}$ in the $\boldsymbol{w}$ vector and compute the corresponding differentiation in the $\partial\boldsymbol{R}/\partial\boldsymbol{w}$ matrix because it directly impacts the accuracy of adjoint total derivatives. For DAFoam, we assume incompressible flow with a constant density and ignore the energy equation, as mentioned in Sec. 3.3. Therefore the states are $[u, v, w, p, \phi, \tilde{\nu}]$, where ϕ is the cell face flux. Note that we treat ϕ as a state variable as a result of the segregated SIMPLE algorithm with Rhie–Chow interpolation, as elaborated on in our previous work [86].

In the discrete adjoint, $\boldsymbol{R}$ and $\boldsymbol{w}$ contain variables for all discrete cells. For example, in ADflow,

$$\boldsymbol{w} = \left[\rho_1, u_1, v_1, w_1, (\rho E)_1, \tilde{\nu}_1, \rho_2, u_2, v_2, w_2, (\rho E)_2, \tilde{\nu}_2, ..., \rho_{n_C}, u_{n_C}, v_{n_C}, w_{n_C}, (\rho E)_{n_C}, \tilde{\nu}_{n_C}\right]^T,$$

and

$$\boldsymbol{R} = \left[R_1^{\rho}, R_1^{u}, R_1^{v}, R_1^{w}, R_1^{\rho E}, R_1^{\tilde{\nu}}, R_2^{\rho}, R_2^{u}, R_2^{v}, R_2^{w}, R_2^{\rho E}, R_2^{\tilde{\nu}}, ..., R_{n_C}^{\rho}, R_{n_C}^{u}, R_{n_C}^{v}, R_{n_C}^{w}, R_{n_C}^{\rho E}, R_{n_C}^{\tilde{\nu}}\right]^T,$$

where the subscripts are the cell indices and n_C is the total number of cells. The superscripts for R denote the residual type. The length for the $\boldsymbol{R}$ and $\boldsymbol{w}$ vectors is $6n_C$.

Similarly, in DAFoam,

$$\boldsymbol{w} = \left[u_1, v_1, w_1, p_1, \phi_1, \tilde{\nu}_1, u_2, v_2, w_2, p_2, \phi_2, \tilde{\nu}_2, ..., u_{nC}, v_{nC}, w_{nC}, p_{nC}, \phi_{nC}, \tilde{\nu}_{nC}\right]^T,$$

and

$$\boldsymbol{R} = \left[R_1^u, R_1^v, R_1^w, R_1^p, R_1^\phi, R_1^{\tilde{\nu}}, R_2^u, R_2^v, R_2^w, R_2^p, R_2^\phi, R_2^{\tilde{\nu}}, ..., R_{nC}^u, R_{nC}^v, R_{nC}^w, R_{nC}^p, R_{nC}^\phi, R_{nC}^{\tilde{\nu}}\right]^T.$$

Note that the ϕ vector contains ϕ variables on all the faces of a cell. Since we have mixed cell-centered and face-centered state variables, the length for $\boldsymbol{R}$ and $\boldsymbol{w}$ depends on the mesh topology and is approximately $8n_C$.

In the above examples we use the cell-by-cell ordering for $\boldsymbol{R}$ and $\boldsymbol{w}$; that is, we start from the state variables associated with the first cell and then repeat for all the other cells. An alternative option is to use the state-by-state ordering; that is, start from the first state variable for all the cells and repeat for all the other state variables. Although the total derivatives do not depend on the ordering, we use the cell-by-cell ordering because the incomplete lower and upper preconditioning technique is more effective when using block matrices.

As mentioned in Sec. 4, the $\boldsymbol{x}$ vector can contain both aerodynamic and geometric design variables. For aerodynamic variables, the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$ is computed directly. However, for geometric design variables, we first compute the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}_v$, which is done by replacing $\boldsymbol{x}$ with $\boldsymbol{x}_v$ in Eq. (13), and then relate it to the design variables by using the chain rule,

$$\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} = \frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}_v}\frac{\mathrm{d}\boldsymbol{x}_v}{\mathrm{d}\boldsymbol{x}_s}\frac{\mathrm{d}\boldsymbol{x}_s}{\mathrm{d}\boldsymbol{x}}, \tag{A.1}$$

where $\boldsymbol{x}_v$ and $\boldsymbol{x}_s$ are the vectors for volume and design surface coordinates, respectively. The derivatives $\mathrm{d}\boldsymbol{x}_s/\mathrm{d}\boldsymbol{x}$ and $\mathrm{d}\boldsymbol{x}_v/\mathrm{d}\boldsymbol{x}_s$ are computed by two external open-source geometry modules: pyGeo[6] and IDWarp.[7] pyGeo parameterizes and manipulates the design surface by using a free-form deformation approach [188]. IDWarp implements an efficient analytic inverse-distance method [189] for volume mesh deformation. pyGeo was described in detail in our previous work [188].

The benefit of computing $\mathrm{d}f/\mathrm{d}\boldsymbol{x}_v$ instead of $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$ is that the adjoint implementation is independent of external geometric parametrization and mesh deformation modules [190,191]. ADflow (all three approaches) and DAFoam (Jacobian-free approach) use the above strategy. However, for DAFoam with the FD Jacobian approach, we compute $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$ for geometric design variables directly in practice. In other words, instead of using the chain rule in Eq. (A.1), we directly compute $\partial\boldsymbol{R}/\partial\boldsymbol{x}$ and $\partial f/\partial\boldsymbol{x}$ for $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$ [86]. This is because $\partial\boldsymbol{R}/\partial\boldsymbol{x}_v$ is very dense, so computing $\partial\boldsymbol{R}/\partial\boldsymbol{x}_v$ by using a coloring scheme is too expensive.

A.2. Graph coloring implementation

As shown in Figs. 4 and 6, we need to use coloring to accelerate the partial derivative computation. The graph coloring for single block structured meshes is straightforward: Since the mesh topology is fixed, we use analytic colorings for a given residual stencil. This is extended to multiblock structured meshes by treating each of the blocks independently and using the halo cells to handle the cross block boundary derivatives. ADflow uses the above strategy when computing the preconditioner matrix and explicitly storing the full state Jacobian $[\partial\boldsymbol{R}/\partial\boldsymbol{w}]^T$. Fig. A.10(a) shows the sparsity pattern for the state Jacobian in ADflow. We achieve a compact coloring scheme, requiring only 210 colors to compute the state Jacobian $[\partial\boldsymbol{R}/\partial\boldsymbol{w}]^T$ and only 162 colors for the first-order preconditioner matrix $[\partial\boldsymbol{R}/\partial\boldsymbol{w}]^T_{\mathrm{PC}}$. Note that the number of colors is computed based on *all* state variables that are associated with a residual. The details of the analytical coloring are elaborated on in our previous work [55].

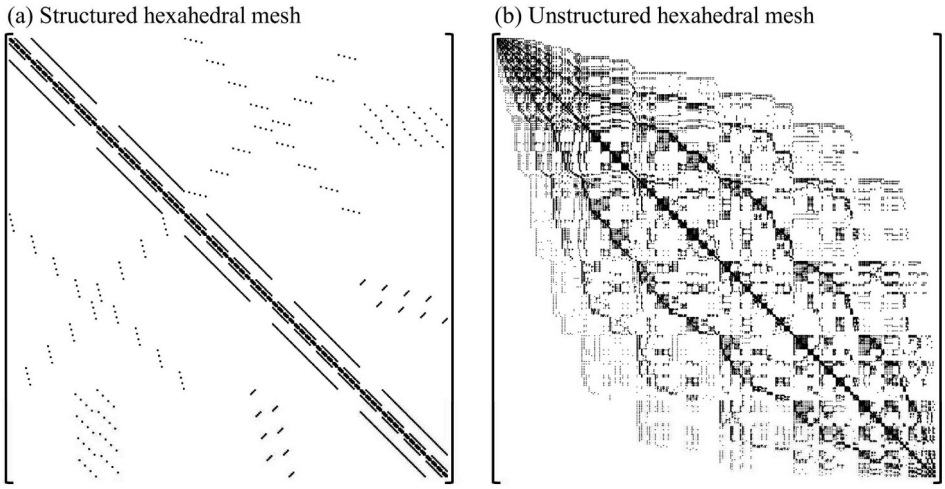

Fig. A.10. Examples of $\partial\boldsymbol{R}/\partial\boldsymbol{w}$ sparsity patterns for (a) ADflow with structured hexahedral mesh and (b) DAFoam with unstructured hexahedral mesh. For DAFoam, the Jacobian matrix is denser because of the larger stencil. Moreover, the sparsity pattern for DAFoam is much more irregular because the use of unstructured meshes.

For unstructured and structured overset meshes, it is no longer possible to use local colorings. The connectivity of unstructured meshes is global, so the entire mesh needs to be colored at one time, creating a much more complex problem without an optimal analytic solution (an NP-hard problem). Fig. A.10(b) shows the sparsity pattern for the state Jacobian in DAFoam. The Jacobian matrix is much denser, owing to the larger stencil for unstructured meshes (see examples in Fig. 1). Moreover, the sparsity pattern is much more irregular, which prevents us from using an analytic

[6] https://github.com/mdolab/pygeo.
[7] https://github.com/mdolab/idwarp.

graph coloring algorithm. Therefore, we adopt a parallel, heuristic graph coloring algorithm that is applicable for any mesh topology and size. The central idea is to tentatively assign colors for local meshes owned by local processors and then resolve the conflicts by globally exchanging data between meshes [86]. The flexibility of the heuristic coloring algorithm comes at a price: the number of colors for $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ is on the order of 1000, which is about three times the maximum number of nonzero entries in a row of $\partial \boldsymbol{R}/\partial \boldsymbol{w}$. The speed of state Jacobian computation is proportional to the number of colors: a larger number of colors leads to a larger computational cost. However, we have shown that the number of colors depends only weakly on mesh size and number of CPU cores [86]. This behavior alleviates the drawback of the relatively large number of colors required by the heuristic graph coloring method. Section 5 evaluates the impact of coloring schemes on the performance of the adjoint derivative.

A.3. Preconditioner matrix construction

As mentioned in Sec. 4.4, having an effective preconditioner is critical to ensure fast convergence of the adjoint equations. We thus need to construct a preconditioner matrix $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$ that approximates $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$ but is easily invertible. To this end, we approximate the residuals and their linearizations with the following solver-specific details:

To construct $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$ in ADflow, we exactly linearize an alternative simple flux formulation that results in only level-one stencils (refer to Fig. 1 for an example of a level-one stencil). The level-one stencil yields a maximum of seven nonzero blocks per row for multiblock meshes and greatly reduces the memory required to store the preconditioner. For the JST scheme, we use only second-order dissipation. This dissipation, which is normally only active at flow discontinuities, is augmented with $\sigma\kappa_4$ [192]. The shock sensor is also frozen for the linearization. If an upwind scheme is used, only first-order reconstruction is employed. For the viscous fluxes, velocity and temperature gradients at the cell face center are approximated by a spatial difference between adjacent cell centers, as described by Yildirim et al. [144]. For the SA turbulence model, we use a first-order upwind method, which also uses only level-one stencils. For overset mesh, due to the complex mesh topology, we compute the preconditioner only on local blocks by using the analytical coloring method mentioned in Appendix A.2; we ignore preconditioner matrix values on any overset stencils.

In DAFoam, we use the first-order upwind scheme to compute the inviscid fluxes (convective terms) to reduce the stiffness of the matrix. We then shrink the residual stencils by reducing the maximal level of connected states for p and ϕ residuals from three to two. We also ignore the turbulence production term to further decrease the linear system stiffness, especially for the viscous layer near the wall when a $y^+ = 1$ mesh is used. Finally, to ensure a better diagonal dominance for both $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$ and $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$, we scale the partial derivatives such that their magnitudes are as close as possible, which is done by normalizing the cell and face residuals by their volume and face area, respectively. We also normalize each column of state Jacobians by their corresponding reference values at the far field [86].

A.4. Differentiation of residuals and functions

We first elaborate on the differentiation of residuals and functions for ADflow. As shown in Fig. 5, before computing the partial derivatives by using AD, we first need to construct a routine to compute $\boldsymbol{R}$ and f, which we call the master routine. This is a collection of functions that reproduces the residual and function computations from the flow solver. The master routine is not used during the actual flow solution; its sole purpose is to serve as a source for AD to produce differentiated code that computes all the required partial derivatives.

The major steps of the master routine are listed in Algorithm 1. The routine computes all intermediate variables that depend on $\boldsymbol{w}$ (e.g., p given T and ρ, and μ using the Sutherland law), updates the boundary conditions (e.g., halo cells through MPI communication, far-field Mach number, and angle of attack), computes the geometric metrics (e.g., cell volume, surface area, surface normal, and wall distance), and then computes $\boldsymbol{R}$ and f. We need to compute and update *all* the intermediate variables and parameters before computing $\boldsymbol{R}$ and f in the master routine, even though some of them may have been set in other routines. This is to make sure the derivatives are properly accumulated through all these intermediate variables and parameters when differentiated with AD.

Algorithm 1 (*master routine*).

1:	function `master`$(\boldsymbol{x}, \boldsymbol{w})$	▷ input: $\boldsymbol{x}$, $\boldsymbol{w}$; output: $\boldsymbol{R}$, f
2:	$\boldsymbol{w}_{\text{int}} \leftarrow F_{\text{int}}(\boldsymbol{x}, \boldsymbol{w})$	▷compute intermediate variables ($\boldsymbol{w}_{\text{int}}$) that depend on $\boldsymbol{w}$ and $\boldsymbol{x}$
3:	$\boldsymbol{w}_{\text{bc}} \leftarrow F_{\text{bc}}(\boldsymbol{x}, \boldsymbol{w})$	▷update variables for boundary faces and halo cells ($\boldsymbol{w}_{\text{bc}}$)
4:	$\boldsymbol{x}_{\text{geo}} \leftarrow F_{\text{geo}}(\boldsymbol{x})$	▷compute geometric metrics such as cell volume and face area ($\boldsymbol{x}_{\text{geo}}$)
5:	$\boldsymbol{R} \leftarrow F_{\text{res}}(\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{w}_{\text{int}}, \boldsymbol{w}_{\text{bc}}, \boldsymbol{x}_{\text{geo}})$	▷compute flow residuals
6:	$f \leftarrow F_{\text{obj}}(\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{w}_{\text{int}}, \boldsymbol{w}_{\text{bc}}, \boldsymbol{x}_{\text{geo}})$	▷compute functions of interests
7:	**return** $\boldsymbol{R}, f$	
8:	**end function**	

We use the source code transformation AD tool Tapenade [174] to differentiate the Fortran master routine. As discussed in Sec. 4.6, we need to differentiate the master routine by using both forward- and reverse-mode AD to implement the Jacobian-free adjoint.

There are two options to differentiate the master routine. We can use Tapenade to directly differentiate the entire master routine. Alternatively, we can use Tapenade to first differentiate all the subroutines in the master routine and then manually assemble these differentiated subroutines. The first option is straightforward to implement but has two significant drawbacks: The first is that this approach leads to sub-optimal differentiated code. Because the AD tool is designed to differentiate any Fortran code, it must be designed to be general. This makes correctness the absolutely highest priority, often at the expense of maximum computational efficiency. It also means that the AD cannot take advantage of any specific code structure related to the CFD code. Furthermore, even with the most advanced AD tools currently available, some code structures remain a challenge to differentiate, including pointers and MPI calls. Therefore, to get a more efficient code, we use the second option. We differentiate the individual subroutines in the master routine and manually assemble routines for the forward and reverse derivatives by using specific properties of the code structure that the AD code cannot, in general, safely assume. This is especially advantageous for the routine used for transpose matrix-vector products in the Jacobian-free version of the adjoint solver, which is because we call this routine multiple times in the GMRES solver, so the speed improvement in the second option outweighs the low implementation effort in the first option.

We call the forward- and reverse-mode master routines master_d and master_b, respectively. A high-efficiency masterfast_b routine is also available to compute just the derivative of the residuals with respect to the states, which allows a reduced set of routines to be used for that computation. The internal structures for these routines are shown in Algorithms 2, 3, and 4. To construct the differentiated master routines, we first use Tapenade to differentiate all the subroutines in the master routine either in forward or reverse mode. We then call these differentiated functions in sequence to compute the necessary derivatives. Note that, in forward mode, the differentiated routines are called in the same order as the master routine whereas, for the reverse-mode computation, the differentiated routines are called in reverse order, as expected. Fig. A.11 shows the inputs and outputs for the original and differentiated master function.

Algorithm 2 (*master_d routine*).

1: **function** `master_d`($\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{w}, \dot{\boldsymbol{w}}$) ▷input: $\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{w}, \dot{\boldsymbol{w}}$; output: $\dot{\boldsymbol{R}}, \dot{f}$
2: $\boldsymbol{w}_{\text{int}}, \dot{\boldsymbol{w}}_{\text{int}} \leftarrow F\_D_{\text{int}}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{w}, \dot{\boldsymbol{w}})$
3: $\boldsymbol{w}_{\text{bc}}, \dot{\boldsymbol{w}}_{\text{bc}} \leftarrow F\_D_{\text{bc}}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{w}, \dot{\boldsymbol{w}})$
4: $\boldsymbol{x}_{\text{geo}}, \dot{\boldsymbol{x}}_{\text{geo}} \leftarrow F\_D_{\text{geo}}(\boldsymbol{x}, \dot{\boldsymbol{x}})$
5: $\dot{\boldsymbol{R}} \leftarrow F\_D_{\text{res}}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{w}, \dot{\boldsymbol{w}}, \boldsymbol{w}_{\text{int}}, \dot{\boldsymbol{w}}_{\text{int}}, \boldsymbol{w}_{\text{bc}}, \dot{\boldsymbol{w}}_{\text{bc}}, \boldsymbol{x}_{\text{geo}}, \dot{\boldsymbol{x}}_{\text{geo}})$
6: $\dot{f} \leftarrow F\_D_{\text{obj}}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{w}, \dot{\boldsymbol{w}}, \boldsymbol{w}_{\text{int}}, \dot{\boldsymbol{w}}_{\text{int}}, \boldsymbol{w}_{\text{bc}}, \dot{\boldsymbol{w}}_{\text{bc}}, \boldsymbol{x}_{\text{geo}}, \dot{\boldsymbol{x}}_{\text{geo}})$
7: **return** $\dot{\boldsymbol{R}}, \dot{f}$
8: **end function**

Algorithm 3 (*master_b routine*).

1: **function** `master_b`($\boldsymbol{x}, \boldsymbol{w}, \bar{\boldsymbol{R}}, \bar{f}$) ▷input: $\boldsymbol{x}, \boldsymbol{w}, \bar{\boldsymbol{R}}, \bar{f}$; output: $\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}}$
2: $\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}}, \bar{\boldsymbol{w}}_{\text{int}}, \bar{\boldsymbol{w}}_{\text{bc}}, \bar{\boldsymbol{x}}_{\text{geo}} \leftarrow F\_B_{\text{obj}}(\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{w}_{\text{int}}, \boldsymbol{w}_{\text{bc}}, \boldsymbol{x}_{\text{geo}}, \bar{f})$
3: $\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}}, \bar{\boldsymbol{w}}_{\text{int}}, \bar{\boldsymbol{w}}_{\text{bc}}, \bar{\boldsymbol{x}}_{\text{geo}} \leftarrow F\_B_{\text{res}}(\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{w}_{\text{int}}, \boldsymbol{w}_{\text{bc}}, \boldsymbol{x}_{\text{geo}}, \bar{\boldsymbol{R}})$
4: $\bar{\boldsymbol{x}} \leftarrow F\_B_{\text{geo}}(\boldsymbol{x}, \bar{\boldsymbol{x}}_{\text{geo}})$
5: $\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}} \leftarrow F\_B_{\text{bc}}(\boldsymbol{x}, \boldsymbol{w}, \bar{\boldsymbol{w}}_{\text{bc}})$
6: $\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}} \leftarrow F\_B_{\text{bc}}(\boldsymbol{x}, \boldsymbol{w}, \bar{\boldsymbol{w}}_{\text{bc}})$
7: **return** $\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}}$
8: **end function**

Algorithm 4 (*masterfast_b routine*).

1: **function** `masterfast_b`($\boldsymbol{w}, \bar{\boldsymbol{R}}$) ▷input: $\boldsymbol{w}, \bar{\boldsymbol{R}}$; output: $\bar{\boldsymbol{w}}$
2: $\bar{\boldsymbol{w}}, \bar{\boldsymbol{w}}_{\text{int}}, \bar{\boldsymbol{w}}_{\text{bc}} \leftarrow F\_B_{\text{res}}(\boldsymbol{w}, \boldsymbol{w}_{\text{int}}, \boldsymbol{w}_{\text{bc}}, \bar{\boldsymbol{R}})$
3: $\bar{\boldsymbol{w}} \leftarrow F\_B_{\text{bc}}(\boldsymbol{w}, \bar{\boldsymbol{w}}_{\text{bc}})$
4: $\bar{\boldsymbol{w}} \leftarrow F\_B_{\text{int}}(\boldsymbol{w}, \bar{\boldsymbol{w}}_{\text{int}})$
5: **return** $\bar{\boldsymbol{w}}$
6: **end function**

The manual implementation described above requires a greater development effort than simply applying AD to the entire master routine. However, we obtain much more efficient code by using this approach than by using the straightforward approach, especially in the reverse-mode case. This efficiency is derived from two main factors: The first factor is that, by manually assembling the derivative routine, we take advantage of the fact that many of the forward pass variables from the residual computation are computed and are up to date from the flow solution and do not need to be recomputed for each derivative computation. This is particularly important when we compute several transpose matrix-vector products in succession at the same solution point, as when doing a Krylov-type solution. In this case, a single forward pass computation can be reused for all of the matrix-vector products.

The second factor is that we can be more selective about the variables that we push onto the AD stack during the reverse pass. There are several variables that, based on careful inspection of the code, we know are not updated between subsequent subroutine calls and can therefore be left off the stack. A general purpose AD routine cannot make this assumption. This allows for a reduction in the number of variables stored during the reverse computation, thereby reducing memory requirements and computational cost. These efficiencies are especially important when solving the adjoint equation by using the Jacobian-free GMRES method because the reverse-mode computation is called repeatedly to compute the transpose matrix-vector products required for each GMRES iteration, as mentioned above.

Fortunately, most of the work involved in implementing this approach occurs in the initial creation of the various master routines. Once the master_d and master_b routines are constructed, they are easy to maintain and extend. All the differentiated subroutines called in the master_d and master_b routines are automatically generated by Tapenade before compiling the adjoint code. Therefore, when we add new terms to the flow residuals, we need only tell Tapenade to differentiate the subroutines that compute this new term and call these newly differentiated subroutines in the master_d and master_b routines to get the correct derivatives.

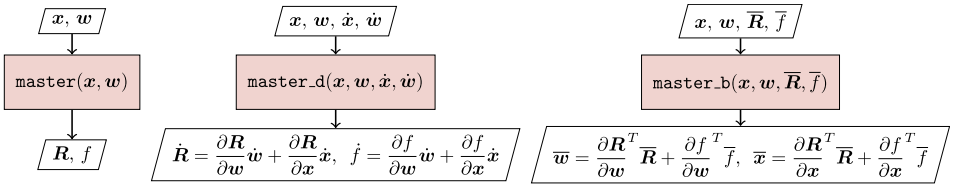

Fig. A.11. Inputs and outputs for the original and differentiated $\boldsymbol{R}$ and f functions in ADflow.

Once the master routine is differentiated with AD, we use the resulting code to compute all the partial derivatives and solve the linear equation. The Jacobian-free option in ADflow is the same as shown in Fig. 6 except that $\dot{\boldsymbol{R}}(\boldsymbol{x}, \boldsymbol{w}, \dot{\boldsymbol{x}}, \dot{\boldsymbol{w}})$ in process 1 is replaced by master_d, $\bar{f}\,(\boldsymbol{x}, \boldsymbol{w}, \bar{f})$ in process 3 and $\bar{\boldsymbol{R}}(\boldsymbol{x}, \boldsymbol{w}, \bar{\boldsymbol{R}})$ in process 7 are replaced by master_b, and $\bar{\boldsymbol{R}}(\boldsymbol{x}, \boldsymbol{w}, \bar{\boldsymbol{R}})$ in process 5 is replaced by masterfast_b. The procedure then consists of the following major steps:

1. Set multiple rows associated with the first color to ones for the input vector $\dot{\boldsymbol{w}}$ and use master_d to compute the partial derivatives for these columns. Move to the next color until the partial derivatives for all the columns are computed. Assemble all the columns to construct the full preconditioner matrix $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$.
2. Set the input vector $\bar{f} = 1$ and use master_b to compute the partial derivative $[\partial f/\partial \boldsymbol{w}]^T$.
3. Set $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{PC}$ as preconditioner matrix and $[\partial f/\partial \boldsymbol{w}]^T$ as right-hand-side vector for the GMRES solver. Set $\boldsymbol{r}_0$ as input vector and use masterfast_b to compute the matrix-vector products $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T \boldsymbol{r}_0$ for the GMRES iteration. Solve the adjoint linear equation until converged and output $\boldsymbol{\psi}$.
4. Set $\bar{\boldsymbol{R}} = \boldsymbol{\psi}$ and $\bar{f} = 1$ as the input seeds for master_b to compute the total derivative $\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} = \frac{\partial f}{\partial \boldsymbol{x}} - \boldsymbol{\psi}^T \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}$.

The FD Jacobian and AD Jacobian in ADflow are modified versions of Fig. 6 with two exceptions: First, in FD Jacobian we compute $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$ and $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{\mathrm{PC}}$ by using the finite-difference method, whereas in AD Jacobian we use the forward-mode AD. Both approaches use a coloring scheme. Second, instead of using the Jacobian-free GMRES approach, we store the $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T$ matrix and explicitly pass it to the GMRES solver to compute the matrix-vector products.

We now elaborate on the differentiation of residuals and functions for DAFoam. The construction of the differentiated $\boldsymbol{R}$ and f has two major differences compared with ADflow. First, no single master routine is constructed to compute all flow residuals and objective functions, primarily because DAFoam solves the flow in a segregated manner, as mentioned in Sec. 3.3. Therefore, we reuse the original OpenFOAM codes and construct multiple residual and objective computation functions. In other words, we construct $\boldsymbol{R}_U(\boldsymbol{x}, \boldsymbol{w})$, $\boldsymbol{R}_p(\boldsymbol{x}, \boldsymbol{w})$, $\boldsymbol{R}_\phi(\boldsymbol{x}, \boldsymbol{w})$, $\boldsymbol{R}_{\tilde{\nu}}(\boldsymbol{x}, \boldsymbol{w})$, and $f(\boldsymbol{x}, \boldsymbol{w})$ separately. These functions are similar to the master routine, where we need to update all intermediate variables, boundary conditions, and geometric metrics before computing the flow residuals and objective functions. Fig. A.12 shows as an example the inputs and outputs for the differentiated $\boldsymbol{R}_U$ and f functions. Second, we use an operator-overloading AD tool to differentiate the above functions because OpenFOAM is written in C++ with complex objective-oriented code structures. Moreover, to generalize solver development, OpenFOAM uses templates extensively. Given the above two features, it is impractical for a source code transformation tool to differentiate an OpenFOAM function.

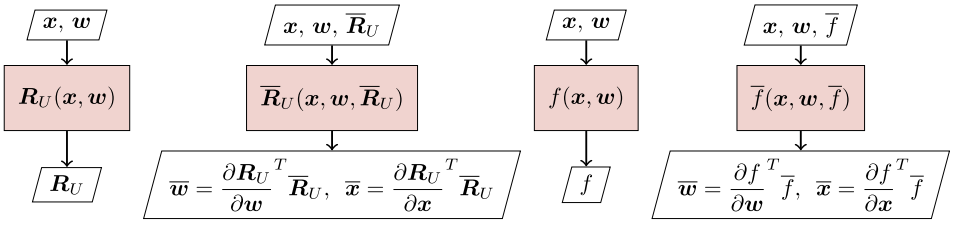

Fig. A.12. Inputs and outputs for the original and differentiated $\boldsymbol{R}$ and f functions in DAFoam.

We build our discrete adjoint solver based on an OpenFOAM version developed by Towara and Naumann [22], who differentiated OpenFOAM by using the operator-overloading tool dco/c++. As mentioned in Sec. 4.5, one of their adjoint solvers (adjointSimpleShapeCheckpointingFoam) computes the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$ by using AD to differentiate the entire primal solver. In Sec. 5, we use adjointSimpleShapeCheckpointingFoam to compare the AD Jacobian, FD Jacobian, Jacobian free, and full-code AD options. Note that we do not use the full-code AD option for our adjoint solvers. Instead, we selectively differentiate parts of the code (e.g., the residual and objective computation functions). Since OpenFOAM was differentiated by using operator-overloading AD, it may be straightforwardly applied to any function without generating new differentiated code. Given this feature, differentiating the flow residual and objective computation functions requires much less development effort compared with manually constructing the master routine and their differentiated versions in ADflow. Moreover, it may be straightforwardly extended to include new terms or governing equations. However, the convenience of operator overloading comes at a price: Its computational cost and memory usage exceed that of the adjoint codes generated by source code transformation. Section 5 compares results for these options.

Overall, the implementation of Jacobian free is similar to that shown in Fig. 6 with one exception: we compute $[\partial \boldsymbol{R}/\partial \boldsymbol{w}]^T_{\mathrm{PC}}$ by using finite-difference instead of forward-mode AD. The FD Jacobian in DAFoam is similar to that shown in Fig. 4, except that we directly compute $\partial \boldsymbol{R}/\partial \boldsymbol{x}$ and $\partial f/\partial \boldsymbol{x}$ by using brute-force finite differencing and then compute $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$, as discussed in Appendix A.1.

References

[1] Y. Yu, Z. Lyu, Z. Xu, J.R.R.A. Martins, On the influence of optimization algorithm and starting design on wing aerodynamic shape optimization, Aero. Sci. Technol. 75 (2018) 183–199, https://doi.org/10.1016/j.ast.2018.01.016.

[2] Z. Lyu, Z. Xu, J.R.R.A. Martins, Benchmarking optimization algorithms for wing aerodynamic design optimization, Proceedings of the 8th International Conference on Computational Fluid Dynamics, Chengdu, Sichuan, China, 2014 ICCFD8-2014-0203.

[3] Z. Lyu, G.K.W. Kenway, J.R.R.A. Martins, Aerodynamic shape optimization investigations of the Common Research Model wing benchmark, AIAA J. 53 (2015) 968–985, https://doi.org/10.2514/1.J053318.

[4] G. Carrier, D. Destarac, A. Dumont, M. Méheut, I.S.E. Din, J. Peter, S.B. Khelil, J. Brezillon, M. Pestana, Gradient-based aerodynamic optimization with the *els*A software, 52nd Aerospace Sciences Meeting, 2014, https://doi.org/10.2514/6.2014-0568 aIAA 2014-0568.

[5] S.T. LeDoux, J.C. Vassberg, D.P. Young, S. Fugal, D. Kamenetskiy, W.P. Huffman, R.G. Melvin, M.F. Smith, Study based on the AIAA aerodynamic design optimization discussion group test cases, AIAA J. 53 (2015) 1910–1935, https://doi.org/10.2514/1.j053535.

[6] S. Chen, Z. Lyu, G.K.W. Kenway, J.R.R.A. Martins, Aerodynamic shape optimization of the Common Research Model wing-body-tail configuration, J. Aircr. 53 (2016) 276–293, https://doi.org/10.2514/1.C033328.

[7] M.B. Giles, N.A. Pierce, An introduction to the adjoint approach to design, Flow, Turbul. Combust. 65 (2000) 393–415.

[8] J.E.V. Peter, R.P. Dwight, Numerical sensitivity analysis for aerodynamic optimization: a survey of approaches, Computers and Fluids 39 (2010) 373–391, https://doi.org/10.1016/j.compfluid.2009.09.013.

[9] J. Elliott, J. Peraire, Practical three-dimensional aerodynamic design and optimization using unstructured meshes, AIAA J. 35 (1997) 1479–1485.

[10] E.J. Nielsen, W.K. Anderson, Aerodynamic design optimization on unstructured meshes using the Navier–Stokes equations, AIAA J. 37 (1999) 1411–1419.

[11] H.-J. Kim, D. Sasaki, S. Obayashi, K. Nakahashi, Aerodynamic optimization of supersonic transport wing using unstructured adjoint method, AIAA J. 39 (2001) 1011–1020.

[12] M. Nemec, M. Aftosmis, S. Murman, T. Pulliam, Adjoint formulation for an embedded-boundary Cartesian method, 43rd AIAA Aerospace Sciences Meeting and Exhibit, 2005, p. 877.

[13] D.J. Mavriplis, Multigrid solution of the discrete adjoint for optimization problems on unstructured meshes, AIAA J. 44 (2006) 42–50.

[14] J. Peter, Discrete adjoint method in *els*A (part I): method/theory, 7th ONERA-DLR Aerospace Symposium, 2006.

[15] R.P. Dwight, J. Brezillon, Effect of approximations of the discrete adjoint on gradient-based optimization, AIAA J. 44 (2006) 3022–3031.

[16] J.E. Hicken, D.W. Zingg, Aerodynamic optimization algorithm with integrated geometry parameterization and mesh movement, AIAA J. 48 (2010) 400–413, https://doi.org/10.2514/1.44033.

[17] A. Griewank, Evaluating Derivatives, SIAM, Philadelphia, 2000.

[18] J.R.R.A. Martins, J.T. Hwang, Review and unification of methods for computing derivatives of multidisciplinary computational models, AIAA J. 51 (2013) 2582–2599, https://doi.org/10.2514/1.J052184.

[19] B. Mohammadi, A new optimal shape design procedure for inviscid and viscous turbulent flows, Int. J. Numer. Methods Fluids 25 (1997) 183–203.

[20] M. Fagan, A. Carle, Reducing reverse-mode memory requirements by using profile-driven checkpointing, Future Gener. Comput. Syst. 21 (2005) 1380–1390.

[21] S. Schlenkrich, A. Walther, N.R. Gauger, R. Heinrich, Differentiating fixed point iterations with ADOL-C: gradient calculation for fluid dynamics, Modeling, Simulation and Optimization of Complex Processes, Springer, 2008, pp. 499–508.

[22] M. Towara, U. Naumann, A discrete adjoint model for OpenFOAM, Proc. Compute. Sci 18 (2013) 429–438.

[23] B. Christianson, Reverse accumulation and attractive fixed points, Optim. Methods Software 3 (1994) 311–326.

[24] A. Griewank, C. Faure, Piggyback differentiation and optimization, Large-scale PDE-Constrained Optimization, Springer, 2003, pp. 148–164.

[25] M.B. Giles, M.C. Duta, J.-D. Müller, N.A. Pierce, Algorithm developments for discrete adjoint methods, AIAA J. 41 (2003) 198–205.

[26] S. Xu, D. Radford, M. Meyer, J.-D. Müller, Stabilisation of discrete steady adjoint solvers, J. Comput. Phys. 299 (2015) 175–195, https://doi.org/10.1016/j.jcp.2015.06.036.

[27] J.-D. Müller, O. Mykhaskiv, J. Hückelheim, STAMPS: a finite-volume solver framework for adjoint codes derived with source-transformation AD, 2018 Multidisciplinary Analysis and Optimization Conference, 2018, https://doi.org/10.2514/6.2018-2928.

[28] T.A. Albring, M. Sagebaum, N.R. Gauger, Efficient aerodynamic design using the discrete adjoint method in SU2, 17th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, 2016, p. 3518.

[29] T.D. Economon, F. Palacios, S.R. Copeland, T.W. Lukaczyk, J.J. Alonso, SU2: an open-source suite for multiphysics simulation and design, AIAA J. 54 (2016) 828–846, https://doi.org/10.2514/1.j053813.

[30] C.A. Mader, J.R.R.A. Martins, J.J. Alonso, E. van der Weide, ADjoint: an approach for the rapid development of discrete adjoint solvers, AIAA J. 46 (2008) 863–873, https://doi.org/10.2514/1.29123.

[31] J.R.R.A. Martins, P. Sturdza, J.J. Alonso, The complex-step derivative approximation, ACM Trans. Math Software 29 (2003) 245–262, https://doi.org/10.1145/838250.838251.

[32] Y. Gao, Y. Wu, J. Xia, Automatic differentiation based discrete adjoint method for aerodynamic design optimization on unstructured meshes, Chin. J. Aeronaut. 30 (2017) 611–627.

[33] C.B. Dilgen, S.B. Dilgen, D.R. Fuhrman, O. Sigmund, B.S. Lazarov, Topology optimization of turbulent flows, Comput. Methods Appl. Mech. Eng. 331 (2018) 363–393.

[34] M. Towara, Discrete Adjoint Optimization with OpenFOAM, Ph.D. Thesis, RWTH Aachen University, 2018.

[35] M. Nemec, D.W. Zingg, Newton–Krylov algorithm for aerodynamic design using the Navier–Stokes equations, AIAA J. 40 (2002) 1146–1154.

[36] R.P. Dwight, J. Brezillon, Efficient and robust algorithms for solution of the adjoint compressible Navier–Stokes equations with applications, Int. J. Numer. Methods Fluids 60 (2009) 365–389, https://doi.org/10.1002/fld.1894.

[37] E.J. Nielsen, W.L. Kleb, Efficient construction of discrete adjoint operators on unstructured grids using complex variables, AIAA J. 44 (2006) 827–836, https://doi.org/10.2514/1.15830.

[38] D.J. Mavriplis, Discrete adjoint-based approach for optimization problems on three-dimensional unstructured meshes, AIAA J. 45 (2007) 741–750, https://doi.org/10.2514/1.22743.

[39] G.K.W. Kenway, C.A. Mader, P. He, J.R.R.A. Martins, CFD Discrete Adjoint Benchmarks, Mendeley Data (2019), https://doi.org/10.17632/w4hsj8wzm8.

[40] A.E. Bryson, Y.C. Ho, Applied Optimal Control; Optimization, Estimation, and Control, Blaisdell Publishing, 1969.

[41] J.L. Lions, Optimal Control of Systems Governed by Partial Differential Equations, Springer–Verlag, New York, 1971.

[42] E.J. Haug, J.S. Arora, Optimal mechanical design techniques based on optimal control methods, Proceedings of the 1st ASME Design Technology Transfer Conference, New York, NY, 1974, pp. 65–74.

[43] J.S. Arora, E.J. Haug, Efficient optimal design of structures by generalized steepest descent programming, Int. J. Numer. Methods Eng. 10 (1976) 747–766.

[44] O. Pironneau, On optimum profiles in Stokes flow, J. Fluid Mech. 59 (1973) 117–128, https://doi.org/10.1017/S002211207300145X.

[45] A. Jameson, Aerodynamic design via control theory, J. Sci. Comput. 3 (1988) 233–260.

[46] A. Jameson, L. Martinelli, N.A. Pierce, Optimum aerodynamic design using the Navier–Stokes equations, Theor. Comput. Fluid Dynam. 10 (1998) 213–237, https://doi.org/10.1007/s001620050060.

[47] A. Jameson, A perspective on computational algorithms for aerodynamic analysis and design, Prog. Aero. Sci. 37 (2001) 197–243, https://doi.org/10.1016/S0376-0421(01)00004-5.

[48] A. Jameson, Aerodynamic Shape Optimization Using the Adjoint Method, Lecture Series, Von Karman Institute for Fluid Dynamics, Rode Saint Genèse, Belgium, 2003.

[49] M.M. Tekitek, M. Bouzidi, F. Dubois, P. Lallemand, Adjoint lattice Boltzmann equation for parameter identification, Comput. Fluid 35 (2006) 805–813.

[50] K. Yaji, T. Yamada, M. Yoshino, T. Matsumoto, K. Izui, S. Nishiwaki, Topology optimization in thermal-fluid flow using the lattice Boltzmann method, J. Comput. Phys. 307 (2016) 355–377.

[52] I. Cheylan, G. Fritz, D. Rico, P. Sagaut, Shape optimization using the adjoint lattice Boltzmann method for aerodynamic applications, AIAA J. (2019), https://doi.org/10.2514/1.J057955 In press.

[53] W.K. Anderson, D.L. Bonhaus, Aerodynamic design on unstructured grids for turbulent flows, NASA TM-112867, NASA Langley Research Center, Hampton, Virginia, 1997 23681-0001.

[54] E.J. Nielsen, B. Diskin, Discrete adjoint-based design for unsteady turbulent flows on dynamic overset unstructured grids, AIAA J. 51 (2013) 1355–1373, https://doi.org/10.2514/1.J051859.

[55] Z. Lyu, G.K. Kenway, C. Paige, J.R.R.A. Martins, Automatic differentiation adjoint of the Reynolds-averaged Navier–Stokes equations with a turbulence model, 21st AIAA Computational Fluid Dynamics Conference, San Diego, CA, 2013, https://doi.org/10.2514/6.2013-2581.

[56] I. Kavvadias, E. Papoutsis-Kiachagias, G. Dimitrakopoulos, K. Giannakoglou, The continuous adjoint approach to the k-omega SST turbulence model with applications in shape optimization, Eng. Optim. 47 (2015) 1523–1542.

[57] P. He, C.A. Mader, J.R.R.A. Martins, K.J. Maki, An object-oriented framework for rapid discrete adjoint development using OpenFOAM, AIAA Science and Technology Forum and Exposition, 2019, https://doi.org/10.2514/6.2019-1210.

[58] J. Driver, D.W. Zingg, Numerical aerodynamic optimization incorporating laminar-turbulent transition prediction, AIAA J. 45 (2007) 1810–1818, https://doi.org/10.2514/1.23569.

[59] Z. Yang, D.J. Mavriplis, Discrete adjoint formulation for turbulent flow problems with transition modelling on unstructured meshes, AIAA SciTech Forum, San Diego, CA, 2019, https://doi.org/10.2514/6.2019-0294.

[60] J.P. Thomas, K.C. Hall, E.H. Dowell, Discrete adjoint approach for modeling unsteady aerodynamic design sensitivities, AIAA J. 43 (2005) 1931, https://doi.org/10.2514/1.731.

[61] C.A. Mader, J.R.R.A. Martins, Derivatives for time-spectral computational fluid dynamics using an automatic differentiation adjoint, AIAA J. 50 (2012) 2809–2819, https://doi.org/10.2514/1.J051658.

[62] C. Ma, X. Su, X. Yuan, An efficient unsteady adjoint optimization system for multistage turbomachinery, J. Turbomach. 139 (2017) 011003.

[63] A. Rubino, M. Pini, P. Colonna, T. Albring, S. Nimmagadda, T. Economon, J. Alonso, Adjoint-based fluid dynamic design optimization in quasi-periodic unsteady flow problems using a harmonic balance method, J. Comput. Phys. 372 (2018) 220–235.

[64] S. He, E. Jonsson, C.A. Mader, J.R.R.A. Martins, Aerodynamic shape optimization

with time spectral flutter adjoint, 2019 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, American Institute of Aeronautics and Astronautics, San Diego, CA, 2019, , https://doi.org/10.2514/6.2019-0697.
[65] C.A. Mader, J.R.R.A. Martins, Computation of aircraft stability derivatives using an automatic differentiation adjoint approach, AIAA J. 49 (2011) 2737–2750, https://doi.org/10.2514/1.J051147.
[66] C.A. Mader, J.R.R.A. Martins, Stability-constrained aerodynamic shape optimization of flying wings, J. Aircr. 50 (2013) 1431–1449, https://doi.org/10.2514/1.C031956.
[67] P. Luchini, A. Bottaro, Adjoint equations in stability analysis, Annu. Rev. Fluid Mech. 46 (2014).
[68] J. Reuther, A. Jameson, J. Farmer, L. Martinelli, D. Saunders, Aerodynamic shape optimization of complex aircraft configurations via an adjoint formulation, Proceedings of the 34th AIAA Aerospace Sciences Meeting and Exhibit, AIAA, Reno, Nevada, 19961996-0094.
[69] M. Drela, Frontiers of Computational Fluid Dynamics, World Scientific, 1998, pp. 363–381, https://doi.org/10.1142/9789812815774_0019.
[70] J.J. Reuther, A. Jameson, J.J. Alonso, M.J. Rimlinger, D. Saunders, Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers, part 1, J. Aircr. 36 (1999) 51–60.
[71] J.J. Reuther, A. Jameson, J.J. Alonso, M.J. Rimlinger, D. Saunders, Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers, part 2, J. Aircr. 36 (1999) 61–74.
[72] S.E. Cliff, J.J. Reuther, D.A. Saunders, R.M. Hicks, Single-point and multipoint aerodynamic shape optimization of high-speed civil transport, J. Aircr. 38 (2001) 997–1005.
[73] M. Nemec, D.W. Zingg, T.H. Pulliam, Multipoint and multi-objective aerodynamic shape optimization, AIAA J. 42 (2004) 1057–1065.
[74] G.K.W. Kenway, J.R.R.A. Martins, Multipoint aerodynamic shape optimization investigations of the Common Research Model wing, AIAA J. 54 (2016) 113–128, https://doi.org/10.2514/1.J054154.
[75] R.P. Liem, J.R.R.A. Martins, G.K. Kenway, Expected drag minimization for aerodynamic design optimization based on aircraft operational data, Aero. Sci. Technol. 63 (2017) 344–362, https://doi.org/10.1016/j.ast.2017.01.006.
[76] Z. Lyu, J.R.R.A. Martins, Aerodynamic design optimization studies of a blended-wing-body aircraft, J. Aircr. 51 (2014) 1604–1617, https://doi.org/10.2514/1.C032491.
[77] N.R. Secco, J.R.R.A. Martins, RANS-based aerodynamic shape optimization of a strut-braced wing with overset meshes, J. Aircr. 56 (2019) 217–227, https://doi.org/10.2514/1.C034934.
[78] T. Dhert, T. Ashuri, J.R.R.A. Martins, Aerodynamic shape optimization of wind turbine blades using a Reynolds-averaged Navier–Stokes model and an adjoint method, Wind Energy 20 (2017) 909–926, https://doi.org/10.1002/we.2070.
[79] M.H.A. Madsen, F. Zahle, N.N. Sørensen, J.R.R.A. Martins, Multipoint high-fidelity CFD-based aerodynamic shape optimization of a 10 MW wind turbine, Wind. Energy. Sci. 4 (2019) 163–192, https://doi.org/10.5194/wes-4-163-2019.
[80] D. Wang, L. He, Adjoint aerodynamic design optimization for blades in multistage turbomachines—part I: methodology and verification, J. Turbomach. 132 (2010) 021011.
[81] D. Wang, L. He, Y. Li, R. Wells, Adjoint aerodynamic design optimization for blades in multistage turbomachines—part II: validation and application, J. Turbomach. 132 (2010) 021012.
[82] N. Garg, G.K.W. Kenway, Z. Lyu, J.R.R.A. Martins, Y.L. Young, High-fidelity hydrodynamic shape optimization of a 3-D hydrofoil, J. Ship Res. 59 (2015) 209–226, https://doi.org/10.5957/JOSR.59.4.150046.
[83] J. Kröger, N. Kühl, T. Rung, Adjoint volume-of-fluid approaches for the hydrodynamic optimisation of ships, Ship Technol. Res. 65 (2018) 47–68.
[84] P. He, G. Filip, J.R.R.A. Martins, K.J. Maki, Design Optimization for Self-Propulsion of a Bulk Carrier Hull Using a Discrete Adjoint Method, Comput. Fluid 192 (2019) https://doi.org/10.1016/j.compfluid.2019.104259 .
[85] C. Othmer, Adjoint methods for car aerodynamics, J. Maths. Indus. 4 (2014) 6, https://doi.org/10.1186/2190-5983-4-6.
[86] P. He, C.A. Mader, J.R.R.A. Martins, K.J. Maki, An aerodynamic design optimization framework using a discrete adjoint approach with OpenFOAM, Comput. Fluids 168 (2018) 285–303, https://doi.org/10.1016/j.compfluid.2018.04.012.
[87] E. Papoutsis-Kiachagias, V. Asouti, K. Giannakoglou, K. Gkagkas, S. Shimokawa, E. Itakura, Multi-point aerodynamic shape optimization of cars based on continuous adjoint, Struct. Multidiscip. Optim. (2018) 1–20.
[88] S. Xu, W. Jahn, J.-D. Müller, CAD-based shape optimisation with CFD using a discrete adjoint, Int. J. Numer. Methods Fluids 74 (2014) 153–168.
[89] T. Verstraete, L. Müller, J.-D. Müller, Adjoint-based design optimisation of an internal cooling channel U-bend for minimised pressure losses, Int. J. Turbomachine. Propul. Power. 2 (2017) 10.
[90] P. He, C.A. Mader, J.R.R.A. Martins, K.J. Maki, Aerothermal optimization of internal cooling passages using a discrete adjoint method, AIAA/ASME Joint Thermophysics and Heat Transfer Conference, Atlanta, GA, 2018.
[91] C. Othmer, A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows, Int. J. Numer. Methods Fluids 58 (2008) 861–877, https://doi.org/10.1002/fld.1770.
[92] E. Papoutsis-Kiachagias, K. Giannakoglou, Continuous adjoint methods for turbulent flows, applied to shape and topology optimization: industrial applications, Arch. Comput. Methods Eng. 23 (2016) 255–299.
[93] L. Kedward, A. Payot, T. Rendall, C. Allen, Efficient multi-resolution approaches for exploration of external aerodynamic shape and topology, AIAA AVIATION Forum, 2018.
[94] E. Lee, K.A. James, J.R.R.A. Martins, Stress-constrained topology optimization with design-dependent loading, Struct. Multidiscip. Optim. 46 (2012) 647–661, https://doi.org/10.1007/s00158-012-0780-x.
[95] G.J. Kennedy, J.R.R.A. Martins, A parallel finite-element framework for large-scale gradient-based design optimization of high-performance structures, Finite Elem. Anal. Des. 87 (2014) 56–73, https://doi.org/10.1016/j.finel.2014.04.011.
[96] M.B. Giles, E. Süli, Adjoint methods for PDEs: a posteriori error analysis and postprocessing by duality, Acta Numer. 11 (2002) 145–236.
[97] K.J. Fidkowski, D.L. Darmofal, Review of output-based error estimation and mesh adaptation in computational fluid dynamics, AIAA J. 49 (2011) 673–694.
[98] D.J. Mavriplis, Functional Error Estimation and Control for Time-dependent Problems, Von Karman Institute for Fluid Dynamics, 2015 Lecture Series, (Chapter 2).
[99] B. Lockwood, D.J. Mavriplis, Gradient-based methods for uncertainty quantification in hypersonic flows, Comput. Fluids 85 (2013) 27–38.
[100] D.J. Mavriplis, Adjoint Methods for Uncertainty Quantification, Von Karman Institute for Fluid Dynamics, 2015 Lecture Series, (Chapter 3).
[101] A. Iollo, M. Ferlauto, L. Zannetti, An aerodynamic optimization method based on the inverse problem adjoint equations, J. Comput. Phys. 173 (2001) 87–115.
[102] L. Biegler, G. Biros, O. Ghattas, M. Heinkenschloss, D. Keyes, B. Mallick, L. Tenorio, B.V.B. Waanders, K. Willcox, Y. Marzouk, Large-scale Inverse Problems and Quantification of Uncertainty vol. 712, Wiley Online Library, 2011.
[103] J.R.R.A. Martins, J.J. Alonso, J.J. Reuther, A coupled-adjoint sensitivity analysis method for high-fidelity aero-structural design, Optim. Eng. 6 (2005) 33–62, https://doi.org/10.1023/B:OPTE.0000048536.47956.62.
[104] J.R.R.A. Martins, J.J. Alonso, J.J. Reuther, High-fidelity aerostructural design optimization of a supersonic business jet, J. Aircr. 41 (2004) 523–530, https://doi.org/10.2514/1.11478.
[105] G.K.W. Kenway, G.J. Kennedy, J.R.R.A. Martins, Scalable parallel approach for high-fidelity steady-state aeroelastic analysis and derivative computations, AIAA J. 52 (2014) 935–951, https://doi.org/10.2514/1.J052255.
[106] G.K.W. Kenway, J.R.R.A. Martins, Multipoint high-fidelity aerostructural optimization of a transport aircraft configuration, J. Aircr. 51 (2014) 144–160, https://doi.org/10.2514/1.C032150.
[107] R.P. Liem, G.K.W. Kenway, J.R.R.A. Martins, Multimission aircraft fuel burn minimization via multipoint aerostructural optimization, AIAA J. 53 (2015) 104–122, https://doi.org/10.2514/1.J052940.
[108] T.R. Brooks, G.K.W. Kenway, J.R.R.A. Martins, Benchmark aerostructural models for the study of transonic aircraft wings, AIAA J. 56 (2018) 2840–2855, https://doi.org/10.2514/1.J056603.
[109] P. Zhang, J. Lu, L. Song, Z. Feng, Study on continuous adjoint optimization with turbulence models for aerodynamic performance and heat transfer in turbomachinery cascades, Int. J. Heat Mass Transf. 104 (2017) 1069–1082.
[110] K. Gkaragkounis, E. Papoutsis-Kiachagias, K. Giannakoglou, The continuous adjoint method for shape optimization in conjugate heat transfer problems with turbulent incompressible flows, Appl. Therm. Eng. 140 (2018) 351–362.
[111] P. He, C.A. Mader, J.R.R.A. Martins, K.J. Maki, Aerothermal optimization of a ribbed U-bend cooling channel using the adjoint method, Int. J. Heat Mass Transf. 140 (2019) 152–172, https://doi.org/10.1016/j.ijheatmasstransfer.2019.05.075 .
[112] T.D. Economon, F. Palacios, J. Alonso, A coupled-adjoint method for aerodynamic and aeroacoustic optimization, 12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, 2012, p. 5598.
[113] E. Fabiano, D. Mavriplis, Adjoint-based aeroacoustic design-optimization of flexible rotors in forward flight, J. Am. Helicopter Soc. 62 (2017) 1–17.
[114] J. Thomas, E. Dowell, K.C. Hall, Discrete adjoint method for nonlinear aeroelastic sensitivities for compressible and viscous flows, 54th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, 2013, p. 1860.
[115] J. Thomas, E. Dowell, Discrete adjoint method for aeroelastic design optimization, 15th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, 2014, p. 2298.
[116] J.F. Kiviaho, K. Jacobson, M.J. Smith, G. Kennedy, Application of a time-accurate aeroelastic coupling framework to flutter-constrained design optimization, 2018 Multidisciplinary Analysis and Optimization Conference, 2018, p. 2932.
[117] E. Jonsson, C.A. Mader, G.J. Kennedy, J.R.R.A. Martins, Computational modeling of flutter constraint for high-fidelity aerostructural optimization, 2019 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, American Institute of Aeronautics and Astronautics, San Diego, CA, 2019, , https://doi.org/10.2514/6.2019-2354.
[118] J.T. Hwang, J.R.R.A. Martins, A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives, ACM Trans. Math Software 44 (2018), https://doi.org/10.1145/3182393 Article 37.
[119] J.S. Gray, J.T. Hwang, J.R.R.A. Martins, K.T. Moore, B.A. Naylor, OpenMDAO: an open-source framework for multidisciplinary design, analysis, and optimization, Struct. Multidiscip. Optim. 59 (2019) 1075–1104, https://doi.org/10.1007/s00158-019-02211-z.
[120] A. Ning, D. Petch, Integrated Design of Downwind Land-Based Wind Turbines Using Analytic Gradients, Wind Energy, 2016, pp. 1–17, https://doi.org/10.1002/we.1972.
[121] J.P. Jasa, J.T. Hwang, J.R.R.A. Martins, Open-source coupled aerostructural optimization using Python, Struct. Multidiscip. Optim. 57 (2018) 1815–1827, https://doi.org/10.1007/s00158-018-1912-8.
[122] J.T. Hwang, A. Ning, Large-scale multidisciplinary optimization of an electric aircraft for on-demand mobility, 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Kissimmee, FL, 2018, https://doi.org/10.2514/6.2018-1384.
[123] J.P. Jasa, C.A. Mader, J.R.R.A. Martins, Trajectory optimization of a supersonic air

vehicle with thermal fuel management system, AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Atlanta, GA, 2018, https://doi.org/10.2514/6.2018-3884.
[124] F. Zahle, N.N. Sørensen, M.K. McWilliam, A. Barlas, Computational fluid dynamics-based surrogate optimization of a wind turbine blade tip extension for maximising energy production, Journal of Physics: Conference Series, vol. 1037, The Science of Making Torque from Wind, Milano, Italy, 2018, , https://doi.org/10.1088/1742-6596/1037/4/042013.
[125] J.T. Hwang, J. Jasa, J.R.R.A. Martins, High-fidelity design-allocation optimization of a commercial aircraft maximizing airline profit, J. Aircr. (2019), https://doi.org/10.2514/1.C035082.
[126] J.S. Gray, C.A. Mader, G.K.W. Kenway, J.R.R.A. Martins, Modeling boundary layer ingestion using a coupled aeropropulsive analysis, J. Aircr. 55 (2018) 1191–1199, https://doi.org/10.2514/1.C034601.
[127] J.S. Gray, J.R.R.A. Martins, Coupled aeropropulsive design optimization of a boundary layer ingestion propulsor, Aeronaut. J. 123 (2019) 121–137, https://doi.org/10.1017/aer.2018.120.
[128] W.K. Anderson, V. Venkatakrishnan, Aerodynamic design optimization on unstructured grids with a continuous adjoint formulation, Computers and Fluids 28 (1999) 443–480.
[129] S. Nadarajah, A. Jameson, A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization, Proceedings of the 38th AIAA Aerospace Sciences Meeting and Exhibit, Reno, NV, 2000, https://doi.org/10.2514/6.2000-667.
[130] T.D. Economon, J.J. Alonso, T. Albring, N.R. Gauger, Adjoint formulation investigations of benchmark aerodynamic design cases in SU2, Proceedings of the 35th AIAA Applied Aerodynamics Conference, Denver, CO, 2017.
[131] J.R.R.A. Martins, J.T. Hwang, Multidisciplinary Design Optimization of Aircraft Configurations—Part 1: A Modular Coupled Adjoint Approach, Lecture Series, Von Karman Institute for Fluid Dynamics, Rode Saint Genèse, Belgium, 2016, pp. ISSN0377–8312.
[132] D.J. Lea, M.R. Allen, T.W. Haine, Sensitivity analysis of the climate of a chaotic system, Tellus Dyn. Meteorol. Oceanogr. 52 (2000) 523–532.
[133] Q. Wang, Forward and adjoint sensitivity computation of chaotic dynamical systems, J. Comput. Phys. 235 (2013) 1–13.
[134] P.J. Blonigan, Q. Wang, E.J. Nielsen, B. Diskin, Least-squares shadowing sensitivity analysis of chaotic flow around a two-dimensional airfoil, AIAA J. (2017) 658–672.
[135] P. Spalart, S. Allmaras, A one-equation turbulence model for aerodynamic flows, 30th Aerospace Sciences Meeting and Exhibit, 1992, https://doi.org/10.2514/6.1992-439.
[136] E. van der Weide, G. Kalitzin, J. Schluter, J.J. Alonso, Unsteady turbomachinery computations using massively parallel platforms, Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit, Reno, NV, AIAA, 20062006-0421.
[137] G.K.W. Kenway, N. Secco, J.R.R.A. Martins, A. Mishra, K. Duraisamy, An efficient parallel overset method for aerodynamic shape optimization, Proceedings of the 58th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, AIAA SciTech Forum, Grapevine, TX, 2017, https://doi.org/10.2514/6.2017-0357.
[138] A. Jameson, W. Schmidt, E. Turkel, Numerical Solution of the Euler Equations by Finite Volume Methods Using Runge–Kutta Time Stepping Schemes, (1981) Technical Report AIAA 1981-1259.
[139] E. Turkel, V.N. Vatsa, Effects of artificial viscosity on three-dimensional flow solutions, AIAA J. 32 (1994) 39–45.
[140] B. van Leer, Towards the ultimate conservative difference scheme. V. a second-order sequel to Godunov's method, J. Comput. Phys. 32 (1979) 101–136.
[141] P.L. Roe, Approximate Riemann solvers, parameter vectors, and difference schemes, J. Comput. Phys. 43 (1981) 357–372.
[142] F.R. Menter, Two-equation eddy-viscosity turbulence models for engineering applications, AIAA J. 32 (1994) 1598–1605.
[143] G. Klopfer, C. Hung, R. Van der Wijngaart, J. Onufer, A diagonalized diagonal dominant alternating direction implicit (D3ADI) scheme and subiteration correction, 29th AIAA, Fluid Dynamics Conference, Albuquerque, NM, 1998, https://doi.org/10.2514/6.1998-2824.
[144] A. Yildirim, G.K.W. Kenway, C.A. Mader, J.R.R.A. Martins, A Jacobian-free approximate Newton–Krylov startup strategy for RANS simulations, J. Comput. Phys. (2018), https://doi.org/10.1016/j.jcp.2019.06.018 In press.
[145] X. He, J. Li, C.A. Mader, A. Yildirim, J.R.R.A. Martins, Robust aerodynamic shape optimization—from a circle to an airfoil, Aero. Sci. Technol. 87 (2019) 48–61, https://doi.org/10.1016/j.ast.2019.01.051.
[146] S.V. Patankar, D.B. Spalding, A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows, Int. J. Heat Mass Transf. 15 (1972) 1787–1806, https://doi.org/10.1016/0017-9310(72)90054-3.
[147] C. Rhie, W.L. Chow, Numerical study of the turbulent flow past an airfoil with trailing edge separation, AIAA J. 21 (1983) 1525–1532, https://doi.org/10.2514/3.8284.
[148] J.R.R.A. Martins, G.K.W. Kenway, T.R. Brooks, Multidisciplinary Design Optimization of Aircraft Configurations—Part 2: High-Fidelity Aerostructural Optimization, Lecture Series, Von Karman Institute for Fluid Dynamics, Rode Saint Genèse, Belgium, 2016, pp. ISSN0377–8312.
[149] J.N. Lyness, C.B. Moler, Numerical differentiation of analytic functions, SIAM J. Numer. Anal. 4 (1967) 202–210.
[150] U. Naumann, The Art of Differentiating Computer Programs—An Introduction to Algorithmic Differentiation, SIAM, 2011.
[151] U. Naumann, K. Leppkes, J. Lotz, dco/c++ : Derivative Code by Overloading in C++ Introduction and Summary of Features, RWTH Aachen University, 2011 Technical Report.
[152] M. Sagebaum, T. Albring, N.R. Gauger, High-performance Derivative Computations Using CoDiPack, (2017) 1709.07229.
[153] J.-D. Müller, P. Cusdin, On the performance of discrete adjoint CFD codes using automatic differentiation, Int. J. Numer. Methods Fluids 47 (2005) 939–945.
[154] J. Grabmeier, E. Kaltofen, Computer Algebra Handbook: Foundations, Applications, Systems, Springer Science & Business Media, 2003.
[155] A.H. Gebremedhin, F. Manne, A. Pothen, What color is your Jacobian? Graph coloring for computing derivatives, SIAM Rev. 47 (2005) 629–705, https://doi.org/10.1137/S0036144504444711.
[156] U.M. Ascher, C. Greif, A First Course in Numerical Methods, SIAM, 2011.
[157] Y. Saad, Iterative Methods for Sparse Linear Systems, second ed., SIAM, 2003.
[158] P. Newman, G. Hou, J. Jones, A. Taylor III, V. Korivi, Observations on computational methodologies for use in large-scale, gradient-based, multidisciplinary design, 4th Symposium on Multidisciplinary Analysis and Optimization, 1992.
[159] V.M. Korivi, A.C. Taylor III, P.A. Newman, G.W. Hou, H.E. Jones, An Incremental Strategy for Calculating Consistent Discrete CFD Sensitivity Derivatives, National Aeronautics And Space Admin Langley Research Center Hampton VA, 1992 Technical Report.
[160] M.B. Giles, On the iterative solution of adjoint equations, Automatic Differentiation of Algorithms, Springer, 2002, pp. 145–151.
[161] E.J. Nielsen, J. Lu, M.A. Park, D.L. Darmofal, An implicit, exact dual adjoint solution method for turbulent flows on unstructured grids, Comput. Fluids 33 (2004) 1131–1155.
[162] S. Akbarzadeh, Y. Wang, J.D. Mueller, Fixed point discrete adjoint of SIMPLE-like solvers, 22nd AIAA Computational Fluid Dynamics Conference, 2015, p. 2750.
[163] S. Xu, S. Timme, K.J. Badcock, Enabling off-design linearised aerodynamics analysis using Krylov subspace recycling technique, Comput. Fluids 140 (2016) 385–396, https://doi.org/10.1016/j.compfluid.2016.10.018.
[164] S. Xu, S. Timme, Robust and efficient adjoint solver for complex flow conditions, Comput. Fluids 148 (2017) 26–38, https://doi.org/10.1016/j.compfluid.2017.02.012.
[165] G.K.W. Kenway, J.R.R.A. Martins, Buffet onset constraint formulation for aerodynamic shape optimization, AIAA J. 55 (2017) 1930–1947, https://doi.org/10.2514/1.J055172.
[166] S. Balay, W.D. Gropp, L.C. McInnes, B.F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A.M. Bruaset, H.P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhäuser Press, 1997, pp. 163–202.
[167] S. Balay, K. Buschelman, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith, H. Zhang, PETSc Web page, (2009) http://www.mcs.anl.gov/petsc.
[168] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, D.A. May, L.C. McInnes, R.T. Mills, T. Munson, K. Rupp, P. Sanan, B.F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc Users Manual, Technical Report ANL-95/11 - Revision 3.10, Argonne National Laboratory, 2018, http://www.mcs.anl.gov/petsc.
[169] J. Peter, F. Renac, A. Dumont, M. Méheut, Discrete adjoint method for shape optimization and mesh adaptation in the elsA code, status and challenges, 50h 3AF International Conference on Applied Aerodynamics, 2015.
[170] M.B. Giles, D. Ghate, M.C. Duta, Using automatic differentiation for adjoint CFD code development, Post SAROD Workshop-2005, 2005.
[171] J.E. Hicken, D.W. Zingg, A simplified and flexible variant of GCROT for solving nonsymmetric linear systems, SIAM J. Sci. Comput. 32 (2010) 1672–1694.
[172] L. Osusky, H. Buckley, T. Reist, D.W. Zingg, Drag minimization based on the Navier—Stokes equations using a Newton—Krylov approach, AIAA J. 53 (2015) 1555–1577, https://doi.org/10.2514/1.J053457.
[173] D.J. Mavriplis, Time Dependent Adjoint Methods for Single and Multi-Disciplinary Problems, Von Karman Institute for Fluid Dynamics, 2015 Lecture Series, (Chapter 1).
[174] L. Hascoet, V. Pascual, The Tapenade automatic differentiation tool: principles, model, and specification, ACM Trans. Math Software 39 (2013), https://doi.org/10.1145/2450153.2450158 20:1–20:43.
[175] P.E. Farrell, D.A. Ham, S.W. Funke, M.E. Rognes, Automated derivation of the adjoint of high-level transient finite element programs, SIAM J. Sci. Comput. 35 (2013) C369–C393.
[176] A. Logg, K.-A. Mardal, G. Wells, Automated Solution of Differential Equations by the Finite Element Method: the FEniCS Book vol. 84, Springer Science & Business Media, 2012.
[177] G. Pierrot, Continuous and discrete adjoint methodologies within ESI CFD solvers, Conference on Industrial Design Optimisation for Fluid Flow, 2012.
[178] G. Eggenspieler, ANSYS Fluent Adjoint Solver, ANSYS, 2012.
[179] Star-CCM+, Adjoint solver workshop, STAR South East Asian Conference, 2013.
[180] A.B. Lambe, J.R.R.A. Martins, Extensions to the design structure matrix for the description of multidisciplinary design, analysis, and optimization processes, Struct. Multidiscip. Optim. 46 (2012) 273–284, https://doi.org/10.1007/s00158-012-0763-y.
[181] N. Bons, X. He, C.A. Mader, J.R.R.A. Martins, Multimodality in aerodynamic wing design optimization, AIAA J. 57 (2019) 1004–1018, https://doi.org/10.2514/1.J057294.
[182] M.B. Giles, Collected matrix derivative results for forward and reverse mode algorithmic differentiation, Advances in Automatic Differentiation, Springer, 2008, pp. 35–44.
[183] A. Sen, Effective sensitivity computation for aerodynamic optimization using discrete adjoint OpenFOAM, 19th European Workshop on Automatic Differentiation, 2016.
[184] D.A. Burdette, J.R.R.A. Martins, Design of a transonic wing with an adaptive

morphing trailing edge via aerostructural optimization, Aero. Sci. Technol. 81 (2018) 192–203, https://doi.org/10.1016/j.ast.2018.08.004.
[185] D.A. Burdette, J.R.R.A. Martins, Impact of morphing trailing edge on mission performance for the Common Research Model, J. Aircr. 56 (2019) 369–384, https://doi.org/10.2514/1.C034967.
[186] T.R. Brooks, G.J. Kennedy, J.R.R.A. Martins, High-fidelity multipoint aerostructural optimization of a high aspect ratio tow-steered composite wing, Proceedings of the 58th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, AIAA SciTech Forum, Grapevine, TX, 2017, https://doi.org/10.2514/6.2017-1350.
[187] J.C. Vassberg, M.A. DeHaan, S.M. Rivers, R.A. Wahls, Development of a Common Research Model for Applied CFD Validation Studies, AIAA, 2008, pp. 2008–6919, https://doi.org/10.2514/6.2008-6919.
[188] G.K. Kenway, G.J. Kennedy, J.R.R.A. Martins, A CAD-free approach to high-fidelity aerostructural optimization, Proceedings of the 13th AIAA/ISSMO Multidisciplinary Analysis Optimization Conference, AIAA, 2010, pp. 2010–9231, , https://doi.org/10.2514/6.2010-9231 Fort Worth, TX.
[189] E. Luke, E. Collins, E. Blades, A fast mesh deformation method using explicit interpolation, J. Comput. Phys. 231 (2012) 586–601, https://doi.org/10.1016/j.jcp.2011.09.021.
[190] E.J. Nielsen, M.A. Park, Using an adjoint approach to eliminate mesh sensitivities in computational design, AIAA J. 44 (2006) 948–953.
[191] M. Widhalm, J. Brezillon, C. Ilic, T. Leicht, Investigation on adjoint based gradient computations for realistic 3D aero-optimization, 13th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, September, Fort Worth, TX, 2010, pp. 1–17.
[192] J.E. Hicken, D. Zingg, A parallel Newton–Krylov solver for the Euler equations discretized using simultaneous approximation terms, AIAA J. 46 (2008) 2773–2786.