This is a preprint of the following article, which is available from `http://mdolab.engin.umich.edu`
Ping He, Charles A. Mader, Joaquim R. R. A. Martins, and Kevin J. Maki. An aerodynamic design optimization framework using a discrete adjoint approach with OpenFOAM. *Computers & Fluids*, 168 (2018) 285–303. `doi: 10.1016/j.compfluid.2018.04.012`
The published article may differ from this preprint.

An Aerodynamic Design Optimization Framework Using a Discrete Adjoint Approach with OpenFOAM

Ping He[1], Charles A. Mader[1], Joaquim R. R. A. Martins[1], and Kevin J. Maki[2]

[1] *Department of Aerospace Engineering, University of Michigan, Ann Arbor*
[2] *Department of Naval Architecture and Marine Engineering, University of Michigan, Ann Arbor*

Abstract Advances in computing power have enabled computational fluid dynamics (CFD) to become a crucial tool in aerodynamic design. To facilitate CFD-based design, the combination of gradient-based optimization and the adjoint method for computing derivatives can be used to optimize designs with respect to a large number of design variables. Open field operation and manipulation (OpenFOAM) is an open source CFD package that is becoming increasingly popular, but it currently lacks an efficient infrastructure for constrained design optimization. To address this problem, we develop an optimization framework that consists of an efficient discrete adjoint implementation for computing derivatives and a Python interface to multiple numerical optimization packages. Our adjoint optimization framework has the following salient features: (1) The adjoint computation is efficient, with a computational cost that is similar to that of the primal flow solver and scales up to 10 million cells and 1024 CPU cores. (2) The adjoint derivatives are fully consistent with those generated by the flow solver with an average error of less than 0.1%. (3) The adjoint framework can handle optimization problems with more than 100 design variables and various geometric and physical constraints such as volume, thickness, curvature, and lift constraints. (4) The framework includes additional modules that are essential for successful design optimization: a geometry-parametrization module, a mesh-deformation algorithm, and an interface to numerical optimizations. To demonstrate our design-optimization framework, we optimize the ramp shape of a simple bluff geometry and analyze the flow in detail. We achieve 9.4% drag reduction, which is validated by wind tunnel experiments. Furthermore, we apply the framework to solve two more complex aerodynamic-shape-optimization applications: an unmanned aerial vehicle, and a car. For these two cases, the drag is reduced by 5.6% and 12.1%, respectively, which demonstrates that the proposed optimization framework functions as desired. Given these validated improvements, the developed techniques have the potential to be a useful tool in a wide range of engineering design applications, such as aircraft, cars, ships, and turbomachinery.

Keywords: OpenFOAM, Discrete Adjoint Optimization, Parallel Graph Coloring, Ahmed Body, UAV, Car

1 Introduction

Open field operation and manipulation (OpenFOAM) is an open source software package for computational fluid dynamics (CFD) [1, 2] that contains more than 80 solvers capable of simulating various types of flow processes, including aerodynamics, hydrodynamics, heat transfer, and multiphase flow [3]. OpenFOAM is being actively developed and verified by its users and developers [4–7], and its popularity has been rapidly growing over the past decade. OpenFOAM has become a powerful tool for aerodynamic design of engineering systems such as aircraft, cars, and turbomachinery [8–14]. One of the major tasks in the process of aerodynamic design is to improve system performance (e.g., reduce drag, maximize power, and improve efficiency). Traditionally, it involves manual loops of design modification and performance evaluation, which is not efficient. To improve the efficiency of this process, the combination of gradient-based optimization and the adjoint method for computing derivatives can be used to automatically optimize the design. The

true benefit of using the adjoint method to compute derivatives is that its computational cost is almost independent of the number of design variables, which enables complex industrial design optimization. Given this background, the development of an adjoint optimization framework may facilitate the existing process of OpenFOAM-based aerodynamic-shape design.

The adjoint method was first introduced to fluid mechanics by Pironneau [15] in 1970s. The approach was then extended by Jameson [16] to the optimization of two-dimensional aerodynamic-shape design in the late 1980s. Since then, the adjoint method has been implemented for three-dimensional turbulent flows, and its application has also been generalized to multipoint and multidisciplinary design optimization [17–26]. While the adjoint method is recognized as an efficient method for computing derivatives of a solver based on partial differential equations (PDEs), successful optimization requires a framework that includes other components that go beyond the flow solution and derivative computation. We also require modules for geometry manipulation, mesh deformation, and optimization algorithms. The speed and accuracy of such modules, especially as they pertain to derivative computation, strongly impact the overall optimization. We have developed a full suite of modules to facilitate aerodynamic optimization, some of which have been published previously. The geometry-manipulation module was developed by Kenway *et al.* [27] and has been used in various aerodynamic and aerostructural design-optimization studies [26, 28–31]. Perez *et al.* [32] developed an open source Python interface to various numerical optimization packages that we reuse here.[1] In the present work, we focus on the implementation of the adjoint solver in OpenFOAM, the development of which allows the OpenFOAM solver to be efficiently integrated into our existing optimization framework.

Two different methods exist for formulating the adjoint of a flow solver: continuous and discrete [33]. The continuous approach derives the adjoint formulation from the Navier–Stokes (NS) equations and then discretizes to obtain the numerical solution. In contrast, the discrete approach starts from the discretized NS equations and differentiates the discretized equations to get the adjoint terms. Although these two approaches handle adjoint formulation in different ways, they both converge to the same answer for a sufficiently refined mesh [34].

The adjoint method was first implemented in OpenFOAM by Othmer [35], who used the continuous approach to derive the adjoint formulation for the incompressible flow solver simpleFoam. This continuous adjoint implementation was then integrated as a built-in OpenFOAM solver for computing derivatives. A number of recent studies have reported shape optimization based on derivatives computed from the continuous adjoint [36–40]. Othmer's continuous adjoint framework uses a free-form deformation (FFD) geometry-morphing technique that can handle complex geometries such as full-scale cars. Moreover, the computational cost for the adjoint is similar to that for the primal flow solver, allowing one to tackle cases with more than 10 million cells [38, 39]. However, they used a basic steepest descent optimization algorithm to update the shape, so their optimization problems did not include design constraints.

More recently, Towara and Naumann [41] reported a discrete adjoint implementation for OpenFOAM. They used reverse mode automatic differentiation (AD) to compute derivatives so that the adjoint derivatives are fully consistent with the flow solution, regardless of the mesh refinement. However, they used AD to differentiate the entire OpenFOAM code, requiring all flow variables to be stored to conduct the reverse AD computation. To reduce the memory required to store the flow variables, the checkpointing technique was used to trade speed for memory. As a result, the overall computational cost to compute derivatives is high—the adjoint-flow runtime ratio ranges from 5 to 15 [42–44]. Given the cost of this adjoint computation, it would be hard to use this implementation for practical shape optimization.

Instead of applying AD to the entire code, we implement a discrete adjoint approach where the partial derivatives in the adjoint equations are computed by using finite differences (see Section 2.5). The objective here is to develop an adjoint solver within the limitations of the OpenFOAM framework that is sufficiently efficient for practical shape optimization. We evaluate the performance of our adjoint implementation in terms of speed, scalability, and accuracy, optimize the aerodynamic shape of a bluff geometry representative of a ground vehicle, and validate the optimized result by comparing it with the result of wind tunnel experiments. Furthermore, we demonstrate the constrained optimization capability for two more complex shape-optimization applications: an unmanned aerial vehicle (UAV), and a car. We opt to use the discrete adjoint approach because the adjoint derivative is consistent with the flow solution, as mentioned above. Moreover, we find the discrete adjoint implementation easier to maintain and extend (for example, when

[1] `https://github.com/mdolab/pyoptsparse`

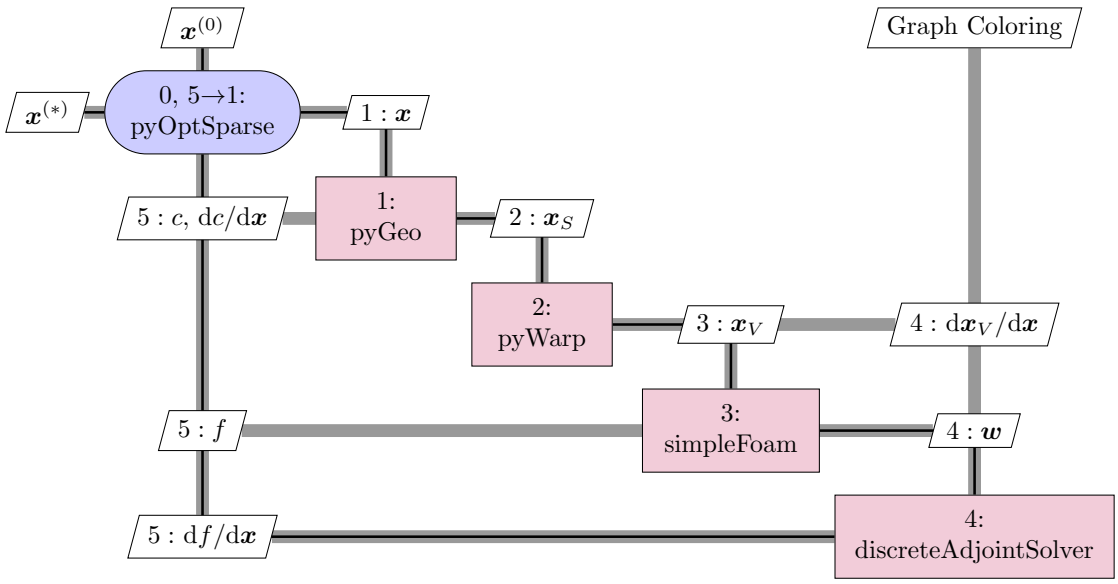

Figure 1: Extended design-structure matrix [47] for discrete adjoint framework for constrained-shape-optimization problems. $\boldsymbol{x}$: design variables; $\boldsymbol{x}^{(0)}$: baseline design variables; $\boldsymbol{x}^{(*)}$: optimized design variables; $\boldsymbol{x}_S$: coordinates of design surface; $\boldsymbol{x}_V$: coordinates of volume mesh; $\boldsymbol{w}$: state variables; c: geometric constraints; f: objective and constraint functions.

adding new objective or constraint functions and boundary conditions).

The rest of the paper is organized as follows: Section 2 introduces the optimization framework along with the theoretical background for each of its modules. Section 3 evaluates its performance and presents the aerodynamic shape optimization results. Finally, we summarize and give conclusions in Section 4.

2 Methodology

The design-optimization framework implements a discrete adjoint for computing the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$, where f is the function of interest (which for optimization will be the objective and constraint functions, e.g., drag, lift, and pitching moment), and $\boldsymbol{x}$ represents the design variables that control the geometric shape via FFD control point movements. The design-optimization framework consists of multiple components written in C++ and Python and depends on the following external libraries and modules: OpenFOAM, portable, extensible toolkit for scientific computation (PETSc) [45, 46], pyGeo [27], pyWarp [27], and pyOptSparse [32]. The framework also requires an external optimization package, which can be any package supported by the pyOptSparse optimization interface. In this section, we elaborate on the overall adjoint optimization framework, the theoretical background for the framework modules, and the code structure and implementation.

2.1 Discrete Adjoint Optimization Framework

Figure 1 shows the modules and data flow for the optimization framework. We use the extended design structure matrix standard developed by Lambe and Martins [47]. 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 thin black lines denote the data and process flow, respectively. The numbers in the entries are their execution order.

The framework consists of two major layers: OpenFOAM and Python, and they interact through input and output files. The OpenFOAM layer consists of a flow solver (simpleFoam), an adjoint solver (discreteAdjointSolver), and a graph-coloring solver (coloringSolver). The flow solver is based on the standard OpenFOAM solver simpleFoam for steady incompressible turbulent flow. The adjoint solver computes the

total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$ based on the flow solution generated by simpleFoam. The mesh deformation derivative matrix ($\mathrm{d}\boldsymbol{x_v}/\mathrm{d}\boldsymbol{x}$, where $\boldsymbol{x_v}$ contains the volume-mesh coordinates) is needed when computing the total derivative and is provided by the Python layer. To accelerate computation of the partial derivatives, we developed a parallel graph-coloring solver, whose algorithm is discussed in Section 2.6.

The Python layer is a high-level interface that takes the user input and the total derivatives computed by the OpenFOAM layer and calls multiple external modules to perform constrained optimization. To be more specific, these external modules include "pyGeo" for the surface-geometry parameterization and computation of geometric constraints c and their derivatives $\mathrm{d}c/\mathrm{d}\boldsymbol{x}$, "pyWarp" for the volume-mesh deformation, and "pyOptSparse" for the optimization setup. In this paper, we use the sparse nonlinear optimizer (SNOPT) package [48], but the pyOptSparse interface provides access to various other optimization algorithms. In addition, the PETSc library is used to efficiently manipulate and store large sparse matrices and vectors and to solve the linear equations. The detailed background for each of these modules is introduced in the following sections.

2.2 Surface Geometry Parameterization—pyGeo

To optimize the shape, one must manipulate the surface of a given geometry. We use a FFD implementation by Kenway *et al.* [27] to parameterize geometries. The FFD approach embeds the geometry into a volume that can then be manipulated by moving points at the surface of that volume (the FFD points). Figure 2(a) shows an example of using the FFD approach to parameterize the surface of a bluff geometry called the Ahmed body [49]. Once an object is embedded into a FFD volume, a Newton search is executed to determine the mapping between the FFD points (parameter space) and the surface geometry (physical space). The FFD volume is a trivariate B-spline volume such that the gradient of any point inside the volume can be easily computed. Note that the FFD volume parameterizes the geometry changes rather than changing the geometry itself, allowing us to choose a more efficient and compact set of design variables.

The geometry parameterization is implemented in the pyGeo module and allows us to control the local shape of the geometry during optimization. Moreover, by moving sets of FFD points together, one can produce rigid motion for surface deformation. This allows us to control the global dimension of a geometry, such as the ramp angle of the Ahmed body, or the twist, sweep, span, and chord of an aircraft wing.

2.3 Volume Mesh Deformation—pyWarp

Once the surface geometry is changed in the optimization process, the corresponding changes need to be applied to the CFD surface mesh. To avoid having negative-volume cells and to maintain mesh quality, we also need to smoothly deform the volume mesh; a process also known as mesh warping or mesh morphing. The mesh-deformation algorithm used in this work is an efficient analytic inverse-distance method similar to that described by Luke *et al.* [50]. The advantage of this approach is that it is highly flexible and can be applied to both structured and unstructured meshes. In addition, compared with the method based on radial basis functions [51], this approach better preserves mesh orthogonality in the boundary layer. Figures 2(b) and 2(c) show examples of the baseline and deformed volume mesh in the symmetry plane for the Ahmed body. The surface-parameterization and mesh-deformation operations are fully parallel and typically require less than 0.1% of the CFD simulation time. Such a speedy mesh deformation is crucial when optimizing, because this operation is called multiple times for each optimization iteration; namely, when the surface geometry is updated and when the mesh-deformation derivative matrix $\mathrm{d}\boldsymbol{x_v}/\mathrm{d}\boldsymbol{x}$ is computed.

2.4 SIMPLE-Algorithm-Based Solver for Steady Incompressible Turbulent Flow—simpleFoam

The standard OpenFOAM solver simpleFoam is used to simulate the steady incompressible turbulent flow by solving the NS equations:

$$\int_S \boldsymbol{U} \cdot \mathrm{d}\boldsymbol{S} = 0, \tag{1}$$

$$\int_S \boldsymbol{U}\boldsymbol{U} \cdot \mathrm{d}\boldsymbol{S} + \int_V \nabla p \,\mathrm{d}V - \int_S (\nu + \nu_t)(\nabla \boldsymbol{U} + \nabla \boldsymbol{U}^T) \cdot \mathrm{d}\boldsymbol{S} = 0, \tag{2}$$

where $\boldsymbol{U} = [u,\ v,\ w]$ is the velocity vector; u, v, and w are the velocity components in the x, y, and z directions, respectively; $\boldsymbol{S}$ is the face-area vector; V is the volume; ν and ν_t are the molecular and turbulent

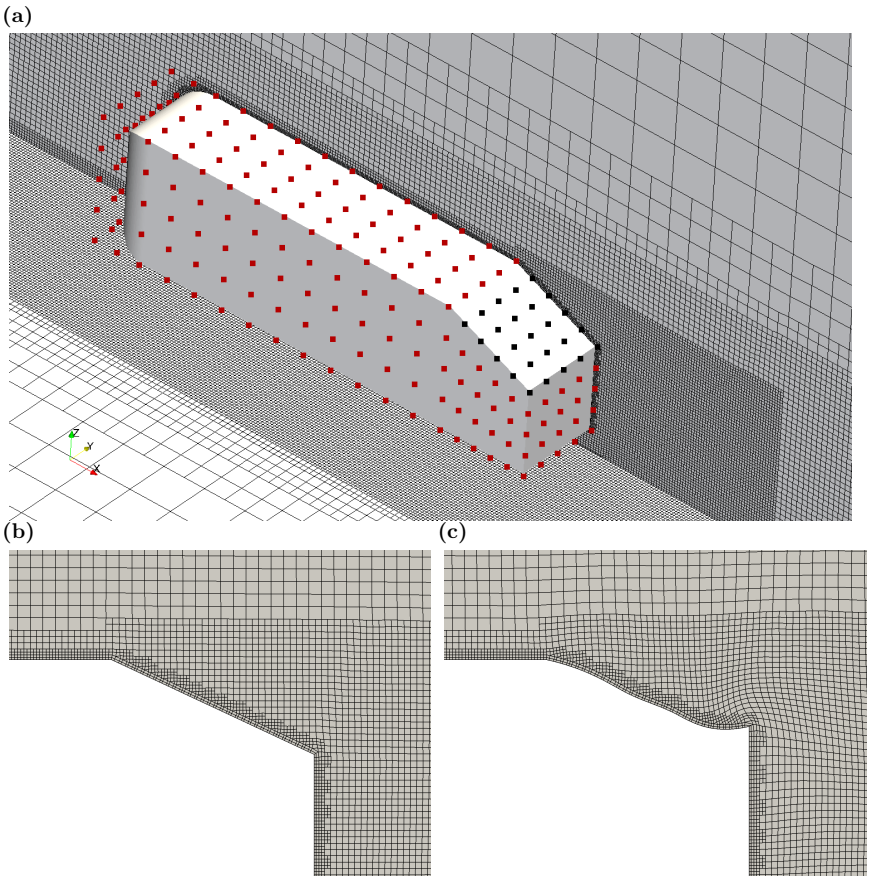

Figure 2: (a) Surface-geometry parameterization for Ahmed body [49] obtained by using FFD approach. The black and red squares are the FFD points. Only the black FFD points are selected as the design variables for manipulating the ramp shape whereas the red FFD points remain unchanged during the optimization. (b) Baseline and (c) deformed volume mesh at symmetry plane for Ahmed body. The surface parameterization and mesh-deformation operations are fully parallel and typically require less than 0.1% of the CFD simulation time.

eddy viscosity, respectively; and p is the pressure. The finite volume method is used to discretize the continuity and momentum equations on collocated meshes. These two equations are coupled by using the semi-implicit method for pressure-linked equations (SIMPLE) algorithm [52] along with the Rhie–Chow interpolation [53]. The detailed implementation is given below, following Jasak [54].

The SIMPLE algorithm starts by discretizing the momentum equation and solving an intermediate velocity field by using the pressure field obtained from the previous iteration or an initial guess (p^0). The momentum equation is then semi-discretized as

$$a_P \boldsymbol{U}_P = -\sum_N a_N \boldsymbol{U}_N - \nabla p^0 = \boldsymbol{H}(\boldsymbol{U}) - \nabla p^0, \tag{3}$$

where a is the coefficient resulting from the finite-volume discretization, subscripts P and N denote the control volume cell and all of its neighboring cells, respectively, and $\boldsymbol{H}(\boldsymbol{U}) = -\sum_N a_N \boldsymbol{U}_N$ represents the influence of the velocity from all the neighboring cells. Note that, to linearize the convective term, a new variable ϕ^0 (the cell-face flux) is introduced into the discretization to give

$$\int_S \boldsymbol{U}\boldsymbol{U} \cdot \mathrm{d}\boldsymbol{S} = \sum_f \boldsymbol{U}_f \boldsymbol{U}_f \cdot \boldsymbol{S}_f = \sum_f \phi^0 \boldsymbol{U}_f, \tag{4}$$

where the subscript f denotes the cell face. The cell-face flux ϕ^0 can be obtained from the previous iteration

or from an initial guess. The above linearization process complicates the discrete adjoint implementation; we elaborate on this issue and its solution in Section 2.5. Solving Eq. (3), we obtain an intermediate velocity field that does not yet satisfy the continuity equation.

Next, the continuity equation is coupled with the momentum equation to construct a pressure Poisson equation, and a new pressure field is computed. The discretized form of the continuity equation is

$$\int_S \boldsymbol{U} \cdot \mathrm{d}\boldsymbol{S} = \sum_f \boldsymbol{U}_f \cdot \boldsymbol{S}_f = 0. \tag{5}$$

Instead of using a simple linear interpolation, $\boldsymbol{U}_f$ in this equation is computed by interpolating the cell-centered velocity $\boldsymbol{U}_P$—obtained from the discretized momentum equation (3)—onto the cell face as follows:

$$\boldsymbol{U}_f = \left(\frac{\boldsymbol{H}(\boldsymbol{U})}{a_P}\right)_f - \left(\frac{1}{a_P}\right)_f (\nabla p)_f. \tag{6}$$

This idea of momentum interpolation was initially proposed by Rhie and Chow [53] and is effective in mitigating the "checkerboard" issue resulting from the collocated mesh configuration. Substituting Eq. (6) into Eq. (5), we obtain the pressure Poisson equation:

$$\nabla \cdot \left(\frac{1}{a_P}\nabla p\right) = \nabla \cdot \left(\frac{\boldsymbol{H}(\boldsymbol{U})}{a_P}\right). \tag{7}$$

Solving Eq. (7), we obtain an updated pressure field p^1.

Finally, the new pressure field p^1 is used to update the cell-face flux by using

$$\phi^1 = \boldsymbol{U}_f \cdot \boldsymbol{S}_f = \left[\left(\frac{\boldsymbol{H}(\boldsymbol{U})}{a_P}\right)_f - \left(\frac{1}{a_P}\right)_f (\nabla p^1)_f\right] \cdot \boldsymbol{S}_f. \tag{8}$$

Next, a new velocity field is computed by using Eq. (3) with the updated pressure and cell-face flux. The above process is repeated until the specified flow-convergence criteria are met.

The Reynolds-averaged Navier–Stokes (RANS) approach is used to model the turbulence in the flow. To connect the mean variables with the turbulence eddy viscosity ν_t, we use the Spalart–Allmaras (SA) one-equation turbulence model:

$$\int_V \nabla \cdot (\boldsymbol{U}\tilde{\nu})\,\mathrm{d}V - \frac{1}{\sigma}\int_V \nabla \cdot [(\nu + \tilde{\nu})\nabla\tilde{\nu}] + C_{b2}|\nabla\tilde{\nu}|^2\,\mathrm{d}V - C_{b1}\int_V \tilde{S}\tilde{\nu}\,\mathrm{d}V + C_{w1}\int_V f_w \left(\frac{\tilde{\nu}}{d}\right)^2 \mathrm{d}V = 0, \tag{9}$$

where $\tilde{\nu}$ is the modified viscosity, which can be related to the turbulent eddy viscosity via

$$\nu_t = \tilde{\nu}\frac{\chi^3}{\chi^3 + C_{v1}^3}, \quad \chi = \frac{\tilde{\nu}}{\nu}. \tag{10}$$

The four terms in Eq. (9) represent the convective, diffusion, production, and near-wall destruction for the turbulent flow, respectively. The detailed definition of these terms and their parameters can be seen in Spalart and Allmaras [55]. Compared with the standard SA model [55], the f_{t2} term is ignored in the OpenFOAM SA model implementation. Moreover, a stability enhancement function f_{v3} is added to ensure a non-negative $\tilde{S}$ term.

2.5 Discrete Adjoint Derivative Computation—discreteAdjointSolver

As mentioned above, to perform gradient-based aerodynamic-shape optimization, we need to compute the total derivative $\mathrm{d}f/\mathrm{d}\boldsymbol{x}$. Note that f depends not only on the design variables, but also on the state variables that are determined by the solution of governing equations, such as Eqs. (1), (2), and (9). Thus,

$$f = f(\boldsymbol{x}, \boldsymbol{w}), \tag{11}$$

where the vector of design variables $\boldsymbol{x} = [x_1, x_2, \ldots, x_{n_x}]^T$ has length n_x, and $\boldsymbol{w} = [w_1, w_2, \ldots, w_{n_w}]^T$ is the vector of state variables with length n_w.

Applying the chain rule for the total derivative, we obtain

$$\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} = \frac{\partial f}{\partial \boldsymbol{x}} + \frac{\partial f}{\partial \boldsymbol{w}} \frac{\mathrm{d}\boldsymbol{w}}{\mathrm{d}\boldsymbol{x}}. \tag{12}$$

A naive computation of $\mathrm{d}\boldsymbol{w}/\,\mathrm{d}\boldsymbol{x}$ via finite differences would require solving the governing equations n_x times, which can be computationally expensive for a large number of design variables. We can avoid this issue by using the fact that the derivatives of the residuals with respect to the design variables must be zero for the governing equations to remain feasible with respect to variations in the design variables. Thus, applying the chain rule to the residuals, we can write

$$\frac{\mathrm{d}\boldsymbol{R}}{\mathrm{d}\boldsymbol{x}} = \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} + \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}} \frac{\mathrm{d}\boldsymbol{w}}{\mathrm{d}\boldsymbol{x}} = 0, \tag{13}$$

where $\boldsymbol{R} = [R_1, R_2, \ldots, R_{n_w}]^T$ is the vector of flow residuals. Substituting Eq. (13) into Eq. (12) and canceling out the $\mathrm{d}\boldsymbol{w}/\,\mathrm{d}\boldsymbol{x}$ term, we get

$$\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} = \frac{\partial f}{\partial \boldsymbol{x}} - \frac{\partial f}{\partial \boldsymbol{w}} \left(\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}} \right)^{-1} \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}. \tag{14}$$

Considering the combination of the $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ and $\partial f/\partial \boldsymbol{w}$ terms in Eq. (14), we can solve the linear equation

$$\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{w}}^T \boldsymbol{\psi} = \frac{\partial f}{\partial \boldsymbol{w}}^T \tag{15}$$

to obtain the adjoint vector $\boldsymbol{\psi} = [\psi_1, \psi_2, \ldots, \psi_{n_w}]^T$. Next, this adjoint vector is substituted into Eq. (14) to compute the total derivative:

$$\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} = \frac{\partial f}{\partial \boldsymbol{x}} - \boldsymbol{\psi}^T \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}. \tag{16}$$

Since the design variable $\boldsymbol{x}$ does not explicitly appear in Eq. (15), we only need to solve Eq. (15) once for each function of interest, and thus the computational cost is (almost) independent of the number of design variables. This is an advantage for three-dimensional aerodynamic-shape-optimization problems, because the number of functions of interest is usually less than 10 but the number of design variables can be a few hundred.

A successful implementation of adjoint-based derivative computation requires an efficient and accurate computation for the partial derivatives—$\partial \boldsymbol{R}/\partial \boldsymbol{w}$, $\partial \boldsymbol{R}/\partial \boldsymbol{x}$, $\partial f/\partial \boldsymbol{w}$, and $\partial f/\partial \boldsymbol{x}$ in Eqs. (15) and (16). Four options are available for computing these partial derivatives [56, 57]: analytical methods, finite differences, the complex-step method [58], and AD [59]. Differentiating these terms analytically requires significant expertise in the particular flow-solver implementation. Given the complex object-oriented code structure in OpenFOAM, the analytical method would require a long development time and is prone to errors. Alternatively, one can use the finite-difference method, which is easy to implement even when residual functions are provided as black-box computations. However, the finite-difference method is subject to truncation and cancellation errors, and the derivative values are sensitive to step size, especially for functions with strong nonlinearity. To circumvent this limitation of finite differences, we could use the complex-step method [60] or AD [20, 21] to compute the partial derivative, both of which would provide very accurate derivatives. In our previous studies [20, 21, 24], we show that selectively applying AD to compute the partial derivatives in the discrete adjoint equations is particularly effective in terms of runtime and memory usage when compared with applying AD to the entire code. However, in the present study, we opt to use the finite-difference method because it is easy to implement and requires minimal modification to the original OpenFOAM code. Fortunately, the nonlinearity in the differentiated functions is relatively weak, and the finite-difference errors can be kept sufficiently small by using carefully chosen step sizes. The finite-difference-based partial derivative computation is detailed in Section 2.6, and in Sections 3.1 and 3.2 we show that its speed is satisfactory and that the accuracy is sufficient to obtain physically reasonable optimization results.

Before applying the finite-difference approach for partial derivative computation, special attention is needed to select state variables and flow residuals. The most straightforward way is to use the primitive variables u, v, w, p, $\tilde{\nu}$ as state variables and their corresponding governing equations (1), (2), and (9) as flow

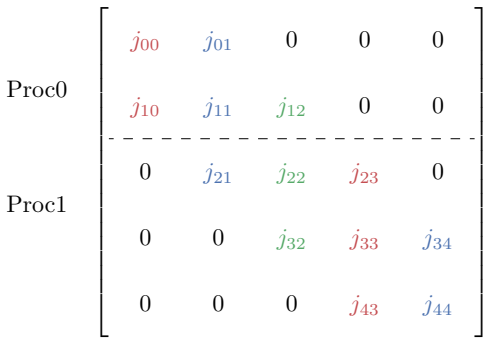

Figure 3: A 5×5 diagonal Jacobian matrix computed with graph coloring. The columns with the same colors are perturbed simultaneously because they affect independent sets of rows, resulting in a maximum of three colors in this case. The dashed line denotes parallel matrix storage in PETSc using two processors.

residuals (i.e., R_u, R_v, R_w, R_p, $R_{\tilde{\nu}}$). Considering the x-momentum residual R_u, as shown in Eq. (4), we introduce the new variable ϕ to linearize the discretized momentum equation, so R_u is a function of u, v, w, p, $\tilde{\nu}$, and ϕ. As shown before, instead of a simple linear interpolation, ϕ is computed by using the momentum-interpolation approach. This implies that ϕ depends on all the primitive variables u, v, w, p, and $\tilde{\nu}$. Given this complicated interconnection, it is easier to treat ϕ as an "independent" state variable when computing partial derivatives, following Roth *et al.* [61]. Therefore, herein, the vector of state variables is chosen to be $\boldsymbol{w} = [u, v, w, p, \tilde{\nu}, \phi]^T$, and the corresponding flow residual vector $\boldsymbol{R} = [R_u, R_v, R_w, R_p, R_{\tilde{\nu}}, R_\phi]^T$. Here, R_u, R_v, R_w can be computed by using Eq. (3), and R_p, R_ϕ, and $R_{\tilde{\nu}}$ can be computed based on Eqs. (7)–(9), respectively. Note that both $\boldsymbol{R}$ and $\boldsymbol{w}$ have mixed cell-centered and face-centered variables. The size of state variables is approximately eight times the number of cells.

2.6 Partial Derivative Computation and Graph-Coloring Method—coloringSolver

As mentioned in Section 2.5, the finite-difference approach is used to compute the partial derivatives $\partial \boldsymbol{R}/\partial \boldsymbol{w}$, $\partial \boldsymbol{R}/\partial \boldsymbol{x}$, $\partial f/\partial \boldsymbol{w}$, and $\partial f/\partial \boldsymbol{x}$. Considering a general Jacobian matrix $\partial \boldsymbol{Y}/\partial \boldsymbol{X}$, its ith and jth element can be computed as

$$\left(\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{X}}\right)_{i,j} = \frac{Y_i(\boldsymbol{X} + \varepsilon \boldsymbol{e}_j) - Y_i(\boldsymbol{X})}{\varepsilon}, \tag{17}$$

where the subscripts i and j are the row and column indices of the matrix, respectively, ε is the finite-difference step, and $\boldsymbol{e}_j$ is a unit vector with unity in row j and zeros in all other rows. A naive implementation is that we perturb each column of the matrix and compute the corresponding partial derivatives for all rows. Although evaluating the $\boldsymbol{Y}$ function is relatively cheap, this process needs to be repeated n_C times, where n_C is the number of columns in the Jacobian matrix. This is not a severe issue for $\partial \boldsymbol{R}/\partial \boldsymbol{x}$ and $\partial f/\partial \boldsymbol{x}$, because the size of $\boldsymbol{x}$ does not typically exceed a few hundred. However, for $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ and $\partial f/\partial \boldsymbol{w}$, the computational cost may be too high, because the size of $\boldsymbol{w}$ can easily escalate to tens of millions for a three-dimensional aerodynamic-optimization problem.

To accelerate the finite-difference-based partial derivative computation, we use a graph-coloring method. The graph-coloring approach exploits the sparsity of the Jacobian matrix and enables us to simultaneously perturb multiple columns by perturbing sets of columns that influence independent sets of rows. Considering the diagonal Jacobian matrix in Fig. 3, as mentioned above, a naive finite-difference implementation would require five function evaluations. To accelerate this process, we partition all the columns of the matrix into different structurally orthogonal subgroups (colors), such that, in one structurally orthogonal subgroup, no two columns have a nonzero entry in a common row. The columns with the same colors are perturbed simultaneously and the number of function evaluations can be reduced to three.

The actual $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ sparsity pattern for a three-dimensional case is obviously much more complicated than the diagonal matrix shown in Fig. 3. Table 1 summarizes the level of connectivity for the simpleFoam flow residuals. Depending on the mesh topology, the number of connected state variables for a flow residual differs from case to case, although their connectivity levels remain the same. As an example, Fig. 4 shows the sparsity

Table 1: Connectivity level for simpleFoam flow residuals with Spalart–Allmaras turbulence model. The numbers denote how many levels of neighboring state variables are connected to a flow residual.

	$\boldsymbol{U}$	p	$\tilde{\nu}$	ϕ
$R_{\boldsymbol{U}}$	2	1	1	0
R_p	3	2	2	1
$R_{\tilde{\nu}}$	1		2	0
R_ϕ	3	2	2	1

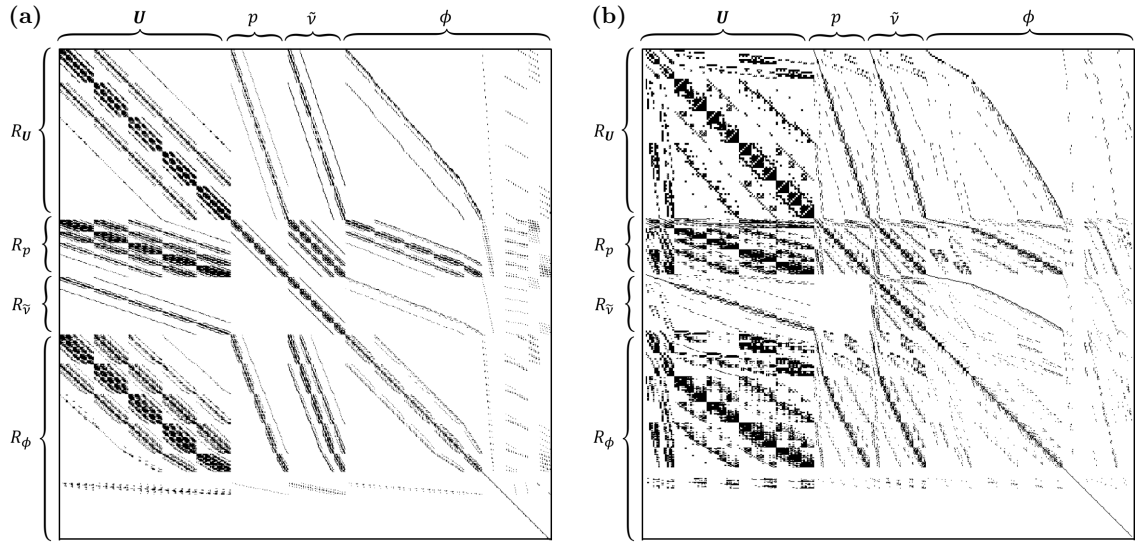

Figure 4: Example of sparsity pattern for $\partial \boldsymbol{R}/\partial \boldsymbol{w}$, generated based on a curved cube geometry. We use a state-by-state matrix ordering (i.e., $\boldsymbol{w} = [\boldsymbol{U}_{1:n_c}, p_{1:n_c}, \tilde{\nu}_{1:n_c}, \phi_{1:n_f}]$). The subscripts denote the cell or face index where n_c and n_f are the total number of cells and faces, respectively. For example, $\boldsymbol{U}_{1:n_c} = [u_1, v_1, w_1, u_2, v_2, w_2, \ldots, u_{n_c}, v_{n_c}, w_{n_c}]$. Similar ordering is also applied to $\boldsymbol{R}$. (a) Structured hexahedral mesh (cell size: 100, face size: 365, Jacobian row size: 865) and (b) unstructured snappy hexahedral mesh (cell size: 94, face size: 345, Jacobian row size: 815).

pattern for $\partial \boldsymbol{R}/\partial \boldsymbol{w}$, which is generated based on the connectivity level and mesh topology information from a curved cube geometry. We show results for both a structured hexahedral mesh and an unstructured snappy hexahedral mesh which is generated by the built-in OpenFOAM mesh tool snappyHexMesh. Given that we mix cell-centered and face-centered variables and residuals, we use a state-by-state matrix ordering because it is straightforward to implement. Note that we reorder the state Jacobian matrix to reduce the memory usage for the adjoint-equation solution (see Section 2.7). As shown in Fig. 4, the p and ϕ residuals have the largest level of connectivity and therefore denser rows. Also, while the structured mesh exhibits a block-diagonal structure, the unstructured mesh is much more irregular. Therefore, an analytical determination of the coloring, such as that described by Lyu *et al.* [21], is impossible for the unstructured mesh $\partial \boldsymbol{R}/\partial \boldsymbol{w}$, and an efficient coloring scheme to partition the Jacobian matrices is needed.

The idea of partitioning the Jacobian matrix was initially proposed by Curtis *et al.* [62]. This process was then modeled as various graph-coloring problems, such as the column intersection (distance-1) graph proposed by Coleman *et al.* [63] and bipartite (distance-2) graph introduced by Gebremedhin *et al.* [64]. The objective is to find the fewest colors possible for a given Jacobian matrix. Note that this number is bounded by the maximum number of nonzero entries in a row of the Jacobian. Unfortunately, it has been shown that the graph-coloring computation is an NP-hard problem [63]; it is very unlikely that a universal algorithm exists to determine the fewest possible colors in polynomial time. This issue becomes even more challenging if one needs to extend the coloring algorithm for parallel computation in a distributed memory system. For example, Nielsen and Kleb [60] used the greedy coloring algorithm [62] to accelerate

Algorithm 1 Parallel graph coloring for sparse Jacobian matrix J

Input: Jacobian matrix: J ▷ Global size: $n_R \times n_C$

Output: Graph-coloring vector: G ▷ Global size: $1 \times n_C$

1: $G_j \leftarrow -1$ for each $j \in C_{\text{all}}$ ▷ Label all columns as uncolored (-1)

2: $R_j \leftarrow \text{rand}$ for each $j \in C_{\text{all}}$ ▷ Initialize conflict-resolution vector with random numbers

3: **for** $t \in \{\text{interior}, \text{global}\}$ **do** ▷ Color interior columns first and then global columns

4: **for** $n \in \{0, \ldots, \infty\}$ **do** ▷ n: current color

5: $G_j \leftarrow n$ for each $j \in C_{\text{uncolored}}$ ▷ Assign current color n to uncolored columns

6: **for** $r \in R_{\text{local}}$ **do** ▷ R_{local}: Local rows owned by the current processor

7: $S = F(t, n, R, G, J_{i=r, j \in C_{\text{nonzeros}}})$ ▷ S: reset columns. F: conflict-resolution function

8: $G_j \leftarrow -1$ for each $j \in S$ ▷ Reset these columns as uncolored

9: **if** $t = \text{interior}$ **and** $G_j \neq -1$ for each $j \in C_{\text{interior}}$ **then**

10: **break** ▷ All interior columns are colored

11: **else if** $t = \text{global}$ **and** $G_j \neq -1$ for each $j \in C_{\text{global}}$ **then**

12: **break** ▷ All global columns are colored

state Jacobian computation in their adjoint implementation, which resulted in a reasonably small number of colors. However, the greedy algorithm is known to be inherently sequential and is difficult to parallelize for tackling large Jacobian matrices [65]. Given this background, a heuristic parallel graph-coloring algorithm is proposed herein.

The general idea of heuristic graph coloring is to tentatively color and then resolve any conflict. The pseudo-algorithm is detailed in Algorithm 1. More specifically, there is a loop over each row that assigns a tentative color to all nonzero, uncolored columns (Algorithm 1, lines 4 to 6). Next, a conflict-resolution function is called to determine the "winner" column in this row (Algorithm 1, line 7), and all the other columns are reset to "uncolored" (Algorithm 1, line 8). The conflict-resolution function is detailed in Algorithm 2. The above process is repeated until all columns are colored (Algorithm 1, lines 9–12). This general idea is similar to that described by Bozdağ *et al.* [66]. The challenge is that the operations above need to be fully parallel and scalable. In addition, the number of colors should be independent of the Jacobian sizes and the number of CPU cores. To achieve these goals, we developed an improved parallel algorithm to tackle the Jacobian matrix with $\mathcal{O}(10 \text{ million})$ rows using $\mathcal{O}(1000)$ CPU cores. This algorithm is optimized based on the parallel storage and communication architecture of PETSc, and it consists of two major steps (i.e., interior and global; see Algorithm 1, line 3).

First, we color the interior columns that do not share any nonzero entry in a common column among the distributed CPU processors. As mentioned before, the PETSc library is used to store and manipulate the large sparse matrices and vectors. An example of the PETSc's parallel matrix storage strategy is shown in Fig. 3. Here each processor owns different portions or rows of the matrix, and a similar parallel storage is also used for vectors. As shown in Fig. 3, column 0 is interior for Proc0 and columns 3 and 4 are interior for Proc1. These interior columns can be safely colored on a local processor without a conflict between processors. Note that, for each processor, we only color the columns in diagonal blocks to avoid any message-passing interface (MPI) communication at this step (Algorithm 1, line 6). Next, we repeat the above coloring strategy for global columns, whereas the colored interior columns remain untouched. Note that we need the coloring information from other processors, and MPI communications are needed at this step. To ensure a successful coloring process, a robust conflict-resolution function is needed. To explore the potential for a smaller number of colors, the "winner" of each conflicting column is determined in a random manner (Algorithm 1, line 2). Note that, when determining the winner for global columns, we need to first complete the coloring for local columns to prevent deadlock (Algorithm 2, line 9).

In this paper, we use the graph-coloring scheme detailed above to accelerate the computation of the partial derivative for both $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ and $\partial f/\partial \boldsymbol{w}$. Special attention is needed for the $\partial f/\partial \boldsymbol{w}$ coloring scheme. Typically, f is computed based on the integration of state variables over a surface (e.g., drag and lift). From a discrete perspective, this implies that the number of connected states for f is on the order of N_D (the total number of cell faces on the surface). To enable the coloring scheme for $\partial f/\partial \boldsymbol{w}$, we divide f into N_D different cell faces. In other words, $\partial f/\partial \boldsymbol{w} = \sum_{i=1}^{N_D} \partial f_i/\partial \boldsymbol{w}$. With this treatment, we can easily obtain the

Algorithm 2 Conflict-resolution function for graph coloring

Input: Task to resolve: t, current color: n, conflict-resolution vector: R, graph-coloring vector: G, and nonzero elements in a given row: $J_{i=r,j\in C_{\text{nonzeros}}}$
Output: Columns to reset: S

function $S = F(t, n, R, G, J_{i=r,j\in C_{\text{nonzeros}}})$
  if $t =$ interior **then**
    $v_{\min} \leftarrow \min(R_j)$ for each $j \in C_{\text{interior}} \cup C_{\text{local}}$
    $j_{\min} \leftarrow$ find_index$(R = v_{\min})$ ▷ Find column index with minimal R value
    for $j \in \{C_{\text{interior}}, C_{\text{local}}\}$ **do**
      if $G_j = n$ **and** $j \neq j_{\min}$ **then**
        S.append(j) ▷ Append all columns but $j_{\min}$ to S for current color n
  if $t =$ global **then**
    for range $\in \{\text{local}, \text{global}\}$ **do**
      $\hat{G} \leftarrow G^k$ for each $k \in$ all processors ▷ Gather all colors to current processor
      $v_{\min} \leftarrow \min(R_j)$ for each $j \in C_{\text{range}}$
      $j_{\min} \leftarrow$ find_index$(R = v_{\min})$
      for $j \in C_{\text{range}}$ **do**
        if $j \in C_{\text{interior}}$ **then**
          S.append$(j \notin C_{\text{interior}})$ ▷ Always keep interior columns
        else if $\hat{G}_j = n$ **and** $j \neq j_{\min}$ **then**
          S.append(j)

connectivity for each $f_i/\partial \boldsymbol{w}$, compute the coloring, calculate the partial derivatives, and sum their values to the final $\partial f/\partial \boldsymbol{w}$ matrix. We observe that the number of colors for $\partial f/\partial \boldsymbol{w}$ is typically one order of magnitude less than the number of colors for $\partial \boldsymbol{R}/\partial \boldsymbol{w}$. Therefore, in Section 3.1, we mainly focus on evaluating the performance of the coloring scheme for $\partial \boldsymbol{R}/\partial \boldsymbol{w}$.

No coloring scheme is needed for $\partial \boldsymbol{R}/\partial \boldsymbol{x}$ and $\partial f/\partial \boldsymbol{x}$. Instead, a brute-force finite-difference approach is used: We first perturb each design variable and deform the surface and volume meshes to compute the mesh-deformation derivative matrix ($\mathrm{d}\boldsymbol{x_v}/\,\mathrm{d}\boldsymbol{x}$, where $\boldsymbol{x_v}$ is the volume-mesh coordinates) in the Python layer. As mentioned in Section 2.3, this operation requires less than 0.1% of the CFD simulation time. This matrix is then passed to the adjoint solver, which allows us to directly compute $\partial \boldsymbol{R}/\partial \boldsymbol{x}$ and $\partial f/\partial \boldsymbol{x}$. Note that the number of operations for computing $\partial \boldsymbol{R}/\partial \boldsymbol{x}$ and $\partial f/\partial \boldsymbol{x}$ is proportional to the number of design variables, which does not typically exceed a few hundred. Finally, we observe that the first-order forward-differencing scheme is the most efficient and robust option, so we use it to compute all partial derivatives in this paper.

2.7 Solution of Adjoint Equations

Adjoint derivative computation requires a robust linear-equation solver, especially for realistic geometry configurations with highly complex flow conditions [30, 67–69]. We use the PETSc library to solve the linear equation shown in Eq. (15). PETSc provides a wide range of parallel linear- and nonlinear-equation solvers with various preconditioning options. We use the generalized minimal residual (GMRES) iterative linear-equation solver with the additive Schwartz method as the global preconditioner. For the local preconditioning, we use the incomplete lower and upper (ILU) factorization approach with one or two levels of fill-in. This strategy is effective for solving the adjoint equation, as reported in previous studies [21, 24]. To reduce the memory usage of ILU fill, we adopt the nested-dissection matrix-reordering approach. The preconditioning matrix is computed by using a coloring approach similar to that used in Section 2.6, except that we use the first-order upwind scheme to compute the convective terms of flow residuals. Since we have a mix of cell- and face-centered state variables and flow residuals, and their magnitudes are quite different (e.g., the magnitudes of $\boldsymbol{U}$ and ϕ), we need to scale all the partial derivatives so that their magnitudes are as similar as possible. By properly scaling the Jacobian matrix, the diagonal dominance and condition number can be improved, thus improving the convergence of the linear equations. The scaled finite-difference

computation for the state Jacobian matrix takes the form

$$\left(\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{W}}\right)^{\text{scaled}}_{i,j} = \frac{C^R_i R_i(\boldsymbol{W} + C^W_j \varepsilon \boldsymbol{e}_j) - C^R_i R_i(\boldsymbol{W})}{\varepsilon}, \tag{18}$$

where C^R and C^W are the scaling factors for the residuals and states, respectively. For the cell-centered residuals, C^R is chosen to be the cell volume whereas, for the face-centered residuals, C^R is set to be the face area S_f. The state scaling factors C^W for $\boldsymbol{U}$, p, $\tilde{\nu}$, and ϕ are selected to be U_0, $U_0^2/2$, $\tilde{\nu}_0$, and $U_0 S_f$, respectively. Here the subscript 0 denotes the inlet (far-field) reference value. Similar scaling is also applied to $\partial f/\partial \boldsymbol{W}$, $\partial \boldsymbol{R}/\partial \boldsymbol{x}$, and $\partial f/\partial \boldsymbol{x}$. Note that the scaling is only applied when computing the partial derivatives $\partial \boldsymbol{R}/\partial \boldsymbol{w}$, $\partial f/\partial \boldsymbol{w}$, $\partial \boldsymbol{R}/\partial \boldsymbol{x}$, and $\partial f/\partial \boldsymbol{w}$, whereas the values of the state variables are unchanged, which ensures a consistent dimension for the flow solver because OpenFOAM uses dimensional variables for the flow solution. Also note that the scaled partial derivative values differ from their original values. However, it is straightforward to prove that the final values of the total derivative are the same. We observe that the convergence of the adjoint equation can be significantly improved by using the above scaling of $\boldsymbol{R}$ and $\boldsymbol{w}$.

2.8 Constrained Nonlinear Optimization—pyOptSparse

We set up our optimization problems by using pyOptSparse, which is an open source, object-oriented Python interface for formulating constrained nonlinear optimization problems. This interface is based on the original pyOpt [32] module but includes extensive modifications to facilitate the assembly of sparse constraint Jacobians. pyOptSparse provides a high-level API for defining the design variables, the objective function, and the constraint functions. pyOptSparse itself does not include optimization problem solvers, but it provides interfaces for several optimization packages, including some open source packages.

In this study, we use SNOPT [48] to solve the optimization problems. SNOPT implements the sequential quadratic programming algorithm to solve the constrained nonlinear optimization problem and uses the quasi-Newton method to solve the quadratic subproblem, where the Hessian of the Lagrangian is approximated by using a Broyden–Fletcher–Goldfarb–Shanno update. The optimality in SNOPT is quantified by the norm of the residual of the first-order Karush–Kuhn–Tucker optimality conditions [48, 70].

As with all gradient-based optimizers, SNOPT requires the derivatives of objective and constraints with respect to all design variables for each optimization iteration. We compute these derivatives efficiently and accurately by using the adjoint approach developed herein.

2.9 Code Structure and Implementation

The adjoint derivative computation is a key module in our optimization framework. We developed a new class called `SIMPLEDerivativeClass` based on the OpenFOAM framework. This class contains all the functions for the coloring solver (coloringSolver) and the discrete adjoint solver (discreteAdjointSolver), including the connectivity computation, parallel graph coloring, partial derivative computation, and adjoint-equation solution. Figure 5 shows the code structure and calling sequence for these two solvers. Note that we only need to run the coloring solver once per optimization, and the coloring information is saved. The discreteAdjointSolver then reads the coloring information as input when computing $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ and $\partial f/\partial \boldsymbol{w}$. As an example, Fig. 6 shows the code structure for the `ComputedRdW` function. Here, we first compute the reference flow residuals based on the unperturbed, converged flow solutions. Next, we start from the first color and simultaneously perturb the state variables associated with it. Based on the perturbed flow solutions, new flow residuals are computed, and the finite-difference approach is used to compute partial derivatives. Finally, we assign the partial derivative values for the $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ matrix stored in PETSc format and reset the perturbations. The above process is repeated until all the colors are done. Note that the boundary conditions must be updated for each state-variable perturbation. For example, when we perturb the velocity of a cell immediately next to an interprocessor boundary patch, we need to interpolate the perturbed velocity onto this boundary patch. This is done by calling `U.correctBoundaryConditions()` in OpenFOAM. We need similar updates for all flow variables. Note that updating the boundary condition is essential for accurately computing the adjoint derivative.

One of the advantages of the discrete adjoint approach is that its formulation starts from the discretized NS equations, and the properties of flow solver (e.g., the flow convergence behavior and derivatives) are

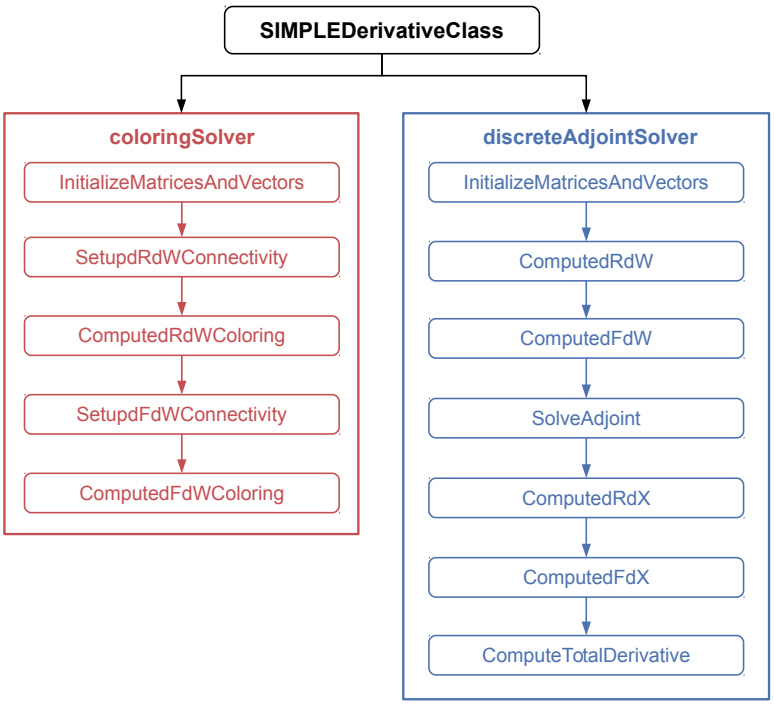

Figure 5: Code structure for coloring and discrete adjoint solvers.

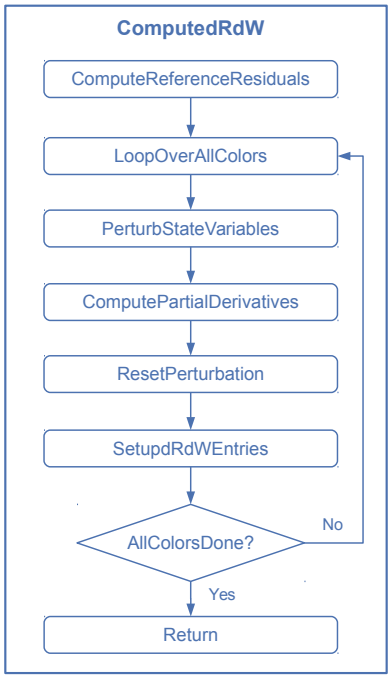

Figure 6: Code structure for state Jacobian matrix ($\partial \boldsymbol{R}/\partial \boldsymbol{w}$) computation function.

preserved. Following this idea, we reuse the code structure in the simpleFoam flow solver as much as possible in the adjoint implementation. This not only facilitates the adjoint code implementation but also reduces the effort required to modify the adjoint code when the flow solver is updated. Listing 1 shows a sample function for computing the momentum-equation residuals. Here we reuse the fvm matrix (`UEqn`;

Listing 1: Sample function for computing momentum residuals. This sample code just illustrates the idea of reusing simpleFoam code (i.e., the `fvm` matrix `UEqn`) for residual computation; the actual `computeURes` function in our code is slightly different.

```
tmp<volVectorField> SIMPLEDerivative::computeURes
(
    const volVectorField& U,
    const volScalarField& p,
    const surfaceScalarField& phi
)
{
    // Reuse the U equation fvm matrix from simpleFoam
    tmp<fvVectorMatrix> UEqn
    (
        fvm::div(phi, U)            // Divergence term
      + turbulence_.divDevReff(U)   // Diffusion term
        ==
        fvOptions_(U)
    );
    // Do a matrix-vector product and add the pressure term for the U residual
    tmp<volVectorField> URes
    (
        (UEqn()&U) + fvc::grad(p)
    );
    return URes;
}
```

the left-hand-side matrix) implemented in the original simpleFoam flow solver and calculate a matrix-vector product for residual computations. A similar implementation is used to compute other flow residuals.

3 Results and Discussion

We now evaluate the performance of adjoint derivative computation in terms of speed, scalability, and accuracy. We then perform a basic optimization of a simple bluff body geometry along with detailed flow analyses and experimental validation of the optimization result. In addition, we apply the optimization framework for two more complex cases: UAV and car aerodynamic design. The main objective of this section is to demonstrate our framework's capabilities for various constrained-optimization applications. A comprehensive optimization study (e.g., multipoint optimization [29, 71] and mesh-refinement studies) for each of these applications is outside the present scope and will be done in future work.

3.1 Performance Evaluation

To evaluate the performance of our adjoint framework, we use a simple bluff geometry—the Ahmed body. The Ahmed body is a rectangular block geometry with a rounded front end and a sharp ramp near the rear end, as shown in Fig. 2(a). It was originally proposed and tested by Ahmed *et al.* [49] in 1984 and has since been widely used as a CFD benchmark. Here we choose the 25° ramp-angle configuration as our baseline geometry.

Only half of the body is simulated, and the simulation domain size is $8L$, $2L$, and $2L$ in the x, y, and z directions, respectively, where L is the length of Ahmed body. The Reynolds number is 1.4×10^6 based on L and U_0. A second-order linear upwind scheme is used to differentiate the divergence terms, whereas the central differential scheme is adopted for the diffusion terms. This differentiation configuration is reportedly the most efficient and accurate for RANS simulations [6] and is used for all the simulations herein.

For the derivative computation, the unstructured snappy hexahedral mesh is generated with approximately 1 million cells. The averaged y^+ is 60, so a wall function is used for ν_t. For the scalability test, we generate meshes with up to 10 million cells. For the coloring scheme, we also show results for a structured hexahedral mesh for comparison. All test simulations in this section were done on Stampede 2, which is a high-performance computing (HPC) system equipped with a second generation Intel Xeon Phi 7250 Processor: Knights Landing (KNL). Each KNL node has 68 CPU cores running at 1.4 GHz and 96 GB

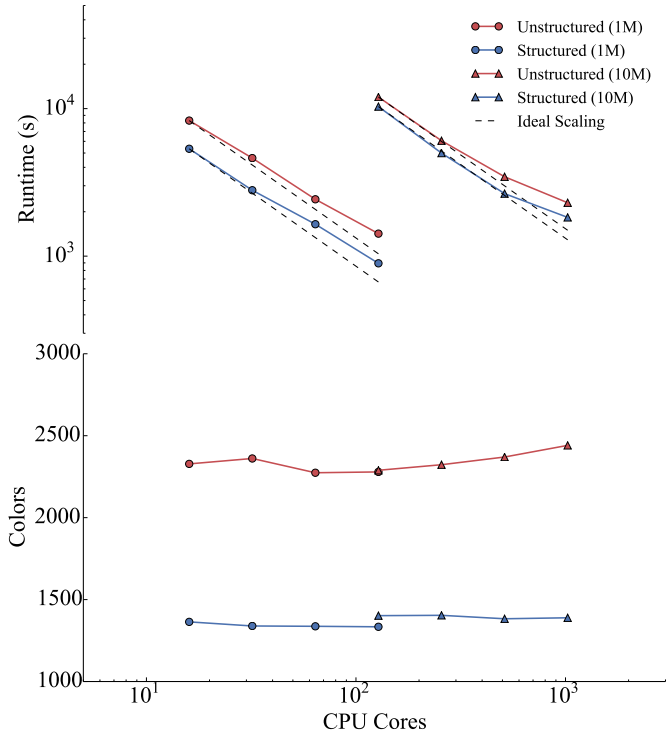

Figure 7: Performance of parallel coloring scheme for state Jacobian matrix $\partial \boldsymbol{R}/\partial \boldsymbol{w}$. The coloring runtime scales up to 10 million cells and 1024 CPU cores, and the number of colors is almost independent of the mesh size and the number of CPU cores.

of DDR4 memory, and they are connected through the Sandy Bridge FDR Infiniband. Stampede 2 has a peak PFLOPS of 18. The OpenFOAM and PETSc versions are 2.4.x and 2.7.6, respectively, and they were compiled with Intel-17.0.4 and IMPI-17.0.3.

We first test the scalability for the coloring solver by using both structured hexahedral and unstructured snappy hexahedral meshes, as shown in Fig. 7. The Jacobian row sizes are 8.8×10^6 and 8.4×10^7 for 1 million and 10 million cells, respectively. The runtime for parallel coloring computation scales well up to 10 million mesh cells and 1024 CPU cores. In addition, the number of colors is almost independent of Jacobian size and number of CPU cores, which confirms the efficiency of our parallel coloring algorithm. In terms of mesh topology, the number of structured mesh colors ($\sim$1300) is generally less than the number of unstructured snappy mesh colors ($\sim$2300) because the stencil for the unstructured snappy mesh is much larger than the structured mesh, resulting in a denser Jacobian matrix and therefore more colors. In the previous adjoint coloring implementations using analytical [21], greedy [60], or heuristic [72] approaches, the number of colors was reported to be $\mathcal{O}(100)$, which is much less than in our implementation. This is primarily due to the level-three connectivity, as shown in Table 1, and especially to the inclusion of the face-center state variable ϕ in our adjoint formulation. As a result, our Jacobian matrix is relatively dense: a maximum of 307 nonzero entries are in a row of the state Jacobian for the structured hexahedral mesh and 811 nonzero entries for the unstructured snappy hexahedral mesh.

Next, we evaluate the speed and scalability of adjoint derivative computation in Fig. 8. The objective function is the drag coefficient C_D, defined as $C_D = D/0.5\rho U_0^2 A_{\text{ref}}$, where D is the drag force and A_{ref} is the frontal area of the Ahmed body. The adjoint derivative computation scales well up to 10 million cells and 1024 CPU cores. This good scalability is primarily due to the performance of the coloring solver, as shown in Fig. 7. Here, we also show the scalability of flow simulation for reference. The derivative computation scales as well as does the flow simulation.

In terms of speed, the derivative computation is faster than the flow solution for 10 million cells, whereas for 1 million cells, the adjoint computation speed is very close to that of the flow simulation and outperforms

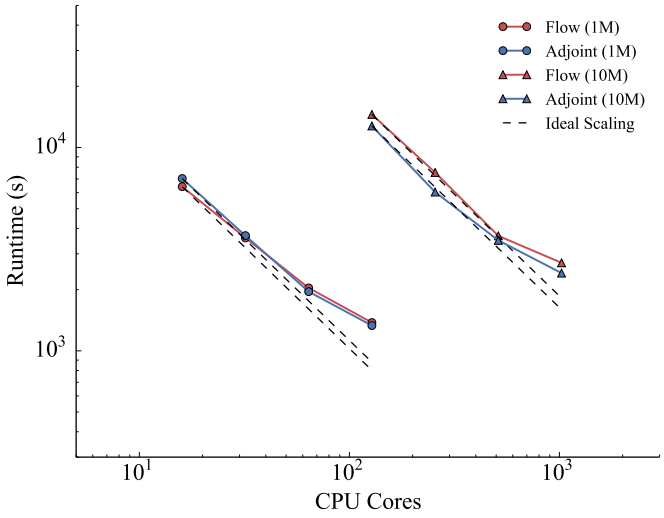

Figure 8: Scalability of flow and adjoint computations. The adjoint runtime scales up to 10 million cells and 1024 CPU cores.

Table 2: Runtimes of flow and adjoint computation for increasing number of CPU cores. The adjoint-flow runtime ratio is of order unity.

1 million cells				
KNL nodes	2	2	2	2
CPU cores	16	32	64	128
Flow runtime (s)	6418	3593	2039	1377
Adjoint runtime (s)	7036	3686	1955	1331
Adjoint-flow runtime ratio	1.10	1.03	0.96	0.97
10 million cells				
KNL nodes	24	24	32	32
CPU cores	128	256	512	1024
Flow runtime (s)	14543	7539	3677	2709
Adjoint runtime (s)	12763	6037	3494	2412
Adjoint-flow runtime ratio	0.88	0.80	0.95	0.89

the flow simulation when using more than 32 CPU cores. The faster derivative computation for greater number of cells is primarily attributed to the fact that the flow requires more iterations to converge. The convergence tolerance of the flow solver is set to 10^{-8}, and the 1- and 10-million-cell cases require 6000 and 8000 steps to converge, respectively.

Overall, the runtime ratio between the adjoint and flow computation is of order unity, independent of Jacobian size (mesh cells) and number of CPU cores, as shown in Table 2. This confirms the efficiency of the adjoint derivative computation. Moreover, the scalability test for the flow and adjoint indicates that good scalability is achieved if each CPU core owns no fewer than 20 000 cells. Table 3 shows the detailed runtime breakdown for the derivative computation using the 1-million-cell Ahmed-body case with 128 CPU cores. Computing the state Jacobian matrix is the most expensive part, followed by computing the state Jacobian preconditioning matrix. Note that the cost of computing the state Jacobian is proportional to $n_c T_{\text{resid}}$, where n_c is the number of colors and T_{resid} is the cost of one flow residual computation. Meanwhile, the adjoint equation converges in about 500 iterations (see Fig. 9) and takes 26.1% of the total adjoint runtime.

In terms of peak resident memory, the flow solutions require 2.9 and 25.2 GB for the 1-million-cell (16 CPU cores) and 10-million-cell (128 CPU cores) cases, respectively, whereas the memory requirement rapidly grows to 101.8 and 886.1 GB to compute the adjoint derivative for 1 and 10 million cells, respectively. The peak memory usage happens at the adjoint-equation solution step when the ILU preconditioner is filled

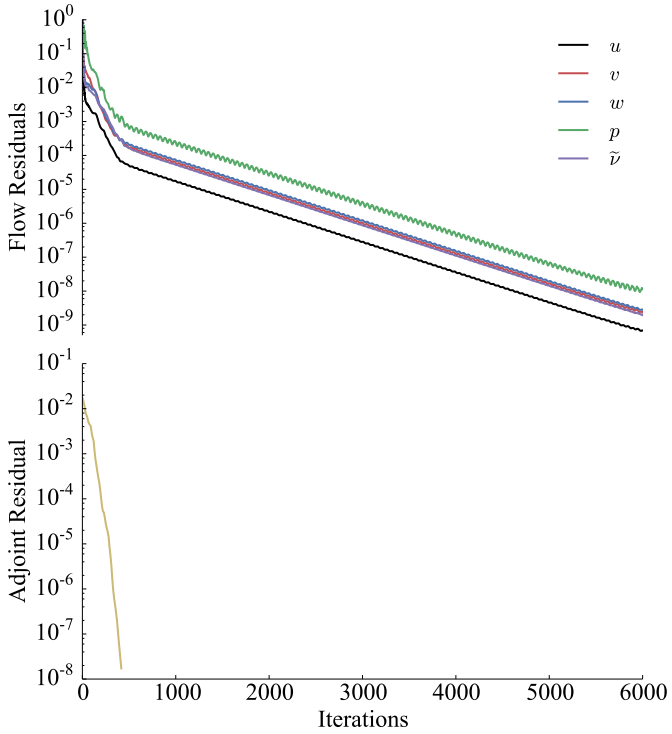

Figure 9: Convergence history of flow and adjoint residuals for Ahmed-body case with 1 million unstructured snappy hexahedral cells.

Table 3: Adjoint computational-time breakdown for Ahmed body. Computation of $\partial \boldsymbol{R}/\partial \boldsymbol{w}$, $\partial \boldsymbol{R}/\partial \boldsymbol{w}$ (PC), and the solution of the adjoint equation are the most expensive whereas the computational cost for $\partial f/\partial \boldsymbol{w}$, $\partial \boldsymbol{R}/\partial \boldsymbol{x}$, and $\partial f/\partial \boldsymbol{x}$ is very small. PC stands for "preconditioning matrix computation."

	Runtime (s)	Percent
$\partial \boldsymbol{R}/\partial \boldsymbol{w}$	547.0	41.1%
$\partial \boldsymbol{R}/\partial \boldsymbol{w}$ (PC)	425.9	32.0%
$\partial f/\partial \boldsymbol{w}$	2.7	0.2%
$\partial \boldsymbol{R}/\partial \boldsymbol{x}$	5.3	0.4%
$\partial f/\partial \boldsymbol{x}$	2.7	0.2%
Adjoint equation	347.4	26.1%

in. This relatively large memory requirement indicates that our discrete adjoint implementation is memory bound. Fortunately, current HPC systems typically have more than 64 GB of memory per node, which alleviates this limitation. As mentioned above, Stampede 2 has 96 GB of memory per node, and we require at least 2 and 10 nodes for the 1- and 10-million-cell cases, respectively. To be conservative, we use at least 24 nodes in practice for the 10 million runs (see Table 2). Note that a KNL node has many more CPU cores than traditional systems, but the processors run at lower frequencies. Therefore, when comparing the speed of KNL against traditional CPUs, a node-to-node comparison is more appropriate than a core-to-core comparison. For example, a combined flow and adjoint computation takes 0.8 hours for the 1 million cells when using 2 KNL nodes with 128 CPU cores (see Table 2).

Finally, we evaluate the accuracy of the adjoint derivative computation by using the derivative computed directly from the finite-difference approach as a reference. The verification of the derivative of the drag coefficient with respect to the far-field velocity ($\mathrm{d}C_D/\,\mathrm{d}u_0$) is shown in Table 4. Because the finite-difference approach is used to compute the partial derivatives in the adjoint implementation, the impact of the finite-difference step size on the accuracy is shown for comparison. The step size used in the partial derivative

Table 4: Verification of adjoint derivative computation for $\mathrm{d}C_D/\,\mathrm{d}u_0$, where u_0 is the far-field velocity in the x direction. For a step size of 10^{-8}, the adjoint derivative matches the reference value to the fifth digit.

Step size	Adjoint	Reference	Relative error
10^{-5}	0.03846861	0.03042447	26.43970%
10^{-6}	0.03093502	0.03042447	1.67809%
10^{-7}	0.03048090	0.03042447	0.18548%
10^{-8}	0.03042432	0.03042447	−0.00049%
10^{-9}	0.03041111	0.03042447	−0.04391%
10^{-10}	0.03040858	0.03042447	−0.05222%

Table 5: Verification of adjoint derivative computation for $\mathrm{d}C_D/\,\mathrm{d}\boldsymbol{x}$, where $\boldsymbol{x} = (x_0, x_1, x_2, x_3)$ is the FFD design variable vector. The average error is less than 0.1%.

Variable	Adjoint	Reference	Relative error
x_0	0.92671214	0.92699603	−0.03062%
x_1	0.66670412	0.66674205	−0.00569%
x_2	−0.52731369	−0.52721475	0.01877%
x_3	−0.05693056	−0.05678746	0.25200%

computations is defined in Eq. (18). Note that, for the scaled partial derivatives, the actual perturbation is the step size multiplied by the scaling factor ($C^W\varepsilon$). The best agreement occurs for a step size of 10^{-8}, for which the adjoint derivative matches the reference value to the fifth digit. Note that similar step-size studies are conducted for the reference derivative value in Table 4 (central difference with a step size of 10^{-5}), and for all other results shown later in this paper. We find that the errors in reference derivative values under different finite-differencing step sizes are much smaller than the errors of adjoint derivative computation; we thus conclude that using the finite-difference derivatives as references is reasonable. In addition to the far-field velocity, we evaluate the accuracy of the derivative with respect to the FFD design variables ($\mathrm{d}C_D/\,\mathrm{d}\boldsymbol{x}$), shown in Table 5, where the design variables are four FFD points covering the Ahmed-body ramp. Again, the reference derivative is computed by using the finite-difference approach. For the best-case scenario ($\mathrm{d}C_D/\,\mathrm{d}x_1$), the error is less than 0.01%, whereas for the worst case ($\mathrm{d}C_D/\,\mathrm{d}x_3$), the error is no more than 0.3%. Overall, the average error is less than 0.1%. The derivative accuracy from our discrete adjoint implementation suffices for numerical optimization, as shown in Section 3.2.

3.2 Aerodynamic-Shape Optimization of Ahmed Body

In this section, we optimize the aerodynamic shape of the Ahmed body. Here we choose the 25° ramp-angle configuration with 1 million cells as our baseline case. This simple geometry allows us to demonstrate the basic optimization capability and to verify the optimization framework. Moreover, extensive experimental results are available for Ahmed body, which provide a better understanding of the flow fields and optimization results.

First, a single design variable—the ramp angle—is optimized to verify our adjoint framework. As reported in the original Ahmed-body experiment [49], reducing the ramp angle decreases the ramp-surface contribution to the drag; however, the contribution from the vertical rear-end surface increases. Thus, an optimal ramp angle between 0° and 40° minimizes the drag. This trend is reproduced by our CFD simulations (shown in Fig. 10 as the black line). Here, we perform CFD simulations for ramp angles ranging from 0° to 30° in 5° increments. As mentioned in Section 2.2, the ramp angle is controlled by using global shape control to move a set of FFD points together, while keeping the upper edge of the ramp fixed. The result indicates that the optimal ramp angle is approximately 15°. Next, we conduct an adjoint optimization for drag minimization, starting from the 25° ramp angle; the result is shown as the red dots in Fig. 10. The optimization converges to the optimum in only four iterations.

We next optimize an aerodynamic shape with multiple local design variables, as described in Table 6. We set 25 FFD points on the ramp surface, forcing the top edge of the ramp to remain unchanged. The FFD volume and points are shown in Fig. 2. We set up five linear constraints to force the y-direction slope at the

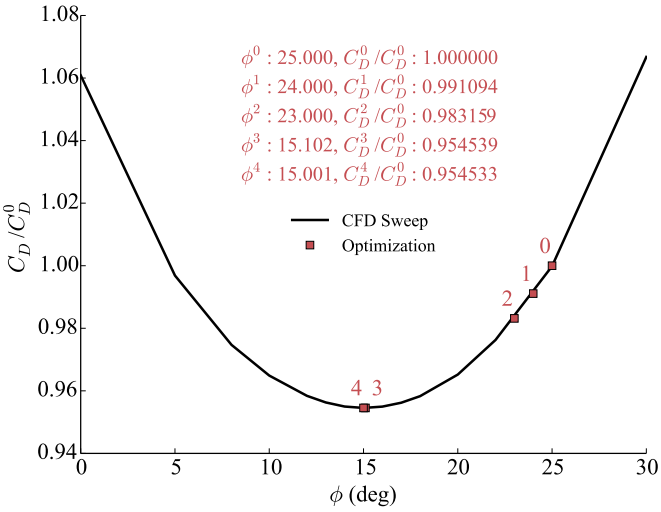

Figure 10: Optimization of ramp angle for Ahmed-body case. The optimization converges to the optimum in only four iterations.

Table 6: Ahmed-body ramp-shape optimization problem.

	Function or variable	Description	Quantity
Minimize	C_D	Drag coefficient	
with respect to	z	z coordinate of FFD points	**25**
subject to	g_y^{sym}=0	y direction slope at the symmetry plane is zero	**5**
	0 m $< \Delta z <$ 0.1 m	Design variable bounds	

symmetry plane to remain zero. In addition, we only allow the ramp surface to move upward, resulting in a shape that is easier to manufacture for wind tunnel experiments.

Figure 11 shows the convergence history of C_D and optimality for optimizing the Ahmed-body ramp shape. Both C_D and optimality converge in 23 iterations. The simulated drag coefficient for the baseline Ahmed body (C_D^0) is 0.310, which is 3.4% less than the experimental result (0.321) at the same Reynolds number [73]. We speculate that this lower C_D value is partially due to the exclusion of four supporting legs in our simulations. For the optimized geometry, C_D drops to 0.281—a 9.4% reduction in drag. In terms of speed, the optimization takes 24 hours when using 32 CPU cores (Intel Xeon E5-2680 v3 at 2.5 GHz). In total, we perform 24 adjoint computations and 29 flow solutions for the optimization. There are 2361 colors for the state Jacobian.

To further validate the optimization results, we conducted wind tunnel experiments using the intermediate and final Ahmed body ramp shapes during the optimization. The reduction in drag predicted by our numerical optimization agrees well with the experimental data, as shown in Fig. 11; the difference is 0.6% for the final shape.

Finally, we analyze the flow fields to better understand the results of optimization. For flows over bluff bodies, the flow separation near the rear end provides the major contribution to drag; the pressure drag dominates and the friction drag is relatively small [49]. Therefore, maintaining a favorable pressure distribution (i.e., an effective pressure recovery near the rear end) is critical for drag reduction. Figure 12 shows the pressure coefficient contours for baseline and optimized Ahmed bodies. Although we only simulate and optimize half of the Ahmed body, we show the full geometry for a better illustration. We also show the isosurface of the Q criterion, which is a useful metric to identify the vortex structure and is defined as

$$Q = \frac{1}{2}(\Omega_{i,j}\Omega_{i,j} - S_{i,j}S_{i,j}), \tag{19}$$

where Ω and S are the rotation and strain rates, respectively. A large positive Q implies that the rotational

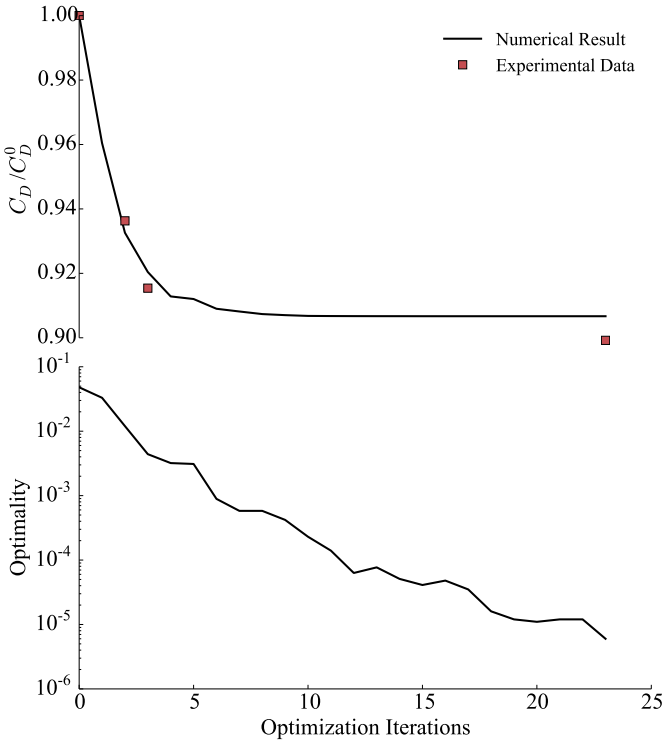

Figure 11: Convergence history of C_D and optimality for optimizing Ahmed-body ramp shape. Drag reduction is 9.4%. The optimization results are validated by wind tunnel experiments.

motion of the fluid dominates its strain (expanding, shrinking, or shearing) motion. For the baseline geometry, a low-pressure region appears near the top and side edges of the ramp surface, matching the experimental observation reported by Lienhart and Becker [74, Fig. 12]. Moreover, a distinct vortex structure appears in the simulation results: A corner vortex originates from the top corner of the ramp surface and propagates downstream. In addition, strong vorticity occurs on the ramp surface, implying a mild flow separation. Also, behind the vertical rear-end surface, two counter-rotating vortices form as a result of flow recirculation. The simulated vortex structure qualitatively agrees with the results observed in Ahmed's original experiments [49, Figs. 6 and 10]. This vortex structure is closely correlated with the pressure distribution, i.e., the low-pressure regions near the side edge and top edge are impacted by the corner vortex and the mild flow separation on the ramp, respectively.

Keeping the above baseline-flow structure in mind, we now examine the pressure distribution over the optimized shape. We find that the pressure distribution changes significantly. For the optimized geometry, the low-pressure regions near the top and side edges shrink, and a high-pressure region emerges in the middle of the ramp surface. By taking a close look at the optimized shape (top-right corner in Fig. 12), we see a bump near the side edge of the ramp. This bump raises the corner vortex and reduces its intensity, as shown by the Q isosurface in Fig. 12. As a result, the low-pressure region near the side edge is reduced. In addition, we find that the local ramp angle near the top edge is reduced in the optimized shape (Fig. 13), resulting in a weaker flow separation and higher pressure in the middle of the ramp surface. Note that this pressure increase, due to better pressure recovery, provides the major contribution to drag reduction. This conclusion is further confirmed by the pressure-coefficient profiles at the ramp and vertical rear-end surfaces of the Ahmed body, as shown in Fig. 13. The pressure on the ramp surface significantly increases, although the pressure on the vertical rear-end surface is slightly reduced.

For this section, we verified the adjoint optimization framework by optimizing with respect to a single design variable (ramp angle). We then optimized the local shape for the ramp surface and validated the optimization results experimentally. Moreover, we analyzed the flow in detail to confirm that the result of optimization is physically reasonable. In the next section, we apply this adjoint framework to two more

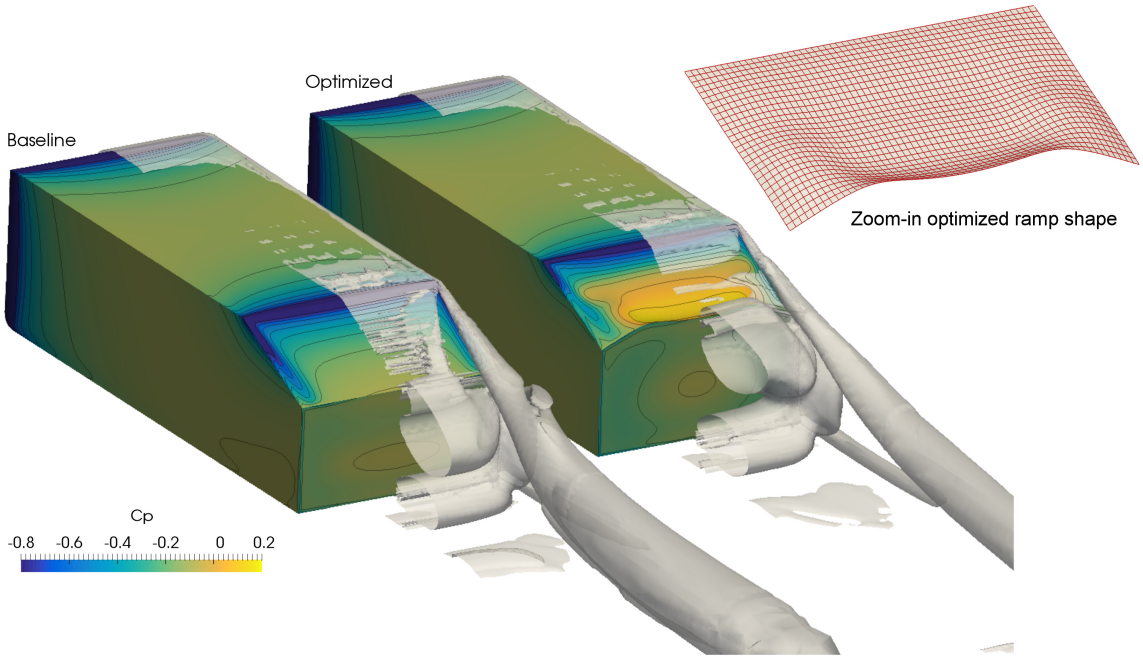

Figure 12: Pressure-coefficient contours for baseline and optimized Ahmed-body geometries. The transparent gray area denotes the isosurface of the Q criterion ($Q = 1000$). The optimized ramp shape provides better pressure recovery by modifying the rear-end vortex structure and the flow separation on the ramp.

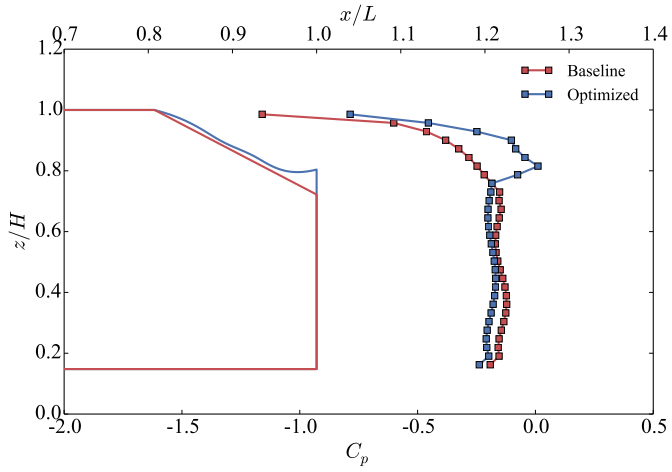

Figure 13: Ramp shape at symmetry plane and C_p profiles on ramp and vertical rear-end surfaces. The C_p profiles are based on the pressure averaged over the lateral (y) direction. The pressure increases significantly on the ramp surface, which contributes to the reduced drag.

complex applications: the aerodynamic-shape optimization of a UAV and of a car.

3.3 Aerodynamic-Shape Optimization of Unmanned Aerial Vehicle

Compared with full-size piloted aircraft, UAVs have the advantage of lower cost, longer endurance, and more flexibility with respect to the operating environment. Over the last few years, UAVs have gained unprecedented popularity for applications such as surveillance, reconnaissance, search and rescue, and scientific-research support [75]. However, most of the existing aerodynamic-shape-optimization studies have focused

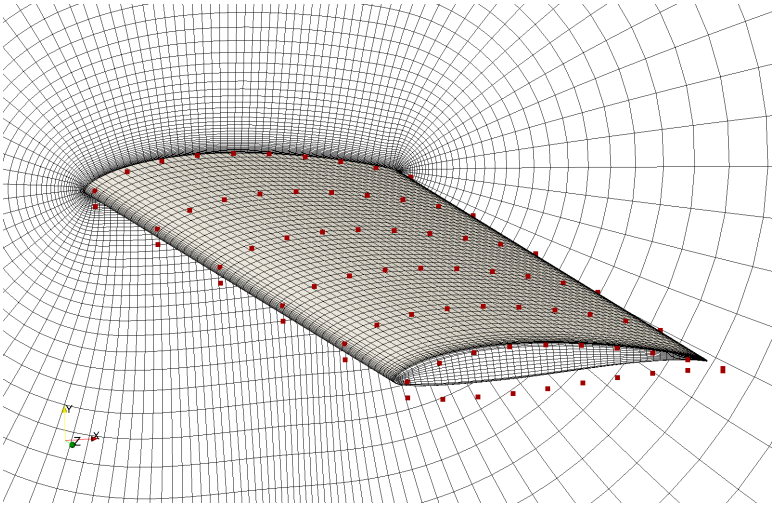

Figure 14: Structured hexahedral mesh for UAV wing. The red squares are the 120 FFD points to control the wing shape at six different spanwise locations.

Table 7: Setup for wing-shape optimization for UAV (Odyssey).

	Function or variable	Description	Quantity
minimize	C_D	Drag coefficient	
with respect to	y	y coordinate of FFD points	120
	γ	Twist	6
	α	Angle of attack	1
		Total design variables	**127**
subject to	C_L=0.75	Lift-coefficient constraint	1
	$t \geq 0.5t_{\text{baseline}}$	Minimum-thickness constraint	400
	$V \geq V_{\text{baseline}}$	Minimum-volume constraint	1
	$\Delta y_{\text{LE}}^{\text{upper}} = -\Delta y_{\text{LE}}^{\text{lower}}$	Fixed leading-edge constraint	20
	$\Delta y_{\text{TE}}^{\text{upper}} = -\Delta y_{\text{TE}}^{\text{lower}}$	Fixed trailing-edge constraint	20
	$-0.5 \text{ m} < \Delta y < 0.5 \text{ m}$	Design variable bounds	
		Total constraints	**442**

on full-size aircraft whereas relatively little effort has been devoted to UAVs. Compared with full-size aircraft, most UAVs cruise at a low speed and therefore operate in the regime of incompressible flow. This makes our adjoint optimization framework especially relevant, because it is based on the incompressible-flow solver simpleFoam.

In this section, we optimize the aerodynamic shape of a UAV wing. The wing geometry is taken from a multi-mission UAV prototype called Odyssey [76]. The wing planform is rectangular with an aspect ratio of 8.57 and a span of 4.572 m. The wing section profile is Eppler214. No twist or sweep is adopted in the baseline wing geometry. For the CFD, we generate a structured hexahedral mesh with approximately 0.5 million cells and an average y^+ of 31 (Fig. 14). The simulation domain extends to 30 chord lengths. The inlet velocity is 24.81 m/s (Mach number 0.074) and the Reynolds number is 9.0×10^5.

Table 7 summarizes the optimization problem, whose objective is to minimize the drag coefficient. For the design variables, we use 120 FFD points to control the local wing shape at six different spanwise locations, as shown in Fig. 14. In addition, the twists at these six spanwise locations are selected to be the design variables along with the angle of attack. The total number of design variables is 127. We constrain the lift coefficient ($C_L = L/0.5\rho U_0^2 A_{\text{ref}}$, where L is the lift force) to be 0.75. In addition, we limit the local wing thickness to be greater than 50% of the baseline thickness. This is done by sampling the points in a 20×20

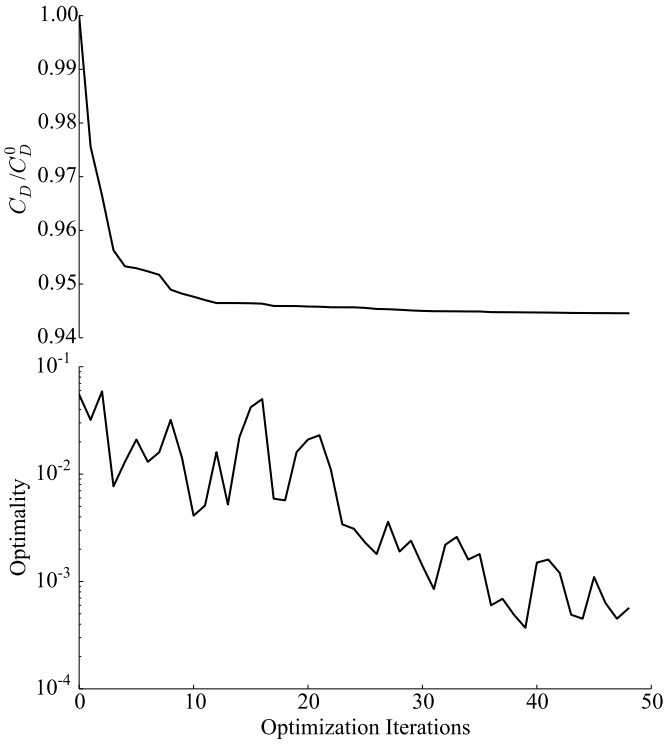

Figure 15: Convergence history of C_D and optimality for optimizing UAV wing shape. Drag is reduced by 5.6%.

grid in the chordwise and spanwise directions, covering full span and 1% to 99% chord. Finally, we constrain the total volume of the optimized wing to be greater than or equal to that of the baseline wing, and the leading and trailing edges of the wing are fixed. In total, we have 442 constraints for this case.

Figure 15 shows the drag coefficient and optimality convergence history for the optimization. The baseline and optimized drag coefficients are 0.0378 and 0.0357, respectively, and the drag is reduced by 5.6%. This reduction in drag is comparable to previous results of aerodynamic-shape optimization for full-size aircraft wings [29, 31, 71]. The optimization took 30 hours on 32 CPU cores (Intel Xeon E5-2680 v3 at 2.5 GHz). In total, we conducted 48 adjoint computations and 79 flow solutions for the optimization. The state Jacobian has 1386 colors.

Figure 16 compares in more detail the baseline and optimized results. According to Lanchester–Prandtl wing theory, the induced drag of a three-dimensional, finite-span wing is minimized if the lift distribution along the spanwise direction is elliptical. As shown in Fig. 16(b), the spanwise lift distribution produced by the baseline wing differs significantly from the elliptical lift distribution, but the optimized wing achieves a elliptical distribution. Because the induced drag typically constitutes the largest proportion of the total drag for subsonic wings, changing the spanwise lift distribution to elliptical is the major driving force for aerodynamic optimization. In our case, the lift distribution can be modified by fine tuning the wing section profiles and twist distributions along the span. Figures 16(e)–16(g) compare the baseline and optimized wing profiles and pressure distributions at three spanwise locations (5% root, 50% mid, and 95% tip). The twist and maximum thickness distributions are shown in Figs. 16(c) and 16(d). We find that the twists increase at all spanwise locations in the optimized shape. Near the tip, the twist increases by almost 1°, which lowers the aerodynamic loading and drives the lift distribution to elliptical. In addition, we observe that the camber increases near the root section, because the airfoils there are subject to higher lift coefficients.

3.4 Optimizing Aerodynamic Shape of a Car

Since Othmer [36] first applied the adjoint method to designing the aerodynamic shape of a car, the popularity of this approach has grown rapidly in the automotive industry [38–40]. However, to date, studies of adjoint-

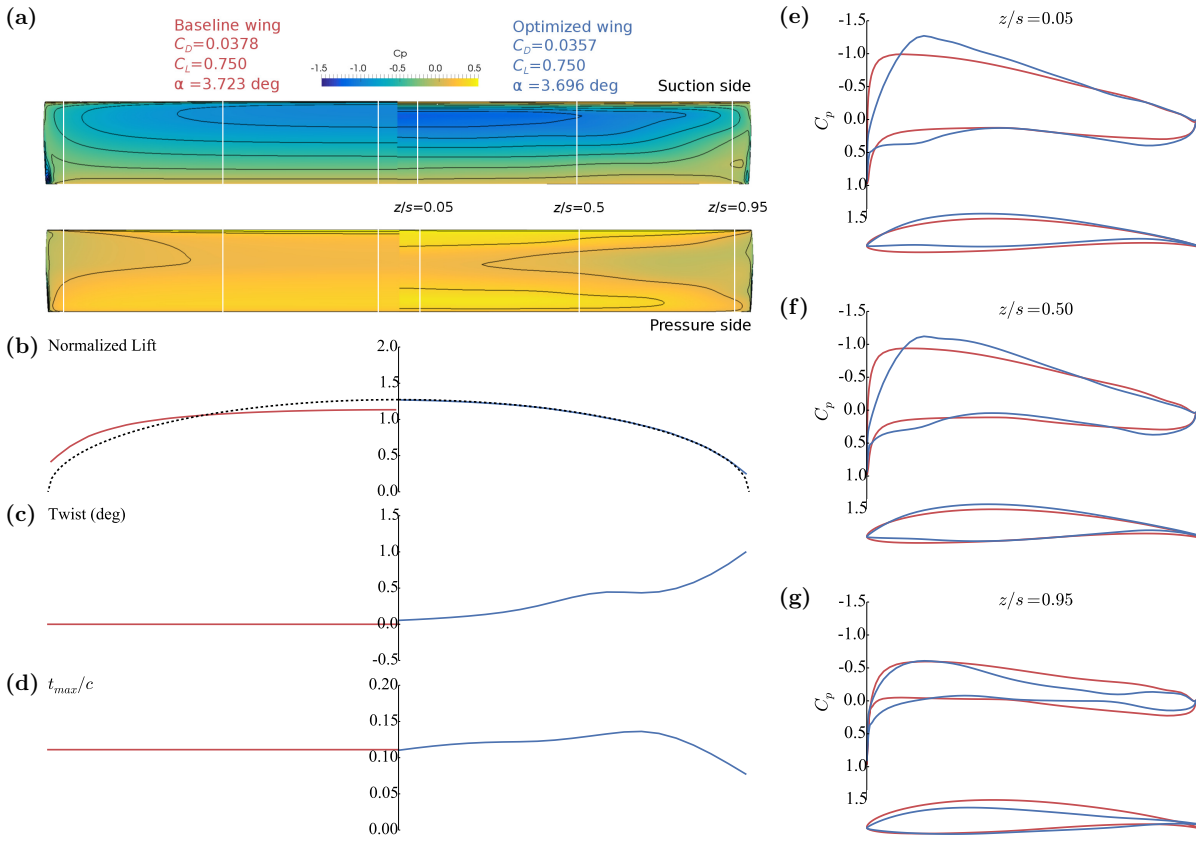

Figure 16: Results of optimizing UAV wing. Fine tuning the wing-section profiles and twist distributions along the span gives the optimized shape the desired elliptical lift distribution. Drag is reduced by 5.6%.

based car-shape optimization have been based on OpenFOAM's built-in continuous adjoint solver, and the optimization problems are unconstrained. Constraints are necessary to obtain practical results for optimization. For example, when optimizing the shape of a car's glass, one needs to limit its curvature and avoid wavy shapes so that it can be manufactured. Instead of imposing a curvature constraint, previous studies limited their optimization to small changes [i.e., $\mathcal{O}(0.01$ m)] in a handful of design variables [36, 38], so they obtained an optimum only in a very limited design space. The approach developed herein addresses this limitation, so we perform a constrained aerodynamic-shape optimization of a car. More specifically, we set up 50 design variables to control the shape along with 38 curvature and slope constraints. Moreover, we set relatively large bounds (± 0.2 m) for the design variables, allowing us to find the optimum in a larger design space.

The car geometry we optimize is a fastback sedan model called DrivAer. The DrivAer geometry is a combination of an Audi A4 and a BMW 3 Series model and was originally proposed by Heft *et al.* [77, 78]. The DrivAer model has been widely used as a benchmark. In this section, we focus on optimizing its rear-end shape. To reduce the mesh size and improve flow convergence, we simplify the geometry by smoothing the underbody and removing wheels, mirrors, and handles. We generate 1 million unstructured snappy hexahedral mesh to simulate half of the DrivAer model with an average y^+ of 46. The geometry and mesh for DrivAer are shown in Fig. 17. The simulation-domain size is $8L$, $2L$, and $2L$ in the x, y, and z directions, respectively, where L is the DrivAer model length. The inlet velocity is 10 m/s and the corresponding Reynolds number is 3.1×10^6.

The objective of the optimization is to minimize the drag coefficient. We define 50 FFD points to control the rear-end shape, as shown in Fig. 17. Similar to what was done for the Ahmed body, the y slope at the symmetry plane is constrained to be zero. Moreover, we take the manufacturing constraints into consideration by imposing a mean curvature constraint on the back glass. The mean curvature of the

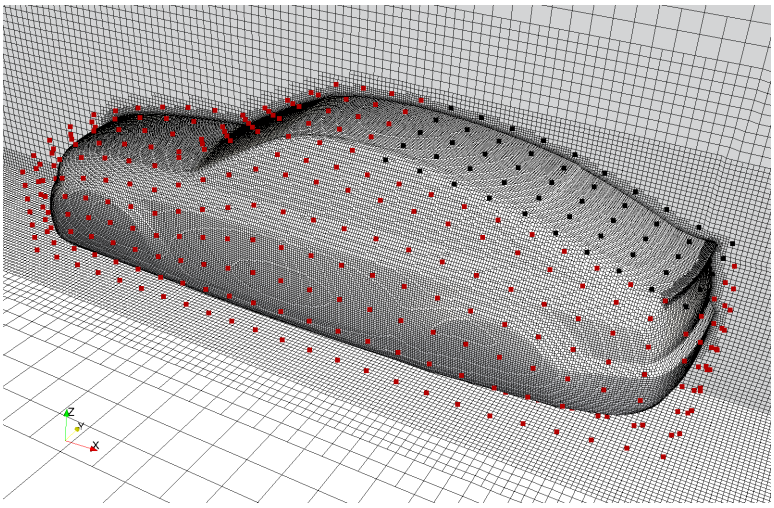

Figure 17: Unstructured snappy hexahedral mesh for DrivAer model. The black and red squares are the FFD points. Only the black FFD points are selected as design variables for manipulating the rear-end shape, whereas the red FFD points remain unchanged during the optimization.

Table 8: DrivAer rear-end-shape optimization problem.

	Function or variable	Description	Quantity
minimize	C_D	Drag coefficient	
with respect to	z	z coordinate of FFD points	**50**
subject to	g_y^{symm}=0	y-direction slope at symmetry plane is zero	10
	$H^{\text{glass}} \leq H^{\text{glass}}_{\text{baseline}}$	Mean curvature constraint for back glass	1
	$\Delta z_{i,j} \geq \Delta z_{i,j+1}$	Monotonic constraint in y direction	27
	-0.2 m $< \Delta z < 0.2$ m	Design-variable bounds	
		Total constraints	**38**

optimized back glass is enforced to be less than that of the baseline DrivAer model. The mean curvature of a parametric surface (surface mesh), $\boldsymbol{x}_S = \boldsymbol{x}_S(u, v)$, is defined as

$$H = \frac{EN - 2FM + GL}{2(EG - F^2)}, \tag{20}$$

where $\boldsymbol{x}_S$ is the surface-coordinates vector, u and v are the parameterization variables, and E, F, G, L, M, and N are the coefficients from the first and second fundamental forms of parametric surfaces,

$$E = \boldsymbol{x}_u \cdot \boldsymbol{x}_u, \quad F = \boldsymbol{x}_u \cdot \boldsymbol{x}_v, \quad G = \boldsymbol{x}_v \cdot \boldsymbol{x}_v, \quad L = \boldsymbol{x}_{uu} \cdot \boldsymbol{n}, \quad M = \boldsymbol{x}_{uv} \cdot \boldsymbol{n}, \quad N = \boldsymbol{x}_{vv} \cdot \boldsymbol{n}, \tag{21}$$

where $\boldsymbol{n}$ is the surface unit normal vector and $\boldsymbol{x}_u$ denotes the first-order derivative of $\boldsymbol{x}$ with respect to u, similarly to the other derivatives. Given Eqs. 20 and 21, we can compute $\mathrm{d}H/\,\mathrm{d}\boldsymbol{x}_S$. To connect surface coordinates and design variables, we map $\mathrm{d}H/\,\mathrm{d}\boldsymbol{x}_S$ to $\mathrm{d}H/\,\mathrm{d}\boldsymbol{x}$ by using the FFD approach introduced in Section 2.2. To avoid wavy shapes, we also impose monotonic constraints in the y direction. In total, we set 38 constraints for this case. The full setup of the optimization problem is summarized in Table 8.

The convergence history of C_D and optimality for the DrivAer rear-end-shape optimization are shown in Fig. 18. The drag coefficient drops from 0.140 to 0.123 in 21 iterations—a 12.1% reduction in drag. Given that DrivAer is designed by experienced aerodynamicists, this drag reduction is a significant improvement. However, the optimality only drops one order of magnitude and stops decreasing at about 1×10^{-3}. The optimization took 58 hours on 32 CPU cores (Intel Xeon E5-2680 v3 at 2.5 GHz). In total, we conducted 21 adjoint computations and 71 flow solutions for the optimization. There were 2377 colors for the state

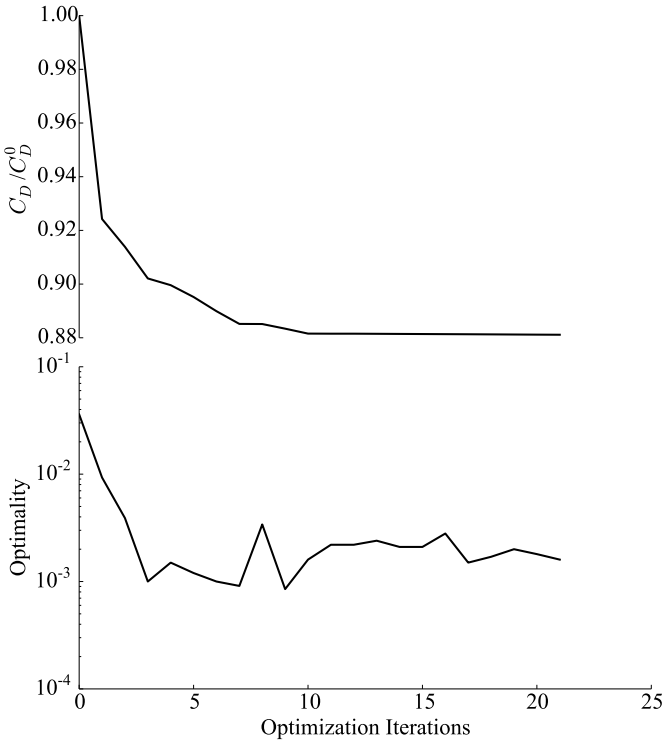

Figure 18: Convergence history of C_D and optimality for DrivAer rear-end-shape optimization. Drag is reduced by 12.1%.

Jacobian. This is a longer runtime than the previous two cases, which is primarily due to the poorer flow convergence for the DrivAer model. As a result, the adjoint-equation solution takes longer to converge. Moreover, the adjoint derivative accuracy is degraded. For instance, the far-field velocity derivative $\mathrm{d}C_D/\,\mathrm{d}u_0$ is 0.02765 based on finite differences, whereas the adjoint computation predicts 0.02744 with an error of 0.76%. Although this level of error remains acceptable for optimization, it increases the overall runtime of the optimization: the optimization requires more function evaluations (flow solutions) at each iteration. The poor flow convergence is known to be the one of the most challenging issues in car aerodynamic-shape optimization [36], so improvements are needed in this area.

To analyze the optimization results, we show in Fig. 19 the pressure distributions and the velocity contours for the baseline and optimized geometries. As discussed by Hucho and Sovran [79], the goal of car rear-end-shape design is to maximize the pressure (also known as the base pressure) to the extent possible in this region. We know that the drag for a bluff body can be impacted by the rear-end slant angle and rear-end height. However, for a fastback car geometry with smoothed top rear-end contour, the rear-end height becomes the main influencing factor, and its optimal value depends strongly on the upstream flow conditions [79]. As shown in Figs. 19(a) and 19(b), the rear-end slant angle remains almost unchanged at the symmetry plane for the optimized shape; however, the rear-end height decreases, pushing the rear-end recirculation vortex downward. This reduction in rear-end height is more evident near the top corners of the vertical rear-end surface, as shown in Figs. 19(d) and 19(e). As a result, the base pressure increases for the optimized shape, which contributes to drag reduction. To confirm this, we plot in Fig. 19(f) the C_p distribution on the back glass and the vertical rear-end surface for the baseline and optimized shapes. Although the pressure on the back glass decreases for the optimized shape, a larger portion of pressure increases on the vertical rear-end surface.

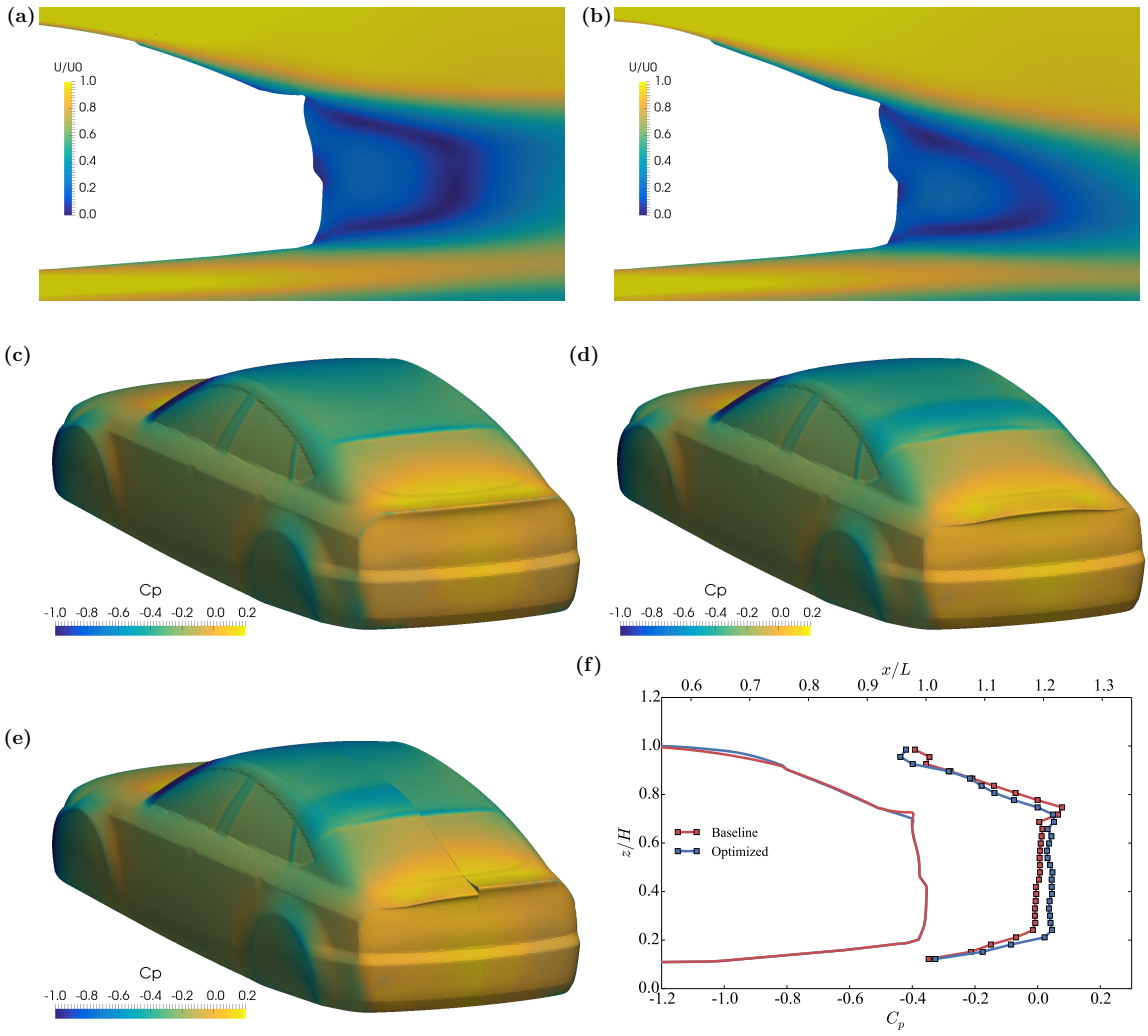

Figure 19: DrivAer rear-end-shape optimization results. Panels (a) and (c) show the baseline geometry, and panels (b) and (d) show the optimized geometry. Panels (e) shows a side-by-side comparison, and panel (f) shows the C_p and rear-end profiles at the symmetry plane. The C_p profiles are based on the pressure averaged over the lateral (y) direction. The optimized shape decreases the rear-end height, thereby increasing the base pressure on the vertical rear-end surface for a total reduction in drag of 12.1%.

4 Conclusions

In this work, we develop an efficient discrete adjoint within a constrained shape-design optimization framework based on OpenFOAM. We demonstrate the capability of this framework by using it to optimize the aerodynamic design of a UAV and a car. This optimization framework consists of multiple modules: a geometric parameterization module based on FFDs, an inverse-distance method for volume-mesh deformation, a standard OpenFOAM incompressible solver for flow simulation, a discrete adjoint solver for derivative computation, and a sequential quadratic programming gradient-based optimizer (SNOPT) for nonlinear constrained-optimization problems.

We then evaluate the performance of our framework by using the Ahmed body as a baseline. We find that the derivatives are computed as fast as the primal flow simulation, with a ratio of adjoint runtime to flow runtime ranging from 0.8 to 1.1. Furthermore, the adjoint derivative computation scales well up to 10 million cells and 1024 CPU cores. This speed and scalability is primarily attributed to the efficient parallel graph-coloring acceleration algorithm developed herein. The average error in the derivative computation is less than 0.1%

We further verify the optimization framework by using it to optimize the ramp angle and shape of the Ahmed body. The result for optimizing the ramp angle matches the optimal value predicted by CFD simulations. We also compare the simulated pressure distributions and rear-end vortex structures with experimental observations for the baseline Ahmed body geometry and obtain consistent results. Building on this, we explain the underlying physics of the optimized result. We find that the optimized ramp shape provides better pressure recovery by modifying the rear-end vortex structure and the flow separation on the ramp, achieving a 9.4% reduction in drag. The intermediate and final shapes of the ramp during the optimization were built and tested in a wind tunnel experiment, and we observed a remarkably good agreement in drag reduction. These results and analyses confirm that the optimized shapes are physically valid and can yield usable designs.

Finally, we use the framework to optimize the aerodynamic design of a UAV wing and a car rear-end design with various physical and geometric constraints (e.g., volume, thickness, curvature, and lift constraints). In the UAV wing shape optimization, we find that the optimized result provides a theoretically optimal elliptical lift distribution by fine-tuning the wing-section profiles and twist distributions, thereby reducing the drag by 5.6%. Upon optimizing the rear-end shape of the car, the drag is reduced by 12.1%, most of which is attributed to the decrease in rear-end height, which increases the base pressure on the vertical rear-end surface.

In this paper, we demonstrate that the proposed adjoint framework can tackle shape-optimization problems with over 100 design variables subject to various geometric and physical constraints (volume, thickness, slope, curvature, and lift). Moreover, our adjoint framework is easily adaptable to other OpenFOAM flow solvers to handle shape optimizations involving heat transfer, hydrodynamics, and internal flows. Given the popularity of OpenFOAM in industry, the proposed optimization framework has the potential to become a useful design tool in a wide range of applications, such as aircraft, cars, ships, and turbomachinery.

One limitation of our adjoint implementation is its use of finite differences to compute the state Jacobian and other partial derivatives. To address this limitation, we plan to use automatic differentiation to compute partial derivatives in the future. Moreover, given that high memory usage is the bottleneck of our adjoint implementation for handling cases with $\mathcal{O}$(100 million) cells, we plan to implement the matrix-free adjoint approach to avoid explicitly storing the state Jacobian matrix. We also plan to improve the preconditioning matrix construction to reduce the memory required by ILU fill-in for the adjoint solution.

Acknowledgments

The computations were done in the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation Grant No. ACI-1548562, as well as the Flux HPC cluster at the University of Michigan Center of Advanced Computing. The authors would like to thank Ney R. Secco and Timothy R. Brooks for their helpful comments and discussion to improve the manuscript. The authors also thank Peter Bachant for providing the wind-tunnel-experiment data.

References

[1] H. G. Weller, G. Tabor, H. Jasak, C. Fureby, A tensorial approach to computational continuum mechanics using object-oriented techniques, Computers in Physics 12 (6) (1998) 620–631.

[2] H. Jasak, A. Jemcov, Z. Tuković, OpenFOAM: A C++ library for complex physics simulations, in: International Workshop on Coupled Methods in Numerical Dynamics, IUC, Citeseer, 2007.

[3] H. Jasak, OpenFOAM: Introduction, capabilities and HPC needs, in: Cyprus Advanced HPC Workshop, 2012.

[4] D. A. Lysenko, I. S. Ertesvåg, K. E. Rian, Modeling of turbulent separated flows using OpenFOAM, Computers & Fluids 80 (2013) 408–422. doi:10.1016/j.compfluid.2012.01.015.

[5] W. Wu, M. M. Bernitsas, K. Maki, RANS simulation versus experiments of flow induced motion of circular cylinder with passive turbulence control at 35,000< Re <130,000, Journal of Offshore Mechanics and Arctic Engineering 136 (4) (2014) 041802. doi:10.1115/1.4027895.

[6] E. Robertson, V. Choudhury, S. Bhushan, D. Walters, Validation of OpenFOAM numerical methods and turbulence models for incompressible bluff body flows, Computers & Fluids 123 (2015) 122–145. doi:10.1016/j.compfluid.2015.09.010.

[7] V. D'Alessandro, S. Montelpare, R. Ricci, Detached–eddy simulations of the flow over a cylinder at Re=3900 using OpenFOAM, Computers & Fluids 136 (2016) 152–169. doi:10.1016/j.compfluid.2016.05.031.

[8] H. Jasak, M. Beaudoin, OpenFOAM turbo tools: From general purpose CFD to turbomachinery simulations, 2011.

[9] R. Bouwman, Design of wind turbines using OpenFOAM as part of the CAE chain–overview, in: First Symposium on OpenFOAM in Wind Energy, 2013.

[10] S. Nakao, M. Kashitani, T. Miyaguni, Y. Yamaguchi, A study on high subsonic airfoil flows in relatively high reynolds number by using OpenFOAM, Journal of Thermal Science 23 (2) (2014) 133–137. doi:10.1007/s11630-014-0687-5.

[11] H. Medina, A. Beechook, J. Saul, S. Porter, S. Aleksandrova, S. Benjamin, Open source computational fluid dynamics using OpenFOAM, in: Royal Aeronautical Society, General Aviation Conference, London,, 2015.

[12] R. Bevan, D. Poole, C. Allen, T. Rendall, Adaptive surrogate-based optimization of vortex generators for tiltrotor geometry, Journal of Aircraft 54 (3) (2017) 1011–1024. doi:10.2514/1.C033838.

[13] T. Blacha, M. Islam, The aerodynamic development of the new Audi Q5, SAE International Journal of Passenger Cars-Mechanical Systems 10 (2017-01-1522). doi:10.4271/2017-01-1522.

[14] R. Lietz, L. Larson, P. Bachant, J. Goldstein, R. Silveira, M. Shademan, P. Ireland, K. Mooney, An extensive validation of an open source based solution for automobile external aerodynamics, Tech. rep., SAE Technical Paper (2017-01-1524) (2017).

[15] O. Pironneau, On optimum profiles in Stokes flow, Journal of Fluid Mechanics 59 (01) (1973) 117–128. doi:10.1017/S002211207300145X.

[16] A. Jameson, Aerodynamic design via control theory, Journal of Scientific Computing 3 (3) (1988) 233–260. doi:10.1007/BF01061285.

[17] A. Jameson, L. Martinelli, N. A. Pierce, Optimum aerodynamic design using the Navier–Stokes equations, Theoretical and Computational Fluid Dynamics 10 (1–4) (1998) 213–237. doi:10.1007/s001620050060.

[18] E. J. Nielsen, W. K. Anderson, Aerodynamic design optimization on unstructured meshes using the Navier-Stokes equations, AIAA Journal 37 (11). doi:10.2514/2.640.

[19] D. J. Mavriplis, Discrete adjoint-based approach for optimization problems on three-dimensional unstructured meshes, AIAA Journal 45 (4) (2007) 740. doi:10.2514/1.22743.

[20] 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 Journal 46 (4) (2008) 863–873. doi:10.2514/1.29123.

[21] 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, in: 21st AIAA Computational Fluid Dynamics Conference, San Diego, CA, 2013. doi:10.2514/6.2013-2581.

[22] J. R. R. A. Martins, A. B. Lambe, Multidisciplinary design optimization: A survey of architectures, AIAA Journal 51 (9) (2013) 2049–2075. doi:10.2514/1.J051895.

[23] C. A. Mader, J. R. R. A. Martins, Stability-constrained aerodynamic shape optimization of flying wings, Journal of Aircraft 50 (5) (2013) 1431–1449. doi:10.2514/1.C031956.

[24] 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 Journal 52 (5) (2014) 935–951. doi:10.2514/1.J052255.

[25] S. Xu, D. Radford, M. Meyer, J.-D. Müller, Stabilisation of discrete steady adjoint solvers, Journal of Computational Physics 299 (2015) 175–195. doi:10.1016/j.jcp.2015.06.036.

[26] N. Garg, G. K. W. Kenway, J. R. R. A. Martins, Y. L. Young, High-fidelity multipoint hydrostructural optimization of a 3-D hydrofoil, Journal of Fluids and Structures 71 (2017) 15–39. doi:10.1016/j.jfluidstructs.2017.02.001.

[27] G. K. Kenway, G. J. Kennedy, J. R. R. A. Martins, A CAD-free approach to high-fidelity aerostructural optimization, in: Proceedings of the 13th AIAA/ISSMO Multidisciplinary Analysis Optimization Conference, Fort Worth, TX, 2010, AIAA 2010-9231. doi:10.2514/6.2010-9231.

[28] G. K. W. Kenway, J. R. R. A. Martins, Multipoint high-fidelity aerostructural optimization of a transport aircraft configuration, Journal of Aircraft 51 (1) (2014) 144–160. doi:10.2514/1.C032150.

[29] Z. Lyu, G. K. W. Kenway, J. R. R. A. Martins, Aerodynamic shape optimization investigations of the Common Research Model wing benchmark, AIAA Journal 53 (4) (2015) 968–985. doi:10.2514/1.J053318.

[30] 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, Journal of Aircraft 53 (1) (2016) 276–293. doi:10.2514/1.C033328.

[31] 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, Aerospace Science and Technology 75 (2018) 183–199. doi:10.1016/j.ast.2018.01.016.

[32] R. E. Perez, P. W. Jansen, J. R. R. A. Martins, pyOpt: A Python-based object-oriented framework for nonlinear constrained optimization, Structural and Multidisciplinary Optimization 45 (1) (2012) 101–118. doi:10.1007/s00158-011-0666-3.

[33] M. B. Giles, N. A. Pierce, An introduction to the adjoint approach to design, Flow, Turbulence and Combustion 65 (3-4) (2000) 393–415. doi:10.1023/A:1011430410075.

[34] S. Nadarajah, A. Jameson, A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization, in: Proceedings of the 38th AIAA Aerospace Sciences Meeting and Exhibit, Reno, NV, 2000. doi:10.2514/6.2000-667.

[35] C. Othmer, A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows, International Journal for Numerical Methods in Fluids 58 (8) (2008) 861–877. doi:10.1002/fld.1770.

[36] C. Othmer, Adjoint methods for car aerodynamics, Journal of Mathematics in Industry 4 (1) (2014) 6. doi:10.1186/2190-5983-4-6.

[37] E. Papoutsis-Kiachagias, N. Magoulas, J. Mueller, C. Othmer, K. Giannakoglou, Noise reduction in car aerodynamics using a surrogate objective function and the continuous adjoint method with wall functions, Computers & Fluids 122 (2015) 223–232. doi:10.1016/j.compfluid.2015.09.002.

[38] T. Han, S. Kaushik, K. Karbon, B. Leroy, K. Mooney, S. Petropoulou, J. Papper, Adjoint-driven aerodynamic shape optimization based on a combination of steady state and transient flow solutions, SAE International Journal of Passenger Cars-Mechanical Systems 9 (2016-01-1599) (2016) 695–709. doi:10.4271/2016-01-1599.

[39] E. Papoutsis-Kiachagias, K. Giannakoglou, Continuous adjoint methods for turbulent flows, applied to shape and topology optimization: industrial applications, Archives of Computational Methods in Engineering 23 (2) (2016) 255–299.

[40] G. K. Karpouzas, E. M. Papoutsis-Kiachagias, T. Schumacher, E. de Villiers, K. C. Giannakoglou, C. Othmer, Adjoint optimization for vehicle external aerodynamics, International Journal of Automotive Engineering 7 (1) (2016) 1–7. doi:10.20485/jsaeijae.7.1_1.

[41] M. Towara, U. Naumann, A discrete adjoint model for OpenFOAM, Procedia Computer Science 18 (2013) 429–438.

[42] A. Sen, Industrial applications of discrete adjoint OpenFOAM, in: 15th European Workshop on Automatic Differentiation, 2014.

[43] A. Sen, M. Towara, U. Naumann, Discrete adjoint of an implicit coupled solver based on foam-extend using algorithmic differentiation, in: 16th European Workshop on Automatic Differentiation, 2014.

[44] A. Sen, Effective sensitivity computation for aerodynamic optimization using discrete adjoint OpenFoam, in: 19th European Workshop on Automatic Differentiation, 2016.

[45] S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith, Efficient Management of Parallelism in Object Oriented Numerical Software Libraries, Birkhäuser Press, 1997, pp. 163–202.

[46] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc users manual, Tech. Rep. ANL-95/11 - Revision 3.4, Argonne National Laboratory (2013).

[47] A. B. Lambe, J. R. R. A. Martins, Extensions to the design structure matrix for the description of multidisciplinary design, analysis, and optimization processes, Structural and Multidisciplinary Optimization 46 (2) (2012) 273–284. doi:10.1007/s00158-012-0763-y.

[48] P. E. Gill, W. Murray, M. A. Saunders, SNOPT: An SQP algorithm for large-scale constrained optimization, SIAM Journal of Optimization 12 (4) (2002) 979–1006. doi:10.1137/S1052623499350013.

[49] S. Ahmed, G. Ramm, G. Faltin, Some salient features of the time-averaged ground vehicle wake, Tech. rep., SAE Technical Paper, No. 840300 (1984).

[50] E. Luke, E. Collins, E. Blades, A fast mesh deformation method using explicit interpolation, Journal of Computational Physics 231 (2) (2012) 586–601. doi:10.1016/j.jcp.2011.09.021.

[51] A. De Boer, M. Van der Schoot, H. Bijl, Mesh deformation based on radial basis function interpolation, Computers & Structures 85 (11) (2007) 784–795. doi:10.1016/j.compstruc.2007.01.013.

[52] S. V. Patankar, D. B. Spalding, A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows, International Journal of Heat and Mass Transfer 15 (10) (1972) 1787–1806. doi:10.1016/0017-9310(72)90054-3.

[53] C. Rhie, W. L. Chow, Numerical study of the turbulent flow past an airfoil with trailing edge separation, AIAA Journal 21 (11) (1983) 1525–1532. doi:10.2514/3.8284.

[54] H. Jasak, Error analysis and estimation for finite volume method with applications to fluid flow, Ph.D. thesis, Imperial College of Science, Technology and Medicine (1996).

[55] P. Spalart, S. Allmaras, A one-equation turbulence model for aerodynamic flows, in: 30th Aerospace Sciences Meeting and Exhibit, 1992. doi:10.2514/6.1992-439.

[56] J. E. Peter, R. P. Dwight, Numerical sensitivity analysis for aerodynamic optimization: A survey of approaches, Computers & Fluids 39 (3) (2010) 373–391. doi:10.1016/j.compfluid.2009.09.013.

[57] M. J.R.R.A., H. J. T., Multidisciplinary design optimization of aircraft configurations. Part 1: A modular coupled adjoint approach, in: Lecture Series, Von Karman Institute for Fluid Dynamics, Sint-Genesius-Rode, Belgium, 2016.

[58] J. R. R. A. Martins, P. Sturdza, J. J. Alonso, The complex-step derivative approximation, ACM Transactions on Mathematical Software 29 (3) (2003) 245–262. doi:10.1145/838250.838251.

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

[60] E. J. Nielsen, W. L. Kleb, Efficient construction of discrete adjoint operators on unstructured grids using complex variables, AIAA Journal 44 (4) (2006) 827–836. doi:10.2514/1.15830.

[61] R. Roth, S. Ulbrich, A discrete adjoint approach for the optimization of unsteady turbulent flows, Flow, Turbulence and Combustion 90 (4) (2013) 763–783. doi:10.1007/s10494-012-9439-3.

[62] A. Curtis, M. J. Powell, J. K. Reid, On the estimation of sparse Jacobian matrices, IMA Journal of Applied Mathematics 13 (1) (1974) 117–119. doi:10.1093/imamat/13.1.117.

[63] T. F. Coleman, J. J. Moré, Estimation of sparse Jacobian matrices and graph coloring blems, SIAM Journal on Numerical Analysis 20 (1) (1983) 187–209. doi:10.1137/0720013.

[64] A. H. Gebremedhin, F. Manne, A. Pothen, What color is your Jacobian? Graph coloring for computing derivatives, SIAM Review 47 (4) (2005) 629–705. doi:10.1137/S0036144504444711.

[65] E. G. Boman, D. Bozdağ, U. Catalyurek, A. H. Gebremedhin, F. Manne, A scalable parallel graph coloring algorithm for distributed memory computers, in: European Conference on Parallel Processing, Springer, 2005, pp. 241–251.

[66] D. Bozdağ, U. Catalyurek, A. H. Gebremedhin, F. Manne, E. G. Boman, F. Özgüner, A parallel distance-2 graph coloring algorithm for distributed memory computers, in: International Conference on High Performance Computing and Communications, Springer, 2005, pp. 796–806.

[67] S. Xu, S. Timme, K. J. Badcock, Enabling off-design linearised aerodynamics analysis using Krylov subspace recycling technique, Computers & Fluids 140 (2016) 385–396. doi:10.1016/j.compfluid.2016.10.018.

[68] S. Xu, S. Timme, Robust and efficient adjoint solver for complex flow conditions, Computers & Fluids 148 (2017) 26–38. doi:10.1016/j.compfluid.2017.02.012.

[69] G. K. W. Kenway, J. R. R. A. Martins, Buffet onset constraint formulation for aerodynamic shape optimization, AIAA Journal 55 (6) (2017) 1930–1947. doi:10.2514/1.J055172.

[70] P. E. Gill, W. Murray, M. A. Saunders, M. H. Wright, User's Guide for SNOPT 5.3: A Fortran Package for Large-scale Nonlinear Programming, Systems Optimization Laboratory, Stanford University, California, 94305-4023, Technical Report SOL 98-1 (1998).

[71] G. K. W. Kenway, J. R. R. A. Martins, Multipoint aerodynamic shape optimization investigations of the Common Research Model wing, AIAA Journal 54 (1) (2016) 113–128. doi:10.2514/1.J054154.

[72] M. Towara, U. Naumann, Implementing the discrete adjoint formulation in OpenFOAM, in: 3rd Argonne AD Workshop, 2015.

[73] W. Meile, G. Brenn, A. Reppenhagen, A. Fuchs, Experiments and numerical simulations on the aerodynamics of the Ahmed body, CFD Letters 3 (1) (2011) 32–39.

[74] H. Lienhart, S. Becker, Flow and turbulence structure in the wake of a simplified car model, Tech. rep., SAE Technical Paper, 2003-01-0656 (2003).

[75] G. J. Vachtsevanos, K. P. Valavanis, Military and civilian unmanned aircraft, in: Handbook of Unmanned Aerial Vehicles, Springer, 2015, pp. 93–103.

[76] H. Ryaciotaki-Boussalis, D. Guillaume, Computational and experimental design of a fixed-wing UAV, in: Handbook of Unmanned Aerial Vehicles, Springer, 2015, pp. 109–141.

[77] A. Heft, T. Indinger, N. Adams, Investigation of unsteady flow structures in the wake of a realistic generic car model, in: 29th AIAA Applied Aerodynamics Conference, 2011, pp. 27–30.

[78] A. I. Heft, T. Indinger, N. Adams, Experimental and numerical investigation of the DrivAer model, in: Proceedings of the ASME 2012 Fluids Engineering Summer Meeting, 2012, pp. FEDSM2012–72272.

[79] W.-H. Hucho, G. Sovran, Aerodynamics of road vehicles, Society of Automotive Engineers, Inc, 1998.