AIAA JOURNAL
Vol. 54, No. 3, March 2016

SU2: An Open-Source Suite for Multiphysics Simulation and Design

Thomas D. Economon*
Stanford University, Stanford, California 94305
Francisco Palacios†
The Boeing Company, Long Beach, California 90808
and
Sean R. Copeland,‡ Trent W. Lukaczyk,‡ and Juan J. Alonso§
Stanford University, Stanford, California 94305

DOI: 10.2514/1.J053813

This paper presents the main objectives and a description of the SU2 suite, including the novel software architecture and open-source software engineering strategy. SU2 is a computational analysis and design package that has been developed to solve multiphysics analysis and optimization tasks using unstructured mesh topologies. Its unique architecture is well suited for extensibility to treat partial-differential-equation-based problems not initially envisioned. The common framework adopted enables the rapid implementation of new physics packages that can be tightly coupled to form a powerful ensemble of analysis tools to address complex problems facing many engineering communities. The framework is demonstrated on a number, solving both the flow and adjoint systems of equations to provide a high-fidelity predictive capability and sensitivity information that can be used for optimal shape design using a gradient-based framework, goal-oriented adaptive mesh refinement, or uncertainty quantification.

Nomenclature

$\boldsymbol{A}^c$	=	Jacobian of the convective flux with respect to $\boldsymbol{U}$
$\boldsymbol{A}^{vk}$	=	Jacobian of the viscous fluxes with respect to $\boldsymbol{U}$
B	=	column vector or matrix B, unless capitalized symbol clearly defined otherwise
$\boldsymbol{B}$	=	(B_x, B_y) in two dimensions, or (B_x, B_y, B_z) in three dimensions
B^T	=	transpose operation on column vector or matrix B
$\boldsymbol{b}$	=	spatial vector $\boldsymbol{b} \in \mathbb{R}^n$, where n is the dimension of the physical Cartesian space (in general, two or three)
C_D	=	coefficient of drag
C_L	=	coefficient of lift
C_{M_y}	=	pitching-moment coefficient
C_p	=	coefficient of pressure
c	=	airfoil chord length
c_p	=	specific heat at constant pressure
$\bar{\bar{D}}^{vk}$	=	Jacobian of the viscous fluxes with respect to ∇U
d_s	=	nearest wall distance
$\boldsymbol{d}$	=	force projection vector
E	=	total energy per unit mass
$\tilde{F}^c_{ij}$	=	numerical convective flux between nodes i and j
$\tilde{F}^{vk}_{ij}$	=	numerical viscous fluxes between nodes i and j
$\boldsymbol{F}^c$	=	convective flux
$\boldsymbol{F}^{vk}$	=	viscous fluxes
$\boldsymbol{f}$	=	force vector on the surface
$\bar{\bar{I}}$	=	identity matrix
J	=	cost function defined as an integral over S
j	=	scalar function defined at each point on S
k	=	turbulent kinetic energy
$\mathcal{N}(i)$	=	set of all neighboring nodes of node i
$\boldsymbol{n}$	=	unit normal vector
P	=	shear-stress transport turbulent kinetic energy production term
Pr_d	=	dynamic Prandtl number
Pr_t	=	turbulent Prandtl number
p	=	static pressure
Q	=	vector of source terms
q_ρ	=	generic density source term
$q_{\rho E}$	=	generic density source term
$\boldsymbol{q}_{\rho \boldsymbol{v}}$	=	generic momentum source term
R	=	gas constant
$\mathcal{R}(U)$	=	system of governing flow equations
Re	=	Reynolds number
$\mathcal{R}_i$	=	system of governing equation residual at node i
S	=	solid wall flow domain boundary
$\hat{S}$	=	Spalart–Allmaras turbulence production term
T	=	temperature
t	=	time variable
U	=	vector of conservative variables
W	=	vector of characteristic variables
W_+	=	vector of positive characteristic variables
W_∞	=	far-field characteristic variables
Γ	=	flow domain boundary
Γ_∞	=	far-field domain boundary
γ	=	ratio of specific heats, equal to 1.4 for air
ΔS_{ij}	=	interface area between nodes i and j
$\delta(\cdot)$	=	first variation of a quantity
$\partial_n(\cdot)$	=	normal gradient operator at a surface point, $\boldsymbol{n}_S \cdot \nabla(\cdot)$
μ_{dyn}	=	laminar dynamic viscosity
μ_{tur}	=	turbulent eddy viscosity
μ^{v1}	=	total viscosity as a sum of dynamic and turbulent components, $\mu_{\text{dyn}} + \mu_{\text{tur}}$
μ^{v2}	=	effective thermal conductivity; $(\mu_{\text{dyn}}/Pr_d)+(\mu_{\text{tur}}/Pr_t)$
$\boldsymbol{v}$	=	flow velocity vector
ρ	=	fluid density
τ	=	pseudotime

Presented as Paper 2013-0287 at the 51st AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, Grapevine (Dallas/Ft. Worth Region), TX, 07–10 January 2013; received 1 September 2014; revision received 7 July 2015; accepted for publication 2 September 2015; published online 28 December 2015.

*Postdoctoral Scholar, Department of Aeronautics and Astronautics. Senior Member AIAA.
†Engineer, Advanced Concepts Group. Senior Member AIAA.
‡Ph.D. Candidate, Department of Aeronautics and Astronautics. Student Member AIAA.
§Professor, Department of Aeronautics and Astronautics. Associate Fellow AIAA.

$\bar{\bar{\tau}}$ = strain rate tensor; $[\nabla \boldsymbol{v} + \nabla \boldsymbol{v}^T - (2/3)\bar{\bar{I}}(\nabla \cdot \boldsymbol{v})]$
$\boldsymbol{\varphi}$ = adjoint velocity vector
Ψ = vector of adjoint variables
Ω = flow domain
Ω_i = control volume surrounding node i
ω = shear-stress transport specific turbulent dissipation
$\boldsymbol{\omega}$ = fluid vorticity
$\cdot$ = vector inner product
$\times$ = vector cross product
$\otimes$ = vector outer product
$\nabla(\cdot)$ = gradient operator
$\nabla \cdot (\cdot)$ = divergence operator
$\nabla_S(\cdot)$ = tangential gradient operator at a surface point; $\nabla(\cdot) - \partial_n(\cdot)\boldsymbol{n}_S$

I. Introduction

THE SU2 software suite has been recently developed for the task of solving partial differential equation (PDE) analyses and PDE-constrained optimization problems on general unstructured meshes. Although the framework is extensible to arbitrary sets of governing equations for solving multiphysics analysis and design problems, the core of the suite is a Reynolds-averaged Navier–Stokes (RANS) solver capable of simulating the compressible, turbulent flows that are representative of many problems in aerospace and mechanical engineering. But, more importantly, through the use of an adjoint method, SU2 is capable of providing gradient information that can be used for optimal shape design, uncertainty quantification, and goal-oriented adaptive mesh refinement. This gradient information enables powerful analysis and design strategies for complex, multiphysics engineering systems.

Previous work [1] has presented a detailed overview of the objectives, implementation, and capabilities of the SU2 analysis and optimization suite; and a follow-on effort [2] described a comprehensive verification and validation (V&V) process for the RANS solver using both the Spalart–Allmaras (S-A) and Menter shear-stress transport (SST) turbulence models. The test cases in the V&V set spanned a wide range of flow regimes pertinent to applications of broad interest. For these selected test cases, SU2 solutions were shown to be in excellent agreement with both the available experimental data and numerical simulation results from other well-established computational tools, which demonstrated the credibility of the solver for research and industrial applications.

Although it is possible to identify the key characteristics that computational analysis and design suites must have to provide the capabilities and efficiencies mentioned previously, one rarely has the opportunity and the resources to create such environments from the ground up. As a consequence, typical architectures for such environments lack the necessary flexibility and sophistication to overcome all of the challenges. Without a careful rethinking of the organization of an entire software suite, engineers and computational scientists are left with a collection of separate tools that they must combine for their work, resulting in a slower pace of research and innovation.

SU2, on the other hand, has been developed from scratch to overcome most of these limitations. The suite is also released under a nonviral open-source license and is freely available to the community so that users and developers around the world can continue the V&V process, contribute to the development of the source code, and further improve the accuracy and capabilities. To overcome challenges and develop a lasting infrastructure for future efforts, the basic philosophy during the development of the SU2 framework has been to ensure the following:

1) To encourage community involvement via an open-source model, the SU2 suite is a tailor-made testbed for the advancement of numerical methods and computational fluid dynamics for researchers worldwide. Additionally, we seek to provide the global community with a state-of-the-art analysis and design capability to address the challenges facing the aerospace community.

2) For reusability and encapsulation, SU2 is architected with high-level abstractions for the major code components (geometry, grid, governing equations, numerical methods, etc.). These abstractions promote code reusability and enable the rapid implementation of new capabilities by combining classes.

3) For portability and ease of use, SU2 has been developed using standard C++ as defined by the International Organization of Standardization and relies solely on widely available, well-supported, open-source software including Message Passing Interface standard (MPI) implementations, mesh partitioning packages, and popular scripting languages. As such, SU2 can be executed on any computing platform for which a C++ compiler is available.

4) For performance, the future of numerical simulation requires efficient algorithms for massively parallel architectures. Though some performance has been traded for flexibility within the class-inheritance structures native to C++, we have developed and implemented numerical algorithms and convergence acceleration techniques that result in a scalable and modular tool for large-scale simulations.

5) For many applications (optimization, response surface formulations, and uncertainty quantification, among others), it is important to obtain gradients of the outputs computed by SU2 to variations of, potentially, very large numbers of input parameters. For this reason, SU2 relies on adjoint solver implementations that can be used to compute the necessary gradients. In addition, these adjoint solutions can be used to drive functional-based mesh adaptation techniques.

Using this philosophy, it is possible to assemble tightly coupled physics packages relying on both finite volume and finite element methods to perform complex, multiphysics simulations for aeroelastic [3], aeroacoustic [4], and chemically reactive, non-equilibrium hypersonics [5] problems, among others. A library of available numerical schemes and linear solvers reduces development time for new feature additions, and common solver structure and parallelization approaches are shared by all members of the SU2 suite. It is important to highlight that the ability to easily integrate these solvers ensures that new features or updated models can be included without affecting the main infrastructure and with a reasonably low degree of difficulty.

The contributions of this paper are as follows. First, the software architecture itself is a primary contribution of this work due to its novel class design, flexibility, and ease of use. It will be described, and several specific examples will illustrate key abstractions. A number of industry-relevant problems requiring high-fidelity tools for analysis and design are used to demonstrate the tools in the Results section (Sec. V). Second, important lessons learned about the execution of an open-source package, including software engineering strategies that enable and sustain this type of open-source project, will be detailed so that others may benefit from the model employed for SU2.

The organization of this paper is as follows. Section II describes the object-oriented class structure of SU2, the flexibility of the implementation, and several components of the open-source strategy. Section III describes the set of RANS equations (including the S-A and SST turbulence models) and the corresponding adjoint RANS equations used in our work. Details of the numerical implementation are provided in Sec. IV. Section V describes several industry-relevant examples using the SU2 framework: a supersonic aircraft configuration, the DLR-F6 aircraft configuration, the National Renewable Energy Laboratory (NREL) Phase VI wind turbine geometry, and the RAM-C II hypersonic flight test vehicle. Finally, the conclusions are summarized in Sec. VI.

II. Code Framework and Design

The SU2 software suite was conceived as a common infrastructure for solving PDE-based problems on unstructured meshes. The full suite is composed of compiled C++ executables and high-level Python scripts that perform a wide range of tasks related to PDE analysis and PDE-constrained optimization. A basic description of the C++ core tools is included in the following in order to give an overall perspective of the available capabilities. Each of the modules

can be executed individually (most notably, SU2_CFD for high-fidelity PDE analysis), but the real power of the suite lies in the coupling of the modules to perform complex activities, including design optimization or adaptive grid refinement.

Most of the C++ class design in SU2 is shared by all of the core modules (in particular, the geometry, integration, and output class structures), and only specific numerical methods for the convective, viscous, and source terms are reimplemented for different physical models where necessary. There is no fundamental limitation on the number of state variables or governing equations that can be solved simultaneously in a coupled or segregated way (other than the physical memory available on a given computer architecture), and the more complicated algorithms and numerical methods (including parallelization, multigrid, and linear solvers) have been implemented in such a way that they can be applied without special consideration during the implementation of a new physical model. We will illustrate these key abstractions in SU2 by detailing a typical edge loop found within the solver.

At the end of this section, we have included a discussion of the critical components of our software engineering strategy for sustaining and growing SU2 as an open-source project. We present this information with hope that it proves useful to those currently pursuing open-source projects of their own or that it might inspire others to release their projects as open source.

A. Software Components

The core tools of the SU2 suite are the C++ executables. A key feature of these modules is that each has been designed for specific functionality while leveraging the advantages of the modern programming language, such as class inheritance and polymorphism. This level of abstraction encourages code reuse and forms a well-defined structure for quickly implementing new algorithms and numerical methods within or on top of the existing framework.

Note that all of the modules share a common C++ class structure; thus, classes and capabilities can be easily ported into other modules. For example, all of the grid deformation capabilities can be integrated directly into the computational fluid dynamics (CFD) solver (for instance, this has been done for simulating unsteady flows on dynamic meshes) or used separately as an independent code: SU2_DEF. A brief description of each of the C++ core tools is provided next. These modules are available at the time of writing, but additional modules can be added or removed from the framework with relative ease.

1) SU2_CFD solves direct, adjoint, and linearized (steady or unsteady) problems for the Euler, Navier–Stokes, and Reynolds-averaged Navier–Stokes nonequilibrium, free-surface, Poisson, heat, wave, etc., equation sets. SU2_CFD can be run serially or in parallel using a mesh partitioning approach (built around the ParMETIS [6] software) and an implementation of the message-passing interface standard. It uses either a finite volume method (FVM) or finite element method with an edge-based data structure. Explicit and implicit time integration methods are available with centered or upwind spatial integration schemes. The software also has several advanced features to improve robustness and convergence, including residual smoothing, agglomeration multigrid, or preconditioners for low-speed applications or the linear solvers. The capabilities of this tool are the subject of much of this paper.

2) SU2_DEF (mesh deformation) computes the geometrical deformation of surfaces within the computational mesh and the surrounding nodes making up the volumetric grid. A number of geometry parameterization techniques are currently available, including free-form deformation (FFD) [7] in two dimensions and and three dimensions, as well as several types of bump functions in two dimensions, such as those of Hicks and Henne [8]. After perturbing the geometry with a chosen parameterization, an approach based on the linear elasticity equations [9] is used to deform the surrounding volume mesh.

3) SU2_DOT (gradient projection) computes the partial derivative of a functional with respect to the shape design variables from a suitable surface geometry parameterization (FFD, bumps, etc.). SU2_DOT uses the surface sensitivities at each mesh node on the geometry provided by an adjoint solution from SU2_CFD and the definition of the geometrical design variables to evaluate the derivative of a particular functional (e.g., drag, lift, etc.) through a dot product operation.

4) SU2_GEO (geometry definition and constraints) evaluate (or constrain during optimization) a number of geometric quantities of interest, such as volumes, section thicknesses, etc.

5) SU2_MSH (mesh adaptation) performs grid adaptation using various techniques (including goal-oriented) based on the analysis of a converged flow, adjoint, or linearized solution in order to strategically refine the mesh about key flow features. SU2_MSH also manages the creation of ghost cells for performing simulations with periodic boundary conditions and outputs a new mesh containing the proper communication structure between periodic faces. This module must be run before SU2_CFD for any simulation that uses periodic boundary conditions.

6) SU2_SOL (solution export) generates volume and surface solution files on request in any available format. This module is automatically called by parallel_computation.py after completing a parallel calculation with SU2_CFD, but it can also be called independently at any time to create a new set of solution files if given a mesh, configuration file, and a restart file containing the solution at each node.

Apart from the core C++ tools, a Python framework has been written around SU2 to enable vertical integration with optimizers and to reduce the amount of user overhead required for setup. There are five levels of components in the optimization control architecture, and most rely on Python scripts to modify the configuration input, execute lower-level components, and automatically postprocess any resulting data. To simplify and shorten overhead time during problem setup, all levels start from a common configuration file (in the same format as that of the C++ modules), which is modified as needed when passed to lower levels. Listed in order from lowest to highest, these levels are as follows:

1) The first level is the core tools. These tools contain all of the SU2 binary executables, as described previously. As input, they take a custom format, text-based configuration file. As output, they write data such as integrated forces, moments, or other objectives to an iteration history file: field data to files for plotting, or deformed or adapted meshes in the native format, for instance.

2) The second level includes core tool pre- and postprocessing. Any necessary preprocessing activities (preparing a restart or launching of a parallel calculation, for example) and postprocess solution file output (accomplished with SU2_SOL) are managed on this level with Python for each execution of a core C++ tool. The parallel_computation.py script, which will be described in the following, serves as a clear example. However, a user can perform all functions on this level manually using only the C++ modules (i.e., without Python).

3) The third level is sensitivity analysis. This level manages the pre- and postprocessing needed for calculating performance sensitivities with respect to a user-specified surface geometry parameterization (i.e., a set of design variables). Both adjoint and finite differencing approaches have been implemented. For the adjoint approach, direct and adjoint PDE solutions are computed (requiring executions of SU2_CFD), and the resulting adjoint surface sensitivities are projected into the design space during a post-processing step (a single execution of SU2_DOT). In the case of finite differencing, multiple but independent evaluations of the direct problem for a state corresponding to a perturbation in each of the design variables is required before the performance sensitivities can be calculated (one SU2_CFD execution per design variable).

4) The fourth level is design evaluation. For easier integration with optimization packages, SU2 has a design management class that wraps a black box around the previous components and takes only a vector of design variables as input. In this view, the optimizer is allowed to drive the design process by calling the wrapped SU2 functionality. To accomplish this, the management class interprets special configuration file options for setting up the full design space. When it receives a design state vector from the optimizer, it executes

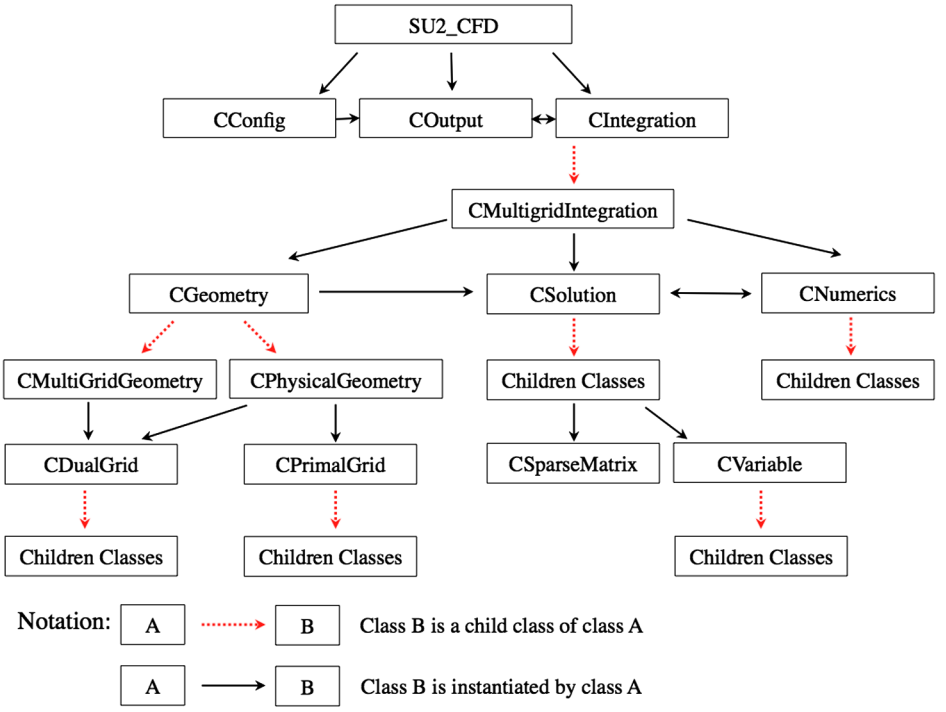


Fig. 1 Class hierarchy in SU2_CFD.

any surface geometry perturbations, mesh deformations, flow solutions, and sensitivity analyses as needed, and then it finally returns performance data (objective, any constraints, and possibly sensitivities). As it operates, it archives restart and plot data in an organized folder structure, which may be useful for secondary analyses or debugging. Evaluations of multiple design requests can be submitted in parallel if the resources are available.

5) The fifth level is design optimization. Single-objective constrained design optimization is the highest level of architecture that is currently available. The primary optimization strategy for SU2 is gradient based and takes full advantage of the built-in adjoint approach to sensitivity analysis. The default optimizer is the sequential least squares programming optimizer found in the open-source SciPy package,[¶] though additional options are available and can be easily interfaced with the design evaluation class.

Interfaces to the aforementioned levels can be scripted in Python to couple the various software modules of SU2 and perform complex analysis and design tasks. To expose the basic functionality of these interfaces that manage typical design optimization problems, several command line scripts are provided in the distribution. It is straightforward for users and developers to build new scripts that import the Python framework and complete new tasks not envisioned by the authors. Brief descriptions of the most commonly used scripts are provided here:

1) The parallel_computation.py script handles the setup and execution of parallel CFD jobs on distributed memory architectures with MPI. The script executes SU2_CFD in parallel and calls SU2_SOL to generate solution output upon completion of the calculation.

2) The continuous_adjoint.py script automatically computes the sensitivities of a functional with respect to design parameter perturbations using the continuous adjoint method. The objective function and design variables are specified in the common configuration file. The SU2_CFD and SU2_DOT modules are called to complete the task.

3) The finite_differences.py script automatically computes the sensitivities of a functional with respect to design parameter perturbations using a finite difference method. As with the continuous_adjoint.py script, design variable information is read from the configuration file, and SU2_CFD is called repeatedly to calculate the appropriate gradient elements.

4) The shape_optimization.py script orchestrates all required SU2 modules to perform shape optimization. The choices for the objective function, design variables, and additional module settings that define the optimization problem are controlled through options in the shared configuration file.

B. C++ Class Design

The object-oriented framework in SU2 makes it an ideal platform for prototyping or researching new physical models, discretization schemes, etc. Ultimately, this philosophy enables the extension of the suite to a wide variety of PDE analysis and design problems. It also allows a researcher to apply their own domain-specific knowledge to problems of interest within SU2 while many unrelated but necessary complexities are abstracted away. We will highlight several examples of this flexibility. This furthers the goal of providing an open platform to a global community in support of increased research and innovation in the computational sciences.

The objective of this section is to introduce the C++ class structure of SU2 at a high level along with specific examples that demonstrate the architecture. The class descriptions that follow focus on the structure within SU2_CFD (the main component of SU2), but many of the classes are also used in the other modules. Maximizing the flexibility of the code was a fundamental driver for the design of the class architecture, and an overview of it is shown in Fig. 1.

As a starting point, the module SU2_CFD instantiates three basic classes, namely, the following:

1) CConfig class reads and stores the problem configuration, including all options and settings from the input file (a custom-format ASCII file with extension. cfg). The option data are encapsulated within the class with "getter" and "setter" methods for access. A pointer to the CConfig object is passed to most routines in SU2 so that the options are readily available.

2) COutput class manages the merging and writing of the outputs for a simulation in a user-specified format (ParaView, Tecplot, CFD General Notation System, comma-separated values, etc.). This includes restart files in a native format, volume and surface solution files, convergence history, a breakdown of forces, etc.

3) CIntegration class integrates any system of governing equations to a solution by instantiating its child classes: CMultiGridIntegration

[¶]Data available online at http://www.scipy.org [retrieved 2015].

and CSingleGridIntegration. The integration classes contain all of the high-level loops for integration in space and time without specialization to a particular PDE. For example, CIntegration contains the spatial integration loop that defines the order of execution for computing the convective, viscous, and source terms, as well as a loop over all boundary conditions on a single grid level. As an example for time integration, CMultiGridIntegration drives the recursive multigrid algorithm and provides all of the necessary restriction and prolongation routines across grid levels without regard to the particular PDE being solved. It connects the CGeometry, CSolver, and CNumerics subclasses; the latter two classes contain the routines that define the terms within a particular set of equations and corresponding numerical methods for solving them.

The core capabilities of the computational tool are embedded within the CGeometry, CSolver, and CNumerics classes that manage the geometry/grids, the main solver functionality (definition of the various terms in a particular PDE), and the numerical methods, respectively. In the next several subsections, these three classes will be discussed.

1. *CGeometry Class*

This class reads and processes the input mesh file. Several input formats are possible, including a custom native mesh format that carries the ".su2" file extension. CGeometry has several child classes, most notably the following:

1) CPhysicalGeometry Constructs the dual-mesh structure from the original primal mesh provided to SU2. Note that the FVM formulation in SU2 is vertex based and operates on the dual mesh through an edge-based data structure. CPhysicalGeometry features the routines necessary for reading grids in various formats: partitioning grids, visualizing grids, checking cell quality, computing wall distances, etc.

2) CMultiGridGeometry If multigrid is requested, this class automatically creates consecutively coarser meshes from the original input mesh using a control volume agglomeration procedure (one instantiation per coarse grid level). These coarse grid levels contain the same type of edge-based data structure as the fine grid, which makes it possible to reuse the same edge loops that are defined only once in the CSolver classes for computing terms in the PDE.

The CPrimalGrid and CDualGrid classes (as seen in Fig. 2) are used for defining the geometrical characteristics of the primal and dual grids. These objects are instantiated by CGeometry to store the smallest geometric components from the meshes, i.e., the individual points and edges making up the dual mesh or the individual triangles and rectangles [two-dimensional (2-D)] or tetrahedra, hexahedra, prisms, and pyramids [three-dimensional (3-D)] that compose the primal mesh.

Although not shown here, additional geometric classes are available to manage other operations related to the embedded geometry and meshes. Routines for performing grid adaption, including feature- and adjoint-based methods, are contained within the CGridAdaptation class. All capabilities related to the movement of grids as part of either shape design or calculations on dynamic meshes can be found within the child classes of CGridMovement. Here, all geometry parameterizations (design variable definitions) for controlling boundary shapes can be found, along with a technique for volume mesh deformation based on the linear elasticity equations.

2. *CSolver Class*

In this class, the solution procedure is defined. Each child class of CSolver represents a solver for a particular set of governing equations. These solver classes contain subroutines with instructions for computing each spatial term of the PDE, e.g., loops over the mesh edges to compute convective and viscous fluxes, loops over the mesh nodes to compute source terms, and routines for imposing various boundary condition types for the specific PDE. The CSolver class also contains the details of how to form the particular solution update in time when an explicit or implicit method is executed for the solver by the CIntegration class.

One or more of these child classes will be instantiated, depending on the desired physics, and several examples are the CEulerSolver class for the Euler equations (compressible or incompressible), the CTurbSolver class for a turbulence model, and the CAdjEulerSolver for the Euler adjoint equations. The solver containers also lead to one of the defining features of SU2: the ability to easily construct multiphysics problems by simultaneously instantiating multiple solvers representing different physics. For example, the mean flow equations are easily coupled to the S-A turbulence model by instantiating both the CNSSolver class and the CTurbSASolver class. These two solver containers will control the solution of the different PDEs while being integrated simultaneously, and the information they contain can be freely passed back and forth. Alternatively, both the Navier–Stokes and S-A equations could be combined within a new CSolver class to form a tightly coupled approach.

Another example of this flexibility arises when solving the adjoint equations. For instance, when solving the adjoint Euler equations, both the CAdjEulerSolver and the CEulerSolver classes are instantiated, as the adjoint equations require a copy of the flow solution that will be read from a solution file and stored by the CEulerSolver class. Furthermore, one could orchestrate a one-shot approach by instantiating the CEulerSolver and CAdjEulerSolver classes and integrating the two solvers simultaneously (or create a new solver class altogether with these equation sets tightly coupled).

The solver classes instantiate a vector of CVariable objects for storing unknowns and other variables pertinent to the PDE at each mesh node. Several objects from the CNumerics class list will be created in order to specify a spatial discretization of the governing equations (to be discussed in the following). If necessary, container classes for holding the matrices and vectors needed by linear solvers will also be created. Detailed lists of all child classes found within

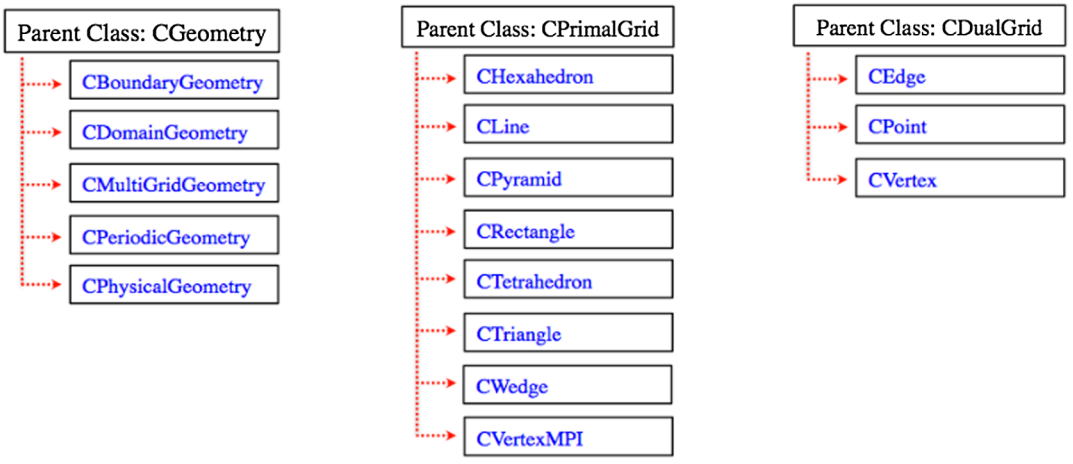


Fig. 2 Classes related to geometry processing.

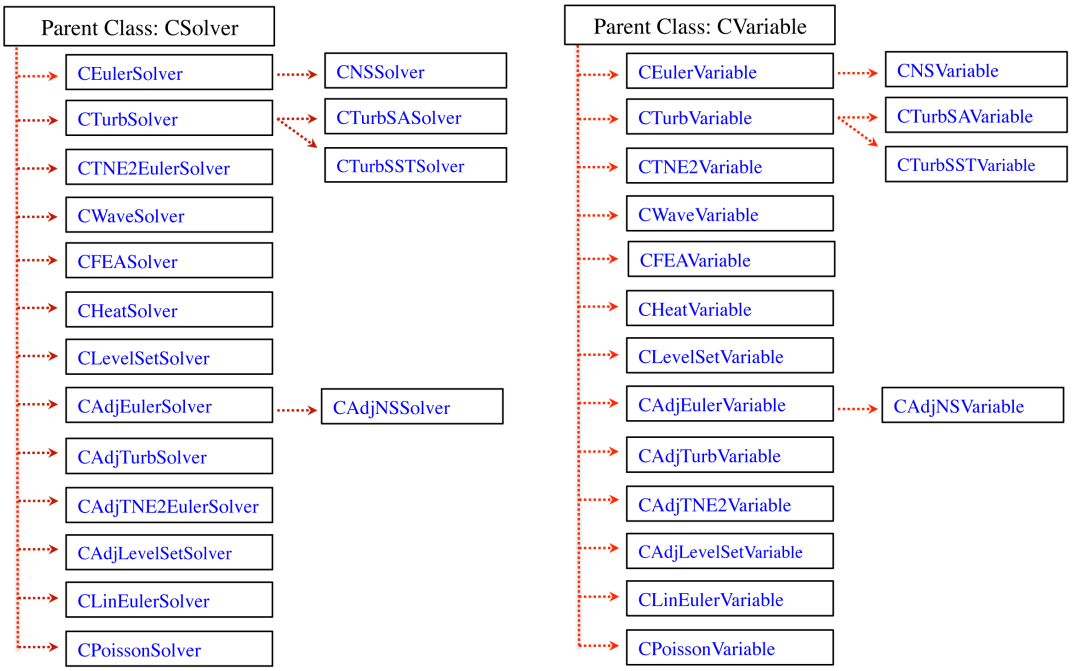


Fig. 3 List of child classes for the CSolver and the CVariable classes.

CSolver and CVariable are given in Fig. 3. These key classes can be briefly described as follows:

1) CVariable is used to store variables at every vertex in the grid, such as the conservative variables (unknowns of the PDE). Depending on the system of equations being solved, CVariable instantiates a certain child class and stores a set of variables particular to that problem at each grid node. For example, the CNSVariable child class stores the variables for the Navier–Stokes equations, which will include viscosity, whereas the CEulerVariable child class does not need to store viscosity. A list of the currently available child classes is given in Fig. 3.

2) CSysMatrix stores values for the Jacobians of fluxes and source terms in a sparse matrix structure for implicit calculations in a block compressed row format. It includes several preconditioning techniques, such as the lower–upper symmetric Gauss–Seidel, Jacobi, or line implicit preconditioning. These preconditioners can also be applied as classical iterative smoothing techniques on their own.

3) CSysVector holds and manipulates vectors needed by the linear solvers, i.e., the solution and right-hand side (residual) vectors. This class is also used to hold the residual vector for explicit calculations.

4) CSysSolve solves the linear system using the CSysMatrix (Jacobian) and CSysVector (right-hand side and solution) objects as input. This includes Krylov-based methods such as the generalized minimal residual method (GMRES) and biconjugate gradient stabilized.

3. CNumerics Class

This class contains many child classes that provide a wide range of discretization techniques for convective fluxes, viscous fluxes, and any source terms that might be present in a given PDE. For example, if one is interested in solving the Navier–Stokes equations expressed in a noninertial frame, CNumerics would call one child class corresponding to the convective scheme (centered or upwind), one corresponding to the viscous terms, and a third for the discretization of the momentum source term that arises from the transformation of the equations to a rotating reference frame.

Within the CNumerics classes, the general idea is to distill residual calculations down to the operations needed to compute a flux across a single edge between two nodes. This flux kernel will then be called repeatedly for all edges in a mesh after loading the pertinent data for a given edge and its endpoints (area normals, state variables, gradients, etc.). These methods are also responsible for computing the flux Jacobian when integrating the equations implicitly.

A defining attribute of the CNumerics class is its polymorphism: each child class contains a routine named ComputeResidual that contains its particular implementation. In this manner, many different discretization schemes can be interchanged without modifying the higher-level edge loop, which will be shown in a code example later in this paper. Moreover, this abstraction makes it very easy for a researcher to develop and prototype new discretization schemes by focusing on the operations at the level of a single edge (similar to a one-dimensional problem).

In practice, the workflow for a single iteration of an implicit calculation proceeds as follows. Methods in the CNumerics classes compute the flux contributions and Jacobians at each node using the variables stored in the CVariable class. These flux and Jacobian values are transferred back to the CSolver class, and the CSolver class executed routines within CSysSolve (operating on CSysMatrix and CSysVector) in order to solve the resulting linear system of equations for the solution update. Figure 4 shows a list of the capabilities in the CNumerics class that are currently available for the mean flow solver.

C. Demonstration of the Class Architecture

The flexible class structure is a unique and differentiating feature that makes SU2 easily extensible to treat entirely new PDE systems or coupled analyses of multiple physics. To take advantage of this, a developer can build a new CSolver class by defining a state vector (of any size) for their PDE, along with the loops that control the computation of each term in the PDE and its appropriate boundary conditions. A corresponding CVariable class that contains the state vector at each mesh node, along with auxiliary data, should also be created. CNumerics classes for the particular convective, viscous, and source schemes can be added or existing implementations can be repurposed from other solvers. Once these components are defined, the functionality of the remaining classes (CGeometry, CIntegration, COutput, CConfig, CSysSolve, CSysMatrix, CSysVector, etc.) can be leveraged with little modification to complete the solver framework.

To give a more concrete example of the class design in action, we present an example of a typical edge loop from the CEulerSolver class, as seen in Fig. 5. The loop depicts the process of computing convective fluxes using a second-order upwind approach with

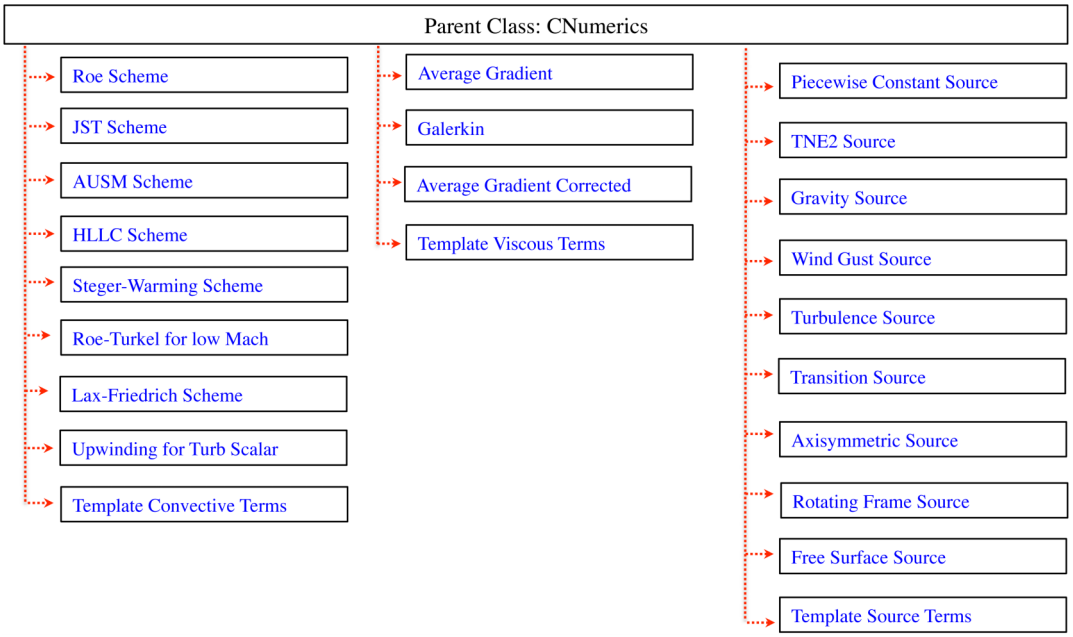


Fig. 4 List of child class capabilities found under the CNumerics parent class.

limiting (MUSCL). Several simplifications of the code have been made to give a more streamlined example.

Before entering this routine within the CEulerSolver class, objects have been instantiated for CGeometry (geometry), CVariable (node is an array of CVariables), CNumerics (numerics), CSysVector (LinSysRes), and CSysMatrix (Jacobian). It is important to note that this edge loop could be executed in either 2-D or 3-D; on any grid level during a multigrid calculation; and for any available gradient calculation method, limiter, and upwind flux scheme. These selections are made as part of a preprocessing given the options specified in the configuration file.

The loop begins by identifying the two grid points associated with the current edge from CGeometry and proceeds to access the values at those nodes from the CVariable class necessary for computing the flux and to store these data within the CNumerics class. Along the way, a higher-order reconstruction with gradient limiting is performed on the flow variables. Once all of the data are stored within the CNumerics class, the ComputeResidual method is called, which will execute the particular upwind flux [e.g., Roe, Harten–Lax–van Leer–contact (HLLC), advection upstream splitting method (AUSM), etc.]. Finally, the resulting values of the convective residual and Jacobian (for implicit calculations) will be stored within the CSysVector and CSysMatrix classes, respectively, which will be used later to complete the solution update for the current time step.

We believe that the class design results in a very approachable and easy to modify codebase. This is of critical importance in an open-source environment: new developers depend upon clear and readable code to lower the barrier to entry and to ease the implementation of additional features. As previously mentioned, the abstractions also allow researchers to focus on isolated pieces of the code that fit within their areas of expertise without needing to be familiar with the entire codebase.

However, we note that this class design, and object-oriented codes in general, can be susceptible to performance losses due to excessive levels of indirection or the encapsulation of data within multiple classes, which can lead to poor data locality (increased cache-miss rate during loop traversal). The CVariable class represents an example of this compromise between object-oriented design and performance. More specifically, the original design of the CVariable class results in an "array of structures," where the quantities needed for a single computation between two grid points may be stored far away from each other in memory within separate CVariable objects. One solution is to move to a "structure of arrays" approach where the data within CVariable are transformed into a set of arrays that contain each variable laid out in a contiguous fashion for all grid points (similar to an allocation in the C language). This reduces the memory access footprint and significantly improves the cache use efficiency. The tradeoffs between performance and usability are always being considered, and we are actively researching techniques for improving the performance and scalability of the suite [10], especially on emerging high-performance hardware architectures.

D. Open-Source Software Engineering

As an open-source package, SU2 is uniquely positioned to serve not only as an example to computational scientists around the world but also as a common baseline for future development by the community. Instead of starting from scratch, any researcher can leverage the platform as a testbed for work in their area of expertise. The current model enables the leading experts across many technical areas, anywhere in the world, to work together in creating new capabilities. A number of examples of this type of collaboration already exist in areas such as computational performance optimizations [10], nonideal compressible flow effects [11], or discrete adjoints via algorithmic differentiation [12]; and the list continues to grow. Furthermore, its open-source nature allows for rapid and effective technology transfer from these efforts back to the community at large.

Given these advantages, an open-source model can directly increase the pace of innovation in the computational sciences. However, an open model can also be susceptible to disarray or fractured code developments (i.e., "spaghetti" code) without proper coordination. In this section, we will briefly describe the salient aspects of our software engineering strategy that have enabled organized and sustainable development activities in an open-source environment. Although many of these processes are standard practices for code development, they are reported here for the benefit of others in the community that may be considering placing their tools in the open source.

Here, a number of our guiding principles for developing and maintaining the SU2 suite in accord with the aforementioned goals are offered:

1) The first principle is open access. This is perhaps the most obvious, but also most important, aspect. Free access to the source code is provided to anyone in the world, at any time, through a Web-based platform.

2) The second principle is version control and collaboration. Development of the code proceeds under a version control system (such as the Git or Subversion version control systems), so all

Listing 1 Example of a typical edge loop in SU2 for computing an upwind flux.

```
for (iEdge = 0; iEdge < geometry->GetnEdge(); iEdge++) {

  /*--- Get points and normal associated with this edge. ---*/
  iPoint = geometry->edge[iEdge]->GetNode(0);
  jPoint = geometry->edge[iEdge]->GetNode(1);
  numerics->SetNormal(geometry->edge[iEdge]->GetNormal());

  /*--- Get primitive variables at points i & j. ---*/
  V_i = node[iPoint]->GetPrimitive();
  V_j = node[jPoint]->GetPrimitive();

  /*--- Compute position vectors from points to mid-point of edge. ---*/
  for (iDim = 0; iDim < nDim; iDim++) {
    Vector_i[iDim] = 0.5*(geometry->node[jPoint]->GetCoord(iDim) - geometry->node[iPoint]->GetCoord(iDim));
    Vector_j[iDim] = 0.5*(geometry->node[iPoint]->GetCoord(iDim) - geometry->node[jPoint]->GetCoord(iDim));
  }

  /*--- Get pre-computed gradients at points i & j. ---*/
  Gradient_i = node[iPoint]->GetGradient_Primitive();
  Gradient_j = node[jPoint]->GetGradient_Primitive();

  /*--- Get pre-computed limiter values at points i & j. ---*/
  Limiter_i = node[iPoint]->GetLimiter_Primitive();
  Limiter_j = node[jPoint]->GetLimiter_Primitive();

  for (iVar = 0; iVar < nPrimVarGrad; iVar++) {
    /*--- Project gradient along the edge direction. ---*/
    Project_Grad_i = 0.0; Project_Grad_j = 0.0;
    for (iDim = 0; iDim < nDim; iDim++) {
      Project_Grad_i += Vector_i[iDim]*Gradient_i[iVar][iDim];
      Project_Grad_j += Vector_j[iDim]*Gradient_j[iVar][iDim];
    }
    /*--- High-order reconstruction using a MUSCL strategy. ---*/
    Primitive_i[iVar] = V_i[iVar] + Limiter_i[iVar]*Project_Grad_i;
    Primitive_j[iVar] = V_j[iVar] + Limiter_j[iVar]*Project_Grad_j;
  }

  /*--- Store the reconstructed primitive variables. ---*/
  numerics->SetPrimitive(Primitive_i, Primitive_j);

  /*--- Compute the residual with chosen convective scheme. ---*/
  numerics->ComputeResidual(Res_Conv, Jacobian_i, Jacobian_j, config);

  /*--- Update residual value. ---*/
  LinSysRes.AddBlock(iPoint, Res_Conv);
  LinSysRes.SubtractBlock(jPoint, Res_Conv);

  /*--- Store Jacobians at i & j, if implicit. ---*/
  if (implicit) {
    Jacobian.AddBlock(iPoint, iPoint, Jacobian_i);
    Jacobian.AddBlock(iPoint, jPoint, Jacobian_j);
    Jacobian.SubtractBlock(jPoint, iPoint, Jacobian_i);
    Jacobian.SubtractBlock(jPoint, jPoint, Jacobian_j);
  }
}
```

Fig. 5 Example of a typical edge loop in SU2 for computing an upwind flux.

changes to the codebase and their author are tracked. The branching system provided by git enables parallel development lines so that conflicts are avoided and new features can be easily folded into the main repository (under supervision) when complete. Hosting the code on an open Web-based platform allows anyone in the world to submit code that can be considered for inclusion in the main repository. A publicly visible list of issues, bugs, and feature requests is maintained alongside the repository.

3) The third principle is continuous integration. Open-source tools often carry a stigma of unreliability due to their open and quickly evolving nature. This sentiment can be overcome by exercising control over the process of incorporating new code changes and by continuously subjecting the code to a rigorous series of regression tests whenever changes in the repository are detected. These activities are closely related to formal verification and validation activities, which are critical in establishing the accuracy of the code and

building confidence in the code output. The regression test system is publicly visible, and the test cases (configuration files and meshes) are provided openly so that results can be reproduced at any time. Many of the provided tests are industry-standard V&V cases.

4) The fourth principle is portability. A difficult build process can quickly discourage adoption of the code by new users and developers. We highly value portability and easy installation on many platforms. By relying on standard C++ alone, we ensure that a basic version of the code can always be compiled with only a C++ compiler and widely available build tools (autoconf/automake). Philosophically, we favor in-house solutions over third-party libraries or additional dependencies when possible. However, we recognize that some users or developers need specialized capabilities provided by external libraries, and we can optionally include these packages in the build process.

5) The fifth principle is documentation. Full documentation, including a comprehensive set of tutorials, is provided on the Web. The tutorials gradually build upon each other to expose the full set of options and features in the code to the practitioner. The documentation itself is also open (in a public wiki format) so that developers can add new information or tutorials as the code evolves.

6) The sixth principle is community involvement. Interaction with the community is absolutely essential for the growth and sustainability of the project. The open-source community provides valuable feedback on the code that directly results in improvements (e.g., the build process, performance considerations, bugs, etc.) and helps identify future development directions. We employ an email list for announcements to users, an email list for discussion among developers, and a dedicated online forum where users and developers can post and answer questions on the code.

At the time of writing, many of the aforementioned considerations are managed through the GitHub ecosystem where the code is currently hosted.** However, these ideas are independent of a particular software solution: many version control, code hosting, or regression test systems are available for accomplishing these tasks.

III. Physical Modeling

The flexible class structure of SU2 has been designed to solve PDEs that are defined on a domain $\Omega \subset \mathbb{R}^3$. The PDE system resulting from physical modeling of a particular problem can be cast in the following structure:

$$\frac{\partial \boldsymbol{U}}{\partial t} + \nabla \cdot \boldsymbol{F}^c - \nabla \cdot (\mu^{vk} \boldsymbol{F}^{vk}) = \boldsymbol{Q} \qquad \text{in}\,\Omega, \qquad t > 0 \tag{1}$$

with appropriate boundary and temporal conditions that will be problem dependent. In this general framework, $\boldsymbol{U}$ represents the vector of state variables, $\boldsymbol{F}^c(U)$ are the convective fluxes, $\boldsymbol{F}^{vk}(U)$ are the viscous fluxes, and $\boldsymbol{Q}(U)$ is a generic source term.

In this section, we will focus on the RANS and adjoint RANS equations as implemented in SU2, and they will be described using Eq. (1) as a baseline PDE. However, it is important to note that other PDEs can be readily solved in the current version of SU2, including the heat and wave equations; Poisson equation, the equations of linear elasticity, and a two-temperature model for high-speed non-equilibrium flows, to name a few. More detail on other physical models available in SU2 can be found in previous work by Palacios et al. [1].

A. Reynolds-Averaged Navier–Stokes Equations

We are concerned with the general scenario of time-accurate viscous flow around aerodynamic bodies that is governed by the compressible, unsteady Navier–Stokes equations. Consider the equations in a domain $\Omega \subset \mathbb{R}^3$ with a disconnected boundary that is divided into a far-field component Γ_∞ and an adiabatic wall boundary S, as seen in Fig. 6. For instance, the surface S could represent the outer mold line of an aerodynamic body. These conservation

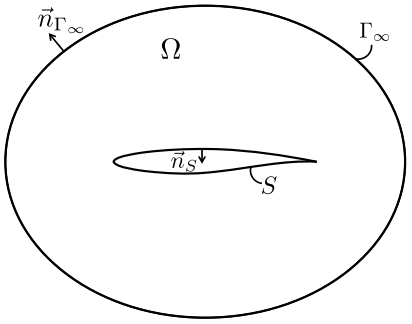


Fig. 6 Notional schematic of the flow domain Ω and the boundaries Γ_∞ and S, as well as the definition of the surface normals.

equations along with a generic source term $\boldsymbol{Q}$ can be expressed in differential form as

$$\begin{cases} \mathcal{R}(U) = \frac{\partial U}{\partial t} + \nabla \cdot \boldsymbol{F}^c - \nabla \cdot (\mu^{vk} \boldsymbol{F}^{vk}) - \boldsymbol{Q} = 0 & \text{in}\,\Omega, \qquad t > 0 \\ \boldsymbol{v} = \boldsymbol{0} & \text{on}\,S, \\ \partial_n T = 0 & \text{on}\,S, \\ (\boldsymbol{W})_+ = W_\infty & \text{on}\,\Gamma_\infty, \end{cases} \tag{2}$$

where the conservative variables are given by $\boldsymbol{U} = \{\rho, \rho\boldsymbol{v}, \rho E\}^T$; and the convective fluxes, viscous fluxes, and source term are

$$\boldsymbol{F}^c = \begin{Bmatrix} \rho\boldsymbol{v} \\ \rho\boldsymbol{v} \otimes \boldsymbol{v} + \bar{\bar{I}}p \\ \rho E\boldsymbol{v} + p\boldsymbol{v} \end{Bmatrix}, \qquad \boldsymbol{F}^{v1} = \begin{Bmatrix} \cdot \\ \bar{\bar{\tau}} \\ \bar{\bar{\tau}} \cdot \boldsymbol{v} \end{Bmatrix},$$

$$\boldsymbol{F}^{v2} = \begin{Bmatrix} \cdot \\ \cdot \\ c_p \nabla T \end{Bmatrix}, \qquad \boldsymbol{Q} = \begin{Bmatrix} q_\rho \\ \boldsymbol{q}_{\rho v} \\ q_{\rho E} \end{Bmatrix} \tag{3}$$

where ρ is the fluid density, $\boldsymbol{v} = \{v_1, v_2, v_3\}^T \in \mathbb{R}^3$ is the flow speed in a Cartesian system of reference, E is the total energy per unit mass, p is the static pressure, c_p is the specific heat at constant pressure, T is the temperature, and the viscous stress tensor can be written in vector notation as

$$\bar{\bar{\tau}} = \nabla\boldsymbol{v} + \nabla\boldsymbol{v}^T - \frac{2}{3}\bar{\bar{I}}(\nabla \cdot \boldsymbol{v}) \tag{4}$$

The second line of Eq. (2) represents the no-slip condition at a solid wall, the third line represents an adiabatic condition at the wall, and the final line represents a characteristic-based boundary condition at the far field [13], with $\boldsymbol{W}$ being the characteristic variables.

Including the boundary conditions given in Eq. (2), the compressible RANS solver in SU2 currently supports the following boundary condition types: Euler (flow tangency) and symmetry wall, no-slip wall (adiabatic and isothermal), far-field and near-field boundaries, characteristic-based inlet boundaries (stagnation, mass flow, or supersonic conditions prescribed), characteristic-based outlet boundaries (back pressure prescribed), periodic boundaries, nacelle inflow boundaries (fan face Mach number prescribed), and nacelle exhaust boundaries (total nozzle temp and total nozzle pressure prescribed).

The boundary conditions listed make SU2 suitable for computing both external and internal flows. SU2 is also capable of solving unsteady flows on both rigidly transforming and dynamically deforming meshes. For unsteady problems, the temporal conditions will be problem dependent. For steady problems, we will use the freestream fluid state as the initial condition for the mean flow, and this is a typical practice in external aerodynamics.

Assuming a perfect gas with a ratio of specific heats γ and gas constant R, the pressure is determined from $p = (\gamma - 1)\rho[E - 0.5(\boldsymbol{v} \cdot \boldsymbol{v})]$, the temperature is given by $T = p/(\rho R)$, and

**Data available online at https://github.com/su2code/SU2 [retrieved 2015].

$c_p = \gamma R/(\gamma - 1)$. In accord with the standard approach to turbulence modeling based upon the Boussinesq hypothesis [14], which states that the effect of turbulence can be represented as an increased viscosity, the total viscosity is divided into a laminar μ_{dyn} and a turbulent μ_{tur} component. To close the system of equations, the dynamic viscosity μ_{dyn} is assumed to satisfy Sutherland's law [15] (a function of temperature alone), the turbulent viscosity μ_{tur} is computed via a turbulence model, and

$$\mu^{v1} = \mu_{\text{dyn}} + \mu_{\text{tur}}, \qquad \mu^{v2} = \frac{\mu_{\text{dyn}}}{Pr_d} + \frac{\mu_{\text{tur}}}{Pr_t} \tag{5}$$

where Pr_d and Pr_t are the dynamic and turbulent Prandtl numbers, respectively.

The turbulent viscosity is obtained from a suitable turbulence model involving the flow state and a set of new variables. The Menter shear-stress transport model and the Spalart–Allmaras model are two of the most common and widely used turbulence models for the analysis and design of engineering applications in turbulent flows. Brief descriptions of the two models are given in the following.

1. Spalart–Allmaras Model

In the case of the one-equation Spalart–Allmaras [16] turbulence model, the turbulent viscosity is computed as

$$\mu_{\text{tur}} = \rho\hat{\nu} f_{v1}, \qquad f_{v1} = \frac{\chi^3}{\chi^3 + c_{v1}^3}, \qquad \chi = \frac{\hat{\nu}}{\nu}, \qquad \nu = \frac{\mu_{\text{dyn}}}{\rho} \tag{6}$$

The new variable $\hat{\nu}$ is obtained by solving a transport equation that includes the following convective, viscous, and source terms:

$$\begin{aligned} &\boldsymbol{F}^c = \boldsymbol{v}\hat{\nu}, \qquad \boldsymbol{F}^v = -\frac{\nu + \hat{\nu}}{\sigma}\nabla\hat{\nu}, \\ &\boldsymbol{Q} = c_{b1}\hat{S}\hat{\nu} - c_{w1} f_w \left(\frac{\hat{\nu}}{d_S}\right)^2 + \frac{c_{b2}}{\sigma}|\nabla\hat{\nu}|^2 \end{aligned} \tag{7}$$

where the production term $\hat{S}$ is defined as

$$\hat{S} = |\boldsymbol{\omega}| + \frac{\hat{\nu}}{\kappa^2 d_S^2} f_{v2}$$

$\boldsymbol{\omega} = \nabla \times \boldsymbol{v}$ is the fluid vorticity, d_S is the distance to the nearest wall, and

$$f_{v2} = 1 - \frac{\chi}{1 + \chi f_{v1}}$$

The function f_w is computed as

$$f_w = g\left[\frac{1 + c_{w3}^6}{g^6 + c_{w3}^6}\right]^{1/6}$$

where $g = r + c_{w2}(r^6 - r)$ and

$$r = \frac{\hat{\nu}}{\hat{S}\kappa^2 d_S^2}$$

Finally, the set of closure constants for the model is given by

$$\begin{aligned} &\sigma = 2/3, \qquad c_{b1} = 0.1355, \qquad c_{b2} = 0.622, \qquad \kappa = 0.41, \\ &c_{w1} = \frac{c_{b1}}{\kappa^2} + \frac{1 + c_{b2}}{\sigma}, \qquad c_{w2} = 0.3, \qquad c_{w3} = 2, \qquad c_{v1} = 7.1 \end{aligned} \tag{8}$$

The physical meaning of the far-field boundary condition for the turbulent viscosity is the imposition of some fraction of the laminar viscosity at the far field. On viscous walls, $\hat{\nu}$ is set to zero, corresponding to the absence of turbulent eddies very near to the wall. The formulation presented here is the original S-A model without corrections, but the S-A negative variant of the model [17] has been recently made available in the code.

2. Menter Shear-Stress Transport Model

The Menter SST turbulence model [18] is a two-equation model for the turbulent kinetic energy k and specific dissipation ω that consists of the blending of the traditional k-ω and k-ϵ models. The definition of the eddy viscosity, which includes the shear-stress limiter, can be expressed as

$$\mu_{\text{tur}} = \frac{\rho a_1 k}{\max(a_1\omega, SF_2)} \tag{9}$$

where $S = \sqrt{2S_{ij}S_{ij}}$, and F_2 is the second blending function. The convective, viscous, and source terms for the turbulent kinetic energy are

$$\boldsymbol{F}^c = \rho k\boldsymbol{v}, \qquad \boldsymbol{F}^v = -(\mu_{\text{dyn}} + \sigma_k\mu_{\text{tur}})\nabla k, \qquad \boldsymbol{Q} = P - \beta^*\rho\omega k \tag{10}$$

where P is the production of turbulent kinetic energy. The convective, viscous, and source terms for the specific dissipation are given by

$$\begin{aligned} &\boldsymbol{F}^c = \rho\omega\boldsymbol{v}, \qquad \boldsymbol{F}^v = -(\mu_{\text{dyn}} + \sigma_\omega\mu_{\text{tur}})\nabla\omega, \\ &\boldsymbol{Q} = \frac{\gamma}{\nu_t}P - \beta^*\rho\omega^2 + 2 * (1 - F_1)\frac{\rho\sigma_{\omega 2}}{\omega}\nabla k\nabla\omega \end{aligned} \tag{11}$$

where F_1 is the first blending function. The values for the constants and the forms for the blending functions and auxiliary relations are detailed in the paper by Menter [18].

B. Continuous Adjoint Navier–Stokes Equations

A typical aerodynamic shape optimization problem seeks the minimization of a cost function $J(S)$ (lift, drag, moment, etc.), as chosen by the designer, with respect to changes in the shape of the boundary S. For the present description, we will focus on integrated forces and moments on the solid surface that depend on a scalar j evaluated at each point on S. Other objectives are possible, such as functions based on surface temperature or surface heat flux, for instance.

We note that any changes to the shape of S will result in perturbations in the fluid state $\boldsymbol{U}$ in the domain and that these variations in the state are constrained to satisfy the RANS equations, i.e., $\mathcal{R}(U) = 0$ must be satisfied for any candidate shape of S. Therefore, the optimal shape design problem can be formulated as a PDE-constrained optimization problem:

$$\min_S J(S) = \int_S j(\boldsymbol{f}, \boldsymbol{n})\,\mathrm{d}s$$

subject to

$$\mathcal{R}(U) = 0 \tag{12}$$

where $\boldsymbol{f} = (f_1, f_2, f_3)$ is the force on the surface (from fluid pressure and viscous stresses), and $\boldsymbol{n}$ is the outward-pointing unit vector normal to the surface S. We assume the surface is continuously differentiable (C^1) and will parameterize the shape by an infinitesimal deformation of size δS along the normal direction $\boldsymbol{n}$ to the surface S. The new surface obtained after the deformation is then given by $S' = \{\boldsymbol{x} + \delta S\boldsymbol{n}, \boldsymbol{x} \in S\}$.

Using the continuous adjoint approach, the computation of the objective function gradient with respect to perturbations of the geometry will require the solution of the adjoint RANS equations given by

$$\begin{cases} -\frac{\partial \boldsymbol{\Psi}^T}{\partial t} - \nabla \boldsymbol{\Psi}^T \cdot (\boldsymbol{A}^c - \mu^{vk} \boldsymbol{A}^{vk}) - \nabla \cdot (\nabla \boldsymbol{\Psi}^T \cdot \mu^{vk} \bar{\bar{D}}^{vk}) - \boldsymbol{\Psi}^T \frac{\partial Q}{\partial U} = 0 & \text{in } \Omega, \quad t > 0 \\ \boldsymbol{\varphi} = \boldsymbol{d} & \text{on } S, \\ \partial_n(\psi_{\rho E}) = 0 & \text{on } S, \end{cases} \tag{13}$$

where $\boldsymbol{\Psi}$ are the adjoint variables, and we have introduced the following Jacobian matrices from the linearization of the governing equations (full form found within previous work by Bueno-Orovio et al. [19]):

$$\left.\begin{aligned} &\boldsymbol{A}^c = (A_x^c, A_y^c, A_z^c), && A_i^c = \left.\frac{\partial \boldsymbol{F}_i^c}{\partial \boldsymbol{U}}\right|_{U(x,y,z)} \\ &\boldsymbol{A}^{vk} = (A_x^{vk}, A_y^{vk}, A_z^{vk}), && A_i^{vk} = \left.\frac{\partial \boldsymbol{F}_i^{vk}}{\partial \boldsymbol{U}}\right|_{U(x,y,z)} \\ &\bar{\bar{D}}^{vk} = \begin{pmatrix} D_{xx}^{vk} & D_{xy}^{vk} & D_{xz}^{vk} \\ D_{yx}^{vk} & D_{yy}^{vk} & D_{yz}^{vk} \\ D_{zx}^{vk} & D_{zy}^{vk} & D_{zz}^{vk} \end{pmatrix}, && D_{ij}^{vk} = \left.\frac{\partial \boldsymbol{F}_i^{vk}}{\partial (\partial_j \boldsymbol{U})}\right|_{U(x,y,z)} \end{aligned}\right\}$$

$$i, j = 1 \ldots 3, \qquad k = 1, 2 \tag{14}$$

After satisfying the adjoint system, the final expression for the functional variation will become a surface integral that contains terms involving only the flow and adjoint variables multiplied by δS:

$$\delta J(S) = \int_S (\boldsymbol{n} \cdot \bar{\bar{\Sigma}}^{\varphi} \cdot \partial_n \boldsymbol{v} - \mu^{v2} c_p \nabla_S \psi_5 \cdot \nabla_S T) \delta S \, \mathrm{d}s \tag{15}$$

where ∇_S represents the tangential gradient operator on S, and

$$\bar{\bar{\Sigma}}^{\varphi} = \mu^{v1} \left(\nabla \boldsymbol{\varphi} + \nabla \boldsymbol{\varphi}^T - \frac{2}{3} \bar{\bar{I}} \nabla \cdot \boldsymbol{\varphi} \right)$$

which depends on the gradient of the adjoint variables. This computable formula is what we call the surface sensitivity, and it is the key result of the continuous adjoint derivation. The surface sensitivity provides a measure of the variation of the objective function with respect to infinitesimal variations of the surface shape in the direction of the local surface normal. This value is computed at every surface node of the numerical grid with negligible computational cost. In this manner, the functional variation for an arbitrary number of shape perturbations will be computable at the fixed cost of solving the flow and adjoint PDE systems.

The ability to recover an analytic expression as a surface integral for the variation of the functional is commonly referred to as a surface formulation for computing gradients (with no dependence on volume mesh sensitivities). After early work in the area of continuous adjoints on unstructured meshes [20,21], this type of surface formulation based on shape calculus was first demonstrated by Castro et al. [22] for inviscid and laminar flows, and it was later extended to turbulent flows using the S-A turbulence model [19].

Extensions and advances of this formulation form much of the recent research activity within the SU2 suite. In particular, the formulation has been extended to sonic boom minimization for supersonic aircraft [23], aerodynamic design for unsteady problems on dynamic meshes [24–26], mesh adaptation and design in nonequilibrium hypersonic flows [5], and design for free-surface flows [27,28]. Since debuting in the initial public release of SU2, the continuous adjoint solver has been extensively used and rigorously verified [29,30] for both inviscid and viscous problems across many flow regimes. Finally, we note that, although the continuous formulation has been the primary focus of adjoint research up to this point, a discrete formulation via algorithmic differentiation has been implemented, and its incorporation into SU2 is ongoing [12].

IV. Numerical Implementation in the SU2 Suite

The following sections contain an overview of the numerical implementation strategies for solving PDEs in SU2. Both the flow and adjoint problems are solved numerically on unstructured meshes with an edge-based data structure. Following the method of lines, the governing equations are discretized in space and time separately. This decoupling of space and time allows for the selection of different types of schemes for the spatial and temporal integration. In general, spatial integration is performed using the finite volume method, whereas integration in time is achieved through several available explicit and implicit methods. For time-accurate calculations, a dual-time-stepping approach is used.

A. Spatial Integration via the Finite Volume Method

PDEs in SU2 are discretized using a finite volume method [13,31–38] with a standard edge-based structure on a dual grid with control volumes constructed using a median-dual vertex-based scheme. Median-dual control volumes are formed by connecting the centroids, face, and edge midpoints of all cells sharing the particular node. After integrating the governing equations over a control volume and applying the divergence theorem, the semidiscretized, integral form of a typical PDE (such as the RANS equations given previously) is given by

$$\int_{\Omega_i} \frac{\partial \boldsymbol{U}}{\partial t} \, \mathrm{d}\Omega + \sum_{j \in \mathcal{N}(i)} (\tilde{F}_{ij}^c + \tilde{F}_{ij}^{vk}) \Delta S_{ij} - \boldsymbol{Q} |\Omega_i| = \int_{\Omega_i} \frac{\partial U}{\partial t} \, \mathrm{d}\Omega + R_i(U) = 0 \tag{16}$$

where $\boldsymbol{U}$ is the vector of state variables, and $R_i(U)$ is the numerical residual representing the integration of all spatial terms at node i. $\tilde{F}_{ij}^c$ and $\tilde{F}_{ij}^{vk}$ are the numerical approximations of the convective and viscous fluxes projected into the local normal direction, respectively; and Q is a source term. ΔS_{ij} is the area of the face associated with the edge ij, $|\Omega_i|$ is the volume of the dual control volume, and $\mathcal{N}(i)$ is the set of neighboring nodes to node i.

The convective and viscous fluxes are evaluated at the midpoint of an edge. The numerical solver loops through all of the edges in the primal mesh in order to calculate these fluxes and then integrates them to evaluate the residual $R_i(U)$ at every node in the numerical grid. The convective fluxes can be discretized using centered or upwind schemes in SU2. A number of numerical schemes have been implemented [Jameson–Schmidt–Turkel (JST) [39], Roe [40], AUSM [41], HLLC [38], and Roe–Turkel [42], to name a few), and the code architecture allows for the rapid implementation of new schemes. Limiters are available for use with higher-order reconstructions for the upwind convective schemes. To evaluate the viscous fluxes using a finite volume method, flow quantities and their first derivatives are required at the faces of the control volumes. The gradients of the flow variables are calculated using either a Green–Gauss or weighted least-squares method at all grid nodes, and then they are averaged to obtain the flow variable gradients at the cell faces. Source terms are approximated at each node using piecewise-constant reconstruction within each of the dual control volumes.

B. Time Integration

Equation (16) must be valid over the entire time interval, so one can choose to evaluate $R_i(U)$ either at time t^n (explicit methods) or t^{n+1} (implicit methods). Focusing on implicit integration (SU2 also has an Euler explicit and a Runge–Kutta explicit method), we find that the

following linear system should be solved in order to find the solution update (ΔU_i^n) after linearizing the equations about the current state:

$$\left(\frac{|\Omega_i|}{\Delta t_i^n}\delta_{ij} + \frac{\partial R_i(U^n)}{\partial U_j}\right) \cdot \Delta U_j^n = -R_i(U^n) \tag{17}$$

where $\Delta U_i^n = U_i^{n+1} - U_i^n$ and, if a flux $\tilde{F}_{ij}$ has a stencil of points $\{i, j\}$, then contributions are made to the Jacobian at four points:

$$\frac{\partial R}{\partial U} := \frac{\partial R}{\partial U} + \begin{bmatrix} \ddots & & & \\ & \frac{\partial \tilde{F}_{ij}}{\partial U_i} & \cdots & \frac{\partial \tilde{F}_{ij}}{\partial U_j} \\ & \vdots & \ddots & \vdots \\ & -\frac{\partial \tilde{F}_{ij}}{\partial U_i} & \cdots & -\frac{\partial \tilde{F}_{ij}}{\partial U_j} \\ & & & \ddots \end{bmatrix} \tag{18}$$

The SU2 framework includes the implementation of several linear solvers for solving Eq. (17). Currently, the following two Krylov subspace methods are available: the generalized minimal residual method [43] and the biconjugate gradient stabilized method [44].

For unsteady flows, a dual-time-stepping strategy [45,46] has been implemented to achieve high-order accuracy in time. In this method, the unsteady problem is transformed into a series of steady problems at each physical time step that can each be solved consecutively by using all of the well-known convergence acceleration techniques for steady problems. The current implementation of the dual-time-stepping approach solves the following problem:

$$\frac{\partial \boldsymbol{U}}{\partial \tau} + R_i^*(U) = 0 \tag{19}$$

with

$$R_i^*(U) = \frac{3}{2\Delta t}U_i + \frac{1}{|\Omega_i|^{n+1}}\left(R_i(U) - \frac{2}{\Delta t}|\Omega_i|^n U_i^n + \frac{1}{2\Delta t}|\Omega_i|^{n-1}U_i^{n-1}\right) \tag{20}$$

for second-order accuracy in time (backward difference formula), where Δt is the physical time step, τ is a fictitious time used to converge the steady-state problem, $R_i(U)$ denotes the residual of the governing equations, and $U = U^{n+1}$ once the steady problem is satisfied. A first-order backward difference in time is also available.

C. Convergence Acceleration

Due to the nature of most iterative relaxation schemes, high-frequency errors are usually well damped (local errors), but low-frequency errors (global error spanning the larger solution domain) are less damped by the action of iterative methods that have a stencil with a local area of influence. To combat this, SU2 contains an agglomeration multigrid implementation that generates effective convergence at all length scales of a problem by employing a sequence of grids of varying resolution (SU2 will automatically generate the coarse grids from the provided fine grid at runtime). In short, the goal is to accelerate the convergence of the numerical solution of a set of equations by computing corrections to the fine-grid solutions on coarser grids and applying this idea recursively [47–50].

Preconditioning is the application of a transformation to the original system that makes it more suitable for numerical solution [51]. In particular, Jacobi, lower–upper symmetric Gauss–Seidel, and line implicit preconditioners have been implemented to improve the convergence rate of the available linear solvers [48,52]. A Roe–Turkel [42] preconditioner for low-Mach-number flows is also available.

V. Results

In this section, we compute both the direct and adjoint solutions for two separate full-aircraft configuration cases and a wind turbine geometry. These simulations demonstrate the capability of SU2 to solve industry-sized problems, using the container code structures to store the flow, turbulence, and adjoint solutions simultaneously, enabling high-fidelity analysis and design using a common code infrastructure. We also show results for the RAM-C II hypersonic flight-test vehicle to illustrate the flexibility of SU2 for rapidly implementing additional physical models.

A. DLR-F6 Transonic Aircraft

Transonic flow over the DLR-F6 aircraft (wing/body configuration) is computed with SU2. For the baseline geometry and case definition, we have chosen the DLR-F6 configuration without a fairing that was used in the 3rd CFD Drag Prediction Workshop (Mach number 0.75, and Reynolds number 5E6). To match the experimental lift coefficient of 0.498, a zero angle of attack (AOA) was required in the numerical settings. It is important to remark that the wind-tunnel experiments were performed at a Reynolds number of 3E6 and an angle of attack of 0.49 deg.

A detailed description of the geometry and experimental results can be found in the documentation produced by the 3rd CFD Drag Prediction Workshop.[††] The original reference for the baseline DLR-F6 geometry was by Brodersen and Stürmer [53].

The mesh used in this study is a mixed-element grid composed of 8,773,810 total elements and 3,059,189 nodes (generated with the ANSYS ICEM CFD Mesh Generation Software). The mesh is composed of tetrahedra, prisms, and pyramids around a surface that has been discretized using triangles (see Fig. 7 for a view of the geometry and surface mesh). The far-field boundary is located approximately 20 body lengths away from the aircraft, with a suitable spacing in the boundary layer to allow for $y^+ \approx 1$.

A JST centered spatial discretization is used to calculate convective fluxes. Turbulent variables for the S-A and SST models are convected using a first-order scalar upwind method, and the viscous fluxes are calculated using the corrected average-gradient method. Implicit, local time stepping is used to converge the problem to the steady-state solution, and the linear system is solved using the iterative GMRES method with a maximum error tolerance of $\mathcal{O}(10^{-6})$.

In this particular study, four representative sections of the wing ($y/b = 0.150, 0.150, 0.331, 0.409, 0.844$) are presented in Fig. 8, including experimental data from the S2MA wind tunnel at ONERA. To obtain these results, two different sets of conditions have been used:

1) The wind-tunnel lift coefficient $C_L = 0.5$ is matched at Reynolds number 5E6. In this case, the results are compared with those obtained by the code Tau (DLR, German Aerospace Center) with very good agreement. It is also important to highlight the small differences introduced by the turbulence models (which are more important in the inboard section close to a well-known recirculation region in the wing/fuselage intersection).

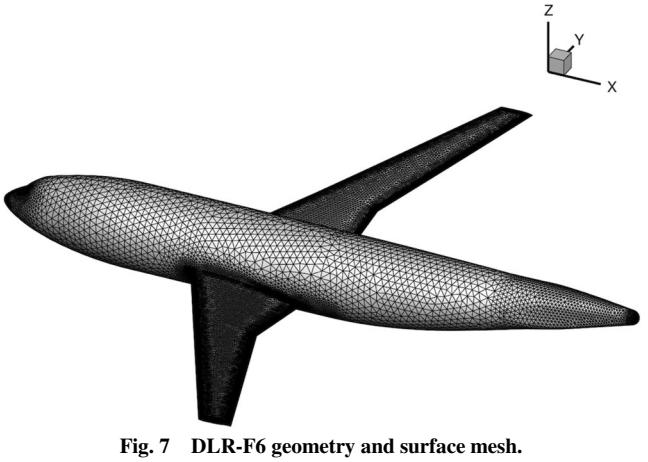
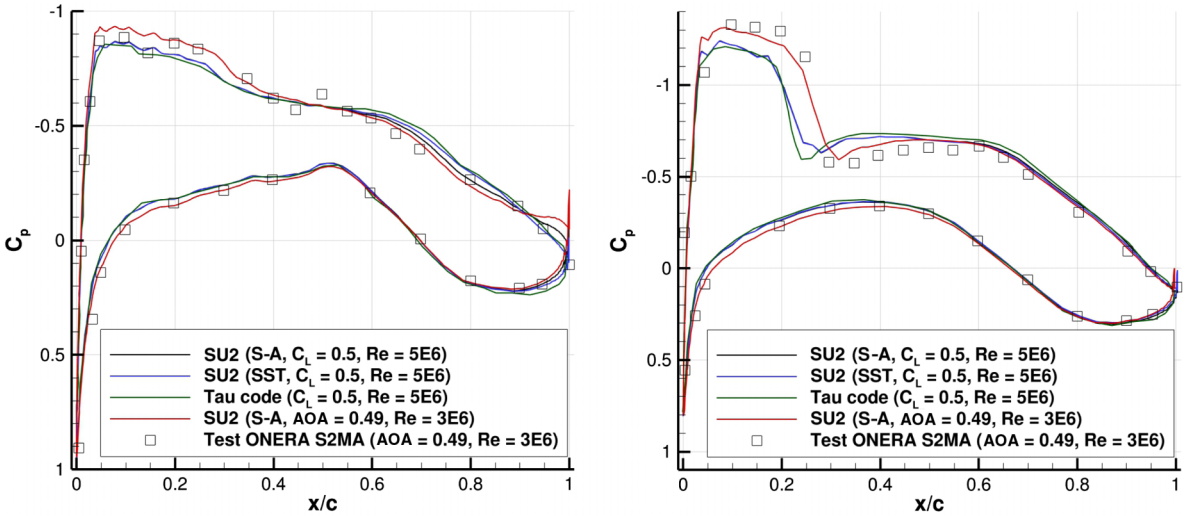
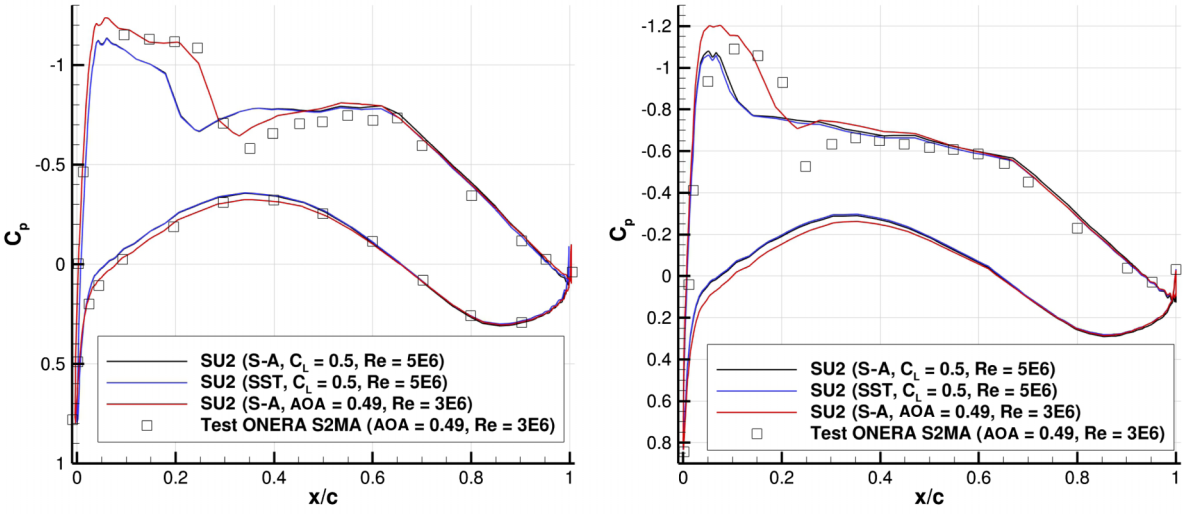


Fig. 7 DLR-F6 geometry and surface mesh.



a) C_p distribution at wing section y/b = 0.150

b) C_p distribution at wing section y/b = 0.331



c) C_p distribution at wing section y/b = 0.409

d) C_p distribution at wing section y/b = 0.844

Fig. 8 C_p distributions at $C_L = 0.5$, $Re = 5E6$ (workshop), and AOA = 0.49, $Re = 3E6$ (wind-tunnel experiment), RANS simulations.

2) The conditions from the wind-tunnel experiment are matched. In this case, the angle of attack is set to 0.49 deg with a Reynolds number of 3E6. With this particular setting, despite the fact that the lift coefficient is overpredicted ($C_L = 0.53$), we obtain very good agreement with the experimental data, except near the most outboard section of the wing where there is a mismatch in the shock wave location (probably due to the low resolution of the numerical grid).

This complex, full-aircraft configuration is a perfect example for demonstrating the adjoint RANS solver that is integrated in SU2 for obtaining the sensitivities needed for shape design. After solving the RANS equations, the direct flow solution and the same computational mesh can be immediately reused as inputs for solving the adjoint RANS equations in the solver (while taking advantage of similar numerical methods and the same code structure). Although using both less computational time and memory resources than in the direct problem with the present continuous adjoint formulation, it is possible to evaluate the surface sensitivity after solving the RANS adjoint equations for a particular objective function.

The pressure distributions on the upper and lower surfaces and the surface sensitivity (for the drag, lift, and pitching-moment coefficients) are shown in Fig. 9. This sensitivity information reveals the impact of a particular geometrical change on the selected objective function and can be used for gradient-based shape optimization or directly by the designer to manually improve the shape of the aircraft.

B. Lockheed Martin Lockheed Martin 1021 Supersonic Aircraft

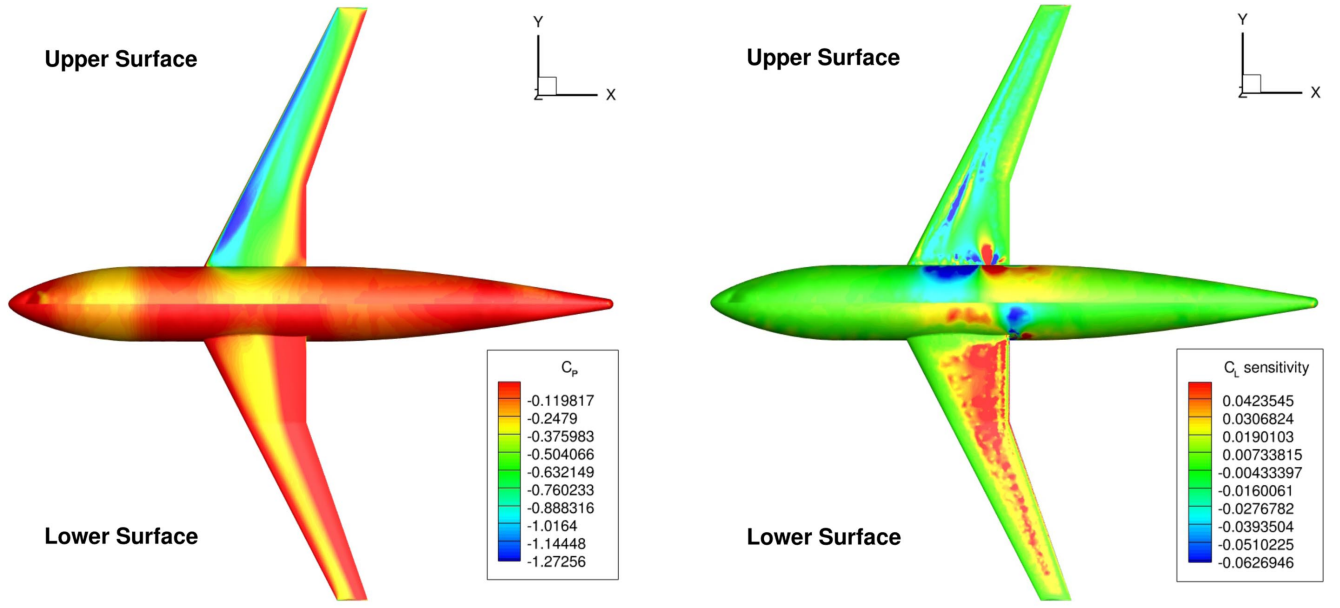
Our second demonstration is the computation of supersonic flow (viscous and inviscid) over the Lockheed Martin 1021 supersonic aircraft concept. Aircraft geometry and flight conditions [54] used in this simulation were specified in the 1st AIAA Sonic Boom Prediction Workshop.[‡‡]

A mixed-element grid composed of 5,730,841 total elements and 2,034,476 nodes is used for the inviscid simulation ($M_\infty = 1.6$, AOA = 2.1), and the grid provided by the workshop organizers (13,737,358 total elements and 2,395,158 nodes) is used for the RANS simulation ($M_\infty = 1.6$, AOA = 2.1, Re /ft = 2.55e6). Both meshes are constructed from tetrahedral, prismatic, and pyramidal elements atop a triangular surface mesh (shown in Fig. 10). The volumetric domain is a priori adapted along the freestream Mach lines for accurate shock capturing to ensure a well-resolved sonic boom signature at the far field, which is located three body lengths from the aircraft.

The validation of the CFD solution is performed via comparison with experimental results. In Fig. 11, the pressure at the near field is presented ($h = 0.4998$, $h = 0.8077$, and zero azimuth angle). There

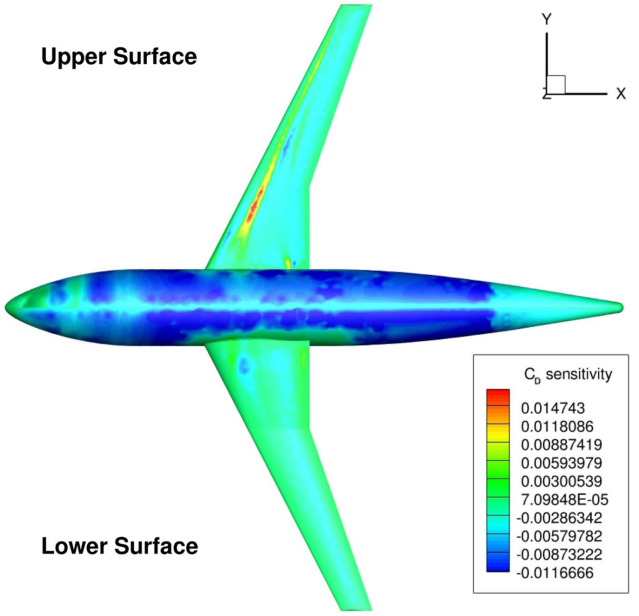
[††]Data available online at http://aaac.larc.nasa.gov/tsab/cfdlarc/aiaa-dpw/Workshop3/ [retrieved 2015].

[‡‡]Data available online at http://lbpw.larc.nasa.gov/sbpw1/ [retrieved 2014].

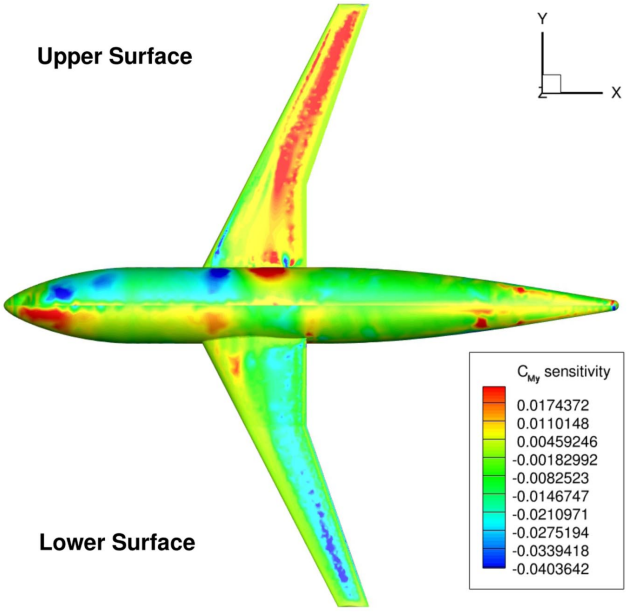


a) C_p contours

b) C_L surface sensitivity contours



c) C_D surface sensitivity contours



d) C_{M_y} surface sensitivity contours

Fig. 9 Pressure and surface sensitivity contours on the DLR-F6 aircraft geometry (lower and upper surfaces).

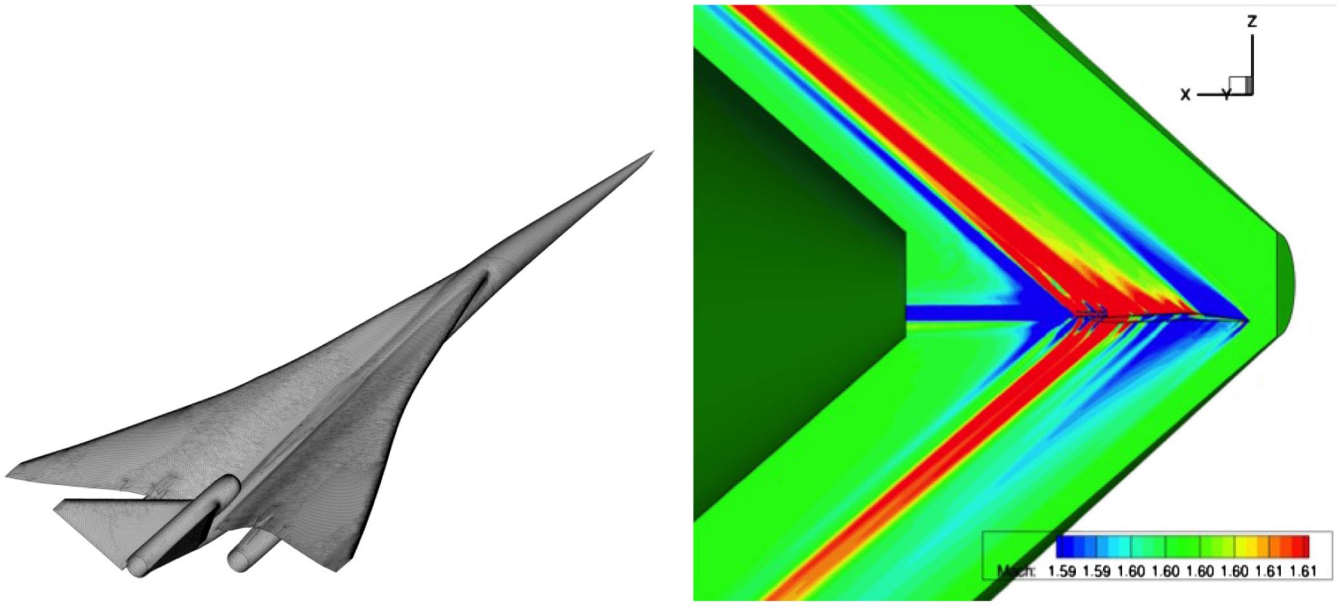
is an excellent agreement for the first two-thirds of the pressure signature, but some small discrepancies appear at the end of the signature. The end of the signature corresponds with the aft section of the aircraft and is affected by the sting of the wind-tunnel model, which was coarsely meshed in this simulation. A refinement of the numerical grid in that location will likely improve the results. As shown in [55], an excellent agreement with experimental data can be also obtained using inviscid simulations.

With respect to the numerical methods, a Roe upwind spatial discretization has been used to calculate convective fluxes, the turbulent variable for the S-A model is convected using a first-order scalar upwind method, and the viscous fluxes are calculated using the corrected average-gradient method. Implicit, local time stepping is used to converge the problem to the steady-state solution, and the linear system is solved using the iterative GMRES method with a maximum error tolerance of $\mathcal{O}(10^{-6})$.

The results for the inviscid simulation (using a JST centered spatial discretization) are presented in Fig. 12. In particular, the surface C_p contours and the adjoint-computed sensitivities of C_D, C_L, and C_{My} are plotted. These sensitivity maps show the influence of local normal geometry perturbations on the respective performance coefficients. With this geometric sensitivity information, engineers can choose an appropriate design variable parameterization and use gradient-based optimization methods for optimal shape design.

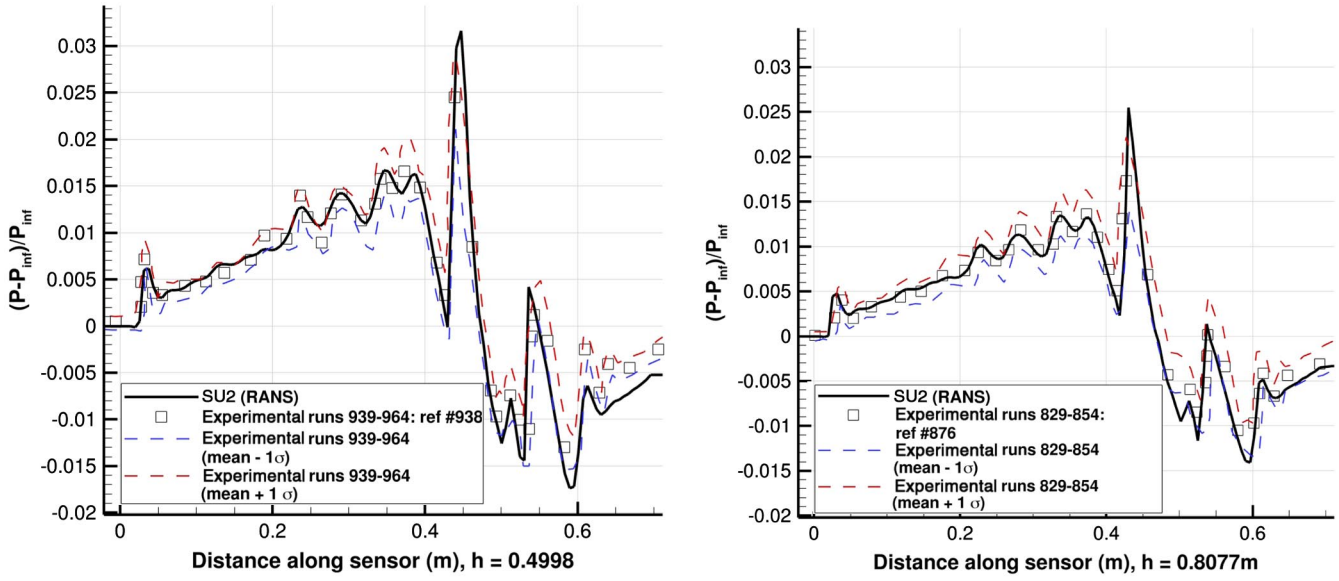
C. National Renewable Energy Laboratory Phase VI Wind Turbine

The NREL Phase VI wind turbine geometry consists of two blades with a radius of 5.029 m and a constant S809 airfoil section along the entire span. This geometry has been used widely for computational fluid dynamics validation using the data from the NREL Phase VI Unsteady Aerodynamics Experiment [56]. The chosen test case for the present study is sequence S with a 7 m/s windspeed and 72 rpm. The mixed-element computational mesh consists of 3.2 million nodes and 7.9 million elements, with triangles on the surface of the blade and prismatic elements in the boundary layer [57]. A layer of pyramids allows for the transition to tetrahedral elements outside of the boundary layer out to the far field.



a) View of the surface mesh b) Mach contours in the symmetry plane

Fig. 10 Lockheed Martin 1021 computational grid and solution.



a) Near-field pressure (h = 0.4998, Φ = 0 deg) b) Near-field pressure (h = 0.8077, Φ = 0 deg)

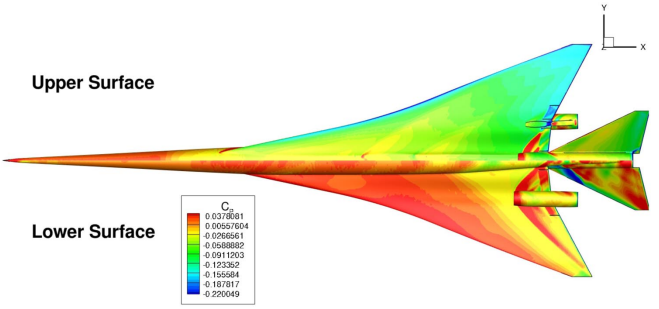
Fig. 11 Lockheed Martin 1021 pressure signature at the near field ($\Phi = 0$ deg): RANS simulation.

The noninertial RANS equations with the standard S-A turbulence model were chosen for analyzing the turbine. For validation purposes, Fig. 13 gives C_p distributions at two radial stations as computed by SU2 and compared to experiment, and Fig. 14 presents the C_p contours on the blade surface. Excellent agreement is seen overall, apart from near the trailing edge of the blade where some discrepancies are found (large spikes in C_p are also seen at the sharp trailing edge due to the geometry/mesh). The surface sensitivity was also computed for a torque objective function and can be seen in Fig. 14. It should be noted that the most sensitive locations on the blade surface are outboard locations along the span highlighted by the surface sensitivity contours. More details on the noninertial formulation and shape design examples (including the use of this geometry as a baseline) can be found in previous work by Economon et al. [24,26].

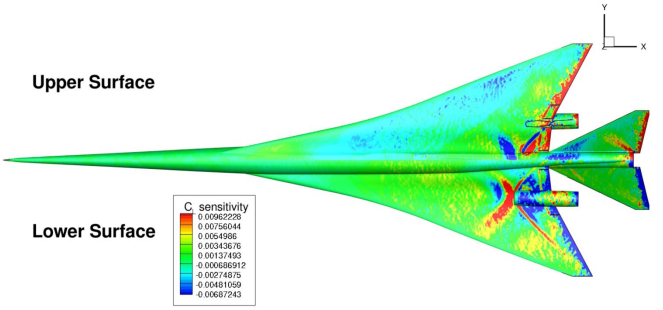
D. Hypersonic Flight-Test Vehicle

As a demonstration of the flexibility in the SU2 framework, hypersonic flow in thermochemical nonequilibrium is simulated for the RAM-C II flight-test article [58]. Additional convection equations and source terms are added to accommodate the new physical modeling via a definition of the appropriate CSolver, CVariable, and CNumerics child classes, whereas the code infrastructure (including the linear solvers, pre- and postprocessors, and MPI architecture) are all shared with the core software infrastructure. This framework allows for rapid feature expansion to include a variety of complex physical models with minimal development time, and these new features are linked to the existing mesh-adaptation, deformation, and gradient-projection modules, quickly providing a variety of powerful analysis and design techniques to bring to bear on multiphysics systems.

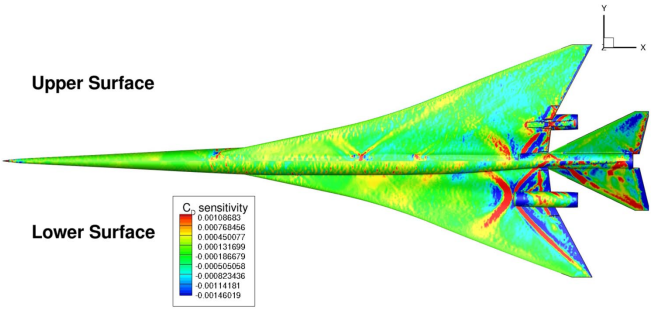
The RAM-C II flight-test article is a 9 deg sphere-cone geometry with a nose radius of 1524 m and a total length of 1.295 m. To simulate the hypersonic flight environment, a body-conformal $129 \times 96 \times 5$ hexahedral mesh is extruded over a 10 deg sector of the axisymmetric body. A rigid-rotator-harmonic-oscillator two-temperature thermodynamic model is used, and the finite rate chemical



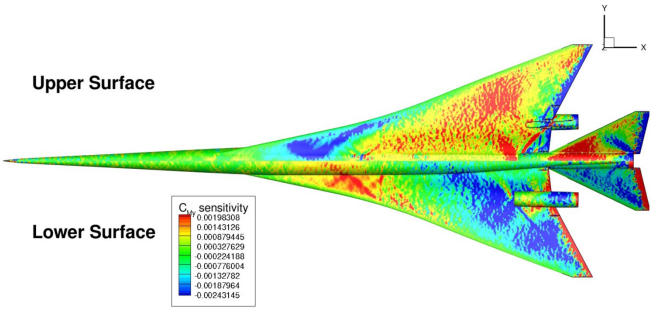
a) C_p contours



b) C_L surface sensitivity contours

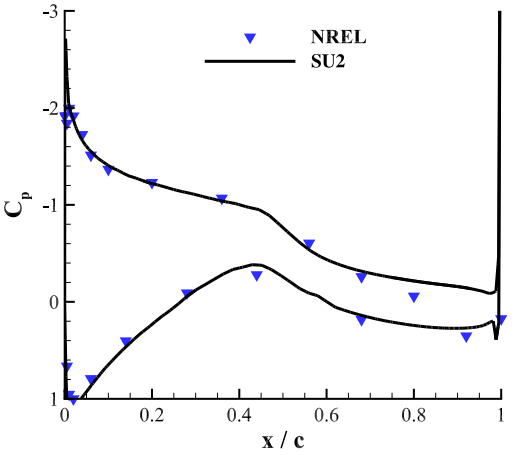


c) C_D surface sensitivity contours

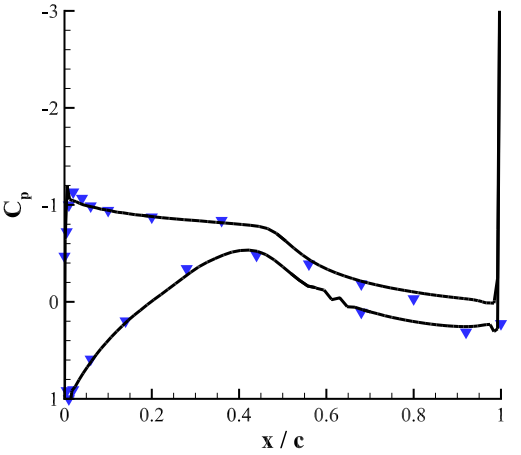


d) C_{My} surface sensitivity contours

Fig. 12 Pressure and surface sensitivity contours on the Lockheed Martin 1021 aircraft geometry (lower and upper surfaces): inviscid simulations.

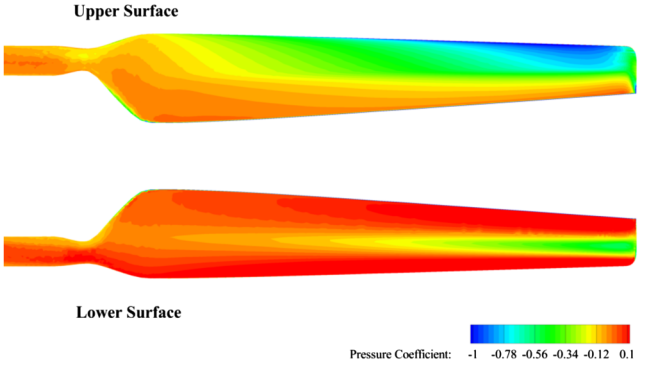


a) $r/R = 0.63$

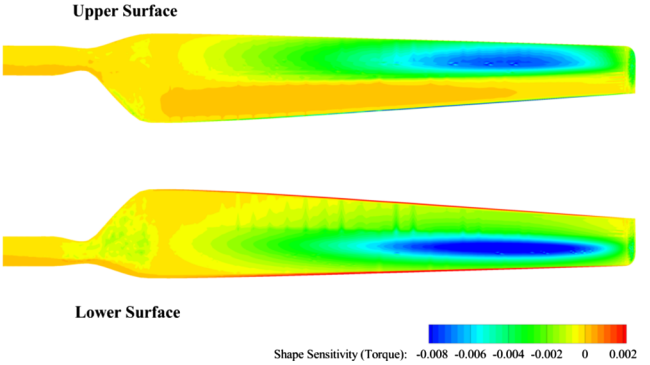


b) $r/R = 0.95$

Fig. 13 C_p distributions at multiple radial blade stations compared with experimental data.

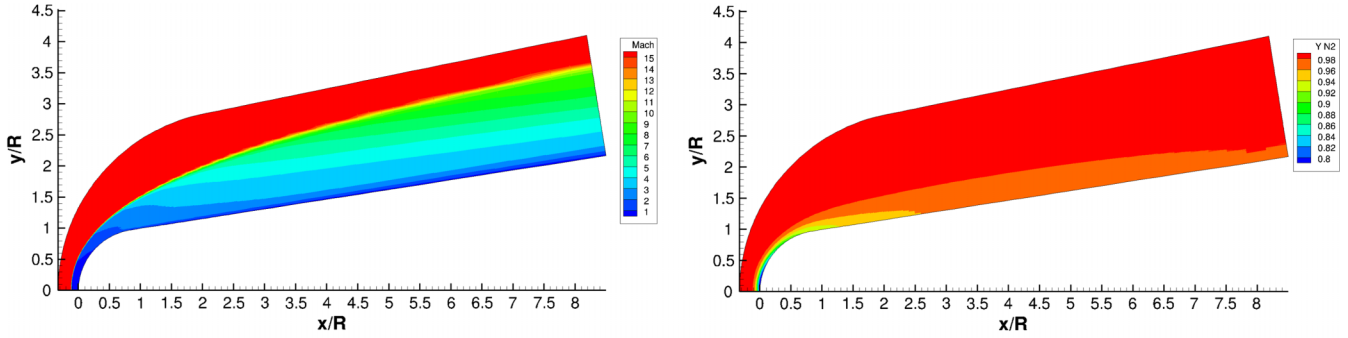


a) Surface contours of pressure coefficient



b) Surface sensitivity contours for a torque objective function

Fig. 14 C_p and surface sensitivity contours on the NREL Phase VI wind turbine blade.



a) Mach number b) N_2 mass fraction

Fig. 15 Slices of the RAM-C II volumetric solution.

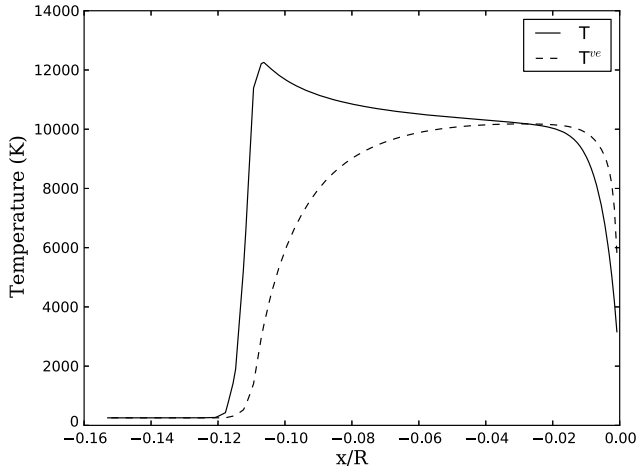


Fig. 16 Stagnation line temperature profiles.

model is used for a nonionizing $N_2 - N$ gas mixture. A full description of the physical modeling can be found in [5].

Slices of the volume solution for a $M_\infty = 16$, $P_\infty = 4000$ Pa, $T_\infty = T_\infty^{ve} = 254$ K freestream are shown in Fig. 15, and stagnation line temperature profiles can be found in Fig. 16. The strong, detached bow shock leads to N_2 dissociation and thermodynamic nonequilibrium in the shock layer. Vibrational relaxation and nitrogen recombination occur as the gas expands around the vehicle shoulder and near the cold-wall (1500 K) boundary condition.

VI. Conclusions

This paper has presented a detailed overview of the objectives, software framework, modeling capabilities, and numerical implementation of the SU2 analysis and optimization suite. The suite can be used to analyze the behavior of problems governed by arbitrary PDEs that are discretized on general, unstructured meshes. Moreover, SU2 readily provides the sensitivity information that is necessary for solving PDE-constrained optimization problems (shape design), goal-oriented mesh adaptation, or uncertainty quantification. The suite takes advantage of modern programming techniques, resulting in a code that is portable, reusable, modular, and freely available through an open-source license.

In addition to describing many of the details of the software suite, this paper also discussed several recent examples of our work using SU2 for flow and sensitivity analyses. The DLR-F6 and the Lockheed Martin 1021 configurations are representative of the scale and complexity of the cases found in the aerospace industry today. The NREL Phase VI wind turbine and RAM-C II hypersonic flight test vehicle test cases illustrate the flexibility of the suite for handling a wide range of applications that may also require the implementation of additional physical models. The flow analysis capabilities in SU2 have been rigorously verified and validated to ensure that accurate high-fidelity performance predictions for complex configurations are obtained. Furthermore, the solution of the corresponding adjoint systems for these problems provides sensitivity information that can be immediately used for shape optimization. The ability to efficiently solve the flow and adjoint equations combined with the surrounding infrastructure for automatic shape design make SU2 a uniquely powerful software suite.

Lastly, it is important to highlight that SU2 is connected to a global community of researchers and developers in the field of scientific computing for engineering applications. The release of the software under an open-source license has enabled engineers and scientists from around the world to work from a common codebase and provides worldwide access to industry-standard analysis tools. Advances in CFD, shape design, and numerical methods can be rapidly disseminated to a wide, knowledgeable user base in an established, online community. Moreover, at the time of writing, multiple institutions around the world are beginning to contribute new capabilities to the source code, which is a trend that will continue to be encouraged and supported in the future.

Acknowledgments

The authors are grateful to Antony Jameson, Enrique Zuazua, Carlos Castro, Nicolas Gauger, Piero Colonna, Jason Hicken, Karthik Duraisamy, Alberto Guardone, Matteo Pini, Michael R. Colonno, Alfonso Bueno, Gérald Carrier, Ben Kirk, Justin Gray, Travis Carrigan, Michael Buonanno, Aniket Aranake, Alejandro Campos, Giulio Gori, Heather Kline, Amrita Lonkar, David Manosalvas, Kedar Naik, Santiago Padrón, Thomas Taylor, Brendan Tracey, Anil Variyar, Salvatore Vitale, and Andrew Wendorff for helpful discussions and their efforts in the development and validation of the code.

References

[1] Palacios, F., Colonno, M. R., Aranake, A. C., Campos, A., Copeland, S. R., Economon, T. D., Lonkar, A. K., Lukaczyk, T. W., Taylor, T. W. R., and Alonso, J. J., "Stanford University Unstructured (SU^2): An open-Source Integrated Computational Environment for Multi-Physics Simulation and Design," AIAA Paper 2013-0287, 2013.

[2] Palacios, F., Economon, T. D., Aranake, A. C., Copeland, S. R., Lonkar, A. K., Lukaczyk, T. W., Manosalvas, D. E., Naik, K. R., Padron, A. S., Tracey, B., Variyar, A., and Alonso, J. J., "Stanford University Unstructured (SU^2): Open-Source Analysis and Design Technology for Turbulent Flows," AIAA Paper 2014-0243, 2014.

[3] Colonno, M. R., Naik, K., Duraisamy, K., and Alonso, J. J., "An Adjoint-Based Multidisciplinary Optimization Framework for Rotorcraft Systems," AIAA Paper 2012-5656, 2012.

[4] Economon, T. D., Palacios, F., and Alonso, J. J., "A Coupled-Adjoint Method for Aerodynamic and Aeroacoustic Optimization," AIAA Paper 2012-5598, 2012.

[5] Copeland, S. R., Palacios, F., and Alonso, J. J., "Adjoint-Based Aerothermodynamic Shape Design of Hypersonic Vehicles in Non-Equilibrium Flows," AIAA Paper 2014-0513, 2014.

[6] Karypis, G., Schloegel, K., and Kumar, V., "Parmetis: Parallel Graph Partitioning and Sparse Matrix Ordering Library: Version 1.0," Dept. of Computer Science, Univ. of Minnesota, Minneapolis, MN, 1997.

[7] Samareh, J. A., "Aerodynamic Shape Optimization Based on Free-Form Deformation," AIAA Paper 2004-4630, 2004.

[8] Hicks, R. M., and Henne, P. A., "Wing Design by Numerical Optimization," *Journal of Aircraft*, Vol. 53, No. 12, 1978, pp. 407–412. doi:10.2514/3.58379

[9] Dwight, R. P., "Robust Mesh Deformation using the Linear Elasticity Equations," *Proceedings of the Fourth International Conference on Computational Fluid Dynamics*, Springer, Berlin, July 2006, pp. 401–406.

[10] Economon, T. D., Palacios, F., Alonso, J. J., Bansal, G., Mudigere, D., Deshpande, A., Heinecke, A., and Smelyanskiy, M., "Towards High-Performance Optimizations of the Unstructured Open-Source SU2 Suite," AIAA Paper 2015-1949, 2015.

[11] Vitale, S., Gori, G., Pini, M., Guardone, A., Economon, T. D., Palacios, F., Alonso, J. J., and Colonna, P., "Extension of the SU2 Open Source CFD Code to the Simulation of Turbulent Flows of Fuids Modelled with Complex Thermophysical Laws," AIAA Paper 2015-2760, 2015.

[12] Zhou, B. Y., Albring, T. A., Gauger, N. R., Economon, T. D., Palacios, F., and Alonso, J. J., "A Discrete Adjoint Framework for Unsteady Aerodynamic and Aeroacoustic Optimization," AIAA Paper 2015-3355, 2015.

[13] Hirsch, C., *Numerical Computation of Internal and External Flows*, Wiley, New York, 1984, pp. 372–384.

[14] Wilcox, D., *Turbulence Modeling for CFD*, 2nd ed., DCW Industries, Inc., La Cañada, CA, 1998, pp. 53–59.

[15] White, F. M., *Viscous Fluid Flow*, McGraw–Hill, New York, 1974, pp. 28–29.

[16] Spalart, P., and Allmaras, S., "A One-Equation Turbulence Model for Aerodynamic Flows," AIAA Paper 1992-0439, 1992.

[17] Allmaras, S. R., Johnson, F. T., and Spalart, P. R., "Modifications and Clarifications for the Implementation of the Spalart–Allmaras Turbulence Model," *The International Conference on Computational Fluid Dynamics (ICCFD)*, ICCFD7 Paper 1902, 2012.

[18] Menter, F. R., "Two-Equation Eddy-Viscosity Turbulence Models for Engineering Applications," *AIAA Journal*, Vol. 32, No. 8, 1994, pp. 1598–1605. doi:10.2514/3.12149

[19] Bueno-Orovio, A., Castro, C., Palacios, F., and Zuazua, E., "Continuous Adjoint Approach for the Spalart–Allmaras Model in Aerodynamic Optimization," *AIAA Journal*, Vol. 50, No. 3, 2012, pp. 631–646. doi:10.2514/1.J051307

[20] Anderson, W. K., and Venkatakrishnan, V., "Aerodynamic Design Optimization on Unstructured Grids with a Continuous Adjoint Formulation," AIAA Paper 1997-0643, 1997.

[21] Jameson, A., and Kim, S., "Reduction of the Adjoint Gradient Formula for Aerodynamic Shape Optimization Problems," *AIAA Journal*, Vol. 41, No. 11, 2003, pp. 2114–2129. doi:10.2514/2.6830

[22] Castro, C., Lozano, C., Palacios, F., and Zuazua, E., "Systematic Continuous Adjoint Approach to Viscous Aerodynamic Design on Unstructured Grids," *AIAA Journal*, Vol. 45, No. 9, 2007, pp. 2125–2139. doi:10.2514/1.24859

[23] Palacios, F., Alonso, J. J., Colonno, M., Hicken, J., and Lukaczyk, T., "Adjoint-Based Method for Supersonic Aircraft Design Using Equivalent Area Distributions," AIAA Paper 2012-0269, 2012.

[24] Economon, T. D., Palacios, F., and Alonso, J. J., "Optimal Shape Design for Open Rotor Blades," AIAA Paper 2012-3018, 2012.

[25] Economon, T. D., Palacios, F., and Alonso, J. J., "Unsteady Aerodynamic Design on Unstructured Meshes with Sliding Interfaces," AIAA Paper 2013-0632, 2013.

[26] Economon, T. D., Palacios, F., and Alonso, J. J., "A Viscous Continuous Adjoint Approach for the Design of Rotating Engineering Applications," AIAA Paper 2013-2580, 2013.

[27] Palacios, F., Alonso, J. J., and Jameson, A., "Optimal Design of Free-Surface Interfaces Using a Level Set Methodology," AIAA Paper 2012-3341, 2012.

[28] Palacios, F., Alonso, J. J., and Jameson, A., "Design of Free-Surface Interfaces Using RANS Equations," AIAA Paper 2013-3110, 2013.

[29] Palacios, F., Economon, T. D., and Alonso, J. J., "Large-Scale Aircraft Design Using SU2," AIAA Paper 2015-1946, 2015.

[30] Economon, T., Palacios, F., and Alonso, J., "Unsteady Continuous Adjoint Approach for Aerodynamic Design on Dynamic Meshes," *AIAA Journal*, Vol. 53, No. 9, 2015, pp. 2437–2453,

[31] Barth, T. J., "Aspects of Unstructured Grids and Finite-Volume Solvers for the Euler and Navier–Stokes Equations," *25th Computational Fluid Dynamics*, AGARD, von Kármán Inst. Lecture Series, March 1994.

[32] Quarteroni, A., and Valli, A., *Numerical Approximation of Partial Differential Equations*, Vol. 23, Springer Series in Computational Mathematics, Springer–Verlag, Berlin, 1997, pp. 501–508.

[33] Jameson, A., "A Perspective on Computational Algorithms for Aerodynamic Analysis and Design," *Progress in Aerospace Sciences*, Vol. 37, No. 2, 2001, pp. 197–243. doi:10.1016/S0376-0421(01)00004-5

[34] LeVeque, R., *Finite Volume Methods for Hyperbolic Problems*, Cambridge Univ. Press, New York, 2002, pp. 64–85.

[35] Wesseling, P., *Principles of Computational Fluid Dynamics*, Vol. 29, Springer Series in Computational Mathematics, Springer–Verlag, Berlin, 2000, pp. 81–110.

[36] Jameson, A., "Analysis and Design of Numerical Schemes for Gas Dynamics, 1: Artificial Diffusion, Upwind Biasing, Limiters and Their Effect on Accuracy and Multigrid Convergence," *International Journal of Computational Fluid Dynamics*, Vol. 4, Nos. 3–4, 1995, pp. 171–218. doi:10.1080/10618569508904524

[37] Jameson, A., "Analysis and Design of Numerical Schemes for Gas Dynamics, 2: Artificial Diffusion and Discrete Shock Structure," *International Journal of Computational Fluid Dynamics*, Vol. 5, Nos. 1–2, 1995, pp. 1–38. doi:10.1080/10618569508940734

[38] Toro, E. F., *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*, Springer–Verlag, New York, 1999, pp. 573–580.

[39] Jameson, A., Schmidt, W., and Turkel, E., "Numerical Solution of the Euler Equations by Finite Volume Methods Using Runge-Kutta Time Stepping Schemes," AIAA Paper 1981-1259, 1981.

[40] Roe, P. L., "Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes," *Journal of Computational Physics*, Vol. 43, No. 2, 1981, pp. 357–372. doi:10.1016/0021-9991(81)90128-5

[41] Liou, M.-S., and Steffen, C. J., Jr., "A New Flux Splitting Scheme," *Journal of Computational Physics*, Vol. 107, No. 1, 1993, pp. 23–39. doi:10.1006/jcph.1993.1122

[42] Turkel, E., Vatsa, V. N., and Radespiel, R., "Preconditioning Methods for Low-Speed Flows," AIAA Paper 1996-2460, 1996.

[43] Saad, Y., and Schultz, M. H., "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems," *SIAM Journal on Scientific and Statistical Computing*, Vol. 7, No. 3, July 1986, pp. 856–869. doi:10.1137/0907058

[44] van der Vorst, H. A., "BI-CGSTAB: A Fast and Smoothly Converging Variant of BI-CG for the Solution of Nonsymmetric Linear Systems," *SIAM Journal on Scientific and Statistical Computing*, Vol. 13, No. 2, March 1992, pp. 631–644. doi:10.1137/0913035

[45] Jameson, A., "Time Dependent Calculations Using Multigrid, with Applications to Unsteady Flows Past Airfoils and Wings," AIAA Paper 1991-1596, 1991.

[46] Jameson, A., and Schenectady, S., "An Assessment of Dual-Time Stepping, Time Spectral and Artificial Compressibility Based Numerical Algorithms for Unsteady Flow with Applications to Flapping Wings," AIAA Paper 2009-4273, 2009.

[47] Jameson, A., Martinelli, L., and Grasso, F., "A Multigrid Method for the Navier–Stokes Equations," AIAA Paper 1986-0208, 1986.

[48] Mavriplis, D. J., "On Convergence Acceleration Techniques for Unstructured Meshes," Inst. for Computer Applications on Science and Engineering, NASA Langley Research Center ICASE Rept. TR-98-44, Hampton, VA, 1998.

[49] Mavriplis, D. J., "Multigrid Techniques for Unstructured Meshes," Inst. for Computer Applications on Science and Engineering, NASA Langley Research Center TR-95-27, Hampton, VA, 1995.

[50] Palacios, F., and Alonso, J. J., "New Convergence Acceleration Techniques in the Joe Code," Annual Research Briefs, Center for Turbulence Research, Stanford Univ. and NASA Ames, 2011, pp. 297–308.

[51] Pierce, N. A., and Giles, M. B., "Preconditioned Multigrid Met hods for Compressible Flow Calculations on Stretched Meshes," *Journal of Computational Physics*, Vol. 136, No. 2, 1997, pp. 425–445. doi:10.1006/jcph.1997.5772

[52] Soto, O., Löhner, R., and Camelli, F., "A Linelet Preconditioner for Incompressible Flow Solvers," *International Journal of Numerical Methods for Heat and Fluid Flow*, Vol. 13, No. 1, 2003, pp. 133–147. doi:10.1108/09615530310456796

[53] Brodersen, O., and Stürmer, A., "Drag Prediction of Engine-Airframe Interference Effects Using Unstructured Navier–Stokes Calculations," AIAA Paper 2001-2414, 2001.

[54] Morgenstern, J. M., Buonanno, M., and Marconi, F., "Full Configuration Low Boom Model and Grids for 2014 Sonic Boom Prediction Workshop," AIAA Paper 2013-647, 2013.

[55] Aftosmis, A. J., and Nemec, M., "Cart3-D Simulations for the First AIAA Sonic Boom Prediction Workshop," AIAA Paper 2014-0558, 2014.

[56] Simms, D. A., Schreck, S., Hand, M., and Fingersh, L. J., "NREL Unsteady Aerodynamics Experiment in the NASA-Ames Wind Tunnel: A Comparison of Predictions to Measurements," National Renewable Energy Lab. TR NREL/TP-500-29494, Golden, CO, June 2001.

[57] Potsdam, M. A., and Mavriplis, D. J., "Unstructured Mesh CFD Aerodynamic Analysis of the NREL Phase VI Rotor," AIAA Paper 2009-1221, 2009.

[58] Jones, W. L., and Cross, A. E., "Electrostatic Probe Measurements of Plasma Parameters for Two Reentry Flight Experiments at 25,000 Feet per Second," NASA TR-TN-D-6617, 1972.

J. R. R. A. Martins
Associate Editor