OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization

Justin S. Gray · **John T. Hwang** · **Joaquim R. R. A. Martins** · **Kenneth T. Moore** ·
Bret A. Naylor

Abstract Multidisciplinary design optimization (MDO) is concerned with solving design problems involving coupled numerical models of complex engineering systems. While various MDO software frameworks exist, none of them take full advantage of state-of-the-art algorithms to solve coupled models efficiently. Furthermore, there is a need to facilitate the computation of the derivatives of these coupled models for use with gradient-based optimization algorithms to enable design with respect to large numbers of variables. In this paper, we present the theory and architecture of Open-MDAO, an open-source MDO framework that uses Newton-type algorithms to solve coupled systems and exploits problem structure through new hierarchical strategies to achieve high computational efficiency. OpenMDAO also provides a framework for computing coupled derivatives efficiently and in a way that exploits problem sparsity. We demonstrate the framework's efficiency by benchmarking scalable test prob-lems. We also summarize a number of OpenMDAO applications previously reported in the literature, which include trajectory optimization, wing design, and structural topology optimization, demonstrating that the framework is effective in both coupling existing models and developing new multidisciplinary models from the ground up. Given the potential of the OpenMDAO framework, we expect the number of users and developers to continue growing, enabling even more diverse applications in engineering analysis and design.

Keywords Multidisciplinary design optimization · Coupled systems · Complex systems · Sensitivity analysis · Derivative computation · Adjoint methods · Python

Justin S. Gray
NASA Glenn Research Center,
Cleveland, OH, USA
E-mail: justin.s.gray@nasa.gov

John T. Hwang
University of California, San Diego,
San Diego, CA, USA
E-mail: jhwang@eng.ucsd.edu

Joaquim R. R. A. Martins
Department of Aerospace Engineering, University of Michigan,
Ann Arbor, MI, USA
E-mail: jrram@umich.edu

Kenneth T. Moore
DB Consulting Group (NASA Glenn Research Center),
Cleveland, OH, USA
E-mail: kenneth.t.moore-1@nasa.gov

Bret A. Naylor
DB Consulting Group (NASA Glenn Research Center),
Cleveland, OH, USA
E-mail: bret.a.naylor@nasa.gov

1 Introduction

Numerical simulations of engineering systems have been widely developed and used in industry and academia. Simulations are often used within an engineering design cycle to inform design choices. Design optimization—the use of numerical optimization techniques with engineering simulation—has emerged as a way of incorporating simulation into the design cycle.

Multidisciplinary design optimization (MDO) arose from the need to simulate and design complex engineering systems involving multiple disciplines. MDO serves this need in two ways. First, it performs the coupled simulation of the engineering system, taking into account all the interdisciplinary interactions. Second, it performs the simultaneous optimization of all design variables, taking into account the coupling and the interdisciplinary design tradeoffs. MDO is sometimes referred to as MDAO (multidisciplinary analysis and optimization) to emphasize that the coupled analysis is useful on its own. MDO was first conceived to solve aircraft design problems, where disciplines such as aerodynamics,

structures, and controls are tightly coupled and require design tradeoffs [39]. Since then, numerical simulations have advanced in all disciplines, and the power of computer hardware has increased dramatically. These developments make it possible to advance the state-of-the-art in MDO, but other more specific developments are needed.

There are two important factors when evaluating MDO strategies: implementation effort and the computational efficiency. The implementation effort is arguably the most important because if the work required to implement a multidisciplinary model is too large, the model will simply never be built. One of the main MDO implementation challenges is that each analysis code consists of a specialized solver that is typically not designed to be coupled to other codes or to be used for numerical optimization. Additionally, these solvers are often coded in different programming languages and use different interfaces. These difficulties motivated much of the early development of MDO frameworks, which provided simpler and more efficient ways to link discipline analyses together. While these MDO frameworks introduce important innovations in software design, modular model construction, and user interface design, they treat each discipline analysis as an explicit function evaluation; that is, they assume that each discipline is an explicit mapping between inputs and outputs. This limits the efficiency of the nonlinear solution algorithms that could be used to find a solution to the coupled multidisciplinary system. Furthermore, these MDO frameworks also present the combined multidisciplinary model as an explicit function to the optimizer, which limits the efficiency when computing derivatives for gradient-based optimization of higher-dimensional design spaces. Therefore, while these first framework developments addressed the most pressing issue by significantly lowering the implementation effort for multidisciplinary analysis, they did not provide a means for applying the most efficient MDO techniques.

The computational efficiency of an MDO implementation is governed by the efficiency of the coupled (multidisciplinary) analysis and the efficiency of the optimization. The coupled analysis method that is easiest to implement is a fixed-point iteration (also known as nonlinear block Gauss–Seidel iteration), but for strongly coupled models, Newton-type methods are potentially more efficient [40, 44, 58, 15]. When it comes to numerical optimization, gradient-based optimization algorithms scale much better with the number of design variables than gradient-free methods. The computational efficiency of both Newton-type analysis methods and gradient-based optimization is, in large part, dependent on the cost and accuracy with which the necessary derivatives are computed.

One can always compute derivatives using finite differences, but analytic derivative methods are much more efficient and accurate. Despite the extensive research into analytic derivatives and their demonstrated benefits, they have not been widely supported in MDO frameworks because their implementation is complex and requires deeper access to the analysis code than can be achieved through an approach that treats all analyses as explicit functions. Therefore, users of MDO frameworks that follow this approach are typically restricted to gradient-free optimization methods, or gradient-based optimization with derivatives computed via finite differences.

The difficulty of implementing MDO techniques with analytic derivatives creates a significant barrier to their adoption by the wider engineering community. The OpenMDAO framework aims to lower this barrier and enable the widespread use of analytic derivatives in MDO applications.

Like other frameworks, OpenMDAO provides a modular environment to more easily integrate discipline analyses into a larger multidisciplinary model. However, OpenMDAO V2 improves upon other MDO frameworks by integrating discipline analyses as implicit functions, which enables it to compute derivatives for the resulting coupled model via the unified derivatives equation [67]. The computed derivatives are *coupled* in that they take into account the full interaction between the disciplines in the system. Furthermore, OpenMDAO is designed to work efficiently in both serial and parallel computing environments. Thus, OpenMDAO provides a means for users to leverage the most efficient techniques, regardless of problem size and computing architecture, without having to incur the significant implementation difficulty typically associated with gradient-based MDO.

This paper presents the design and algorithmic features of OpenMDAO V2 and is structured to cater to different types of readers. For readers wishing to just get a quick overview of what OpenMDAO is and what it does, reading this introduction, the overview of applications (Sec. 7, especially Table 13), and the conclusions (Sec. 8) will suffice. Potential OpenMDAO users should also read Sec. 3, which explains the basic usage and features through a simple example. The remainder of the paper provides a background on MDO frameworks and the history of OpenMDAO development (Sec. 2), the theory behind OpenMDAO (Sec. 4), and the details of the major new contributions in OpenMDAO V2 in terms of multidisciplinary solvers (Sec. 5) and coupled derivative computation (Sec. 6).

2 Background

The need for frameworks that facilitate the implementation of MDO problems and their solution was identified soon after MDO emerged as a field. Various requirements have been identified over the years. Early on, Salas and Townsend [88] detailed a large number of requirements that they categorized under software design, problem formulation, problem execution, and data access. Later, Padula and Gillian

[81] more succinctly cited modularity, data handling, parallel processing, and user interface as the most important requirements. While frameworks that fulfill these requirements to various degrees have emerged, the issue of computational efficiency or scalability has not been sufficiently highlighted or addressed.

The development of commercial MDO frameworks dates back to the late 1990s with iSIGHT [29], which is now owned by Dassault Systèmes and renamed Isight/SEE. Various other commercial frameworks have been developed, such as Phoenix Integration's ModelCenter/CenterLink, Esteco's modeFRONTIER, TechnoSoft's AML suite, Noesis Solutions' Optimus, SORCER [59], and Vanderplaats' VisualDOC [2]. These frameworks have focused on making it easy for users to couple multiple disciplines and to use the optimization algorithms through graphical user interfaces (GUIs). They have also been providing wrappers to popular commercial engineering tools. While this focus has made it convenient for users to implement and solve MDO problems, the numerical methods used to converge the multidisciplinary analysis (MDA) and the optimization problem are usually not state-of-the-art. For example, these frameworks often use fixed-point iteration to converge the MDA. When derivatives are needed for a gradient-based optimizer, finite-difference approximations are used rather than more accurate analytic derivatives.

When solving MDO problems, we have to consider how to organize the discipline analysis models, the problem formulation, and the optimization algorithm in order to obtain the optimum design with the lowest computational cost possible. The combination of the problem formulation and organizational strategy is called the *MDO architecture*. MDO architectures can be either monolithic (where a single optimization problem is solved) or distributed (where the problem is partitioned into multiple optimization subproblems). Martins and Lambe [69] describe this classification in more detail and present all known MDO architectures.

In an attempt to facilitate the exploration of the various MDO architectures, Tedford and Martins [93] developed pyMDO. This was the first object-oriented framework that focused on automating the implementation of different MDO architectures [73]. In pyMDO, the user defined the general MDO problem once, and the framework would reformulate the problem in any architecture with no further user effort. Tedford and Martins [94] used this framework to compare the performance of various MDO architectures, concluding that monolithic architectures vastly outperform the distributed ones. Marriage and Martins [66] integrated a semi-analytic method for computing derivatives based on a combination of finite-differencing and analytic methods, showing that the semi-analytic method outperformed the traditional black-box finite-difference approach.

The origins of OpenMDAO began in 2008, when Moore et al. [75] identified the need for a new MDO framework

to address aircraft design challenges at NASA. Two years later, Gray et al. [31] implemented the first version of OpenMDAO (V0.1). An early aircraft design application using OpenMDAO to implement gradient-free efficient global optimization was presented by Heath and Gray [43]. Gray et al. [32] later presented benchmarking results for various MDO architectures using gradient-based optimization with analytic derivatives in OpenMDAO.

As the pyMDO and OpenMDAO frameworks progressed, it became apparent that the computation of derivatives for MDO presented a previously unforeseen implementation barrier that these frameworks needed to address. The methods available for computing derivatives are finite-differencing, complex-step, algorithmic differentiation, and analytic methods. The finite-difference method is popular because it is easy to implement and can always be used, even without any access to source code, but it is subject to large inaccuracies. The complex-step method [91, 70] is not subject to these inaccuracies, but it requires access to the source code to implement. Both finite-difference and complex-step methods become prohibitively costly as the number of design variables increases because they require rerunning the simulation for each additional design variable. Algorithmic differentiation (AD) uses a software tool to parse the code of an analysis tool to produce new code that computes derivatives of that analysis [38, 76]. Although AD can be efficient, even for large numbers of design variables, it does not handle iterative simulations efficiently in general. Analytic methods are the most desirable because they are both accurate and efficient even for iterative simulations [67]. However, they require significant implementation effort. Analytic methods can be implemented in two different forms: the direct method and the adjoint method. The choice between these two methods depends on how the number of functions that we want to differentiate compares to the number of design variables. In practice, the adjoint method tends to be the more commonly used method.

Early development of the adjoint derivative computation was undertaken by the optimal control community in the 1960s and 1970s [11], and the structural optimization community adapted those developments throughout the '70s and '80s [1]. This was followed by the development of adjoint methods for computational fluid dynamics [51], and aerodynamic shape optimization became a prime example of an application where the adjoint method has been particularly successful [84, 12, 16]. When computing the derivatives of coupled systems, the same methods that are used for single disciplines apply. Sobieszczanski-Sobieski [90] presented the first derivation of the direct method for coupled systems, and Martins et al. [72] derived the coupled adjoint method. One of the first applications of the coupled adjoint method was in high-fidelity aerostructural optimization [71]. The results from the work on coupled derivatives

highlighted the promise of dramatic computational cost reductions, but also showed that existing frameworks were not able to handle these methods. Their implementation required linear solvers and support for distributed memory parallelism that no framework had at the time.

In an effort to unify the theory for the various methods for computing derivatives, Martins and Hwang [67] derived the *unified derivatives equation*. This new generalization showed that all the methods for computing derivatives can be derived from a common equation. It also showed that when there are both implicitly and explicitly defined disciplines, the adjoint method and chain rule can be combined in a hybrid approach. Hwang et al. [50] then realized that this theoretical insight provided a sound and convenient mathematical basis for a new software design paradigm and set of numerical solver algorithms for MDO frameworks. Using a prototype implementation built around the unified derivatives equation [50, 68], they solved a large-scale satellite optimization problem with 25,000 design variables and over 2 million state variables. Later, Gray et al. [33] developed OpenMDAO V1, a complete rewrite of the OpenMDAO framework based on the prototype work of Hwang et al. with the added ability to exploit sparsity in a coupled multidisciplinary model to further reduce computational cost.

Collectively, the work cited above represented a significant advancement of the state-of-the-art for MDO frameworks. The unified derivatives equation, combined with the new algorithms and framework design, enabled the solution of significantly larger and more complex MDO problems than had been previously attempted. In addition, OpenMDAO had now integrated three different methods for computing total derivatives into a single framework: finite-difference, analytic, and semi-analytic. However, this work was all done using serial discipline analyses and run on a serial computing environment. The serial computing environment presented a significant limitation, because it precluded the integration of high-fidelity analyses into the coupled models.

To overcome the serial computing limitation, Hwang and Martins [47] parallelized the data structures and solver algorithms from their prototype framework, which led to the modular analysis and unified derivatives (MAUD) architecture. Hwang and Martins [45] used the new MAUD prototype to solve a coupled aircraft allocation-mission-design optimization problem. OpenMDAO V1 was then modified to incorporate the ideas from the MAUD architecture. Gray et al. [36] presented an aeropropulsive design optimization problem constructed in OpenMDAO V1 that combined a high fidelity aerodynamics model with a low fidelity propulsion model, executed in parallel. One of the central features of the MAUD architecture, enabling the usage of parallel computing and high-fidelity analyses, was the use of hierarchical, matrix-free linear solver design. While advan-

tageous for large parallel models, this feature was inefficient for smaller serial models. The need to support both serial and parallel computing architectures led to the development of OpenMDAO V2, a second rewrite of the framework, which is presented in this paper.

Recently, the value of analytic derivatives has also motivated the development of another MDO framework, GEMS, which is designed to implement bi-level distributed MDO architectures that are more useful in some industrial settings [27]. This stands in contrast to OpenMDAO, which is focused mostly on the monolithic MDO architectures for best possible computational efficiency.

3 Overview of OpenMDAO V2

In this section, we introduce OpenMDAO V2, present its overall approach, and discuss its most important feature—efficient derivative computation. To help with the explanations, we introduce a simple model and optimization problem that we use throughout Secs. 3 and 4.

3.1 Basic description

OpenMDAO is an open-source software framework for multidisciplinary design, analysis, and optimization (MDAO), also known as multidisciplinary design optimization (MDO). It is primarily designed for gradient-based optimization; its most useful and unique features relate to the efficient and accurate computation of the model derivatives. We chose the Python programming language to develop OpenMDAO because it makes scripting convenient, it provides many options for interfacing to compiled languages (e.g., SWIG and Cython for C and C++, and F2PY for Fortran), and it is an open-source language. OpenMDAO facilitates the solution of MDO problems using distributed-memory parallelism and high-performance computing (HPC) resources by leveraging MPI and the PETSc library [3].

3.2 A simple example

This example consists of a model with one scalar input, x, two "disciplines" that define state variables (y_1, y_2), and one scalar output, f. The equations for the disciplines are

$$\text{(Discipline 1)} \quad y_1 = y_2^2 \tag{1}$$

$$\text{(Discipline 2)} \quad \exp(-y_1 y_2) - x y_2 = 0, \tag{2}$$

where Discipline 1 computes y_1 explicitly and Discipline 2 computes y_2 implicitly. The equation for the model output f is

$$f = y_1^2 - y_2 + 3. \tag{3}$$

Figure 1 visualizes the variable dependencies in this model using a design structure matrix. We show components that compute variables on the diagonal and dependencies on the off-diagonals. From Fig. 1, we can easily see the feedback loop between the two disciplines, as well as the overall sequential structure with the model input, the coupled disciplines, and the model output. We will refer back to this model and optimization problem periodically throughout Secs. 3 and 4.

To minimize f with respect to x using gradient-based optimization, we need the total derivative $\mathrm{d}f/\mathrm{d}x$. In Sec. 3.4 we use this example to demonstrate how OpenMDAO computes the derivative.
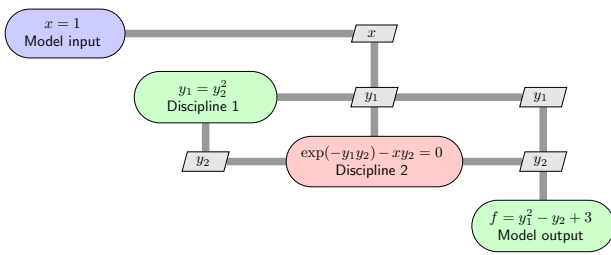
Fig. 1: Extended design structure matrix (XDSM) [62] for the simple model. Components that compute variables are on the diagonal, and dependencies are shown on the off-diagonals, where an entry above the diagonal indicates a forward dependence and vice versa. Blue indicates an independent variable, green indicates an explicit function, and red indicates an implicit function.

3.3 Approach and nomenclature

OpenMDAO uses an object-oriented programming paradigm and an object composition design pattern. Specific functionality via narrowly focused classes are combined to achieve the desired functionality during execution. In this section, we introduce the four most fundamental types of classes in OpenMDAO: *Component*, *Group*, *Driver*, and *Problem*. Note that for the *Component* class, the end user actually works with one of its two derived classes, *ExplicitComponent* or *ImplicitComponent*, which we describe later in this section.

MDO has traditionally considered multiple "disciplines" as the units that need to be coupled through coupling variables. In OpenMDAO, we consider more general *components*, which can represent a whole discipline analysis or can perform a smaller sub-set of calculations representing only a portion of a whole discipline model. Components share a common interface that allows them to be integrated to form a larger model. This modular approach allows OpenMDAO

to automate tasks that are performed repeatedly when building multidisciplinary models. Instances of the *Component* class provide the lowest-level functionality representing basic calculations. Each component instance maps input values to output values via some calculation. A component could be a simple explicit function, such as $y = \sin(x)$; it could involve a long sequence of code; or it could call an external code that is potentially written in another language. In multidisciplinary models, each component can encapsulate just a part of one discipline, a whole discipline, or even multiple disciplines. In our simple example, visualized in Fig. 1, there are four components: Discipline 1 and the model output are components that compute explicit functions, Discipline 2 is a component that computes an implicit function, and the model input is a special type of component with only outputs and no inputs.

Another fundamental class in OpenMDAO is *Group*, which contains components, other groups, or a mix of both. The containment relationships between groups and components form a hierarchy tree, where a top-level group contains other groups, which contain other groups, and so on, until we reach the bottom of the tree, which is composed only of components. Group instances serve three purposes: (1) they help to package sets of components together, e.g., the components for a given discipline; (2) they help create better-organized namespaces (since all components and variables are named based on their ancestors in the tree); and (3) they facilitate the use of hierarchical nonlinear and linear solvers. In our simple example, the obvious choice is to create a group containing Discipline 1 and Discipline 2, because these two form a coupled pair that needs to be converged for any given value of the model input. The hierarchy of groups and components collectively form the *model*.

Children of the *Driver* base class define algorithms that iteratively call the model. For example, a sub-class of *Driver* might implement an optimization algorithm or execute design of experiments (DOE). In the case of an optimization algorithm, the design variables are a subset of the model inputs, and the objective and constraint functions are a subset of the model outputs.

Instances of the *Problem* class perform as a top-level container, holding all other objects. A problem instance contains both the groups and components that constitute the model hierarchy, and also contains a single driver instance. In addition to serving as a container, a problem also provides the user interface for model setup and execution.

Figure 2 illustrates the relationships between instances of the *Component*, *Group*, and *Driver* classes, and introduces the nomenclature for derivatives. The driver repeatedly calls model (i.e., the top-level instance of *Group*, which in turn contains groups that contain other groups that contain the component instances). The derivatives of the model outputs with respect to the model inputs are considered to

be *total* derivatives, while the derivatives of the component outputs with respect to the component inputs are considered to be *partial* derivatives. This is not the only way to define the difference between partial and total derivatives, but this is a definition that suits the present context and is consistent with previous work on the computation of coupled derivatives [72]. In the next section, we provide a brief explanation of how OpenMDAO computes derivatives.

3.4 Derivative computation

As previously mentioned, one of the major advantages of OpenMDAO is that it has the ability to compute total derivatives for complex multidisciplinary models very efficiently via a number of different techniques. Total derivatives are derivatives of model outputs with respect to model inputs. In the example problem from Sec. 3.2, the total derivative needed to minimize the objective function is just the scalar df/dx. Here, we provide a high-level overview of the process for total derivative computation because the way it is done in OpenMDAO is unique among computational modeling frameworks. The mathematical and algorithmic details of total derivative computation are described in Sec. 4.

Total derivatives are difficult and expensive to compute directly, especially in the context of a framework that must deal with user-defined models of various types. As mentioned in the introduction, there are various options for computing derivatives: finite differencing, complex step, algorithmic differentiation, and analytic methods. The finite-difference method can always be used because it just requires re-running the model with a perturbation applied to the input. However, the accuracy of the result depends heavily on the magnitude of the perturbation, and the errors can be large. The complex-step method yields accurate results, but it requires modifications to the model source code to work with complex numbers. The computational cost of these methods scales with the number of input variables, since the model needs to be re-run for a perturbation in each input. OpenMDAO provides an option to use either of these methods, but their use is only recommended when the ease of implementation justifies the increase in computational cost and loss of accuracy.

As described in the introduction, analytic methods have the advantage that they are both efficient and accurate. OpenMDAO facilitates the derivative computation for coupled systems using analytic methods, including the direct and adjoint variants. To use analytic derivative methods in OpenMDAO, the model must be built such that any internal implicit calculations are exposed to the framework. This means that the model must be cast as an implicit function of design variables and implicit variables with associated residuals that must be converged. For explicit calculations, OpenMDAO performs the implicit transformation automatically,

as discussed in Sec. 4.3. When integrating external analysis tools with built-in solvers, this means exposing the residuals and the corresponding state variable vector. Then, the total derivatives are computed in a two-step process: (1) compute the partial derivatives of each component; and (2) solve a linear system of equations that computes the total derivatives. The linear system in Step 2 can be solved in a forward (direct) or a reverse (adjoint) form. As mentioned in the introduction, the cost of the forward method scales linearly with the number of inputs, while the reverse method scales linearly with the number of outputs. Therefore, the choice of which form to use depends on the ratio of the number of outputs to the number of inputs. The details of the linear systems are derived and discussed in Sec. 4. For the purposes of this section, it is sufficient to understand that the total derivatives are computed by solving these linear systems, and that the terms in these linear systems are partial derivatives that need to be provided.

In the context of OpenMDAO, partial derivatives are defined as the derivatives of the outputs of each component with respect to the component inputs. For an *ExplicitComponent*, which is used when outputs can be computed as an analytic function of the inputs, the partial derivatives are the derivatives of these outputs with respect to the component inputs. For an *ImplicitComponent*, which is used when a component provides OpenMDAO with residual equations that need to be solved, the partial derivatives are of these residuals with respect to the component input and output variables. Partial derivatives can be computed much more simply and with lower computational cost than total derivatives. OpenMDAO supports three techniques for computing partial derivatives: full-analytic, semi-analytic, and mixed-analytic.

When using the full-analytic technique, OpenMDAO expects each and every component in the model to provide partial derivatives. These partial derivatives can be computed either by hand differentiation or via algorithmic differentiation. For the example model in Sec. 3.2, the partial derivatives can easily be hand-derived. Discipline 1 is an *ExplicitComponent* defined as $y_1 = y_2^2$ (one input and one output), so we only need the single partial derivative:

$$\frac{\partial y_1}{\partial y_2} = 2y_2. \tag{4}$$

Discipline 2 is an *ImplicitComponent*, so it is defined as a residual that needs to be driven to zero, $R = \exp(-y_1 y_2) - xy_2 = 0$. In this case, we need the partial derivatives of this
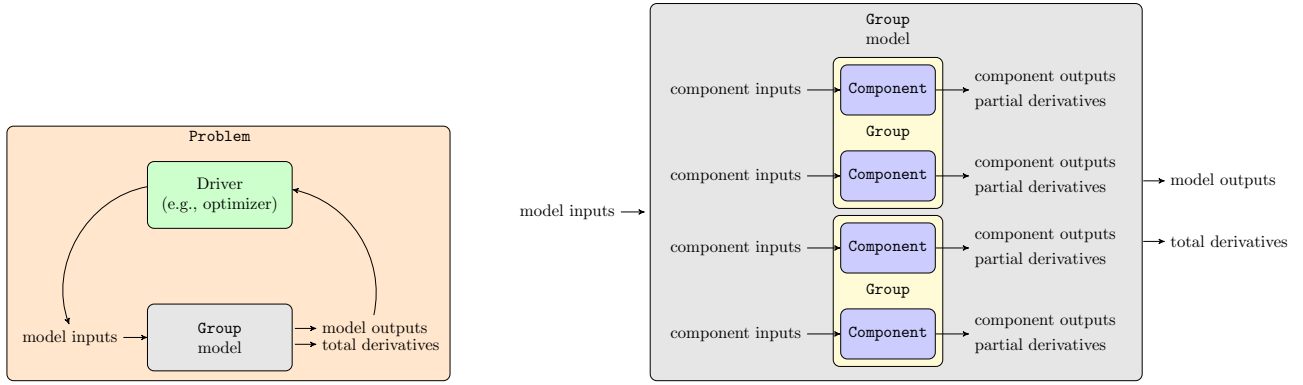
Fig. 2: Relationship between *Driver*, *Group*, and *Component* classes. An instance of *Problem* contains a *Driver* instance, and the *Group* instance named "model". The model instance holds a hierarchy of *Group* and *Component* instances. The derivatives of a model are *total* derivatives, and the derivatives of a component are *partial* derivatives.

residual function with respect to all the variables:

$$\frac{\partial R}{\partial y_1} = -y_2 \exp(-y_1 y_2), \tag{5}$$

$$\frac{\partial R}{\partial y_2} = -y_1 \exp(-y_1 y_2) - x, \tag{6}$$

$$\frac{\partial R}{\partial x} = -y_2. \tag{7}$$

Finally, we also need the partial derivatives of the objective function component:

$$\frac{\partial f}{\partial y_1} = 2y_1, \quad \frac{\partial f}{\partial y_2} = -1. \tag{8}$$

When using the semi-analytic technique, OpenMDAO automatically computes the partial derivatives for each component using either the finite-difference or complex-step methods. This is different from applying these methods to the whole model because it is done component by component, and therefore it does not require the re-convergence of the coupled system. For instances of an *ImplicitComponent*, only partial derivatives of the residual functions are needed (e.g., Eqs. (5), (6), and (7) in the example). Since residual evaluations do not involve any nonlinear solver iterations, approximating their partial derivatives is much less expensive and more accurate. The technique is called "semi-analytic" because while the partial derivatives are computed numerically, the total derivatives are still computed analytically by solving a linear system.

In the mixed-technique, some components provide analytic partial derivatives, while others approximate the partials with finite-difference or complex step methods. The mixed-technique offers great flexibility and is a good option for building models that combine less costly analyses without analytic derivatives and computationally expensive analyses that do provide them. If the complex-step method

is used for some of the partial derivatives, the net result is effectively identical to the fully-analytic method. If finite differences are used to compute some of the partial derivatives, then some accuracy is lost, but overall the net result is still better than either the semi-analytic approach or finite differencing the coupled model to compute the total derivatives.

3.5 Implementation of the Simple Example

We now illustrate the use of the OpenMDAO basic classes by showing the code implementation of the simple model we presented in Sec. 3.2.

The run script is listed in Fig. 3. In *Block 1*, we import several classes from the OpenMDAO API, as well as the components for Discipline 1 and Discipline 2, which we show later in this section. In *Block 2*, we instantiate the four components shown in Fig. 1, as well as a group that combines the two disciplines, called `states_group`. In this group, we connect the output of Discipline 1 to the input of Discipline 2 and vice-versa. Since there is coupling within this group, we also assign a Newton solver to be used when running the model and a direct (LU) solver to be used for the linear solutions required for the Newton iterations and the total derivative computation. For the model output, we define a component "inline", using a convenience class provided by the OpenMDAO standard library. In *Block 3*, we create the top-level group, which we appropriately name as `model`, and we add the relevant subsystems to it and make the necessary connections between inputs and outputs.

In *Block 4*, we specify the model inputs and model outputs, which in this case correspond to the design variable and objective function, respectively, since we are setting up the model to solve an optimization problem. In *Block 5*, we create the problem, assign the model and driver, and run `setup` to signal to OpenMDAO that the problem construc-

```python
import numpy as np
# Block 1: OpenMDAO and component imports
from openmdao.api import Problem, Group, ScipyOptimizeDriver
from openmdao.api import IndepVarComp, ExecComp
from openmdao.api import NewtonSolver, DirectSolver
from disciplines import Discipline1, Discipline2
# Block 2: creation of all the components and groups
#          except the top-level group
input_comp = IndepVarComp('x')
states_group = Group()
states_group.add_subsystem('discipline1_comp', Discipline1())
states_group.add_subsystem('discipline2_comp', Discipline2())
states_group.connect('discipline1_comp.y1', 'discipline2_comp.y1')
states_group.connect('discipline2_comp.y2', 'discipline1_comp.y2')
states_group.nonlinear_solver = NewtonSolver(iprint=0)
states_group.linear_solver = DirectSolver(iprint=0)
output_comp = ExecComp('f=y1**2-y2+3.')
# Block 3: creation of the top-level group
model = Group()
model.add_subsystem('input_comp', input_comp)
model.add_subsystem('states_group', states_group)
model.add_subsystem('output_comp', output_comp)
model.connect('input_comp.x', 'states_group.discipline2_comp.x')
model.connect('states_group.discipline1_comp.y1', 'output_comp.y1')
model.connect('states_group.discipline2_comp.y2', 'output_comp.y2')
# Block 4: specification of the model input (design variable)
#          and model output (objective)
model.add_design_var('input_comp.x')
model.add_objective('output_comp.f')
# Block 5: creation of the problem and setup
prob = Problem()
prob.model = model
prob.driver = ScipyOptimizeDriver()
prob.setup()
# Block 6: set a model input; run the model; and print a model output
prob['input_comp.x'] = 1.
prob.run_model()
print(prob['output_comp.f'])
# Block 7: solve the optimization problem and print the results
prob.run_driver()
print(prob['input_comp.x'], prob['output_comp.f'])
```

Fig. 3: Run script for the simple example. This script depends on a `disciplines` file that defines the components for Disciplines 1 and 2 (see Fig. 4)

tion is complete so it can perform the necessary initialization. In *Block 6*, we illustrate how to set a model input, run the model, and read the value of a model output, and in *Block 7*, we run the optimization algorithm and print the results.

In Fig. 4, we define the actual computations and partial derivatives for the components for the two disciplines. Both classes inherit from OpenMDAO base classes and implement methods in the component API, but they are different because Discipline 1 is explicit while Discipline 2 is implicit. For both, `setup` is where the component declares its inputs and outputs, as well as information about the partial derivatives (e.g., sparsity structure and whether to use finite differences to compute them). In Discipline 1, `compute` maps inputs to outputs, and `compute_partials` is responsible for providing partial derivatives of the outputs with respect to inputs. In Discipline 2, `apply_nonlinear` maps inputs and outputs to residuals, and `linearize` computes the partial derivatives of the residuals with respect to inputs and outputs. More details regarding the API can be found in the documentation on the OpenMDAO website [1].

The component that computes the objective function is built using the inline `ExecComp`. `ExecComp` is a helper class in the OpenMDAO standard library that provides a convenient shortcut for implementing an `ExplicitComponent` for simple and inexpensive calculations. This provides the user a quick mechanism for adding basic calculations like summing values or subtracting quantities. However, `ExecComp` uses the complex-step method to compute the derivatives, so it should not be used for expensive calculations or where there is a large input array.

Figure 5 shows a visualization of the model generated automatically by OpenMDAO. The hierarchy structure of the groups and components is shown on the left, and the dependency graph is shown on the right. This diagram is useful for understanding how data is exchanged between components in the model. Any connections above the diagonal indicate feed-forward data relationships, and connections below the diagonal show feedback relationships that require a nonlinear solver.

---

[1] http://www.openmdao.org/docs

```python
import numpy as np
from openmdao.api import ExplicitComponent, ImplicitComponent
class Discipline1(ExplicitComponent):
    def setup(self):
        self.add_input('y2')
        self.add_output('y1')
        self.declare_partials('y1', 'y2')
    def compute(self, inputs, outputs):
        outputs['y1'] = inputs['y2'] ** 2
    def compute_partials(self, inputs, partials):
        partials['y1', 'y2'] = 2 * inputs['y2']
class Discipline2(ImplicitComponent):
    def setup(self):
        self.add_input('x')
        self.add_input('y1')
        self.add_output('y2')
        self.declare_partials('y2', 'x')
        self.declare_partials('y2', 'y1')
        self.declare_partials('y2', 'y2')
    def apply_nonlinear(self, inputs, outputs, residuals):
        residuals['y2'] = (np.exp(-inputs['y1'] * outputs['y2']) -
                           inputs['x'] * outputs['y2'])
    def linearize(self, inputs, outputs, partials):
        partials['y2', 'x'] = -outputs['y2']
        partials['y2', 'y1'] = (-outputs['y2'] * np.exp(-inputs['y1'] *
                                outputs['y2']))
        partials['y2', 'y2'] = (-inputs['y1'] *
                                np.exp(-inputs['y1'] * outputs['y2']) - inputs['x'])
```

Fig. 4: Definition of the components for Discipline 1 and Discipline 2 for the simple example, including the computation of the partial derivatives.
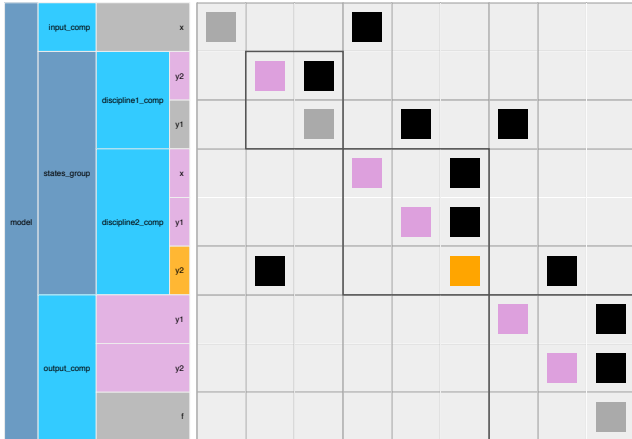

Fig. 5: Visualization of the simple model generated automatically by OpenMDAO. In the hierarchy tree on the left, the darker blue blocks are groups, the lighter blue blocks are components, pink blocks are component inputs, and grey blocks are component outputs.

4 Theory

As previously mentioned, one of the main goals in Open-MDAO is to efficiently compute the total derivatives of the model outputs (f) with respect to model inputs (x), and we stated that we could do this using partial derivatives computed with analytic methods. For models consisting purely of explicit functions, the basic chain rule can be used to achieve this goal. However, when implicit functions are present in the model (i.e., any functions that require iterative nonlinear solvers), the chain rule is not sufficient. In this section, we start by deriving these methods and then explain how they are implemented in OpenMDAO.

4.1 Analytic Methods: Direct and Adjoint

Given a function $F(x,y)$, where x is a vector n inputs, and y is a vector of m variables that depends implicitly on x, then

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} + \frac{\partial F}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}x}, \tag{9}$$

where we distinguish the quantity f from the function F that computes it using lowercase and uppercase, respectively. Using this notation, total derivatives account for the implicit relation between variables, while the partial derivatives are just explicit derivatives of a function [47]. The only derivative in the right-hand side of Eq. (9) that is not partial is $\mathrm{d}y/\mathrm{d}x$, which captures the change in the converged values for y with respect to x. Noting the implicit dependence by $R(x,y) = 0$, we can differentiate it with respect to x to obtain

$$\frac{\mathrm{d}r}{\mathrm{d}x} = \frac{\partial R}{\partial x} + \frac{\partial R}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}x} = 0. \tag{10}$$

Re-arranging this equation, we get the linear system

$$\underbrace{\left[\frac{\partial R}{\partial y}\right]}_{m\times m} \underbrace{\frac{\mathrm{d}y}{\mathrm{d}x}}_{m\times n} = -\underbrace{\left[\frac{\partial R}{\partial x}\right]}_{m\times n}. \tag{11}$$

Now dy/dx can be computed by solving this linear system, which is constructed using only partial derivatives. This linear system needs to be solved n times, once for each component of x, with the column of $\partial R/\partial x$ that corresponds to the element of x as the right-hand side. Then, dy/dx can be used in Eq. (9) to compute the total derivatives. This approach is known as the direct method.

There is another way to compute the total derivatives based on these equations. If we substitute the linear system (11) into the total derivative equation (9), we obtain

$$\frac{df}{dx} = \frac{\partial F}{\partial x} - \overbrace{\underbrace{\frac{\partial F}{\partial y}}^{1 \times m} \overbrace{\left[\frac{\partial R}{\partial y}\right]^{-1}}^{m \times m}}_{\psi^T} \left[\frac{\partial R}{\partial x}\right]. \tag{12}$$

By grouping the terms $[\partial R/\partial y]^{-1}$ and $\partial F/\partial y$, we get an m-vector, ψ, which is the adjoint vector. Instead of solving for dy/dx with Eq. (11) (the direct method), we can instead solve a linear system with $[\partial F/\partial y]^T$ as the right-hand side to compute ψ:

$$\underbrace{\left[\frac{\partial R}{\partial y}\right]^T}_{m \times m} \underbrace{\psi}_{m \times 1} = \underbrace{\left[\frac{\partial F}{\partial y}\right]^T}_{m \times 1}. \tag{13}$$

This linear system needs to be solved once for each function of interest f. If f is a vector variable, then the right-hand side for each solution is the corresponding row of $\partial F/\partial y$. The transpose of the adjoint vector, ψ^T, can then be used to compute the total derivative,

$$\frac{df}{dx} = \frac{\partial F}{\partial x} - \psi^T \frac{\partial R}{\partial x}. \tag{14}$$

This is the *adjoint method*, and the derivation above shows why the computational cost of this method is proportional to the number of outputs and independent of the number of inputs. Therefore, if the number of inputs exceeds the number of outputs, the adjoint method is advantageous, while if the opposite is true, then the direct method has the advantage. The main idea of these analytic methods is to compute total derivatives (which account for the solution of the models) using only partial derivatives (which do not require the solution of the models).

As mentioned in Sec. 2, these analytic methods have been extended to MDO applications [90, 72, 67]. All of these methods have been used in MDO applications, but as was discussed in Sec. 2, the implementations tend to be highly application specific and not easily integrated into an MDO framework.

To overcome the challenge of application-specific derivative computations, Hwang and Martins [47] developed the *modular analysis and unified derivatives* (MAUD) architecture, which provides the mathematical and algorithmic framework to combine the chain rule, direct, and adjoint methods into a single implementation that works even when using models that utilize distributed memory parallelism, such as computational fluid dynamics (CFD) and finite element analysis (FEA) codes.

4.2 Nonlinear Problem Formulation

OpenMDAO V1 and V2 were designed based on the algorithms and data structures of MAUD, but V2 includes several additions to the theory and algorithms to enable more efficient execution for serial models. In this section, we summarize the key MAUD concepts and present the new additions in OpenMDAO V2 that make the framework more efficient for serial models. The core idea of MAUD is to formulate any model (including multidisciplinary models) as a single nonlinear system of equations. This means that we concatenate all variables—model inputs and outputs, and both explicit and implicit component variables—into a single vector of unknowns, u. Thus, in all problems, we can represent the model as $R(u) = 0$, where R is a residual function defined in such a way that this system is equivalent to the original model.

For the simple example from Sec. 3.2, our vector of unknowns would be $u = (x, y_1, y_2, f)$, and the correct residual function is

$$R(u) = \begin{bmatrix} r_x \\ r_{y_1} \\ r_{y_2} \\ r_f \end{bmatrix} = \begin{bmatrix} x - x^* \\ y_1 - y_2^2 \\ \exp(-y_1 y_2) - x y_2 \\ f - (y_1^2 - y_2 + 3) \end{bmatrix} = 0. \tag{15}$$

Although the variable x is not an "unknown" (it has a value that is set explicitly), we reformulate it into an implicit form by treating it as an unknown and adding a residual that forces it to the expected value of x^*. Using this approach, any computational model can be written as a nonlinear system of equations such that the solution of the system yields the same outputs and intermediate values as running the original computational model.

Users do not actually need to re-formulate their problems in this fully implicit form because OpenMDAO handles the translation automatically via the *ExplicitComponent* class, as shown in the code snippet in Fig. 4. However, the framework does rely on the fully implicit formulation for its internal representation.

The key benefit of representing the whole model as a single monolithic nonlinear system is that we can use the *unified derivatives equation* [67, 47], which generalizes all analytic derivative methods. The unified derivatives equation can be written as

$$\left[\frac{\partial R}{\partial u}\right] \left[\frac{du}{dr}\right] = \mathcal{I} = \left[\frac{\partial R}{\partial u}\right]^T \left[\frac{du}{dr}\right]^T, \tag{16}$$

where u is a vector containing inputs, implicitly defined variables, and outputs, and R represents the corresponding residual functions. The matrix $\mathrm{d}u/\mathrm{d}r$ contains a block with the total derivatives that we ultimately want (i.e., the derivatives of the model outputs with respect to the inputs, $\mathrm{d}f/\mathrm{d}x$). Again, we use lowercase and uppercase to distinguish between quantities and functions, as well as the convention for total and partial derivatives introduced earlier. For the simple example in Sec. 3.2 the total derivative matrix is

$$
\left[\frac{\mathrm{d}u}{\mathrm{d}r}\right] = \begin{bmatrix} \frac{\mathrm{d}x}{\mathrm{d}r_x} & \frac{\mathrm{d}x}{\mathrm{d}r_{y_1}} & \frac{\mathrm{d}x}{\mathrm{d}r_{y_2}} & \frac{\mathrm{d}x}{\mathrm{d}r_f} \\ \frac{\mathrm{d}y_1}{\mathrm{d}r_x} & \frac{\mathrm{d}y_1}{\mathrm{d}r_{y_1}} & \frac{\mathrm{d}y_1}{\mathrm{d}r_{y_2}} & \frac{\mathrm{d}y_1}{\mathrm{d}r_f} \\ \frac{\mathrm{d}y_2}{\mathrm{d}r_x} & \frac{\mathrm{d}y_2}{\mathrm{d}r_{y_1}} & \frac{\mathrm{d}y_2}{\mathrm{d}r_{y_2}} & \frac{\mathrm{d}y_2}{\mathrm{d}r_f} \\ \frac{\mathrm{d}f}{\mathrm{d}r_x} & \frac{\mathrm{d}f}{\mathrm{d}r_{y_1}} & \frac{\mathrm{d}f}{\mathrm{d}r_{y_2}} & \frac{\mathrm{d}f}{\mathrm{d}r_f} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{\mathrm{d}y_1}{\mathrm{d}x} & \frac{\mathrm{d}y_1}{\mathrm{d}r_{y_1}} & \frac{\mathrm{d}y_1}{\mathrm{d}r_{y_2}} & 0 \\ \frac{\mathrm{d}y_2}{\mathrm{d}x} & \frac{\mathrm{d}y_2}{\mathrm{d}r_{y_1}} & \frac{\mathrm{d}y_2}{\mathrm{d}r_{y_2}} & 0 \\ \frac{\mathrm{d}f}{\mathrm{d}x} & \frac{\mathrm{d}f}{\mathrm{d}r_{y_1}} & \frac{\mathrm{d}f}{\mathrm{d}r_{y_2}} & 1 \end{bmatrix}, \quad (17)
$$

where the middle term shows the expanded total derivative matrix and the right-most term simplifies these derivatives. The middle term is obtained by inserting $u = [x, y_1, y_2, f]^T$ and $r = [r_x, r_{y_1}, r_{y_2}, r_f]^T$. The simplification in the right-most term is possible because from Eq. (15), we know that for example,

$$
\left[\frac{\mathrm{d}x}{\mathrm{d}r_x}\right] = 1, \quad \left[\frac{\mathrm{d}x}{\mathrm{d}r_y}\right] = 0.
$$

In this example, the left equality of the unified derivatives equation (16) corresponds to the forward form, which is equivalent to the direct method, while the right equality corresponds to the reverse form, which is equivalent to the adjoint method. Solving the forward form solves for one column of the total derivative matrix at a time, while the reverse mode solves for one row at a time. Thus, the forward mode requires a linear solution for each input, while the the reverse mode requires a linear solution for each output.

Although the full matrix $\mathrm{d}u/\mathrm{d}r$ is shown in Eq. (17), we do not actually need to solve for the whole matrix; the optimizer only needs the derivatives of the model outputs with respect to model inputs. The needed derivatives are computed by solving for the appropriate columns (forward mode) or rows (reverse mode) one at a time using OpenMDAO's linear solver functionality.

4.3 API for Group and Component Classes

To recast the entire model as a single nonlinear system, the *Component* and *Group* classes both define the following five API methods:

- *apply_nonlinear(p, u, r)*: Compute the residuals (r) given the inputs (p) and outputs (u) of the component or group.
- *solve_nonlinear(p, u)*: Compute the converged values for the outputs given the inputs.

- *linearize(p, u)*: Perform any one-time linearization operations, e.g., computing partial derivatives of the residuals or approximating them via finite differences.
- *apply_linear($\mathrm{d}u, \mathrm{d}r$)*: Compute a Jacobian-vector product, and place the result in the storage vector. For the forward mode this product is

$$
\mathrm{d}r = \left[\frac{\partial R}{\partial u}\right] \mathrm{d}u, \tag{18}
$$

and for the reverse mode, it is

$$
\mathrm{d}u = \left[\frac{\partial R}{\partial u}\right]^T \mathrm{d}r. \tag{19}
$$

- *solve_linear($\mathrm{d}u, \mathrm{d}r$)*: Multiply the inverse of the Jacobian with the provided right-hand side vector (or solve a linear system to compute the product without explicitly computing the inverse), and place the result in the storage vector. For the forward mode,

$$
\mathrm{d}u = \left[\frac{\partial R}{\partial u}\right]^{-1} \mathrm{d}r, \tag{20}
$$

while for the reverse mode,

$$
\mathrm{d}r = \left(\left[\frac{\partial R}{\partial u}\right]^T\right)^{-1} \mathrm{d}u. \tag{21}
$$

Up to this point we have commonly referred to the unknown vector (u) and the residual vector (r), but the API methods above introduce several new vectors that have not been previously discussed. The input vector (p) contains values for any variables that are constant relative to a given location in the model hierarchy. For any given component, the set of inputs are clear. For a group, the inputs are composed of the set of any variables that do not have an associated output owned by one of the children of that group. For example, referring back to Fig. 5, the *states_group* has the variable x in its input vector, and the *output_group* has the variables *y1* and *y2* in its input vector. At the top level of that model, the input vector is empty, and all variables are contained within the unknown vector. There are also the $\mathrm{d}u$ and $\mathrm{d}r$ vectors, which are used to contain the source and product for matrix-vector-product methods. For a detailed description of the relationship between the p and u vectors, and how the API methods as well as the $\mathrm{d}u$ and $\mathrm{d}r$ vectors enable efficient solution of the unified derivatives equations, see the original MAUD paper [47].

Both *Group* and *Component* classes must implement these five methods, but default implementations are provided in many cases. All five methods are implemented in the *Group* class in OpenMDAO's standard library, which is used to construct the model hierarchy. For subclasses of *ExplicitComponent*, such as *Discipline1* in Fig. 4, the user does not

directly implement any of the five basic API methods. Instead, the user implements the `compute` and `compute_partials` methods that the *ExplicitComponent* base class uses to implement the necessary lower level methods, as shown in Algorithm (1). The negative sign in line 8 of Algorithm (1) indicates that the partial derivatives for the implicit transformation are the negative of the partial derivatives for the original explicit function. As shown in Eq. (15), the implicit transformation for the explicit output f is given by

$$r_f = f - (y_1^2 - y_2 + 3) , \qquad (22)$$

which explains the negative sign.

Algorithm 1 *ExplicitComponent* API

1: **function** apply_nonlinear(p, u, r)
2:     $r \leftarrow u - \text{compute}(p)$
3:     **return** r
4: **function** solve_nonlinear(u)
5:     $u \leftarrow \text{compute}(p)$
6:     **return** u
7: **function** linearize(p)
8:     $\left[\frac{\partial R}{\partial u} \right] \leftarrow -\text{compute\_partials}(p)$
9:     **return** $\left[\frac{\partial R}{\partial u} \right]$

For subclasses of *ImplicitComponent*, such as `Discipline2` in Fig. 4, only `apply_nonlinear` is strictly required, and `solve_nonlinear` is optional. (The base class implements a method that does not perform any operations.) For many models, such as the example in Fig. 3, it is sufficient to rely on one of the nonlinear solvers in OpenMDAO's standard library to converge the implicit portions of a model. Alternatively, a component that wraps a complex discipline analysis can use `solve_nonlinear` to call the specialized nonlinear solver built into that analysis code.

In the following section, we discuss the practical matter of using the API methods to accomplish the nonlinear and linear solutions required to execute OpenMDAO models. In both the nonlinear and linear cases, there are two strategies employed, depending on the details of the underlying model being worked with: monolithic and hierarchical. While in our discussion we recommend using each strategy for certain types of models, in actual models, the choice does not need to be purely one or the other. Different strategies can be employed at different levels of the model hierarchy to match the particular needs of any specific model.

In addition, it is important to note that the usage of one strategy for the nonlinear solution does not prescribe that same strategy for the linear solution. In fact, it is often the case that a model using the hierarchical nonlinear strategy would also use the monolithic linear strategy. The converse is also true: Models that use the monolithic nonlinear strategy will often use the hierarchical linear strategy. This asymmetry of nonlinear and linear solution strategies is one of the

central features in OpenMDAO that enables the framework to work efficiently with a range or models that have vastly different structures and computational needs.

5 Monolithic and Hierarchical Solution Strategies

OpenMDAO uses a hierarchical arrangement of groups and components to organize models, define execution order, and control data passing. This hierarchical structure can also be used to define nonlinear and linear solver hierarchies for models. While in some cases it is better to match the solver hierarchy closely to that of the model structure, in most cases, better performance is achieved when the solver structure is more monolithic than the associated model. The framework provides options for both nonlinear and linear solvers, and allows the user to mix them at the various levels of the model hierarchy to customize the solver strategy for any given model.

The hierarchical model structure and solver structure used in OpenMDAO were first proposed as part of the MAUD architecture [47]. In addition, MAUD also included several algorithms that implement monolithic and hierarchical solvers in the model hierarchy that OpenMDAO also adopted: monolithic Newton's method, along with hierarchical versions of nonlinear block Gauss–Seidel, nonlinear block Jacobi, linear block Gauss–Seidel, and linear block Jacobi. In addition to these solvers, OpenMDAO V2 implements a new hierarchical nonlinear solver that improves performance for very tightly coupled models (e.g., hierarchical Newton's method). It also includes a monolithic linear solver strategy that enables much greater efficiency for serial models.

This section describes the new contributions in OpenMDAO, along with a summary of the relevant strategies and solver algorithms adopted from the MAUD architecture.

5.1 Nonlinear Solution Strategy

Although the user may still implement any explicit analyses in the traditional form using *ExplicitComponent*, OpenMDAO internally transforms all models into the implicit form defined by MAUD, i.e., $R(u) = 0$. For the simple example problem from Sec. 3.2, this transformation is given by Eq. (15). While the transformation is what makes it possible to leverage the unified derivatives equation to compute total derivatives, it also yields a much larger implicit system that now represents the complete multidisciplinary model including all intermediate variables. The larger system is more challenging to converge, and may not be solvable in monolithic form. OpenMDAO provides a hierarchical nonlinear strategy that allows individual subsystems in the model to be solved first, which makes the overall problem more tractable. The hierarchical nonlinear strategy rep-

resents a trade-off between solution robustness and solution efficiency because it is typically more robust and more expensive.

5.1.1 Monolithic Nonlinear Strategy

In some cases, treating the entire model as a single monolithic block provides a simple and efficient solution strategy. This is accomplished with a pure Newton's method that iteratively applies updates to the full u vector until the residual vector is sufficiently close to zero, via

$$\left[\frac{\partial R}{\partial u}\right] \Delta u = -r. \tag{23}$$

In practice, pure Newton's method is usually used together with a globalization technique, such as a line search, to improve robustness for a range of initial guesses. OpenMDAO's Newton solver uses these methods in its actual implementation. For simplicity, we omit the globalization techniques from the following descriptions. Since these techniques do not change the fundamentals of Newton's method, we can do this without loss of generality.

Algorithm 2 shows the pseudocode a pure Newton's method implemented using the OpenMDAO API. All variables are treated as implicit and updated in line 4, which uses `solve_linear` to implement Eq. (23). Note that `solve_nonlinear` is never called anywhere in Algorithm 2; only `apply_nonlinear` is called to compute the residual vector, r. This means that no variables—not even outputs of an *ExplicitComponent*—have their values directly set by their respective components. When the pure Newton's method works, as is the case for the *states_group* in the example model shown in Fig. 5, it is a highly efficient algorithm for solving a nonlinear system. The challenge with pure Newton's method is that even with added globalization techniques, it still may not converge for especially complex models with large numbers of states. Pure Newton's method is particularly challenging to apply to large multidisciplinary models built from components that wrap disciplinary analyses with their own highly customized nonlinear solver algorithms. This is because some specialized disciplinary solvers include customized globalization schemes (e.g., pseudo time continuation) and linear solver preconditioners that a pure Newton's method applied at the top level of the model cannot directly take advantage of.

5.1.2 Hierarchical Nonlinear Strategy

For some models, the monolithic nonlinear strategy may be numerically unstable and fail to converge on a solution. In those cases, the hierarchical strategy may provide more robust solver behavior. Consider that each level of the model hierarchy, from the top level *model* group all the way down

---

Algorithm 2 Pure Newton's Method

1: $r \leftarrow \text{apply\_nonlinear}(p, u, r)$
2: **while** $||r|| > \varepsilon$ **do**
3: $\quad \left[\frac{\partial R}{\partial u}\right] \leftarrow \text{linearize}(p, u)$
4: $\quad \Delta u \leftarrow \text{solve\_linear}(-r)$
5: $\quad u \leftarrow u + \Delta u$
6: $\quad r \leftarrow \text{apply\_nonlinear}(p, u, r)$

---

to the individual components, contains a subset of the unknowns vector, u_{child}, and the corresponding residual equations, $R_{\text{child}}(u_{\text{child}}) = 0$. For any level of the hierarchy, a given subsystem (which can be a component or group of components) is a self-contained nonlinear system, where any variables from external components or groups are inputs that are held constant for that subsystem's `solve_nonlinear`. Therefore, we can apply a nonlinear solver to any subsystem in the hierarchy to converge that specific subset of the nonlinear model. The hierarchical nonlinear strategy takes advantage of this subsystem property to enable more robust top level solvers.

OpenMDAO implements a number of nonlinear solution algorithms that employ a hierarchical strategy. The most basic two algorithms are the nonlinear block-Gauss–Seidel and nonlinear block-Jacobi algorithms used by Hwang and Martins [47]. Both of these algorithms use simple iterative strategies that repetitively call `solve_nonlinear` for all the child subsystems in sequence, until the residuals are sufficiently converged.

OpenMDAO V2 introduces a new hierarchical Newton's method solver that extends the use of this strategy to multidisciplinary models composed of a set of more tightly coupled subsystems. Compared to the pure Newton's method of Algorithm (2), the hierarchical Newton algorithm adds an additional step that recursively calls `solve_nonlinear` on all child subsystems of the parent system, as shown in Algorithm 3.

Algorithm 3 Hierarchical Newton's Methods

1: **for all** child in subsystems **do**
2: $\quad u_{\text{child}} \leftarrow \text{child.solve\_nonlinear}(p_{\text{child}}, u_{\text{child}})$
3: $r \leftarrow \text{apply\_nonlinear}(p, u, r)$
4: **while** $||r|| > \varepsilon$ **do**
5: $\quad \left[\frac{\partial R}{\partial u}\right] \leftarrow \text{linearize}(p, u)$
6: $\quad \Delta u \leftarrow \text{solve\_linear}(-r)$
7: $\quad u \leftarrow u + \Delta u$
8: $\quad$ **for all** child in subsystems **do**
9: $\quad\quad u_{\text{child}} \leftarrow \text{child.solve\_nonlinear}(p_{\text{child}}, u_{\text{child}})$
10: $\quad r \leftarrow \text{apply\_nonlinear}(p, u, r)$

We refer to Algorithm 3 as the *hierarchical Newton's method*, because although each child subsystem solves for its own unknowns (u_{child}), the parent groups are responsible

for those same unknowns as well. Since each level of the hierarchy sees the set of residuals from all of its children, the size of the Newton system (the number of state variables it is converging) increases as one moves higher up the hierarchy, making it increasingly challenging to converge. The recursive solution of subsystems acts as a form of nonlinear preconditioning or globalization to help stabilize the solver, but fundamentally, the top level Newton solver is dealing with the complete set of all residual equations from the entire model.

There is another, arguably more common, formulation for applying Newton's method to nested models where the solver at any level of the model hierarchy sees only the subset of the implicit variables that it alone is responsible for. In this formulation, the Newton system at any level is much smaller because it does not inherit the states and residuals from any child systems. Instead, it treats any child calculations as if they were purely explicit. We refer to this formulation as the "reduced-space Newton's method". In Appendix B, we prove that the application of the hierarchical Newton's method yields the exact same solution path as that of a reduced-space Newton's method. The proof demonstrates that exact recursive solutions for u_{child} (i.e., $R_{\text{child}}(u_{\text{child}}) = 0$) (lines 1, 2, 8, and 9 in Algorithm 3) reduce the search space for the parent solver to only the subset of the u vector that is owned exclusively by the current system and not by any of the solvers from its children.

While perfect sub-convergence is necessary to satisfy the conditions of the proof, in practice, it is not necessary to fully converge the child subsystems for every top level hierarchical Newton iteration. Once the nonlinear system has reached a sufficient level of convergence, the recursion can be turned off, reverting the solver to the more efficient monolithic strategy.

A hybrid strategy that switches between monolithic and hierarchical strategies was investigated by Chauhan et al. [15] in a study where they found that the best performing nonlinear solver algorithm changes with the strength of the multidisciplinary coupling. Their results underscore the need for OpenMDAO to support both hierarchical and monolithic nonlinear solver architectures, because they show that different problems require different treatments. The mixture of the two often yields the best compromise between stability and performance.

In addition to the hierarchical Newton's method solver, OpenMDAO also provides a gradient-free hierarchical Broyden solver that may offer faster convergence than the nonlinear Gauss–Seidel or nonlinear Jacobi solvers. In Appendix C, we also prove that the Broyden solver exhibits the same equivalence between the hierarchical and reduced-space formulations.

5.2 Linear Solution Strategy

As discussed above, some nonlinear solvers require their own linear solvers to compute updates for each iteration. OpenMDAO also uses a linear solver to compute total derivatives via Eq. (16). The inclusion of linear solvers in the framework, and the manner in which they can be combined, is one of the unique features of OpenMDAO.

There are two API methods that are useful for implementing linear solvers: `apply_linear` and `solve_linear`. In an analogous fashion to the nonlinear solvers, the linear solvers can employ either a monolithic or hierarchical strategy. In this context, a monolithic strategy is one that works with the entire partial derivatives Jacobian ($\partial R/\partial u$) as a single block in-memory. A hierarchical linear strategy is one that leverages a matrix-free approach.

5.2.1 Hierarchical Linear Strategy

The hierarchical linear solver strategy is an approach that relies on the use of the `apply_linear` and `solve_linear` methods in the OpenMDAO API. As such, it is a matrix-free strategy. This strategy was originally proposed by Hwang and Martins [47], and we refer the reader to that work for a more detailed presentation of these concepts, including an extensive treatment of how parallel data passing is integrated into this linear strategy. OpenMDAO implements the hierarchical linear solver strategy proposed by MAUD to support integration with computationally expensive analyses, i.e., parallel distributed memory analyses such as CFD and FEA. Models that benefit from this strategy tend to have fewer than ten components that are computationally expensive, with at least one component having on the order of a hundred thousand unknowns. The linear block-Gauss–Seidel and linear block-Jacobi solvers are the two solvers in the OpenMDAO standard library that use the hierarchical strategy. Algorithms (4) and (5) detail the forward (direct) formulation of the two hierarchical linear solvers. There are also separate reverse (adjoint) formulations for these solvers, which are explained in more detail by Hwang and Martins [47]. For integration with PDE solvers, the forward and reverse forms of these algorithms allow OpenMDAO to leverage existing, highly specialized linear solvers used by discipline analyses as part of a recursive preconditioner for a top-level OpenMDAO Krylov subspace solver in a coupled multidisciplinary model [47].

Algorithm 4 Linear Block Gauss–Seidel (forward mode)

1: **while** $||r|| > \varepsilon$ **do**
2:     **for all** child$_i$ in subsystems **do**
3:         **for all** child$_j$ in subsystems : $i \neq j$ **do**
4:             $\mathrm{d}r_i \leftarrow \mathrm{d}r_i - \text{child}_j.\text{apply\_linear}(\mathrm{d}u_{\text{child } j})$
5:         $\mathrm{d}u_{\text{child}_i} \leftarrow \text{child}_i.\text{solve\_linear}(\mathrm{d}r_i)$

Algorithm 5 Linear Block Jacobi (forward mode)

1: **while** $||r|| > \varepsilon$ **do**
2:     **for all** child_i in subsystems **do**
3:         $\text{d}u_{\text{child}_i} \leftarrow \text{child}_i.\text{solve\_linear}(\text{d}r_i)$
4:     **for all** child_i in subsystems **do**
5:         **for all** child_j in subsystems $: i \neq j$ **do**
6:             $\text{d}r_i \leftarrow \text{d}r_i - \text{child}_j.\text{apply\_linear}(\text{d}u_{\text{child}_j})$

5.2.2 Monolithic Linear Strategy

Although the hierarchical linear solver strategy is an efficient approach for models composed of computationally expensive analyses, it can introduce significant overhead for models composed of hundreds or thousands of computationally simple components. The hierarchical linear solver strategy relies on the use of the `apply_linear` and `solve_linear` methods, which only provide linear operators that must be recursively called on the entire model hierarchy. While recursion is generally expensive in and of itself, the cost is exacerbated because OpenMDAO is written in Python, an interpreted language where loops are especially costly. For many models, it is feasible to assemble the entire partial derivative Jacobian matrix in memory, which then allows the use of a direct factorization to solve the linear system more efficiently. As long as the cost of computing the factorization is reasonable, this approach is by far the simplest and most efficient way to implement the `solve_linear` method. This represents a significant extension from the previously developed hierarchical formulation [47], and as we will show in Sec. 5.3, this approach is crucial for good computational performance on models with many components.

The matrix assembly can be done using either a dense matrix or a sparse matrix. In the sparse case, OpenMDAO relies on the components to declare the nonzero partial derivatives, as shown in Fig. 4. Broadly speaking, at the model level, the partial derivative Jacobian is almost always very sparse, even for simple models. Figure 5, which includes a visualization of $\left[\partial R / \partial u\right]^T$, shows that even a small model has a very sparse partial derivative Jacobian. In the vast majority of cases, the factorization is more efficient when using a sparse matrix assembly.

The monolithic linear solver strategy is primarily designed to be used with a direct linear solver. A direct factorization is often the fastest, and certainly the simplest type of linear solver to apply. However, this strategy can also be used with a Krylov subspace solver, assuming we either do not need to use a preconditioner or want to use a preconditioner that is also compatible with the monolithic strategy (e.g., incomplete LU factorization). Krylov subspace solvers are unique because they can be used with both the hierarchical and monolithic linear solver strategies, depending on what type of preconditioner is applied.

Monolithic and hierarchical linear solver strategies can be used in conjunction with each other as part of a larger model. At any level of the model hierarchy, a monolithic strategy can be used, which causes all components below that level to store their partial derivatives in the assembled Jacobian matrix. Above that level, however, a hierarchical linear solver strategy can still be used. This mixed linear solver strategy is crucial for achieving good computational efficiency for larger models. Aeropropulsive design optimization is good example where this is necessary. Gray et al. [36] coupled a RANS CFD analysis to a 1-D propulsion model using OpenMDAO with a hierarchical linear solver strategy to combine the matrix-free Krylov subspace from the CFD with the monolithic direct solver used for the propulsion analysis.

5.3 Performance Study for Mixed Linear Solver Strategy

The specific combination of hierarchical and monolithic linear solvers that will give the best performance is very model-specific, which is why OpenMDAO's flexibility to allow different combinations is valuable.

This sensitivity of computational performance to solver strategy can be easily demonstrated using an example model built using the OpenAeroStruct [53] library. OpenAeroStruct is a modular, lower-fidelity, coupled aerostructural modeling tool which is built on top of OpenMDAO V2. Consider a notional model that computes the average drag coefficient for a set of aerostructural wing models at different angles of attack, as shown in Fig. 6. The derivatives of average drag with respect to the shape design variables can be computed via a single reverse model linear solution. This reverse mode solution was tested with two separate solver strategies: (1) pure monolithic with a direct solver at the top of the model hierarchy; (2) mixed hierarchical/monolithic with a linear block Gauss–Seidel solver at the top and direct solver angle of attack case. Figure 7 compares the computational costs of these two linear solver strategies and examines how the computational cost scales with increasing number of components. For this problem, the scaling is achieved by increasing the number of angle of attack conditions included in the average. As the number of angle of attack cases increases, the number of components and number of variables goes up as well, since each case requires its own aerostructural analysis group. The data in Fig. 7 shows that both linear solver strategies scale nearly linearly with an increasing number of variables, which indicates very good scaling for the direct solver. This solver relies on the sparse LU factorization in SciPy [80]. However, the difference between the purely monolithic and the mixed hierarchical/monolithic linear solver strategies is 1.5 to 2 orders of magnitude in total computation time. This difference in computational cost is roughly independent of problem size, which demonstrates

that the mixed strategy is fundamentally more efficient than the pure monolithic one.

6 Efficient Methods for Computing Total Derivatives of Sparse Models

As discussed in Sec. 4, when using the unified derivative equation to analytically compute the total derivative Jacobian, one linear solution is required for each model input in forward mode; alternatively, in reverse mode, one linear solution is required for each model output. However, when the model exhibits certain types of sparsity patterns, it is possible to compute the complete total derivative Jacobian matrix using fewer linear solutions than the number of model inputs or model outputs. Furthermore, when properly exploited,

(a) Pure monolithic

(b) Mixed hierarchical and monolithic
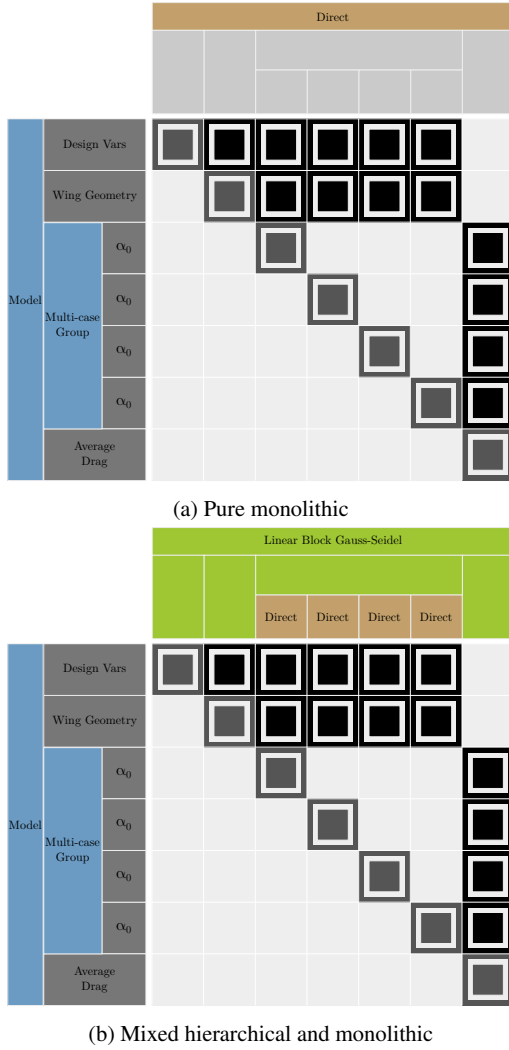
Fig. 6: Model hierarchy, data connectivity, and linear solver strategy for an OpenAeroStruct model with four flight conditions.
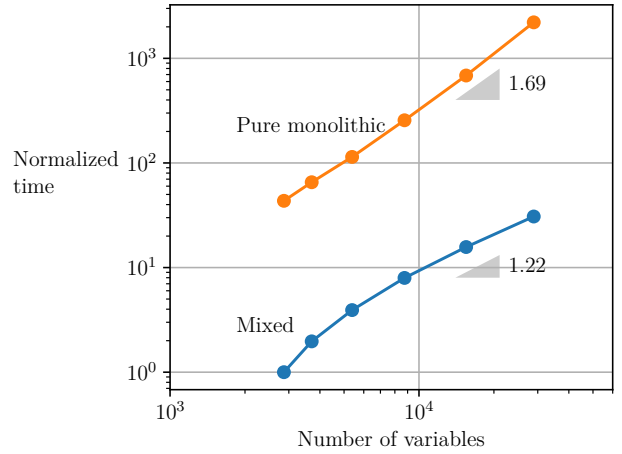
Fig. 7: Scaling of computational cost for a single linear solution versus number of components in the model. Two solver strategies are compared: pure monolithic (orange) and mixed (blue). The average slope of the two data sets indicates nearly linear scaling for both, but there is a two-order of magnitude difference in actual cost.

these sparsity structures can also change the preferred solution mode (forward or reverse) from what would normally be expected. In this section, we present two specific types of sparsity structure and discuss how OpenMDAO exploits them to reduce the computational cost of computing derivatives for models where they are present. The first sparsity pattern arises from separable model variables and allows for the multiple linear solutions to be computed simultaneously using a single right-hand-side vector; thus, it requires only one linear solution. The second sparsity pattern arises from models where one component output provides the input to many downstream components that do not depend on each other. This enables multiple linear solutions to be computed in parallel via multiple parallel linear solutions on multiple right-hand sides.

6.1 Combining Multiple Linear Solutions Using Graph Coloring

Typically, a single linear solution corresponds to a single model input (forward/direct mode) or a single model output (reverse/adjoint mode). When a portion of the total derivative Jacobian has a block-diagonal structure, a coloring technique can be applied to combine linear solutions for multiple model inputs or multiple model outputs into a single linear solution. This reduces the cost for computing the total derivative Jacobian. Consider a notional problem with seven model inputs $(a, b, c_0, c_1, c_2, c_3, c_4)$ and six model outputs $(g_0, g_1, g_2, g_3, g_4, f)$ with the total derivative Jacobian structure illustrated in Fig. 8. Since there are more model

inputs than outputs, the reverse mode would normally be faster.

(a) Seven individual solutions

(b) Two individual solutions and one simultaneous solution
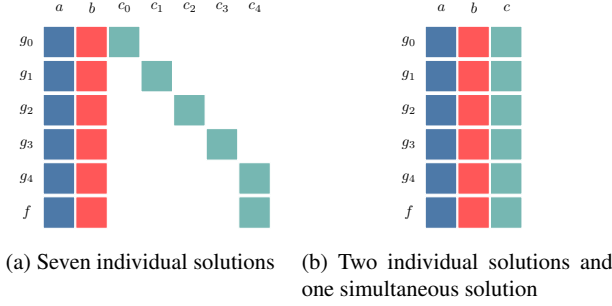
Fig. 8: Total derivative Jacobian structure for a notional model where the model outputs are all separable with respect to the c_i inputs. The left side shows the full Jacobian structure with coloring to indicate the potential for simultaneous solutions. The right side shows the collapsed Jacobian structure that takes advantage of the coloring.

However, as we can infer from Fig. 8, the five c_i inputs affect the six outputs $(g_0 - g_4, f)$ independently. Therefore, these inputs can share a single color, and the linear system only requires a single forward solution. Combining multiple linear solutions is accomplished by combining multiple columns of the identity matrix from Eq. (16) into a single right-hand side, as shown in Fig. 9. Normally, a combined right-hand side would yield sums of total derivatives, but if we know that certain terms are guaranteed to be zero (e.g., $dg_0/dc_1 = 0$, $dg_1/dc_2 = 0$), we can safely combine the right-hand side vectors.

In this case, using forward mode would require two separate linear solutions for a and b, and then a single additional combined linear solution for the set $(c_0, \ldots, c_4)$, as illustrated in Fig. 8(a) and Fig. 9(b). Since this case is colored in forward mode, the subscripts (i, j, k) in Fig. 9(b) are associated with the denominator of the total derivatives, indicating that each solution yields derivatives of all the outputs with respect to a single input. In Fig. 9(a), $\left[du/dr_{i|j|k} \right]$ indicates the need to use three separate linear solutions. In Fig. 9(b), $\left[du/dr_{i+j+k} \right]$ indicates that the three right-hand sides can be added as

$$\left[\frac{du}{dr_{i+j+k}} \right] = \left[\frac{du}{dr_i} \right] + \left[\frac{du}{dr_j} \right] + \left[\frac{du}{dr_k} \right], \tag{24}$$

to form a single linear system that can compute all three sets of total derivatives.

Using coloring, the complete total derivative Jacobian can be constructed using only three linear solutions in a colored forward mode. The original uncolored solution method

(a) Basic forward mode

(b) Simultaneous forward mode
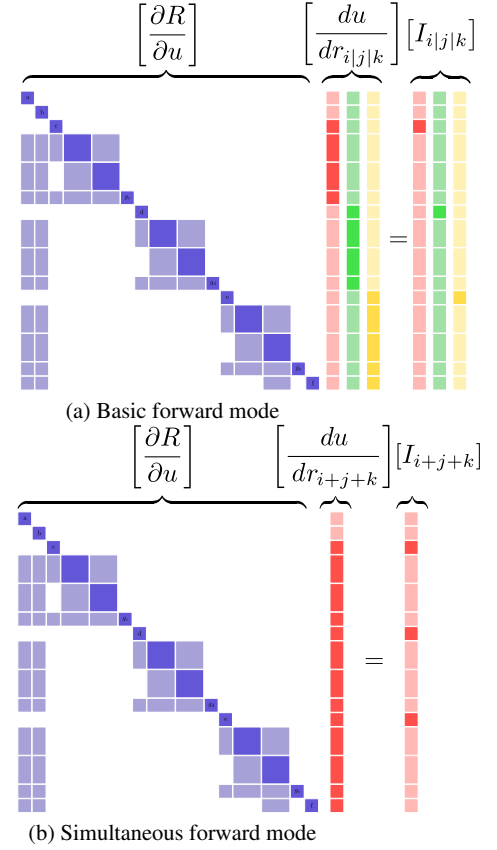
Fig. 9: Combining multiple linear solutions.

would require six linear solutions in reverse mode. Therefore, the colored forward mode is faster than using the uncolored reverse mode.

There is a well known class of optimal control problems that is specifically formulated to create total derivative Jacobians that can be efficiently colored. Betts and Huffman [7] describe their "sparse finite-differencing" method, where multiple model inputs are perturbed simultaneously to approximate total derivatives with respect to more than one input at the same time. Sparse finite-differencing is applicable only to forward separable problems, but OpenMDAO can leverage coloring in both forward and reverse directions because analytic derivatives are computed with linear solutions rather than a numerical approximations of the nonlinear analysis. The ability to leverage both forward and reverse modes for coloring gives the analytic derivatives approach greater flexibility than the traditional sparse finite-differencing, and makes it applicable to a wider range of problems.

Although the notional example used in this section provides a very obvious coloring pattern, in general coloring a total derivative Jacobian is extremely nontrivial. OpenMDAO uses an algorithm developed by Coleman and Verma [18] to perform coloring, and it uses a novel approach to

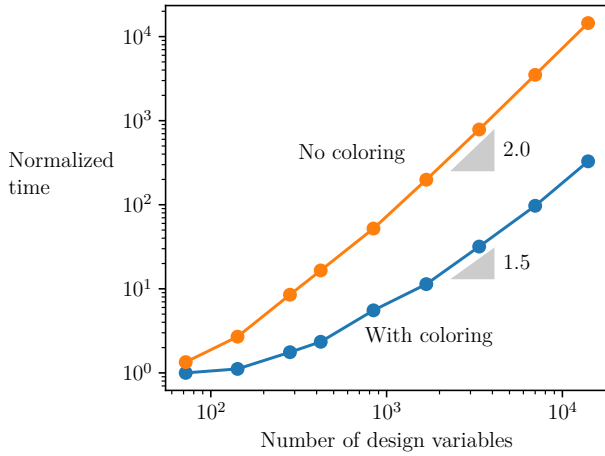computing the total derivative sparsity. More details are included in Appendix A.

Fig. 10: Comparison of total derivatives computation time computed with (blue) and without (orange) the combined linear solution feature.

6.1.1 Computational Savings from Combined Linear Solutions

The combined linear solution feature is leveraged by the Dymos optimal control library, which is built using OpenMDAO. To date, Dymos has been used to solve a range of optimal control problems[24], including canonical problems such as Bryson's minimum time to climb problem [10], as well as the classic brachistochrone problem posed by Bernoulli [6]. It has also been used to solve more complex optimal trajectory problems for electric aircraft [23, 89].

To demonstrate the computational improvement, we present results showing how the cost of solving for the total derivatives Jacobian scales with and without the combined linear solutions feature for Bryson's minimum time to climb problem implemented in the Dymos example problem library. Figure 10 shows the variation of the total derivatives computation time as a function of the number of time steps used in the model. The greater the number of time steps, the greater the number of constraints in the optimization problem for which we need total derivatives. We can see that the combined linear solution offers significant reductions in the total derivative computation time, and more importantly, shows superior computational scaling.

6.2 Sparsity from Quasi-decoupled Parallel Models

Combining multiple linear solutions offers significant computational savings with no requirement for additional memory allocation; thus, it is a highly efficient technique for reducing the computational cost of solving for total derivatives, even when running in serial. However, it is not possible to use that approach for all models. In particular, a common model structure that relies on parallel execution for computational efficiency, which we refer to as "quasi-decoupled", prevents the use of combined linear solutions and demands a different approach to exploit its sparsity. In this section, we present a method for performing efficient linear solutions for derivatives of quasi-decoupled systems that enables the efficient use of parallel computing resources for reverse mode linear solutions.

A quasi-decoupled model is one with an inexpensive serial calculation bottleneck at the beginning, followed by a more computationally costly set of parallel calculations for independent model outputs. The data passing in this model is such that one set of outputs gets passed to multiple downstream components that can run in parallel. A typical example of this structure can be found in multipoint models, where the same analysis is run at several different points, e.g., multiple CFD analyses that are run for the same geometry, but at different flow conditions [85, 77, 26]. In these cases, the geometric calculations that translate the model inputs to the computational grid are the serial bottleneck, and the multiple CFD analyses are the decoupled parallel computations, which can be solved in an embarrassingly parallel fashion. This model can be run efficiently in the forward direction for nonlinear solutions—making it practically forward decoupled—but the linear reverse mode solutions to compute total derivatives can no longer be run in parallel.

One possible solution to address this challenge is to employ a constraint aggregation approach [60, 63]. This approach allows the adjoint method to work efficiently because it collapses many constraint values into a single scalar, hence recovering the adjoint method efficiency. Though this may work in some cases, constraint aggregation is not well-suited to problems where the majority of the constraints being aggregated are active at the optimal solution, as is the case for equality constraints. In these situations, the conservative nature of the aggregations function is problematic because it prevents the optimizer from tightly satisfying all the equalities. Kennedy and Hicken [57] developed improved aggregation methods that offer a less conservative and more numerically stable formulation, but despite the improvements, aggregation is still not appropriate for all applications. In these cases, an alternate approach is needed to maintain efficient parallel reverse mode (adjoint) linear solutions.

When aggregation cannot be used, OpenMDAO uses a solution technique that retains the parallel efficiency at the cost of a slight increase in required memory. First, the memory allocated for the serial bottleneck calculations in the right-hand side and solution vectors is duplicated across all processors. Only the variables associated with the bottleneck

(a) Sequential derivative computation (Group 1)

(b) Sequential derivative computation (Group 2)

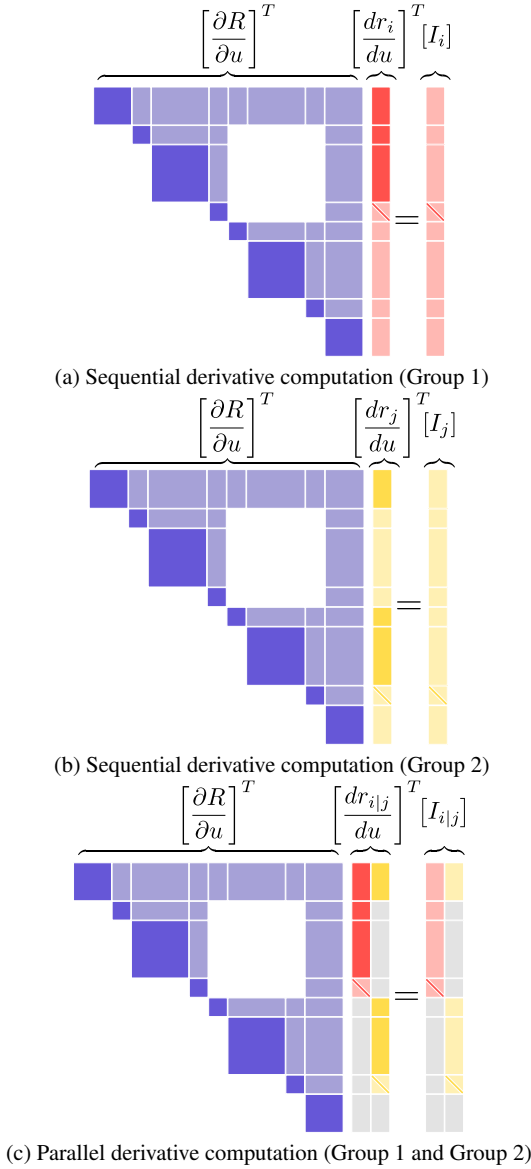(c) Parallel derivative computation (Group 1 and Group 2)

Fig. 11: Parallel derivative computation for a reverse mode solution [45]. Group 1 contains components 2, 3, and 4, and Group 2 contains components 5, 6, and 7. We assume Groups 1 and 2 are allocated on different processors. Without the ability to solve multiple right-hand sides simultaneously, processor 2 would be idling in (a), and processor 1 would be idling in (b). In (c), each processor can solve for the derivatives of its own group simultaneously.

calculation are duplicated, and the variables in the computationally expensive parallel calculations remain the same. This duplication is acceptable because the bottleneck calculation requires an insignificant amount of memory compared to the parallel calculations. The duplication of the memory effectively decouples the portions of the model that require more memory and computational effort, so OpenMDAO can then perform multiple linear solutions in parallel across mul-

tiple processors. This approach for parallel reverse derivative solutions was first proposed by Hwang and Martins [45], and has been adapted into the OpenMDAO framework. It is described below for completeness and to provide context for the performance results presented.

Figure 11 compares the reverse mode linear solution between a basic reverse mode solution an a parallel reverse mode solution. Note that the left-hand side matrices in Fig. 11 are upper triangular because the reverse mode solutions use $[\partial R / \partial u]^T$. In this notional model, the first component on the diagonal is the inexpensive serial bottleneck that all following calculations depend on. Then, there are two parallel expensive computational groups represented by components (2,3,4) and (5,6,7), each of which computes a model output. We assume that Group 1 is allocated in processor 1, and Group 2 is allocated in processor 2.

Using the basic reverse mode requires two sequential linear solutions. During the first solution, illustrated in Fig. 11(a), processor 1 solves for the derivative of the output of Group 1, while processor 2 idles. Similarly, during the second solution, illustrated in Fig. 11(b), processor 2 solves for the derivatives of the outputs of Group 2, while processor 1 idles.

Using the parallel reverse mode, both processors independently loop over the two right-hand sides to compute two linear solutions, as shown in Fig. 11(c). For processor 1, the first right-hand side computes derivatives of the Group 1 model output, while the second performs no operations. For processor 2, the first right-hand side does not require any operations, and the second computes derivatives of the Group 2 model output. Therefore, the parallel reverse mode of Fig. 11(c) takes advantage of embarrassingly parallel execution. Note that in Fig. 11(c), there are grayed out portions of the vectors, indicating that memory is not allocated for those variables on that particular processor.

6.2.1 Computational Savings from Parallel Reverse Mode

We now demonstrate the performance of parallel reverse mode derivatives computation using the combined allocation-mission-design (AMD) problem developed by Hwang and Martins [45]. In this problem, an airline with four existing aircraft and one new aircraft design allocates these aircraft for 128 individual routes to maximize operational profit. This model was executed on a parallel cluster with 140 cores, ensuring that the mission constraints are handled in parallel. A number of the constraints exhibit the quasi-decoupled structure because there are separate sets related to each of the 128 missions which can all be computed in parallel, but they all depend on a single upstream calculation that creates the reverse mode bottleneck.

In Fig. 12, we show the total derivatives computation time with and without parallel reverse mode for a range of
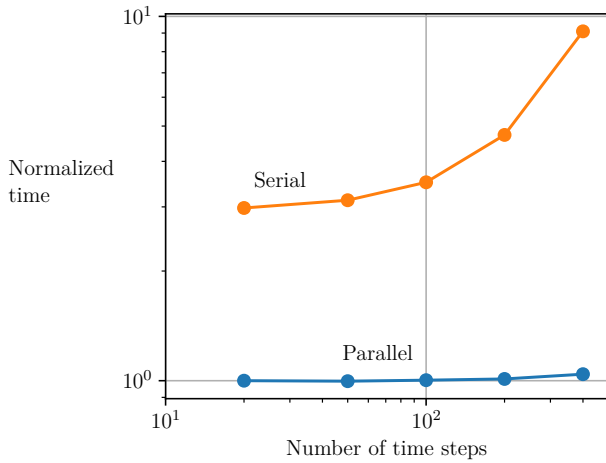
Fig. 12: Comparison of total derivatives computation time calculated with and without parallel reverse mode.

different model sizes. The models were scaled up by refining the time discretization of the mission integration, which also created more physical constraints for each mission. The calculation is significantly faster with parallel reverse mode, but more importantly, the parallel reverse mode makes the cost to compute total derivatives nearly independent of the size of the problem.

7 Applications

The usefulness and efficiency of OpenMDAO has already been demonstrated in a variety of applications. All of these applications have been the subject of previous publications and will not be detailed here. Instead, we present an overview of these applications to illustrate the wide range of model fidelities, problem structures, and disciplines that can be handled by OpenMDAO. For each application, we highlight the computational performance and OpenMDAO features that were used. The applications are listed in Table 13, where we show the extended design structure matrix (XDSM) [62] for each problem and list the design variables, objective functions, and constraints.

One of the first practical engineering problems, and the first example of an optimal-control problem, solved with OpenMDAO V2 was a satellite MDO problem that maximized the data downloaded by varying satellite design parameters (related to solar panels, antenna, and radiators) as well as time dependent variables (power distribution, trajectory, and solar panel controls) [33]. This problem was originally implemented using a bare-bones implementation of MAUD [50]. The modeled disciplines consisted of orbit dynamics, attitude dynamics, cell illumination, temperature, solar power, energy storage, and communication. The optimization involved over 25,000 design variables and 2.2 mil-

lion state variables, and required 100 CPU-hours to converge to the optimum result in a serial computation [50]. The satellite model is broken down into around 100 different components, which greatly simplified the task of deriving the analytic partial derivatives by hand. This problem exhibits the quasi-decoupled structure discussed in Section 6.2, and in OpenMDAO V2, the model was able to run in around 6 hours of wall time, running in parallel on 6 CPUs with parallel derivative computation feature of the framework.

The first integration of a specialized high-fidelity solver in OpenMDAO was done to perform the design optimization of an aircraft considering allocation, trajectory optimization, and aerodynamic performance with the objective of maximizing airline profit [46]. The aerodynamics were modeled using the ADflow CFD solver [64], which has an adjoint implementation to efficiently compute derivatives of the aerodynamic force coefficients with respect to hundreds of wing shape variables. In this work, the ADflow solver was integrated into the overall model in an explicit form, which greatly simplified the integration and reduced the number of variables that OpenMDAO needed to track. The optimization problem consisted of over 6,000 design variables and 23,000 constraints, and it was solved in about 10 hours using 128 processors. This work relied heavily on OpenMDAO's support for parallel computing to efficiently evaluate multiple aerodynamic points simultaneously. Related work on this problem by Roy et al.[86] expanded the problem to a mixed integer optimization that considered the airline allocation portion of the problem in a more correct discrete form, which demonstrated the flexibility of the framework to expand beyond purely gradient-based optimization.

The OpenMDAO interface to ADflow, first developed in the work mentioned above, was later reworked into an implicit form that exposed the full state vector of the flow solution to the framework. The new wrapper was then used in a series of propulsion-airframe integration optimization studies that were the first to demonstrate the framework's ability to compute high-fidelity coupled derivatives. A CFD model of a tightly integrated fuselage and propulsor was coupled to a one-dimensional engine cycle model to build an aeropropulsive model, which enabled the detailed study of boundary layer ingestion (BLI) effects [36] and the simultaneous design of aerodynamic shape and propulsor sizing for BLI configurations [34, 37]. The thermodynamic cycle analysis tool was developed using OpenMDAO as a standalone propulsion modeling library based on a chemical-equilibrium analysis technique [35]. This new modeling library was the first engine cycle analysis capability that included analytic derivative computation [42]. The development of the cycle analysis tool in OpenMDAO was ultimately what motivated the addition of the new monolithic linear solver strategy to the framework, and coupling that model to the high-fidelity aerodynamic solver required the
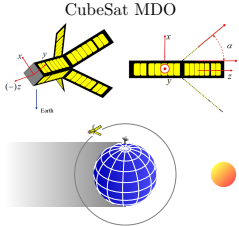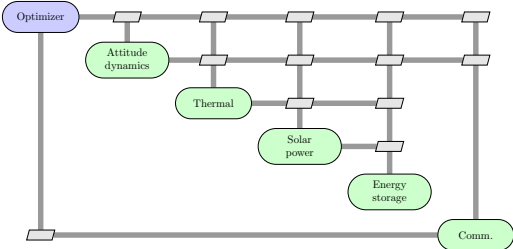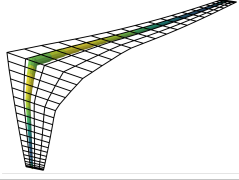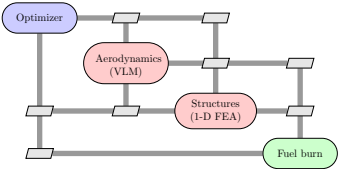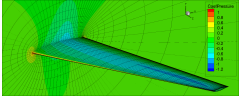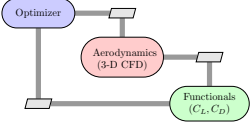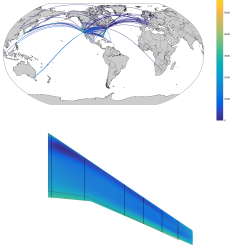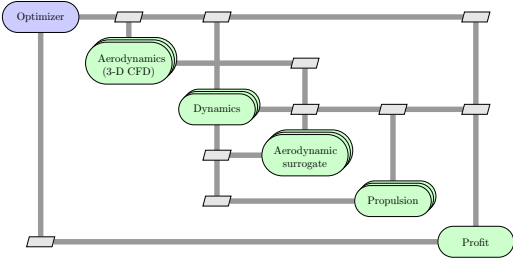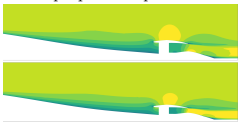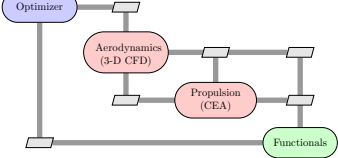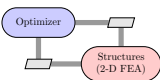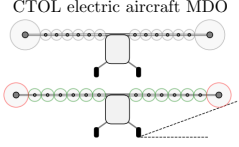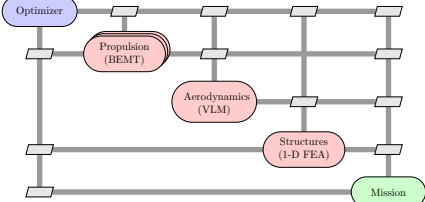
Problem	Model structure	Design variables	Objective	Constraints
CubeSat MDO		Solar panel angle, antenna angle, num. radiators, power distribution, attitude profile, solar panel controls	Data downloaded	Batt. charge rate, batt. charge level
Low-fidelity wing aerostructural opt.		Thicknesses, twists	Fuel burn, range, takeoff weight	Trim, stress
RANS-based wing optimization		Shape	Drag coefficient	Lift coefficient
Allocation-design optimization		Wing shape, altitude profiles, cruise Mach, allocation	Profit	Wing geometry, thrust demand fleet limits
Aero-propulsive optimization		Inlet shape	Fuel burn	Trim
Structural topology optimization		Element densities	Compliance	Mass fraction
CTOL electric aircraft MDO		Altitude prof., velocity prof., prop RPM profs., prop chord, prop twist, prop diam., wing twist, beam thickness	Range	Average speed, eqs. of motion, max. power, min. torque, ground clear., tip speed, wing failure

Fig. 13: Summary of applications of OpenMDAO to engineering design optimization problems.

implementation of a mixed hierarchical-monolithic linear solver strategy in the coupled model.

Jasa et al. [53] developed OpenAeroStruct, a low-fidelity wing aerostructural library whose development was motivated by the absence of a tool for fast wing design optimization. OpenAeroStruct implements a vortex lattice model for the aerodynamic analysis and a beam finite-element model for structural analysis. These analyses are coupled, enabling the aerostructural analysis of lifting surfaces. Each of the models was implemented in OpenMDAO from the ground up, making use of the hierarchical representation and solvers for the best possible coupled solution efficiency (seconds using one processor) as demonstrated in Fig. 7. As a result, OpenAeroStruct efficiently computes aerostructural derivatives through the coupled-adjoint method, enabling fast aerostructural analysis (solutions in seconds) and optimizations (converged results in minutes). OpenAeroStruct has already been used in a number of applications, including the multi-fidelity optimization under uncertainty of a tailless aircraft [25, 13, 82, 20, 21, 8, 61, 96, 19, 4, 83, 14]. Significant computational efficiency was achieved for OpenAeroStruct by using the sparse-assembled Jacobian matrix feature with a monolithic linear solver strategy, thanks to the highly sparse nature of many of the underlying calculations.

Chung et al. [17] developed a framework for setting up structural topology optimization problems and formulations. Using this platform, they implemented three popular topology optimization approaches. Even though structural topology optimization involves only one discipline, they found that the framework benefited from the modularity and the more automated derivative computation. The increased modularity made it easier to restructure and extend the code, allowing the authors to quickly change the order of operations in the process to demonstrate the importance of correct sequencing. This structural topology optimization framework is expected to facilitate future developments in multiscale and multidisciplinary topology optimization. This work, in addition to OpenAeroStruct, provides an excellent example of using OpenMDAO as a low-level software layer to develop new disciplinary analysis solvers.

Hwang and Ning [49] developed and integrated low-fidelity propeller, aerodynamic, structural, and mission analysis models using OpenMDAO for NASA's X-57 Maxwell research aircraft, which features distributed electric propulsion. They solved MDO problems with up to 101 design variables and 74 constraints that converged in a few hundred model evaluations. Numerical experiments showed the scaling of the optimization time with the number of mission points was, at worst, linear. The inclusion of a fully transient mission analysis model of the aircraft performance was shown to offer significantly different results than a basic multipoint optimization formulation. The need to include the transient analysis is an example of why analytic derivatives are needed

for these types of problems: They offer the required computational efficiency and accuracy that could not be achieved using monolithic finite differencing.

Other work that used OpenMDAO V2 includes a framework for the solution of ordinary differential equations [48], a conceptual design model for aircraft electric propulsion [9], and a mission planning tool for the X-57 aircraft [89].

Application-focused work has included the design of a next-generation airliner considering operations and economics [87], design and trajectory optimization of a morphing wing aircraft [52], and trajectory optimization of an aircraft with a fuel thermal management system [54]. OpenMDAO is also being used extensively by the wind energy community for wind turbine design [79, 5, 98, 99, 74, 30, 22] and wind farm layouts [95, 92].

8 Conclusions

The OpenMDAO framework was developed to facilitate the multidisciplinary analysis and design optimization of complex engineering systems. While other frameworks exist that address the same purpose, OpenMDAO has evolved in the last few years to incorporate state-of-the-art algorithms that enable it to address optimization problems of unprecedented scale and complexity.

Two main areas of development made this possible: algorithms for the solution of coupled systems and methods for derivative computation. The development of efficient derivative computation was motivated by the fact that gradient-based optimization is our only hope for solving large-scale problems that involve computationally expensive models; thus, efficient gradient computations that are scalable are required. Because most models and coupled systems exhibit some degree of sparsity in their problem structure, OpenMDAO takes advantage of the sparsity for both storage and computation.

To achieve the efficient solution of coupled systems, OpenMDAO implements known state-of-the-art monolithic methods and has developed a flexible hierarchical approach that enables users to group models according to the problem structure so that computations can be nested, parallelized, or both.

To compute total coupled derivatives efficiently in a scalable way, OpenMDAO uses analytic methods in two modes: forward and reverse. The forward mode is equivalent to the coupled direct method, and its cost scales linearly with the number of design variables. The reverse mode is equivalent to the coupled adjoint method, and its cost scales linearly with the number of functions of interest—but it is independent of the number of design variables. This last characteristic is particularly desirable because many problems have a number of design variables that is larger than the number of functions of interest (objective and constraints). Further-

more, in cases with large numbers of constraints, these can often be aggregated.

Problem sparsity is also exploited in the coupled derivative computation by a new approach we developed that uses graph coloring. We also discussed a few other techniques to increase the efficiency of derivative computations using the hierarchical problem representation.

The algorithms in OpenMDAO work best if the residuals of the systems involved are available, but when they are not available, it is possible to formulate the models solely in terms of their inputs and outputs.

The efficiency and scalability of OpenMDAO was demonstrated in several examples. We also presented an overview of various previously published applications of OpenMDAO to engineering design problems, including satellite, wing, and aircraft design. Some of these problems involved tens of thousands of design variables and similar number of constraints. Other problems involved costly high-fidelity models, such as CFD and finite element structural analysis with millions of degrees of freedom. While the solution of the problems in these applications would have been possible with single purpose implementations, OpenMDAO made it possible to use state-of-the-art methods with a much lower development effort.

Based on the experience of these applications, we conclude that while OpenMDAO can handle traditional disciplinary analysis models effectively, it is most efficient when these models are developed from the ground up using OpenMDAO with a fine-grained modularity to take full advantage of the problem sparsity, lower implementation effort, and built-in derivative computation.

Replication of Results

Most of the codes required to replicate the results in this paper are available under open source licenses and are maintained in version control repositories. The OpenMDAO framework is available from GitHub (github.com/OpenMDAO). The OpenMDAO website (openmdao.org) provides installation instructions and a number of examples. The code for the simple example of Sec. 3 is listed in Figs. 3 and 4 and can be run once OpenMDAO is installed as is. The scripts used to produce the scaling plots in Figs. 7 and 10 are available as supplemental material in this paper. In addition to requiring OpenMDAO to be installed, these scripts require OpenAeroStruct (github.com/mdolab/OpenAeroStruct) and Dymos (github.com/OpenMDAO/dymos). The scaling plots for the AMD problem (Fig. 12) involve a complex framework that includes code that is not open source, and therefore, we are not able to provide scripts for these results. Finally, although no results are shown in applications mentioned in Sec. 7, the code for two of these applications—OpenAeroStruct (github.com/mdolab/OpenAeroStruct) and

the satellite MDO (github.com/OpenMDAO/CADRE)—is also available.

Conflict of interest statement: On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Arora J, Haug EJ (1979) Methods of design sensitivity analysis in structural optimization. AIAA Journal 17(9):970–974, doi:10.2514/3.61260
2. Balabanov V, Charpentier C, Ghosh DK, Quinn G, Vanderplaats G, Venter G (2002) Visualdoc: A software system for general purpose integration and design optimization. In: 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Atlanta, GA
3. Balay S, Abhyankar S, Adams M, Brown J, Buschelman K, Dalcin L, Dener A, Eijkhout V, Gropp W, Karpeyev D, Kaushik D, Knepley M, May D, McInnes LC, Mills R, Munson T, Rupp K, Sanan P, Smith B, Zampini S, Zhang H (2018) PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.10, Argonne National Laboratory
4. Baptista R, Poloczek M (2018) Bayesian optimization of combinatorial structures. In: Dy J, Krause A (eds) Proceedings of the 35th International Conference on Machine Learning, PMLR, Stockholmsmässan, Stockholm Sweden, Proceedings of Machine Learning Research, vol 80, pp 462–471, URL http://proceedings.mlr.press/v80/baptista18a.html
5. Barrett R, Ning A (2018) Integrated free-form method for aerostructural optimization of wind turbine blades. Wind Energy 21(8):663–675, doi:10.1002/we.2186
6. Bernoulli J (1696) A new problem to whose solution mathematicians are invited. Acta Eruditorum 18:269

7. Betts JT, Huffman WP (1991) Trajectory optimization on a parallel processor. Journal of Guidance, Control, and Dynamics 14(2):431–439, doi:10.2514/3.20656

8. Bons NP, He X, Mader CA, Martins JRRA (2017) Multimodality in aerodynamic wing design optimization. In: 18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Denver, CO, doi:10.2514/6.2017-3753

9. Brelje BJ, Martins JRRA (2018) Development of a conceptual design model for aircraft electric propulsion with efficient gradients. In: Proceedings of the AIAA/IEEE Electric Aircraft Technologies Symposium, Cincinnati, OH, doi:10.2514/6.2018-4979

10. Bryson AE (1999) Dynamic optimization. Addison Wesley Longman Menlo Park, CA

11. Bryson AE, Ho YC (1975) Applied Optimal Control: Optimization, Estimation, and Control. John Wiley & Sons

12. Carrier G, Destarac D, Dumont A, Méheut M, Din ISE, Peter J, Khelil SB, Brezillon J, Pestana M (2014) Gradient-based aerodynamic optimization with the elsA software. In: 52nd Aerospace Sciences Meeting, doi:10.2514/6.2014-0568

13. Chaudhuri A, Lam R, Willcox K (2017) Multifidelity uncertainty propagation via adaptive surrogates in coupled multidisciplinary systems. AIAA Journal pp 235–249, doi:10.2514/1.J055678

14. Chauhan SS, Martins JRRA (2018) Low-fidelity aerostructural optimization of aircraft wings with a simplified wingbox model using openaerostruct. In: Proceedings of the 6th International Conference on Engineering Optimization, EngOpt 2018, Springer, Lisbon, Portugal, pp 418–431, doi:10.1007/978-3-319-97773-7_38

15. Chauhan SS, Hwang JT, Martins JRRA (2018) An automated selection algorithm for nonlinear solvers in MDO. Structural and Multidisciplinary Optimization 58(2):349–377, doi:10.1007/s00158-018-2004-5

16. Chen S, Lyu Z, Kenway GKW, Martins JRRA (2016) Aerodynamic shape optimization of the Common Research Model wing-body-tail configuration. Journal of Aircraft 53(1):276–293, doi:10.2514/1.C033328

17. Chung H, Hwang JT, Gray JS, Kim HA (2018) Implementation of topology optimization using OpenMDAO. In: 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, AIAA, AIAA, Kissimmee, FL, doi:10.2514/6.2018-0653

18. Coleman TF, Verma A (1998) The efficient computation of sparse Jacobian matrices using automatic differentiation. SIAM Journal of Scientific Computing 19(4):1210–1233

19. Cook LW (2018) Effective formulations of optimization under uncertainty for aerospace design. PhD thesis, University of Cambridge, doi:10.17863/CAM.23427

20. Cook LW, Jarrett JP, Willcox KE (2017) Extending horsetail matching for optimization under probabilistic, interval, and mixed uncertainties. AIAA Journal pp 849–861, doi:10.2514/1.J056371

21. Cook LW, Jarrett JP, Willcox KE (2017) Horsetail matching for optimization under probabilistic, interval and mixed uncertainties. In: 19th AIAA non-deterministic approaches conference, p 0590, doi:10.2514/6.2017-0590

22. Dykes K, Damiani R, Roberts O, Lantz E (2018) Analysis of ideal towers for tall wind applications. In: 2018 Wind Energy Symposium, AIAA, doi:10.2514/6.2018-0999

23. Falck RD, Chin JC, Schnulo SL, Burt JM, Gray JS (2017) Trajectory optimization of electric aircraft subject to subsystem thermal constraints. In: 18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Denver, CO

24. Falck RD, Gray JS, Naylor B (2019) Optimal control within the context of multidisciplinary design, analysis, and optimization. In: AIAA SciTech Forum, AIAA, doi:10.2514/6.2019-0976

25. Friedman S, Ghoreishi SF, Allaire DL (2017) Quantifying the impact of different model discrepancy formulations in coupled multidisciplinary systems. In: 19th AIAA non-deterministic approaches conference, p 1950, doi:10.2514/6.2017-1950

26. Gallard F, Meaux M, Montagnac M, Mohammadi B (2013) Aerodynamic aircraft design for mission performance by multipoint optimization. In: 21st AIAA Computational Fluid Dynamics Conference, American Institute of Aeronautics and Astronautics, doi:10.2514/6.2013-2582, URL http://dx.doi.org/10.2514/6.2013-2582

27. Gallard F, Lafage R, Vanaret C, Pauwels B, Guénot D, Barjhoux PJ, Gachelin V, Gazaix A (2017) GEMS: A python library for automation of multidisciplinary design optimization process generation. In: 18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference

28. Gebremedhin AH, Manne F, Pothen A (2005) What color is your Jacobian? Graph coloring for computing derivatives. SIAM Review 47(4):629–705

29. Golovidov O, Kodiyalam S, Marineau P, Wang L, Rohl P (1998) Flexible implementation of approximation concepts in an MDO framework. In: 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, American Institute of Aeronautics and Astronautics, doi:10.2514/6.1998-4959

30. Graf P, Dykes K, Damiani R, Jonkman J, Veers P (2018) Adaptive stratified importance sampling: hybridization of extrapolation and importance sampling monte carlo methods for estimation of wind turbine extreme loads.

Wind Energy Science (Online) 3(2), doi:10.5194/wes-3-475-2018

31. Gray J, Moore KT, Naylor BA (2010) OpenMDAO: An open source framework for multidisciplinary analysis and optimization. In: Proceedings of the 13th AIAA/ISSMO Multidisciplinary Analysis Optimization Conference, Fort Worth, TX, aIAA 2010-9101

32. Gray J, Moore KT, Hearn TA, Naylor BA (2013) Standard platform for benchmarking multidisciplinary design analysis and optimization architectures. AIAA Journal 51(10):2380–2394, doi:10.2514/1.J052160

33. Gray J, Hearn T, Moore K, Hwang JT, Martins JRRA, Ning A (2014) Automatic evaluation of multidisciplinary derivatives using a graph-based problem formulation in OpenMDAO. In: Proceedings of the 15th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Atlanta, GA, doi:10.2514/6.2014-2042

34. Gray JS, Martins JRRA (2018) Coupled aeropropulsive design optimization of a boundary layer ingestion propulsor. The Aeronautical Journal (In press)

35. Gray JS, Chin J, Hearn T, Hendricks E, Lavelle T, Martins JRRA (2017) Chemical equilibrium analysis with adjoint derivatives for propulsion cycle analysis. Journal of Propulsion and Power 33(5):1041–1052, doi:10.2514/1.B36215

36. Gray JS, Mader CA, Kenway GKW, Martins JRRA (2017) Modeling boundary layer ingestion using a coupled aeropropulsive analysis. Journal of Aircraft doi:10.2514/1.C034601

37. Gray JS, Kenway GKW, Mader CA, Martins JRRA (2018) Aeropropulsive design optimization of a turbo-electric boundary layer ingestion propulsion. In: 2018 Aviation Technology, Integration, and Operations Conference, AIAA, Atlanta, GA, doi:10.2514/6.2018-3976

38. Griewank A (2000) Evaluating Derivatives. SIAM, Philadelphia

39. Haftka RT (1977) Optimization of flexible wing structures subject to strength and induced drag constraints. AIAA Journal 15(8):1101–1106, doi:10.2514/3.7400

40. Haftka RT, Sobieszczanski-Sobieski J, Padula SL (1992) On options for interdisciplinary analysis and design optimization. Structural Optimization 4:65–74, doi:10.1007/BF01759919

41. He P, Mader CA, Martins JRRA, Maki KJ (2018) An aerodynamic design optimization framework using a discrete adjoint approach with openfoam. Computers & Fluids doi:10.1016/j.compfluid.2018.04.012, ¡p¿in press¡/p¿

42. Hearn DT, Hendricks E, Chin J, Gray JS, Moore DKT (2016) Optimization of turbine engine cycle analysis with analytic derivatives. In: 17th AIAA/ISSMO Multidisciplinary Analysis and Optimization Confer-

ence, part of AIAA Aviation 2016 (Washington, DC), doi:10.2514/6.2016-4297

43. Heath C, Gray J (2012) OpenMDAO: Framework for flexible multidisciplinary design, analysis and optimization methods. In: Proceedings of the 53rd AIAA Structures, Structural Dynamics and Materials Conference, Honolulu, HI, aIAA-2012-1673

44. Heil M, Hazel AL, Boyle J (2008) Solvers for large-displacement fluid–structure interaction problems: segregated versus monolithic approaches. Computational Mechanics 43(1):91–101, doi:10.1007/s00466-008-0270-6

45. Hwang JT, Martins JRRA (2015) Parallel allocation-mission optimization of a 128-route network. In: Proceedings of the 16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Dallas, TX

46. Hwang JT, Martins JRRA (2016) Allocation-mission-design optimization of next-generation aircraft using a parallel computational framework. In: 57th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, American Institute of Aeronautics and Astronautics, doi:10.2514/6.2016-1662

47. Hwang JT, Martins JRRA (2018) A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives. ACM Transactions on Mathematical Software (In press)

48. Hwang JT, Munster DW (2018) Solution of ordinary differential equations in gradient-based multidisciplinary design optimization. In: 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, AIAA, AIAA, Kissimmee, FL, doi:10.2514/6.2018-1646

49. Hwang JT, Ning A (2018) Large-scale multidisciplinary optimization of an electric aircraft for on-demand mobility. In: 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, AIAA, AIAA, Kissimmee, FL, doi:10.2514/6.2018-1384

50. Hwang JT, Lee DY, Cutler JW, Martins JRRA (2014) Large-scale multidisciplinary optimization of a small satellite's design and operation. Journal of Spacecraft and Rockets 51(5):1648–1663, doi:10.2514/1.A32751

51. Jameson A (1988) Aerodynamic design via control theory. Journal of Scientific Computing 3(3):233–260

52. Jasa JP, Hwang JT, Martins JRRA (2018) Design and trajectory optimization of a morphing wing aircraft. In: 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference; AIAA SciTech Forum, Orlando, FL

53. Jasa JP, Hwang JT, Martins JRRA (2018) Open-source coupled aerostructural optimization using Python. Structural and Multidisciplinary Optimization 57:1815–1827, doi:10.1007/s00158-018-1912-8

54. Jasa JP, Mader CA, Martins JRRA (2018) Trajectory optimization of supersonic air vehicle with thermal fuel management system. In: AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Atlanta, GA, doi:10.2514/6.2018-3884

55. Jones M, Plassmann P (1993) A parallel graph coloring heuristic. SIAM Journal on Scientific Computing 14(3):654–669, doi:10.1137/0914041

56. Karp RM, Wigderson A (1985) A fast parallel algorithm for the maximal independent set problem. Journal of the Association for Computing Machinery 32(4):762–773

57. Kennedy GJ, Hicken JE (2015) Improved constraint-aggregation methods. Computer Methods in Applied Mechanics and Engineering 289:332–354, doi:10.1016/j.cma.2015.02.017

58. Keyes DE, McInnes LC, Woodward C, Gropp W, Myra E, Pernice M, Bell J, Brown J, Clo A, Connors J, Constantinescu E, Estep D, Evans K, Farhat C, Hakim A, Hammond G, Hansen G, Hill J, Isaac T, Jiao X, Jordan K, Kaushik D, Kaxiras E, Koniges A, Lee K, Lott A, Lu Q, Magerlein J, Maxwell R, McCourt M, Mehl M, Pawlowski R, Randles AP, Reynolds D, Riviere B, Rude U, Scheibe T, Shadid J, Sheehan B, Shephard M, Siegel A, Smith B, Tang X, Wilson C, Wohlmuth B (2013) Multiphysics simulations: Challenges and opportunities. International Journal of High Performance Computing Applications 27(1):4–83, doi:10.1177/1094342012468181

59. Kolonay RM, Sobolewski M (2011) Service oriented computing environment (SORCER) for large scale, distributed, dynamic fidelity aeroelastic analysis. In: Optimization, International Forum on Aeroelasticity and Structural Dynamics, IFASD 2011, 26–30

60. Kreisselmeier G, Steinhauser R (1979) Systematic control design by optimizing a vector performance index. In: International Federation of Active Controls Symposium on Computer-Aided Design of Control Systems, Zurich, Switzerland, doi:10.1016/S1474-6670(17)65584-8

61. Lam R, Poloczek M, Frazier P, Willcox KE (2018) Advances in bayesian optimization with applications in aerospace engineering. In: 2018 AIAA Non-Deterministic Approaches Conference, p 1656, doi:10.2514/6.2018-1656

62. Lambe AB, Martins JRRA (2012) Extensions to the design structure matrix for the description of multidisciplinary design, analysis, and optimization processes. Structural and Multidisciplinary Optimization 46:273–284, doi:10.1007/s00158-012-0763-y

63. Lambe AB, Martins JRRA, Kennedy GJ (2017) An evaluation of constraint aggregation strategies for wing box mass minimization. Structural and Multidisciplinary Optimization 55(1):257–277, doi:10.1007/s00158-016-1495-1

64. Lyu Z, Kenway GK, Paige C, Martins JRRA (2013) Automatic differentiation adjoint of the Reynolds-averaged Navier–Stokes equations with a turbulence model. In: 21st AIAA Computational Fluid Dynamics Conference, San Diego, CA, doi:10.2514/6.2013-2581

65. Mader CA, Martins JRRA, Alonso JJ, van der Weide E (2008) ADjoint: An approach for the rapid development of discrete adjoint solvers. AIAA Journal 46(4):863–873, doi:10.2514/1.29123

66. Marriage CJ, Martins JRRA (2008) Reconfigurable semi-analytic sensitivity methods and mdo architectures within the πMDO framework. In: Proceedings of the 12th AIAA/ISSMO Multidisciplinary Analysis and Optimizaton Conference, Victoria, British Columbia, Canada, AIAA 2008-5956

67. Martins JRRA, Hwang JT (2013) Review and unification of methods for computing derivatives of multidisciplinary computational models. AIAA Journal 51(11):2582–2599, doi:10.2514/1.J052184

68. Martins JRRA, Hwang JT (2016) Multidisciplinary design optimization of aircraft configurations—Part 1: A modular coupled adjoint approach. Lecture series, Von Karman Institute for Fluid Dynamics, Rode Saint Genèse, Belgium, iSSN0377-8312

69. Martins JRRA, Lambe AB (2013) Multidisciplinary design optimization: A survey of architectures. AIAA Journal 51(9):2049–2075, doi:10.2514/1.J051895

70. Martins JRRA, Sturdza P, Alonso JJ (2003) The complex-step derivative approximation. ACM Transactions on Mathematical Software 29(3):245–262, doi:10.1145/838250.838251, september

71. Martins JRRA, Alonso JJ, Reuther JJ (2004) High-fidelity aerostructural design optimization of a supersonic business jet. Journal of Aircraft 41(3):523–530, doi:10.2514/1.11478

72. Martins JRRA, Alonso JJ, Reuther JJ (2005) A coupled-adjoint sensitivity analysis method for high-fidelity aero-structural design. Optimization and Engineering 6(1):33–62, doi:10.1023/B:OPTE.0000048536.47956.62

73. Martins JRRA, Marriage C, Tedford NP (2009) pyMDO: An object-oriented framework for multidisciplinary design optimization. ACM Transactions on Mathematical Software 36(4):20:1–20:25, doi:10.1145/1555386.1555389

74. McWilliam MK, Zahle F, Dicholkar A, Verelst D, Kim T (2018) Optimal aero-elastic design of a rotor with bend-twist coupling. Journal of Physics: Conference Series 1037(4):042009, URL http://stacks.iop.org/1742-6596/1037/i=4/a=042009

75. Moore K, Naylor B, Gray J (2008) The development of an open-source framework for multidisci-

plinary analysis and optimization. In: Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Victoria, BC, Canada, aIAA 2008-6069

76. Naumann U (2011) The Art of Differentiating Computer Programs—An Introduction to Algorithmic Differentiation, SIAM

77. Nemec M, Zingg DW, , Pulliam TH (2004) Multipoint and multi-objective aerodynamic shape optimization. AIAA Journal 42(6):1057–1065, doi:10.2514/1.10415

78. Nielsen EJ, Kleb WL (2006) Efficient construction of discrete adjoint operators on unstructured grids using complex variables. AIAA Journal 44(4):827–836, doi:10.2514/1.15830

79. Ning A, Petch D (2016) Integrated design of downwind land-based wind turbines using analytic gradients. Wing Energy 19(12):2137–2152, doi:10.1002/we.1972

80. Oliphant TE (2007) Python for scientific computing. Computing in Science and Engineering 9(3):10, doi:10.1109/MCSE.2007.58

81. Padula SL, Gillian RE (2006) Multidisciplinary environments: A history of engineering framework development. In: Proceedings of the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, doi:doi.org/10.2514/6.2006-7083, aIAA 2006-7083

82. Palar PS, Shimoyama K (2017) Polynomial-chaos-kriging-assisted efficient global optimization. In: Computational Intelligence (SSCI), 2017 IEEE Symposium Series on, IEEE, pp 1–8, doi:10.1109/SSCI.2017.8280831

83. Peherstorfer B, Beran PS, Willcox KE (2018) Multi-fidelity monte carlo estimation for large-scale uncertainty propagation. In: 2018 AIAA Non-Deterministic Approaches Conference, p 1660, doi:10.2514/6.2018-1660

84. Peter JEV, Dwight RP (2010) Numerical sensitivity analysis for aerodynamic optimization: A survey of approaches. Computers and Fluids 39(3):373–391, doi:10.1016/j.compfluid.2009.09.013

85. Reuther JJ, Jameson A, Alonso JJ, Rimlinger MJ, , Saunders D (1999) Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers, part 1. Journal of Aircraft 36(1):51–60, doi:https://doi.org/10.2514/2.2413

86. Roy S, Crossley WA, Moore KT, Gray JS, Martins JRRA (2018) Next generation aircraft design considering airline operations and economics. In: 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, AIAA, AIAA, Kissimmee, FL, doi:10.2514/6.2018-1647

87. Roy S, Crossley WA, Moore KT, Gray JS, Martins JRRA (2018) Next generation aircraft design considering airline operations and economics. In:

AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Kissimmee, FL, doi:10.2514/6.2018-1647

88. Salas AO, Townsend JC (1998) Framework requirements for MDO application development. In: 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, pp 98–4740

89. Schnulo SL, Jeff Chin RDF, Gray JS, Papathakis KV, Clarke SC, Reid N, Borer NK (2018) Development of a multi-segment mission planning tool for SCEPTOR X-57. In: 2018 Multidisciplinary Analysis and Optimization Conference, AIAA, Atlanta, GA, doi:10.2514/6.2018-3738

90. Sobieszczanski-Sobieski J (1990) Sensitivity of complex, internally coupled systems. AIAA Journal 28(1):153–160, doi:10.2514/3.10366

91. Squire W, Trapp G (1998) Using complex variables to estimate derivatives of real functions. SIAM Review 40(1):110–112

92. Stanley APJ, Ning A (2018) Coupled wind turbine design and layout optimization with non-homogeneous wind turbines. Wind Energy Science doi:10.5194/wes-2018-54

93. Tedford NP, Martins JRRA (2006) On the common structure of MDO problems: A comparison of architectures. In: Proceedings of the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Portsmouth, VA, AIAA 2006-7080

94. Tedford NP, Martins JRRA (2010) Benchmarking multidisciplinary design optimization algorithms. Optimization and Engineering 11(1):159–183, doi:10.1007/s11081-009-9082-6

95. Thomas J, Gebraad P, , Ning A (2017) Improving the FLORIS wind plant model for compatibility with gradient-based optimization. Wind Engineering 41(5):313–329, doi:10.1177/0309524X17722000

96. Tracey BD, Wolpert D (2018) Upgrading from gaussian processes to student's t processes. In: 2018 AIAA Non-Deterministic Approaches Conference, p 1659, doi:10.2514/6.2018-1659

97. Welsh DJA, Powell MB (1967) An upper bound for the chromatic number of a graph and its application to timetabling problems. Computing Journal 10(1):85–=86, doi:10.1093/comjnl/10.1.85

98. Zahle F, Tibaldi C, Pavese C, McWilliam MK, Blasques JPAA, Hansen MH (2016) Design of an aeroelastically tailored 10 mw wind turbine rotor. Journal of Physics: Conference Series 753(6):062008, URL http://stacks.iop.org/1742-6596/753/i=6/a=062008

99. Zahle F, Sørensen NN, McWilliam MK, Barlas A (2018) Computational fluid dynamics-based surrogate optimization of a wind turbine blade tip extension for maximising energy production. In: Journal of Physics:

Conference Series, The Science of Making Torque
from Wind, Milano, Italy, vol 1037, doi:10.1088/1742-
6596/1037/4/042013

A Coloring of Total Derivative Jacobians

A.1 Determining Total Derivative Coloring

In the toy problem illustrated in Fig. 8, the forward coloring of the model inputs is obvious. However, for a large problem with an unordered total Jacobian matrix, it is not easy to identify coloring. There are a wide variety of serial coloring algorithms [97, 56, 55, 18, 28], originally developed for coloring partial derivative Jacobians. There are also a set of parallel coloring algorithms that have been developed for parallel distributed memory applications, such as CFD [55, 78, 65, 64, 41]. What we propose here is that these coloring algorithms are now also applicable for coloring total derivative Jacobian calculations based on the unified derivatives equations (16).

To apply a coloring algorithm, we need to know the total derivative Jacobian sparsity pattern *a priori*, but this information is not easily available. However, the sparsity pattern of the *partial* derivative Jacobian matrix, $\partial R/\partial u$, is known to OpenMDAO *a priori* from the combination of user-declared partial derivatives and the connections made during model construction. Therefore, we developed a method in OpenMDAO that computes the total derivative sparsity given the partial derivative Jacobian sparsity.

To determine the total Jacobian sparsity pattern for a given state, OpenMDAO computes a randomized partial derivative matrix using linear solutions of Eq. (16) with randomized values for $\partial R/\partial u$. Intuitively, one can understand how using randomized partial derivatives would yield a relatively robust estimate of the total derivative sparsity pattern, but we provide a more detailed logical argument for why this approach is appropriate here. A single randomized total derivative Jacobian is likely to give the correct sparsity pattern, but we can reduce the likelihood of errors in the sparsity by summing the absolute value of multiple randomly generated total derivative Jacobians.

Once we have the total derivative sparsity pattern, OpenMDAO applies a coloring algorithm based on the work of Coleman and Verma [18] to identify the reduced set of linear solutions needed to compute the total derivative Jacobian. As we demonstrate in Sec. 5.3, coloring can offer significant performance improvements for problems that have sparse total derivative Jacobians.

A.2 Justification for Coloring with Randomized Total Derivative Jacobians

In theory, it would be possible to color the total derivative Jacobian based on the actual Jacobian computed around the initial condition of the model. There is a risk, however, that the initial condition of the model will happen to be at a point

where some of the total derivatives in the model are incidentally zero, although they will take nonzero values elsewhere in the design space. The incidental zero would potentially result in an incorrect coloring, and so it is to be avoided if possible. Instead of using the actual Jacobian values, we generate a randomized total derivative Jacobian that is statistically highly unlikely to create any incidental zero values.

From Eq. (16), we know that the total derivative Jacobian matrix, du/dr, is equal to the inverse of the partial derivative Jacobian matrix, $\partial R/\partial u$. Our task reduces to the general mathematical problem of determining the sparsity structure of a matrix inverse given the sparsity structure of the matrix itself, assuming that the matrix is large. First, we note that we do not need the sparsity structure of all of du/dr; we only need the rows and columns corresponding to df/dx, which is almost always a much smaller matrix because du/dr contains all the intermediate model variables as well as the model inputs and model outputs. From Cramer's rule, we know that

$$\left[\frac{\partial R}{\partial u}^{-1}\right]_{ij} = \frac{\mathrm{adj}(\partial R/\partial u)_{ij}}{\det(\partial R/\partial u)}, \tag{25}$$

which is to say that the (i, j)th entry of the inverse of the Jacobian is the quotient of the (i, j)th entry of the adjugate of the Jacobian and the determinant of the Jacobian. The Jacobian is invertible for well-posed models, so the determinant is always nonzero. Thus, only the numerator in Eq. (25) determines if a particular term is nonzero. If the matrix is $n \times n$, where n is large, each term in the adjugate and, thus, the inverse is the sum of a large number of terms that are products of $n-1$ partial derivatives.

If we produce a partial derivative Jacobian matrix with random values for all nonzero terms, it would be highly improbable that this sum of a large number of terms would be incidentally zero. Therefore, we assume that each entry of the inverse of the random partial derivative Jacobian that is zero is actually a zero in the sparsity structure of the true total derivative Jacobian.

B Equivalence between Hierarchical and Reduced-Space Newton's Methods

OpenMDAO introduces a new hierarchical Newton's method formulation for the sake of improved numerical flexibility; however, this proof shows that the new full-space method can be made mathematically identical to the more traditional reduced-space Newton's method if one always fully converges the internal nonlinear system associated with R_y. Consider the residual of an arbitrary implicit function, $R_x(x) = r_x$, at some non-converged value of x. If r_x is actually a function of the nonlinear system $R_y(x, y) = 0$, converged for a

specific value of x:

$$R_x(x) = C(x,y) = r_x \tag{26}$$
$$R_y(y) = D(x,y) = 0, \tag{27}$$

then we call R_x the reduced-space residual function, with the full space composed of the vectors x and y of lengths n and m, respectively.

Our ultimate goal is to solve for x such that $R_x(x) = 0$, using Newton's method. The traditional Newton's method iteration consists in computing Δx by solving a linear system of size n:

$$\left[\frac{\partial R_x}{\partial x} \right] \Delta x = -r_x. \tag{28}$$

Expanding $\partial R_x / \partial x$ to account for the intermediate calculation of y gives

$$\left[\frac{\partial C}{\partial x} + \frac{\partial C}{\partial y} \frac{dy}{dx} \right] \Delta x = -r_x. \tag{29}$$

By differentiating Eq. (27) with respect to x, we find that

$$\frac{dR_y}{dx} = \frac{\partial R_y}{\partial x} + \frac{\partial R_y}{\partial y} \frac{dy}{dx} = 0, \tag{30}$$

$$\frac{dy}{dx} = -\left[\frac{\partial D}{\partial y} \right]^{-1} \frac{\partial D}{\partial x}. \tag{31}$$

Combining Eqs. (29) and (31) gives a formula for the Newton update of the reduced-space function as

$$\left[\frac{\partial C}{\partial x} - \frac{\partial C}{\partial y} \left[\frac{\partial D}{\partial y} \right]^{-1} \frac{\partial D}{\partial x} \right] \Delta x = -r_x. \tag{32}$$

Now, instead of the reduced-space form of Eq. (26), consider a full-space form that deals with both x and y simultaneously as one vector:

$$R_u(u) = R(x,y) = \begin{bmatrix} C(x,y) \\ D(x,y) \end{bmatrix} = r_u = \begin{bmatrix} r_x \\ r_y \end{bmatrix}. \tag{33}$$

This full-space form is the mathematical representation used by OpenMDAO for any system—or subsystem, as demonstrated in Eq. (15). The Newton update for the flattened system must be solved for via a linear system of size $(n+m)$:

$$\left[\frac{\partial R_u}{\partial u} \right] \Delta u = -r_u. \tag{34}$$

If we apply the hierarchical Newton algorithm to the full-space formulation, then we can assume that any time Eq. (34) is solved, y has first been found such that that $r_y = 0$. Expanding $\partial R_u / \partial u$ and setting $r_y = 0$ in Eq. (34) yields

$$\begin{bmatrix} \frac{\partial C}{\partial x} & \frac{\partial C}{\partial y} \\ \frac{\partial D}{\partial x} & \frac{\partial D}{\partial y} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = - \begin{bmatrix} r_x \\ 0 \end{bmatrix}. \tag{35}$$

Solving Eq. (35) for Δy and back substituting yields

$$\left[\frac{\partial C}{\partial x} - \frac{\partial C}{\partial y} \left[\frac{\partial D}{\partial y} \right]^{-1} \frac{\partial D}{\partial x} \right] \Delta x = -r_x. \tag{36}$$

Now note that Eqs. (32) and (36) are identical, and therefore applying the hierarchical Newton algorithm to the full-space model (size $n+m$) gives the exact same Δx as the reduced-space Newton algorithm applied to the smaller, reduced-space model (size n). Since the updates to Δx are the same, then assuming complete convergence of all child subsystems, the path that the hierarchical Newton's method takes on the size $(n+m)$ formulation will be identical to the path the reduced-space Newton's method takes on the smaller size n formulation.

C Equivalence between Recursive and Hierarchical Broyden's Methods

Broyden's Second Method computes an approximate update of the inverse Jacobian via

$$\left[\frac{\partial R_x}{\partial x} \right]_n^{-1} = J_n^{-1} \cong J_{n-1}^{-1} + \frac{\Delta x_n - J_{n-1}^{-1} \Delta r_n}{\| \Delta r_n \|^2} \Delta r_n^T. \tag{37}$$

Then, the Newton update is applied using the approximate inverse Jacobian via

$$\Delta x = -J_n^{-1} r_x. \tag{38}$$

C.1 Reduced-space Broyden

Consider the same composite model structure given in Eqs. (26) and (27). From Eq. (32), we know that

$$\left[\frac{\partial R_x}{\partial x} \right]_n = \left[\frac{\partial C}{\partial x} - \frac{\partial C}{\partial y} \frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \right]_n. \tag{39}$$

To simplify the algebra, we now define a new variable, β, as

$$\beta = \left[\frac{\partial R_x}{\partial x} \right]_{n-1}^{-1} = \left[\frac{\partial C}{\partial x} - \frac{\partial C}{\partial y} \frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \right]_{n-1}^{-1}. \tag{40}$$

Now we can substitute this into Eq. (37) to calculate the Broyden update to the inverse Jacobian as

$$J_n^{-1} = \beta + \frac{\Delta x - \beta \Delta r_x}{\| \Delta r_x \|^2} \Delta r_x^T. \tag{41}$$

Finally, if we substitute this into Eq. (38), we get the commanded update to the state value:

$$\Delta x_n = - \left(\beta + \frac{\Delta x - \beta \Delta r_x}{\| \Delta r_x \|^2} \Delta r_x^T \right) r_x. \tag{42}$$

C.2 Full-space Broyden

To apply Broyden's method to the full-space formulation, we start from Eq. (35) for the Newton update of the full-space system, but now instead of an exact inverse Jacobian, we use the approximate inverse Jacobian, J_n^{-1}, i.e.,

$$[\Delta u] = \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = -J_n^{-1} \begin{bmatrix} r_x \\ 0 \end{bmatrix} = -J_n^{-1} [r_u]. \tag{43}$$

The full-space Broyden's method gives

$$\left[\frac{\partial R_u}{\partial u} \right]_n^{-1} = J_n^{-1} \cong J_{n-1}^{-1} + \frac{\Delta u_n - J_{n-1}^{-1} \Delta r_n}{\|\Delta r_n\|^2} \Delta r_n^T. \tag{44}$$

We use the closed form solution for the block inverse of a 2×2 matrix to obtain

$$J_{n-1}^{-1} = \begin{bmatrix} \frac{\partial C}{\partial x} & \frac{\partial C}{\partial y} \\ \frac{\partial D}{\partial x} & \frac{\partial D}{\partial y} \end{bmatrix}^{-1} =$$

$$\begin{bmatrix} \left[\frac{\partial C}{\partial x} - \frac{\partial C}{\partial y} \frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \right]^{-1} & -\frac{\partial C}{\partial x}^{-1} \frac{\partial C}{\partial y} \left[\frac{\partial D}{\partial y} - \frac{\partial D}{\partial x} \frac{\partial C}{\partial x}^{-1} \frac{\partial C}{\partial y} \right]^{-1} \\ -\frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \left[\frac{\partial C}{\partial x} - \frac{\partial C}{\partial y} \frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \right]^{-1} & \left[\frac{\partial D}{\partial y} - \frac{\partial D}{\partial x} \frac{\partial C}{\partial x}^{-1} \frac{\partial C}{\partial y} \right]^{-1} \end{bmatrix}. \tag{45}$$

We can simplify this by substituting Eq. (40) into Eq. (45), which yields

$$J_{n-1}^{-1} = \begin{bmatrix} \beta & -\frac{\partial C}{\partial x}^{-1} \frac{\partial C}{\partial y} \left[\frac{\partial D}{\partial y} - \frac{\partial D}{\partial x} \frac{\partial C}{\partial x}^{-1} \frac{\partial C}{\partial y} \right]^{-1} \\ -\frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \beta & \left[\frac{\partial D}{\partial y} - \frac{\partial D}{\partial x} \frac{\partial C}{\partial x}^{-1} \frac{\partial C}{\partial y} \right]^{-1} \end{bmatrix} \tag{46}$$

To further simplify, we define one more new variable, γ:

$$\gamma = \left[\frac{\partial D}{\partial y} - \frac{\partial D}{\partial x} \frac{\partial C}{\partial x}^{-1} \frac{\partial C}{\partial y} \right]^{-1} \tag{47}$$

and then,

$$J_{n-1}^{-1} = \begin{bmatrix} \beta & -\frac{\partial C}{\partial x}^{-1} \frac{\partial C}{\partial y} \gamma \\ -\frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \beta & \gamma \end{bmatrix}. \tag{48}$$

Now, returning to Eq. (44) we can compute each of the terms:

$$J_{n-1}^{-1} \Delta r_n = J_{n-1}^{-1} \begin{bmatrix} \Delta r_x \\ 0 \end{bmatrix}_n = \begin{bmatrix} \beta \Delta r_x \\ -\frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \beta \Delta r_x. \end{bmatrix} \tag{49}$$

Since the residual r_y is zero,

$$\|\Delta r_u\| = \|\Delta r_x\|. \tag{50}$$

Putting it all together, we get

$$J_n^{-1} = \begin{bmatrix} \beta & -\frac{\partial C}{\partial x}^{-1} \frac{\partial C}{\partial y} \gamma \\ -\frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \beta & \gamma \end{bmatrix} +$$

$$+ \frac{\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} - \begin{bmatrix} \beta \Delta r_x \\ -\frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \beta \|\Delta r_x\|^2 \end{bmatrix}}{\|\Delta r_x\|^2} \begin{bmatrix} \Delta r_x^T & 0 \end{bmatrix}. \tag{51}$$

Performing the outer product and reducing yields

$$\Delta J_n^{-1} = \begin{bmatrix} \beta + \frac{\Delta x - \beta \Delta r_1}{\|\Delta r_x\|^2} \Delta r_x^T & \frac{\partial C}{\partial x}^{-1} \frac{\partial C}{\partial y} \gamma \\ \frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \beta \frac{\Delta x - \frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \beta \Delta r_x}{\|\Delta r_x\|^2} \Delta r_x^T & \gamma \end{bmatrix}. \tag{52}$$

Now we substitute this into Eq. (23) to get the commanded update to the state value.

$$\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}_n = - \begin{bmatrix} \beta + \frac{\Delta x - \beta \Delta r_x}{\Delta r_x^2} \Delta r_x^T & \frac{\partial C}{\partial x}^{-1} \frac{\partial C}{\partial y} \gamma \\ \frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \beta + \frac{\Delta y - \frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \beta \Delta r_x}{\|\Delta r_x\|^2} \Delta r_x^T & \gamma \end{bmatrix} \begin{bmatrix} r_x \\ 0 \end{bmatrix}_n \tag{53}$$

$$\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}_n = - \begin{bmatrix} \left(\beta + \frac{\Delta x - \beta \Delta r_x}{\|\Delta r_x\|^2} \Delta r_x^T \right) r_x \\ \left(\frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \beta + \frac{\Delta y - \frac{\partial D}{\partial y}^{-1} \frac{\partial D}{\partial x} \beta \Delta r_x}{\|\Delta r_x\|^2} \Delta r_x^T \right) r_x \end{bmatrix}. \tag{54}$$

Finally, we can pull out the Broyden updated Δx:

$$\Delta x_n = - \left(\beta + \frac{\Delta x - \beta \Delta r_x}{\|\Delta r_x\|^2} \Delta r_x^T \right) r_x. \tag{55}$$

This matches the result from recursive Broyden in Eq. (42). We also end up with an update to y, but since we have assumed that a sub-solver will always drive R_y to zero, this update to y does not affect the path taken by the top level full-space solver.