This is a preprint of the following article, which is available from http://mdolab.engin.umich.edu
John T. Hwang and J. R. R. A. Martins. A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives. *ACM TOMS*, 2018. (In press).
The published article may differ from this preprint, and is available by following the DOI above.

A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives

John T. Hwang[1] and Joaquim R. R. A. Martins[1]

Abstract One of the challenges in computational modeling is coupling models to solve multidisciplinary problems. Flow-based computational frameworks alleviate part of the challenge through a modular approach, where data *flows* from component to component. However, existing flow-based frameworks are inefficient when coupled derivatives are needed for optimization. To address this, we develop the *modular analysis and unified derivatives* (MAUD) architecture. MAUD formulates the multidisciplinary model as a nonlinear system of equations, which leads to a linear equation that unifies all methods for computing derivatives. This enables flow-based frameworks that use the MAUD architecture to provide a common interface for the chain rule, adjoint method, coupled adjoint method, and hybrid methods; MAUD automatically uses the appropriate method for the problem. A hierarchical, matrix-free approach enables modern solution techniques such as Newton–Krylov solvers to be used within this monolithic formulation without computational overhead. Two demonstration problems are solved using a Python implementation of MAUD: a nanosatellite optimization with more than 2 million unknowns and 25,000 design variables, and an aircraft optimization involving over 6,000 design variables and 23,000 constraints. MAUD is now implemented in the open-source framework OpenMDAO, which has been used to solve aircraft, satellite, wind turbine, and turbofan engine design problems.

Keywords: Parallel computing; optimization; engineering design; multidisciplinary design optimization; multiphysics simulation; PDE-constrained optimization; high-performance computing; complex systems; adjoint methods; derivative computation; Python

1 Introduction

Computational models are ubiquitous in science and engineering. They provide a cheaper and more convenient alternative to physical experiments and are thus a valuable tool in many applications that involve design, decision-making, research, or forecasting. We consider a *computational model* to be a computer program that represents the behavior of a real-world system or process numerically.

In engineering design, we can leverage a given computational model by using numerical optimization to determine the design variable values that maximize the performance of the system that is being designed. In many cases, the computational model has several *components*, and it might involve multiple engineering disciplines or areas of science, therefore requiring the coupling of multiple numerical models. The field of *multidisciplinary design optimization* (MDO) emerged to solve such problems [4, 16, 51, 62].

The work presented herein is also motivated by the solution of these types of problems. We assume that the number of design variables is $\mathcal{O}(10^2)$ or greater, and thus that only gradient-based optimization is practical. However, gradient-based optimization requires differentiability, as well as accurate and efficient derivative computation, both of which make the implementation more challenging. For coupled computational models, the coupled adjoint method can compute accurate gradients at a cost similar to that of solving the model, regardless of the number of design variables, but it requires an *ad hoc* implementation [41, 49]. Another challenge is that these coupled systems might be composed of *heterogeneous* models. For example,

[1]Department of Aerospace Engineering, University of Michigan, Ann Arbor, MI 48109 (hwangjt@umich.edu, jrram@umich.edu).

the modeling of the full system might require coupling a partial differential equation (PDE) solver, a set of explicit functions, and a time integrator for an ordinary differential equation (ODE).

As we detail in Sec. 2.1, there are currently flow-based computational frameworks that facilitate the coupling of heterogeneous models, providing features like graphical interfaces, built-in solvers and optimizers, and automatic unit conversions. However, they are designed to couple models with a relatively small number of inputs and outputs. As a result, these frameworks are inefficient in converging the coupled models and in computing the coupled derivatives for many types of problems. On the other hand, there are computational frameworks (described in Sec. 2.2) that use sophisticated methods to solve PDEs, including coupled multiphysics problems, and a few of these frameworks automatically compute derivatives via adjoint methods. However, there is currently no framework that computes the coupled derivatives efficiently for problems with heterogeneous coupled models. The existing PDE frameworks that compute coupled derivatives use the same numerical method, e.g., the finite-element method, for all the models.

Therefore, there is a need to make the implementation of coupled heterogeneous systems more convenient, in a way that is scalable and efficient. We address this by developing a new architecture for computational frameworks that we call *modular analysis and unified derivatives* (MAUD). The MAUD architecture uses a new mathematical formulation of the computational model that views each component as an implicit function, rather than an explicit one. MAUD then represents the multidisciplinary computational model as a single system of algebraic equations and solves the resulting nonlinear and linear systems using a hierarchical solution strategy, without adding significant memory or computing overhead. This monolithic formulation enables a unification of the existing methods of computing derivatives, including the adjoint method. As a result, the computational framework can automatically compute the coupled derivatives across the components, provided that the user computes partial derivatives within each component. The significance is that, while many challenges remain, MAUD lowers the entry barrier for large-scale optimization in problems with many heterogeneous components. This is especially true for multidisciplinary models that have a complex network of dependencies among the components, as in the examples described in Sec. 5.

MAUD makes three main assumptions about the components that are part of the multidisciplinary model. The first is that each component is continuous and differentiable. Since we require derivatives of the overall model for gradient-based optimization, each component must have continuous first derivatives. There are problems in which the optimization succeeds despite some discontinuities, so this is not a strict requirement. The second assumption is that each component efficiently computes the derivatives of its outputs with respect to its inputs. If derivatives are not available for a component, one would have to resort to finite differencing that component, which could degrade the overall efficiency. The third assumption is that the Jacobian matrix of the multidisciplinary model is invertible, i.e., the problem is well-posed, although the Jacobian of parts of the model can be singular in some cases. We discuss these assumptions in more detail in Sec. 4.5.

The remainder of the paper proceeds as follows. First, we provide background on existing computational frameworks, highlighting how the MAUD architecture differs from these frameworks. Second, we present the mathematical foundation of MAUD, which is the representation of a computational model as a system of equations and a unified equation for computing derivatives. Next, we describe the MAUD strategy for efficiently solving nonlinear and linear systems. Finally, we present two engineering design problems for which MAUD makes large-scale optimization possible despite the problem complexity: the optimization of a nanosatellite and the simultaneous optimization of the mission profiles and route allocation for commercial aircraft.

2 Background

This section provides the context for the MAUD architecture by reviewing the state of the art in computational frameworks, which are also known as problem-solving environments [2, 30, 39]. We categorize the existing frameworks as either flow-based frameworks (Sec. 2.1) or frameworks designed for the modular solution of PDEs (Sec. 2.2).

2.1 Flow-based frameworks

The existing computational frameworks that couple models from multiple disciplines adopt what we call a *flow-based architecture*. Their defining characteristic is that each component implements an explicit function that maps its inputs to its outputs. The component is considered a black box because its intermediate

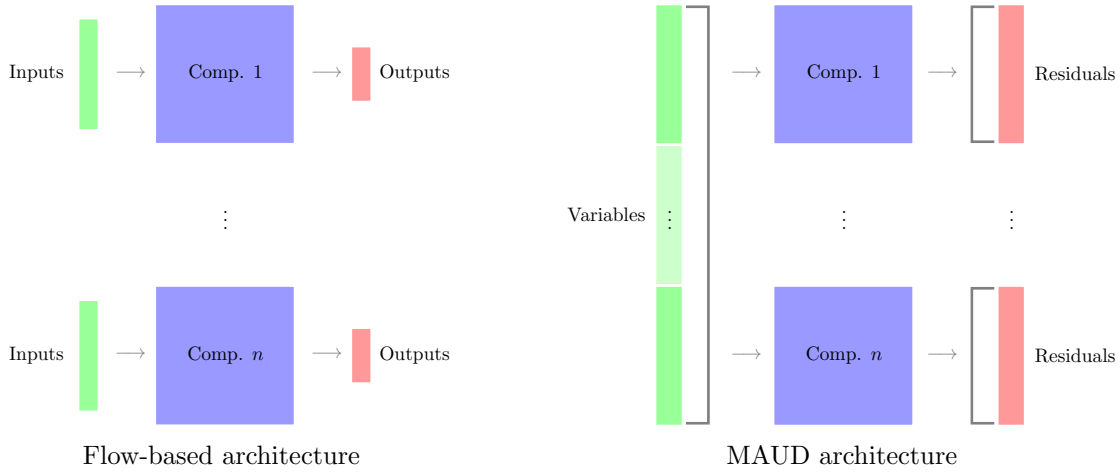


Figure 1: Comparison of the flow-based and MAUD architectures. In the flow-based architecture, each component is an explicit function, and in the MAUD architecture, each component is an implicit function.

variables and functions that are used in the evaluation of the explicit function are hidden from the framework. In contrast, a component in the MAUD architecture implicitly defines its variables by providing corresponding residuals. The MAUD architecture is more centralized because the variables defined by each component are concatenated into a single vector, and every component takes the entire vector as an argument, at least conceptually. The component implementations in the flow-based and MAUD architectures are compared in Fig. 1.

Flow-based frameworks are designed for components with a relatively small number of inputs and outputs. They are typically available as commercial software, e.g., Phoenix Integration's ModelCenter/CenterLink, Dassault Systèmes' Isight/SEE, Esteco's modeFRONTIER, TechnoSoft's AML suite, MathWorks' MATLAB/Simulink, Noesis Solutions' Optimus, Vanderplaats' VisualDOC [7], and SORCER [44]. These frameworks have four fundamental features in common: 1. a graphical user interface (GUI), 2. software libraries, 3. interfaces to external software, and 4. grid computing management.

The GUI provides a drag-and-drop interface for the construction of computational models from smaller components and includes tools for post-processing and visualization of the results. Software libraries are suites of reusable general-purpose components, e.g., optimizers, stochastic modeling tools, and surrogate models. The built-in interfaces to existing commercial computer-aided design and engineering software enable the framework to interact with other specialized software by making it one component within the framework. Finally, grid computing management typically involves a GUI that facilitates the handling of parallel computing tasks running on remote clusters.

Flow-based frameworks integrate multiple components that are not computationally costly to evaluate. These frameworks primarily focus on design of experiments (DOE) and gradient-free optimization, since these methods do not require derivatives. They are flexible: the data can be general objects, files, and databases rather than floating-point numbers. Flow-based frameworks also provide utilities such as automatic unit conversions and data recording. However, the downside is that they are not suitable for the efficient solution of large-scale optimization problems. In particular, they are usually limited to fixed-point iteration for coupled models and finite-difference methods for derivative computation.

Many noncommercial flow-based frameworks have been developed in the context of MDO. These frameworks are reviewed by Padula and Gillian [55], who note that the modularity, data handling, parallel processing, and user interface are the most important features to facilitate optimization based on simulations that involve multiple disciplines. An earlier survey [60] listed more detailed requirements for MDO frameworks, categorized into the framework's architectural design, problem construction, problem execution, and data access.

Some MDO frameworks use coupling variables converged by an optimizer instead of simultaneously converging the residuals of a multidisciplinary system. Examples include reconfigurable multidisciplinary

synthesis [5, 6] and the ψ specification language [65]. Like MAUD, both of these examples enable a decomposition-based approach, but they rely on the optimizer to solve the coupling between disciplines, limiting the scalability with respect to the number of disciplines and the coupling variables. Another MDO framework is pyMDO [52], which shares with MAUD the common objective of facilitating gradient-based optimization through the modular computation of derivatives using direct or adjoint analytic methods [47]. However, all the existing MDO frameworks have the same limitations as the commercial frameworks: they do not scale well with the total number of state variables. On the other hand, MAUD provides an infrastructure for efficient parallel data passing, hierarchically solves the nonlinear and linear systems, adopts a matrix-free approach, incorporates preconditioning, and has other characteristics that make it scalable.

The OpenMDAO framework [26] also has these features since it has recently adopted MAUD as its architecture for the core algorithms and data structures. Via the OpenMDAO framework, MAUD has been used by many universities and institutions for a wide variety of applications including aircraft design [21, 35, 57] using an open-source low-fidelity aerostructural optimization tool [36], satellite design [31], aircraft propulsion and trajectory optimization [28], structural topology optimization [14], electric aircraft design [19, 34], and wind turbine design [23, 54].

2.2 Frameworks for solving PDEs

Frameworks designed for solving PDEs are highly specialized, so they provide a high level of modularity and automation. These frameworks decompose the code into the PDE definition, boundary conditions, embedded problem-specific models (e.g., structural material properties), discretization scheme, element definition, and sometimes adaptive mesh refinement. Frameworks of this type are also motivated by the goal of modularity, but they differ fundamentally from flow-based computational frameworks or the MAUD architecture because they are designed to solve only PDEs. In contrast, the flow-based and MAUD architectures can integrate heterogeneous components, such as a series of explicit formulae, a surrogate model, or the solution of a PDE, all of which are treated the same way—as a set of components with a common interface.

Many frameworks for solving PDEs utilize the finite-element method. Finite-element frameworks provide some combination of the following features: automatic mesh creation for simple geometries, element libraries and quadrature routines, predefined sets of PDEs or a simple interface to define custom PDEs, boundary condition libraries, and linear solvers and preconditioners. Nearly all of these frameworks are written in C or C++, and they use an object-oriented paradigm in which the user instantiates objects for meshes, elements, and solvers.

Many finite-element frameworks use what we call *linear-algebra frameworks* for the underlying objects and solvers. These frameworks provide parallel vector and matrix objects, routines for parallel data scattering, parallel sparse matrix assembly, direct solvers, iterative solvers, and preconditioners. One example of a linear-algebra framework is the portable, extensible toolkit for scientific computation (PETSc) [8]. Several open-source finite-element frameworks rely on PETSc for their linear-algebra operations: MOOSE [22], OOFEM [56], libMesh [43], FEniCS [45], deal.II [9], PetIGA [15], and PHAML [53]. Trilinos [29] can be considered another linear-algebra framework, although it contains subpackages that make it a finite-element framework as well. SUNDANCE [46] and deal.II are examples of finite-element frameworks that use the linear algebra packages in Trilinos. COMSOL Multiphysics is one of the most widely used commercial finite-element frameworks, and it provides dedicated physics interfaces to various engineering applications. Not all of these frameworks are specialized to the solution of PDEs; however, they are all designed to solve systems of equations using finite-element discretizations. Since preconditioning is a critical and challenging part of solving PDEs, there are also frameworks that perform block preconditioning [12, 17] and multilevel preconditioning [20] in high-performance computing applications.

Other object-oriented PDE frameworks and libraries include OVERTURE [10], UG [1], Cactus [25], Roccom [37], MCS [66], UPACS [67], Tellis [11], and SIERRA [64]. This is by no means an exhaustive list, but it demonstrates the level of interest in and historical precedent for software frameworks that simplify the solution of PDEs using a modular, object-oriented approach. Many PDE frameworks are tailored to a specific field or application and offer a combination of features such as adaptive meshing, multigrid, multiphysics coupling, and adjoint computation.

Figure 2 illustrates how a computational framework and a finite-element framework can be used together to couple a finite-element analysis (FEA) to other components, whether those components are other finite-element models or different computations. This figure also shows the differences between integrating an

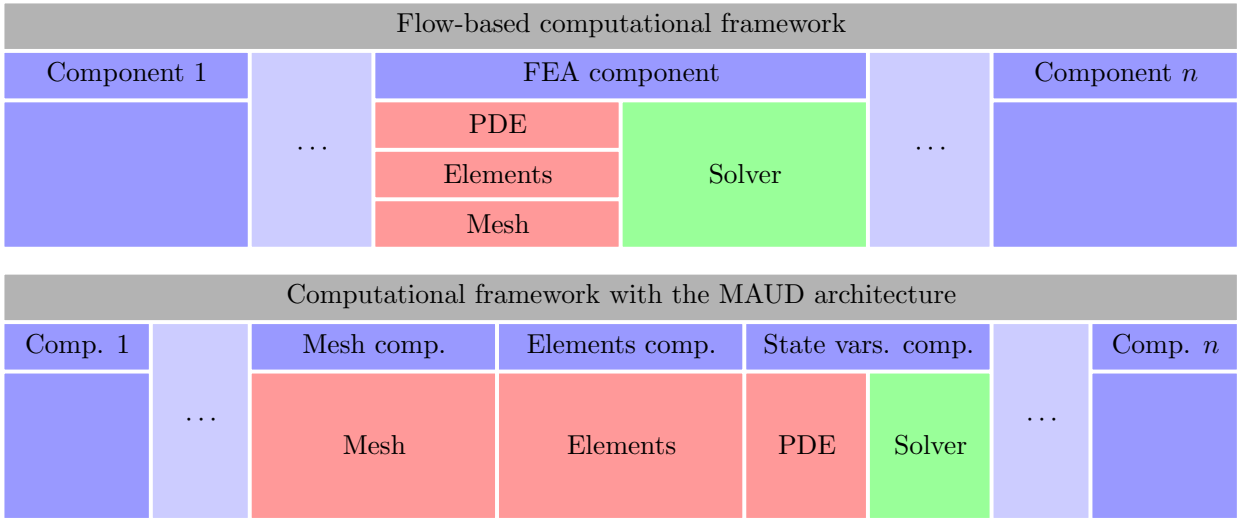


Figure 2: Implementation of a finite-element analysis model in flow-based and MAUD-based computational frameworks. Vertical stacking of blocks indicates containment. Gray blocks are part of the computational framework, blue blocks are components, red blocks are part of a finite-element framework, and green blocks are part of a linear algebra framework.

FEA component using a flow-based framework versus a MAUD-based framework. In the former, the entire FEA model is a single component and the various steps are nested: the FEA residual or matrix assembly calls the PDE definition class, which in turn calls the element class, and so on. In contrast, the MAUD architecture facilitates further modularization of the FEA so that the mesh computation is one component, the element shape function evaluations at the quadrature points are another, and the solution of the linear or nonlinear system to find the PDE state variables is another. The advantage is that it is simpler to compute partial derivatives for three small components than for one complex component. It is also easier to find the derivatives of this PDE's residuals with respect to variables from the other components, enabling Newton's method and the adjoint method to be used when coupling the FEA computation with other components or disciplines.

Many finite-element frameworks have common features with the MAUD architecture. Specifically, SUNDANCE, MOOSE, and TACS [40] also implement the adjoint method and use matrix-free operators. However, once again, these frameworks differ fundamentally from the MAUD architecture because they work only with computational models that solve a PDE using a finite-element discretization and do not provide any automation for computing derivatives involving general components with the efficiency of the adjoint method. Moreover, SUNDANCE takes the "differentiate-then-discretize" approach to the computation of derivatives. In contrast, the MAUD architecture applies the "discretize-then-differentiate" approach, since it views each component as an explicit or implicit function mapping to and from finite-dimensional vectors.

3 A unified theory

We now describe the mathematical formulation of the MAUD architecture and explain its benefits. The two main ideas are (1) to represent the multidisciplinary model as a single nonlinear system, and (2) to linearize this system in a way that unifies the methods for computing derivatives, including the adjoint method.

Section 3.1 defines the notation and form of a general numerical model, and Sec. 3.2 formulates the general model as a single system of algebraic equations. Sec. 3.3 shows that the derivatives of interest can be computed by solving a system of linear equations that unifies the existing methods for computing derivatives. A list of nomenclature is given in Appendix A.

3.1 General numerical model

We use the term *numerical model* to refer to the discretized variables and their explicit or implicit definitions, while *computational model* refers to the code that implements the numerical model. Fundamentally, numerical models capture the relationships between quantities; they determine the response of a set of variables to another set of given variables. Computing the response could involve multiple disciplines; i.e., the numerical model can integrate multiple submodels corresponding to different disciplines. Multidisciplinary models or problems are also characterized as multiphysics in some contexts, with the same meaning.

We frequently use the term *component* in the context of both the numerical model and the computational model. Each component computes one or more variables as a function of the variables of the remaining components. A discipline can be implemented as one or more components; therefore, even a single-discipline problem can have multiple components.

3.1.1 Input, output, and state variables

For our purposes, a *variable* represents a vector of a single type of physical or abstract quantity in a numerical model. In many settings, each individual scalar is referred to as a separate variable. However, in the current context, a group of scalars representing the same quantity—such as a vector comprised of temperature values at different time instances—is collectively referred to as a single variable. The only exception is design variables; we refer to each scalar varied by the optimizer as a design variable, to remain consistent with the terminology used in the optimization literature.

The given quantities in a numerical model can be considered to be *input variables*, which we represent as

$$x = (x_1, \ldots, x_m), \quad \text{where} \quad x_1 \in \mathbb{R}^{N_1^x}, \ldots, x_m \in \mathbb{R}^{N_m^x}, \tag{1}$$

so each variable x_k has size N_k^x. Input variables are independent variables whose values are set manually by a user or set automatically by an optimization algorithm. In the context of numerical optimization, design variables are a subset of the input variables.

The response quantities of the numerical model can be considered to be *output variables*, which we represent as

$$f = (f_1, \ldots, f_q), \quad \text{where} \quad f_1 \in \mathbb{R}^{N_1^f}, \ldots, f_q \in \mathbb{R}^{N_q^f}. \tag{2}$$

Output variables are computed while running the numerical model, and they represent quantities that we are interested in. In the context of numerical optimization, they include the objective and constraints.

In the process of computing the output variables, a numerical model might need to compute as an intermediate step another set of variables, the *state variables*, which we represent as

$$y = (y_1, \ldots, y_p), \quad \text{where} \quad y_1 \in \mathbb{R}^{N_1^y}, \ldots, y_p \in \mathbb{R}^{N_p^y}. \tag{3}$$

These are dependent variables that are defined by implicit or explicit functions of the design variables and other state variables.

All the variables in a numerical model can be classified into one of the categories described above: input variable, state variable, or output variable. As an example, in the finite-element analysis of a truss structure, the input variables would be the bar cross-sectional areas; the state variables would be the displacements; and the output variables could be the total weight and stresses. The total number of degrees of freedom is

$$N = N^x + N^y + N^f \quad \text{where} \quad N^x = \sum_{k=1}^{m} N_k^x, \quad N^y = \sum_{k=1}^{p} N_k^y, \quad N^f = \sum_{k=1}^{q} N_k^f, \tag{4}$$

where N^x is the total size of the m input variables, N^y is the total size of the p state variables, and N^f is the total size of the q output variables.

3.1.2 Explicit and implicit functions

State variables can be classified based on how they are computed. A state variable is either explicitly computed by evaluating functions or implicitly computed by converging residuals to zero. To simplify the presentation, we consider the two extreme cases where all state variables are of an explicit type and where all state variables are of an implicit type.

If all p state variables are individually of the explicit type, they are given by

$$\begin{aligned} y_1 &= \mathcal{Y}_1(x_1, \ldots, x_m, y_2, \ldots, y_p), \\ &\vdots \\ y_p &= \mathcal{Y}_p(x_1, \ldots, x_m, y_1, \ldots, y_{p-1}). \end{aligned} \tag{5}$$

We can also represent these functions with a single, concatenated function, $\mathcal{Y} = (\mathcal{Y}_1, \ldots, \mathcal{Y}_p)$, which is no longer an explicit computation.

We distinguish variables from functions by assigning lowercase and uppercase letters, respectively. This notation clearly distinguishes between the quantity and the function that computes that quantity, making it easier to define total and partial derivatives.

Another possibility is that each of the p state variables is computed implicitly by driving a residual function to zero through numerical solution. The residual function is $\mathcal{R} = (\mathcal{R}_1, \ldots, \mathcal{R}_p) = 0$, where $\mathcal{R}_k : \mathbb{R}^{N^x + N^y} \to \mathbb{R}^{N^y_k}$ and

$$\begin{aligned} \mathcal{R}_1(x_1, \ldots, x_m, y_1, \ldots, y_p) &= 0, \\ &\vdots \\ \mathcal{R}_p(x_1, \ldots, x_m, y_1, \ldots, y_p) &= 0. \end{aligned} \tag{6}$$

The multidisciplinary model can include a mixture of components defined by explicit functions (5) and components defined by residual equations (6). Therefore, in general, the p state variables are a mixture of explicit and implicit variables.

The q output variables are computed via the output function, $\mathcal{F} = (\mathcal{F}_1, \ldots, \mathcal{F}_q)$, where

$$\begin{aligned} f_1 &= \mathcal{F}_1(x_1, \ldots, x_m, y_1, \ldots, y_p), \\ &\vdots \\ f_q &= \mathcal{F}_q(x_1, \ldots, x_m, y_1, \ldots, y_p). \end{aligned} \tag{7}$$

We can assume without loss of generality that all the output variables are computed explicitly because if an implicitly defined state variable also happens to be a quantity of interest, we can define an output variable that is equal to that state variable.

Numerical models can always be represented with these three types of variables and functions. We can always divide the variables in the model into input variables, x, which are independent variables; state variables, y, which are computed explicitly by evaluating functions (5) or implicitly by driving residuals to zero (6) or a combination thereof; and output variables, f, which are computed by evaluating explicit functions (7).

In this view of variables and functions, multiple levels of decomposition are possible. In the least decomposed case, we could consider only input and output variables, by hiding all the state variables in a single black-box component. At the other extreme, we could consider every line of code with a variable assignment to define a new explicit state variable. In the following sections, we combine the three types of variables into one vector and derive a linear equation whose solution yields $\mathrm{d}f/\mathrm{d}x$, the total derivatives of the functions of interest with respect to the input variables. The level of model decomposition determines the derivative computation method.

3.2 A monolithic formulation for the numerical model

This section presents the reformulation of the numerical model defined by functions (5), (6), and (7) as a single system of algebraic equations.

The first step is to concatenate the set of variables into one vector,

$$u = (u_1, \ldots, u_n) = (x_1, \ldots, x_m, y_1, \ldots, y_p, f_1, \ldots, f_q), \tag{8}$$

where $n = m + p + q$. As previously discussed, each of these variables can in general be a vector. We denote the sizes of the variables in this concatenated vector as

$$(N_1, \ldots, N_n) = (N_1^x, \ldots, N_m^x, N_1^y, \ldots, N_p^y, N_1^f, \ldots, N_q^f). \tag{9}$$

Now we define the residual associated with variable u_k as the function $R_k : \mathbb{R}^N \to \mathbb{R}^{N_k}$ for $k = 1, \ldots, n$. These residuals have different forms depending on the type of variable. For the input variables, the residuals are defined as

$$R_k(u) = x_k - x_k^*, \quad \forall k = 1, \ldots, m, \tag{10}$$

where $x_k^* \in \mathbb{R}^{N_k^x}$ is the value of input variable x_k at the point where the numerical model is being evaluated, for $k = 1, \ldots, m$. For instance, if we want to run our model at $x_1 = 3$, then we would have $R_1(u) = x_1 - 3$.

For the state variables, the residuals are defined differently, depending on whether they are determined explicitly or implicitly:

$$R_{m+k}(u) = \begin{cases} y_k - \mathcal{Y}_k(x_1, \ldots, x_m, y_2, \ldots, y_p), & \text{if } y_k \text{ is explicitly defined} \\ -\mathcal{R}_k(x_1, \ldots, x_m, y_1, \ldots, y_p), & \text{if } y_k \text{ is implicitly defined} \end{cases}, \forall k = 1, \ldots, p. \tag{11}$$

For example, an explicit state variable y_2 that is defined as the product of x_1 and y_1 would have the residual, $R_{m+2}(u) = y_2 - x_1 y_1$. We have added a minus sign in the residuals for the implicit state variables because this yields more intuitive formulae for computing the derivatives (Sec. 3.3). Albersmeyer and Diehl [3] introduced residuals for explicit variables as the difference between the variable and the function output in their "lifted Newton method." They start with what is already a nonlinear system and modify Newton's method to improve the convergence properties. In contrast, we take heterogeneous computational models and turn them into a nonlinear system while ensuring that the existing solution methods can be used in the new formulation.

Finally, we use the explicit function residuals for the output variables

$$R_{m+p+k}(u) = f_k - \mathcal{F}_k(x_1, \ldots, x_m, y_1, \ldots, y_p), \quad \forall k = 1, \ldots, q. \tag{12}$$

Concatenating all the residuals as $R = (R_1, \ldots, R_n)$, we can write the complete numerical model as the following algebraic system of equations:

$$\boxed{\left.\begin{array}{c} R_1(u_1, \ldots, u_n) = 0 \\ \vdots \\ R_n(u_1, \ldots, u_n) = 0 \end{array}\right\} \Leftrightarrow R(u) = 0,} \tag{13}$$

where $R : \mathbb{R}^n \to \mathbb{R}^n$. This represents a unified formulation for any numerical model, which we call the *fundamental system*. The vector u^* that solves the fundamental system (13) is also a solution of the numerical model defined by Eqs. (5), (6), and (7).

3.3 Computation of multidisciplinary derivatives

In the context of optimization, the design variables that the optimizer varies are a subset of the numerical model input variables x, and the objective and constraint functions provided to the optimizer are a subset of the output variables f. Thus, the derivatives of interest are the derivatives of the output variables with respect to the input variables, i.e., the Jacobian matrix $\mathrm{d}f/\mathrm{d}x$.

In the next section we will see that we can derive an equation that unifies all derivative computation methods starting from the fundamental system (13), where the resulting method depends on the choice of the state variables. At one extreme, we can consider the whole system as a single component that does not expose any state variables, whose linearization results in a black-box differentiation using, for instance, the finite-difference method. At the other extreme, we can consider every line of code in a computational model to be a component with the associated variable as a state variable, which is equivalent to automatic differentiation using source-code transformation [50].

3.3.1 Connection between the fundamental system and analytic methods

Before we present the equation unifying all the derivative computation methods, we show the connection between the fundamental system and analytic methods. Analytic methods consist of direct or adjoint methods, which are applicable when residuals are available for the state variables. Analytic methods can be applied to accurately compute derivatives of coupled systems. We show this connection because the analytic adjoint method is particularly powerful in that it computes the derivatives of a function of interest with respect to all the input variables at a cost that is of the same order as the cost of running the model.

Assuming that all the model state variables are implicitly defined, and using the numerical definitions in the preceding sections, we can represent the model evaluated at $x = x^*$ as

$$x = x^* \quad \text{with} \quad \mathcal{R}(x, y) = 0 \quad \text{and} \quad f = \mathcal{F}(x, y). \tag{14}$$

The fundamental system then has the form

$$u = \begin{pmatrix} x \\ y \\ f \end{pmatrix} \quad \text{and} \quad R(u) = \begin{pmatrix} x - x^* \\ -\mathcal{R}(x, y) \\ f - \mathcal{F}(x, y) \end{pmatrix}, \tag{15}$$

and the numerical model can be encapsulated in the function

$$\mathcal{G} : x \mapsto \mathcal{F}(x, \mathcal{Y}(x)), \tag{16}$$

which maps the inputs x to the outputs f. The derivatives of interest are then $\partial\mathcal{G}/\partial x$. The following proposition shows how $\partial\mathcal{G}/\partial x$ can be computed at a point x^* using the fundamental system defined by Eq. (15).

Proposition 1. *Let inputs x, states y, and outputs f be defined by Eq. (14), and let $\mathcal{G}$ map the inputs to the outputs as defined in Eq. (16). Let R and u be given by Eq. (15). If $\partial R/\partial u$ is invertible and the inverse is defined as*

$$\frac{\partial R}{\partial u}^{-1} = \begin{bmatrix} A^{xx} & A^{xy} & A^{xf} \\ A^{yx} & A^{yy} & A^{yf} \\ A^{fx} & A^{fy} & A^{ff} \end{bmatrix}, \tag{17}$$

then the following identity holds:

$$\frac{\partial\mathcal{G}}{\partial x} = A^{fx}, \tag{18}$$

where $\partial R/\partial u$ is evaluated at u^* satisfying $R(u^*) = 0$.

Proof. By construction, we have

$$\begin{bmatrix} \mathcal{I} & 0 & 0 \\ -\dfrac{\partial\mathcal{R}}{\partial x} & -\dfrac{\partial\mathcal{R}}{\partial y} & 0 \\ -\dfrac{\partial\mathcal{F}}{\partial x} & -\dfrac{\partial\mathcal{F}}{\partial y} & \mathcal{I} \end{bmatrix} \begin{bmatrix} A^{xx} & A^{xy} & A^{xf} \\ A^{yx} & A^{yy} & A^{yf} \\ A^{fx} & A^{fy} & A^{ff} \end{bmatrix} = \begin{bmatrix} \mathcal{I} & 0 & 0 \\ 0 & \mathcal{I} & 0 \\ 0 & 0 & \mathcal{I} \end{bmatrix}, \tag{19}$$

where the first matrix in the above is $\partial R/\partial u$. Block-forward substitution to solve for the first block-column yields

$$\begin{bmatrix} A^{xx} \\ A^{yx} \\ A^{fx} \end{bmatrix} = \begin{bmatrix} \mathcal{I} \\ -\dfrac{\partial\mathcal{R}}{\partial y}^{-1} \dfrac{\partial\mathcal{R}}{\partial x} \\ \dfrac{\partial\mathcal{F}}{\partial x} - \dfrac{\partial\mathcal{F}}{\partial y} \dfrac{\partial\mathcal{R}}{\partial y}^{-1} \dfrac{\partial\mathcal{R}}{\partial x} \end{bmatrix}. \tag{20}$$

Now, $\mathcal{G}$ is a composition of functions mapping $x \mapsto (x, \mathcal{Y}(x))$ and $(x, y) \mapsto \mathcal{F}(x, y)$, so applying the chain rule yields

$$\frac{\partial \mathcal{G}}{\partial x} = \begin{bmatrix} \dfrac{\partial \mathcal{F}}{\partial x} & \dfrac{\partial \mathcal{F}}{\partial y} \end{bmatrix} \begin{bmatrix} \mathcal{I} \\ \dfrac{\partial \mathcal{Y}}{\partial x} \end{bmatrix}. \tag{21}$$

Using the implicit function theorem, we can write

$$\frac{\partial \mathcal{Y}}{\partial x} = -\left(\frac{\partial \mathcal{R}}{\partial y}\right)^{-1} \frac{\partial \mathcal{R}}{\partial x}. \tag{22}$$

Combining these two equations yields

$$\frac{\partial \mathcal{G}}{\partial x} = \frac{\partial \mathcal{F}}{\partial x} - \frac{\partial \mathcal{F}}{\partial y} \left(\frac{\partial \mathcal{R}}{\partial y}\right)^{-1} \frac{\partial \mathcal{R}}{\partial x}. \tag{23}$$

Therefore, we can see from Eqs. (20) and (23) that

$$\frac{\partial \mathcal{G}}{\partial x} = A^{fx}, \tag{24}$$

as required. □

Note that $\partial \mathcal{R}/\partial y$ must be invertible for a well-posed model. For instance, in a structural finite-element model, a singular $\partial \mathcal{R}/\partial y$ would indicate that there is an infinite number of solutions, e.g., because of insufficient boundary conditions. In a multidisciplinary context where R and y include multiple models, a singular matrix would indicate an ill-posed problem with no solution or an infinite number of solutions, or a physical instability, e.g., divergence in the fluid-structure interaction. It follows from the properties of determinants of block matrices that the invertibility of $\partial \mathcal{R}/\partial y$ implies the invertibility of $\partial R/\partial u$.

Remark 1. *The computation of the A^{fx} block of the $[\partial R/\partial u]^{-1}$ matrix turns out to be mathematically equivalent to the adjoint method. In an* ad hoc *implementation of the adjoint method, there is no advantage in computing the $\mathrm{d}f/\mathrm{d}x$ derivatives via $\partial R/\partial u$ instead of the traditional equations. In fact, if poorly implemented, the $\partial R/\partial u$ approach may be less efficient.*

The advantage of applying the adjoint method via $\partial R/\partial u$ is that this method is much more general. As we will see in the following sections, solving the system with $\partial R/\partial u$ generalizes not just the adjoint method but also the direct method, the coupled adjoint method, chain rule, and the global sensitivity equations [61]. The method used depends on what type of model R and u represent, i.e., whether the state variables are explicit, implicit, or mixed; and whether there are two-way dependencies among them. Therefore, a software framework can adapt and implement each of these derivative computation methods without requiring dedicated code implementations for each method, given that the user implements the partial derivative computation using the finite-difference method, the complex-step method [48, 63], automatic differentiation [27], or by differentiating by hand.

3.3.2 Definition of a total derivative

Before deriving the unifying derivatives equation, we give a precise definition of the total derivative. To start, we introduce $r \in \mathbb{R}^N$ as the value of the residual vector, as we will need it later.

As mentioned previously, we distinguish variables from functions by assigning lowercase and uppercase letters, respectively. We consider a total derivative to be the derivative of a *variable* with respect to another *variable*, e.g., $\mathrm{d}y_1/\mathrm{d}x_1$, whereas a partial derivative is the derivative of a *function* with respect to the *argument* of that function, e.g., $\partial \mathcal{Y}_1/\partial x_1$. Therefore, a total derivative captures the coupling between components, while a partial derivative is local to a component.

The concept of a total derivative is used in many settings with various meanings, but in the context of direct and adjoint analytic methods, the total derivative is written as

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} + \frac{\partial F}{\partial y} \frac{\mathrm{d}y}{\mathrm{d}x}. \tag{25}$$

The $\mathrm{d}f/\mathrm{d}x$ term takes into account both the explicit dependence of F on the argument x and the indirect dependence of F on the state variables y through the solution of a numerical model, for example.

Suppose that R^{-1} exists and is differentiable on a neighborhood of $r = 0$. The Jacobian $\partial(R^{-1})/\partial r$ has a similar meaning as the total derivative because the (i, j)th entry of the matrix $\partial(R^{-1})/\partial r$ captures the dependence of the ith component of u on the jth component of r both explicitly and indirectly via the other components of u. This motivates the following definition of the total derivative.

Definition 1. *Given the algebraic system of equations $R(u) = 0$, assume that $\partial R/\partial u$ is invertible at the solution of this system. The matrix of total derivatives $\mathrm{d}u/\mathrm{d}r$ is defined to be*

$$\frac{\mathrm{d}u}{\mathrm{d}r} = \frac{\partial(R^{-1})}{\partial r}, \tag{26}$$

where $\partial(R^{-1})/\partial r$ is evaluated at $r = 0$.

This defines the total derivative in the context of a model that contains multiple coupled components. Equation (26) states that *the total derivative matrix is the Jacobian of partial derivatives of the inverse of the function that represents the model as a nonlinear system* (13).

We break down the total derivative matrix $\mathrm{d}u/\mathrm{d}r$ into the blocks corresponding to the different types of variables. We write the Jacobian blocks corresponding to the k^{th} input variable as

$$\frac{\mathrm{d}u_i}{\mathrm{d}x_k} := \left[\frac{\mathrm{d}u}{\mathrm{d}r}\right]_{i,k}, \tag{27}$$

for $1 \le k \le m$ and $1 \le i \le n$. The total derivatives related to the state variables depend on whether those variables are explicitly or implicitly defined. If the k^{th} state variable is explicit, we write

$$\frac{\mathrm{d}u_i}{\mathrm{d}y_k} := \left[\frac{\mathrm{d}u}{\mathrm{d}r}\right]_{i,m+k}, \tag{28}$$

and if the k^{th} state variable is implicit, we write

$$\frac{\mathrm{d}u_i}{\mathrm{d}r_{y_k}} := \left[\frac{\mathrm{d}u}{\mathrm{d}r}\right]_{i,m+k}, \tag{29}$$

for $1 \le k \le p$ and $1 \le i \le n$. The blocks corresponding to the k^{th} output variable are

$$\frac{\mathrm{d}u_i}{\mathrm{d}f_k} := \left[\frac{\mathrm{d}u}{\mathrm{d}r}\right]_{i,m+p+k}, \tag{30}$$

for $1 \le k \le q$ and $1 \le i \le n$.

In summary, a total derivative represents how changes in one *variable* result in changes in another *variable* because of the constraints imposed by the residual function R. In contrast, a partial derivative represents how the output of a *function* changes due to changes in an *argument* of that function. Therefore, the total derivative is a property of a *numerical model*, while a partial derivative is a property of a *component* in the MAUD context.

Since the partial derivative has a precise and more familiar meaning, we can understand and interpret the total derivative via the partial derivative with which it is defined. In the representation of the numerical model as a nonlinear system, the total derivative with respect to an input variable is simply the partial derivative of R^{-1} with respect to the residual for the input variable. In other words, if we perturb the right-hand side of the nonlinear system (13) in a row corresponding to an input variable, we would find that the resulting normalized change in the u vector is the vector of total derivatives with respect to that input variable, in the sense of Eq. (25).

3.3.3 The unifying equation for derivative computation

We can now present the unification of methods for computing total derivatives. Assuming $\partial R/\partial u$ is invertible at our point of interest, $r = 0$, the inverse function theorem guarantees the existence of a locally defined inverse function $R^{-1} : r \mapsto u | R(u) = r$ on a neighborhood of $r = 0$ that satisfies

$$\frac{\partial (R^{-1})}{\partial r} = \left[\frac{\partial R}{\partial u}\right]^{-1} . \tag{31}$$

Combining Eqs. (26) and (31), we obtain

$$\frac{\mathrm{d}u}{\mathrm{d}r} = \left[\frac{\partial R}{\partial u}\right]^{-1} . \tag{32}$$

That is, $\mathrm{d}u/\mathrm{d}r$ is equal to the inverse of the Jacobian of the system. Therefore, we can write

$$\frac{\partial R}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}r} = \mathcal{I} = \frac{\partial R}{\partial u}^{T}\frac{\mathrm{d}u}{\mathrm{d}r}^{T} , \tag{33}$$

which we call the *unifying derivative equation*. Recalling that a matrix and its inverse commute, commuting and transposing the leftmost expression leads to the rightmost expression. The left and right equalities represent the *forward mode* and the *reverse mode*, respectively, in the terminology of automatic differentiation. The unifying derivative equation (33) is presented by Martins and Hwang [50] in different notation using an alternative interpretation and derivation. That paper also shows in detail how all differentiation methods can be derived from this equation.

The total derivative Jacobian ($\mathrm{d}u/\mathrm{d}r$) is the matrix of unknowns in the unifying derivative equation (33), and it contains the derivatives we ultimately want to compute ($\mathrm{d}f/\mathrm{d}x$). However, $\mathrm{d}u/\mathrm{d}r$ also contains various other derivatives that need to be computed in the process of obtaining $\mathrm{d}f/\mathrm{d}x$, which we now detail. As previously defined, the inverse of the residual function is $R^{-1} : r \mapsto u | R(u) = r$, where $u = (x, y, f)$ and $r = (r_x, r_y, r_f)$. If all the state variables are implicitly defined, we have

$$\begin{aligned} r &= R(u) \\ \begin{pmatrix} r_x \\ r_y \\ r_f \end{pmatrix} &= \begin{pmatrix} x - x^* \\ -\mathcal{R}(x, y) \\ f - \mathcal{F}(x, y)) \end{pmatrix} \end{aligned} \implies \frac{\mathrm{d}u}{\mathrm{d}r} = \frac{\partial (R^{-1})}{\partial r} = \begin{bmatrix} \frac{\mathrm{d}x}{\mathrm{d}r_x} & \frac{\mathrm{d}x}{\mathrm{d}r_y} & \frac{\mathrm{d}x}{\mathrm{d}r_f} \\ \frac{\mathrm{d}y}{\mathrm{d}r_x} & \frac{\mathrm{d}y}{\mathrm{d}r_y} & \frac{\mathrm{d}y}{\mathrm{d}r_f} \\ \frac{\mathrm{d}f}{\mathrm{d}r_x} & \frac{\mathrm{d}f}{\mathrm{d}r_y} & \frac{\mathrm{d}f}{\mathrm{d}r_f} \end{bmatrix} = \begin{bmatrix} \mathcal{I} & 0 & 0 \\ \frac{\mathrm{d}y}{\mathrm{d}x} & \frac{\mathrm{d}y}{\mathrm{d}r_y} & 0 \\ \frac{\mathrm{d}f}{\mathrm{d}x} & \frac{\mathrm{d}f}{\mathrm{d}r_y} & \mathcal{I} \end{bmatrix} , \tag{34}$$

where the last equality comes from Eqs. (27), (29), (30).

In the first column, $\mathrm{d}x/\mathrm{d}r_x$ is the identity matrix, because $x = r_x + x^*$ when we evaluate R^{-1} and x^* is constant. It then follows that $\mathrm{d}y/\mathrm{d}r_x = \mathrm{d}y/\mathrm{d}x$ and $\mathrm{d}f/\mathrm{d}r_x = \mathrm{d}f/\mathrm{d}x$. These are total derivatives because they capture the *implicit* effect of x on y and f via the condition that the residuals r_y must always be zero.

In the second column, $\mathrm{d}x/\mathrm{d}r_y$ is zero because x depends only on r_x and not on r_y. The $\mathrm{d}y/\mathrm{d}r_y$ and $\mathrm{d}f/\mathrm{d}r_y$ blocks capture the first-order perturbations to y and f given that we solve $\mathcal{R}(x, y) = r_y$ instead of $\mathcal{R}(x, y) = 0$ and then update $f = \mathcal{F}(x, y)$ with the perturbed y.

In the third column, $\mathrm{d}x/\mathrm{d}r_f = 0$ and $\mathrm{d}y/\mathrm{d}r_f = 0$ because x and y do not depend on r_f. It follows that $\mathrm{d}f/\mathrm{d}r_f$ is the identity matrix because x and y are not influenced by a perturbation to r_f, so f is equal to r_f plus a constant.

The block matrices in the unifying derivative equation are shown in Fig. 3, including the blocks in $\mathrm{d}u/\mathrm{d}r$ that we have just explained. The derivatives of interest are $\mathrm{d}f/\mathrm{d}x$, which is a block matrix in $\mathrm{d}u/\mathrm{d}r$, as we have just shown. Computing $\mathrm{d}f/\mathrm{d}x$ involves solving a linear system with multiple right-hand sides. Depending on the relative sizes of f and x, it might be more efficient to solve the forward system (left-side equality) or the reverse system (right-side equality).

If the state variables are explicitly defined, the terms $\mathrm{d}y/\mathrm{d}r_y$ and $\mathrm{d}f/\mathrm{d}r_y$ are more meaningful. We illustrate this by first considering the simplest situation, where each state variable depends only on the

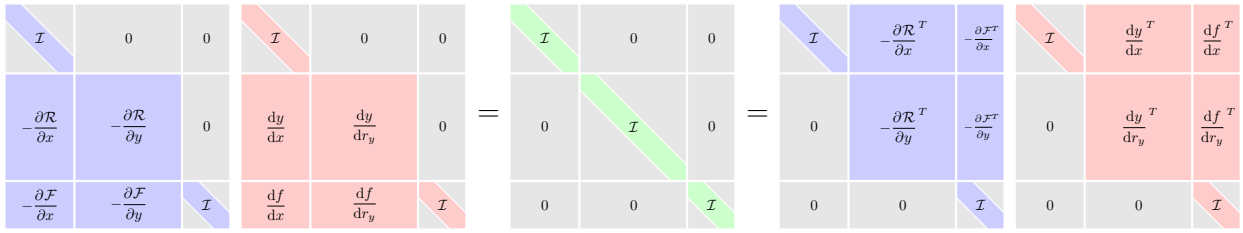


Figure 3: Block matrix structure of the unifying derivative equation (33).

previous state variables, so that $y_k = \mathcal{Y}_k(x, y_1, \ldots, y_{k-1})$. Then, we have, from Eq. (28),

$$\frac{\mathrm{d}y}{\mathrm{d}r_y} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ \dfrac{\mathrm{d}y_2}{\mathrm{d}y_1} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ \dfrac{\mathrm{d}y_p}{\mathrm{d}y_1} & \cdots & \dfrac{\mathrm{d}y_p}{\mathrm{d}y_{p-1}} & 1 \end{bmatrix} \quad \text{and} \quad \frac{\mathrm{d}f}{\mathrm{d}r_y} = \begin{bmatrix} \dfrac{\mathrm{d}f_1}{\mathrm{d}y_1} & \dfrac{\mathrm{d}f_1}{\mathrm{d}y_2} & \cdots & \dfrac{\mathrm{d}f_1}{\mathrm{d}y_{p-1}} & \dfrac{\mathrm{d}f_1}{\mathrm{d}y_p} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \dfrac{\mathrm{d}f_q}{\mathrm{d}y_1} & \dfrac{\mathrm{d}f_q}{\mathrm{d}y_2} & \cdots & \dfrac{\mathrm{d}f_q}{\mathrm{d}y_{p-1}} & \dfrac{\mathrm{d}f_q}{\mathrm{d}y_p} \end{bmatrix}. \tag{35}$$

In this case, the total derivatives are simply computed using the chain rule to multiply and add combinations of partial derivatives to find the derivatives of the composite functions. The general case extends to include implicit variables, coupling between the state variables, or both. In the general case, the $\mathrm{d}y/\mathrm{d}r_y$ or $\mathrm{d}f/\mathrm{d}r_y$ derivatives still have an important meaning: they represent the sensitivity of the state or output variable of interest to changes in the residual variable in the denominator.

4 Algorithmic approach and software design

The overall goal of the MAUD architecture is to facilitate two tasks: the evaluation of a computational model with multiple components and the efficient computation of the model's coupled derivatives across the various components. The significance of the mathematical formulation presented in Sec. 3 is that the different algorithms for performing these two tasks are unified in a way that simplifies the implementation of the computational framework. Specifically, the task of evaluating a coupled computational model reduces to solving a system of nonlinear algebraic equations, and the task of computing derivatives reduces to solving a system of linear equations. To perform these tasks and achieve the overall goal, MAUD assembles and solves four types of systems:

1. Numerical model (nonlinear system)
$$R(u) = 0$$
2. Newton step (linear system)
$$\frac{\partial R}{\partial u}\Delta u = -r$$
3. Forward differentiation (linear system with multiple right-hand-side vectors)
$$\frac{\partial R}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}r} = \mathcal{I}$$
4. Reverse differentiation (linear system with multiple right-hand-side vectors)
$$\frac{\partial R}{\partial u}^T \frac{\mathrm{d}u}{\mathrm{d}r}^T = \mathcal{I}$$

Naively solving these systems of equations could incur large penalties in both computational time and memory usage, in part because MAUD assembles a system of equations that is larger than necessary when compared to an *ad hoc* implementation without a framework. For instance, applying a Newton–Krylov solver to the fundamental system (13) would result in the allocation of larger-than-necessary vectors that contain all the input, state, and output variables, even though only the state variables are coupled. However, this potentiall pitfall is circumvented by adopting a recursive hierarchical solver architecture that allows for the Newton–Krylov solver to be used only on the coupled state variables even though the fundamental system (13) is still formulated.

To illustrate by example, Fig. 4 shows several problems with simple model structures. In each case, the problem structure (transpose of the Jacobian) is shown as a block matrix, and above it is the corresponding hierarchical decomposition that would be the most efficient. Each bar represents an intermediate system in the numerical model, and a different type of solver would be appropriate for each color. For example, a red bar indicates that its children systems have only feed-forward coupling, so a single iteration of nonlinear (linear) block Gauss–Seidel would be the best nonlinear (linear) solver. Systems with green bars would be best solved with a single iteration of block Jacobi, since there is no coupling among the children. MAUD does not automatically select the right combination of solvers, but it provides the infrastructure for users to decompose and solve their models hierarchically, selecting the desired solver at each node. This section describes this infrastructure.

Section 4.1 explains the hierarchical decomposition of the fundamental system (13) into smaller systems of algebraic equations. Section 4.2 describes the object-oriented implementation of this hierarchy of algebraic systems. Section 4.3 presents the hierarchical solution strategy, and Sec. 4.4 describes the data structures. Finally, Sec. 4.5 discusses the assumptions and limitations of the MAUD architecture.

4.1 Mathematical decomposition of the algebraic system

To solve the fundamental system (13) efficiently, we partition the set of unknowns to form smaller systems of equations. Moreover, we apply this partitioning strategy recursively, resulting in a hierarchical decomposition of the fundamental system (13).

We introduce a smaller algebraic system, an *intermediate system*, for a given subset of the residuals and unknowns of the fundamental system. We define the index set

$$S = \{i+1, \ldots, j\}, \quad 0 \leq i < j \leq n, \tag{36}$$

to represent the indices of the variables that make up the unknowns for this smaller algebraic system. Without loss of generality we have chosen these indices to be contiguous, since the variables can always be reordered. The residual function for this intermediate system is

$$R_S : \mathbb{R}^N \to \mathbb{R}^{N_{i+1}} \times \cdots \times \mathbb{R}^{N_j}, \tag{37}$$

where we have concatenated the residual functions corresponding to the indices in S.

This subset of residuals is in general a function of all the unknowns u, but only some of these unknowns are implicitly defined by R_S. Therefore, we divide u into the intermediate system's *unknown vector*, $u_S = (u_{i+1}, \ldots, u_j)$, and the rest of the unknowns, the *parameter vector* $p_S = (u_1, \ldots, u_i, u_{j+1}, \ldots, u_n)$. Then, the intermediate system implicitly defines u_S as a function of p_S:

$$\left.\begin{aligned} R_{i+1}(u_1, \ldots, u_i, u_{i+1}, \ldots, u_j, u_{j+1}, \ldots, u_n) &= 0 \\ &\vdots \\ R_j(\underbrace{u_1, \ldots, u_i}_{p_S}, \underbrace{u_{i+1}, \ldots, u_j}_{u_S}, \underbrace{u_{j+1}, \ldots, u_n}_{p_S}) &= 0 \end{aligned}\right\} \Leftrightarrow R_S(p_S, u_S) = 0. \tag{38}$$

The significance of this definition is that intermediate systems are formulated and solved in the process of solving the fundamental system. Solving an intermediate system that has only one variable or a group of related variables in its unknown vector is analogous to executing a component in the traditional view of a framework, i.e., computing the component's outputs (u_S) given the values of the inputs (subset of p_S). This formulation enables a hierarchical decomposition of all the variables in the computational model by allowing the definition of intermediate systems that group together other intermediate systems that may correspond to components.

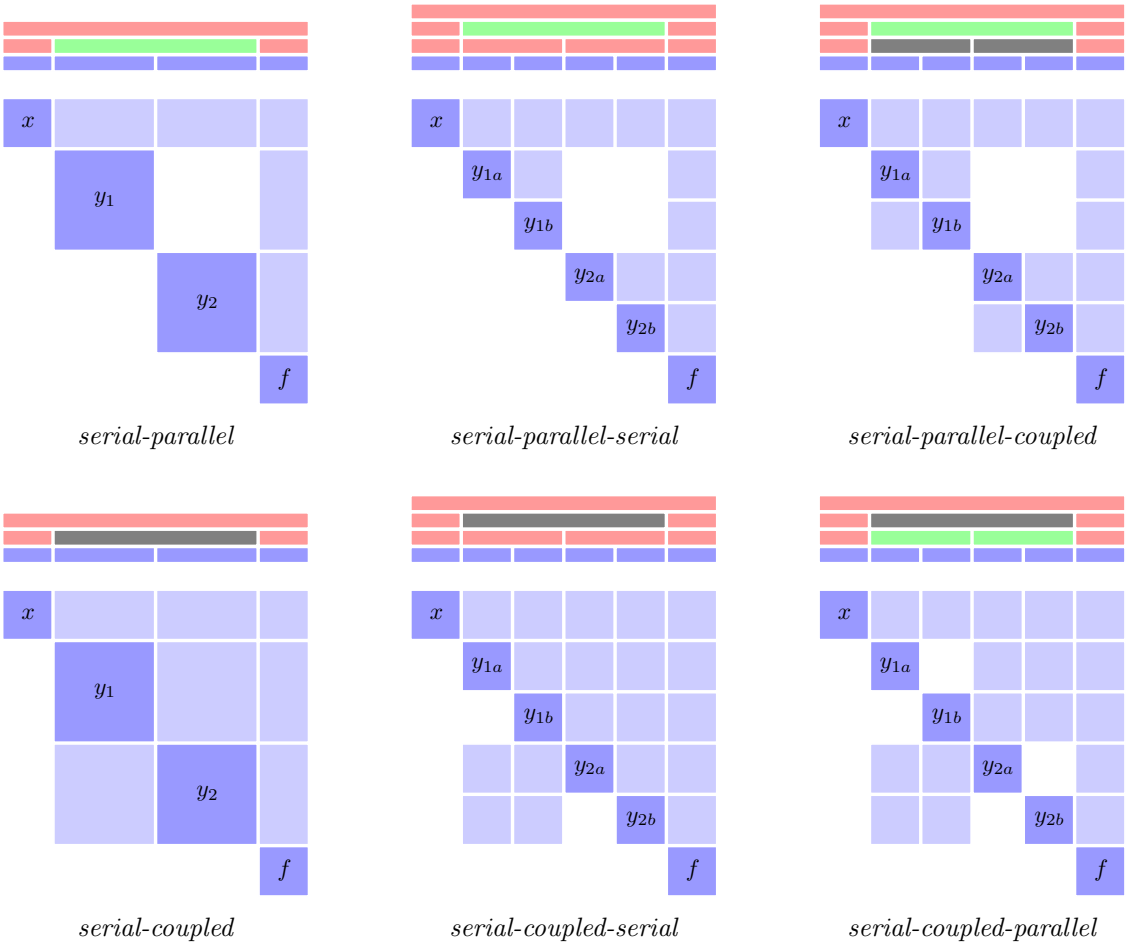


Figure 4: Six sample problem structures and the corresponding MAUD hierarchical decompositions. The problem structure is shown using the design structure matrix, which transposes the structure of the Jacobian matrix, so that feed-forward dependencies are above the diagonal. The hierarchical decompositions are shown above the matrices, where the components are shown in blue, serial groups in red, parallel groups in green, and coupled groups (i.e., those containing feedback loops) in gray. Each blue, red, green, or gray bar corresponds to an intermediate system in the numerical model.

4.2 Object-oriented software design

In the numerical model, the fundamental system includes the full set of n variables and corresponds to the root of the hierarchy tree. It contains a group of intermediate systems that together partition the n variables and form the second level of the tree. Each of these intermediate systems can contain other intermediate systems forming a third level, and so on, until we reach intermediate systems that correspond to components and form the leaves of the tree.

The hierarchy in the computational model mirrors this hierarchy in the numerical model, as shown in Fig. 5. A leaf in the hierarchy tree is an instance of the *Component* class (or a subclass thereof). Components are the user-implemented units of code that own some of the variables in the overall model. Components are grouped together by instances of the *Assembly* class (or a subclass thereof), which can in turn be grouped together by other assemblies. Therefore, all nodes of the hierarchy tree that are not leaves are assemblies.

The class inheritance diagram is shown in Fig. 6 for the various system classes. Both the *Assembly* and *Component* classes inherit from the *System* class, which can be interpreted as any intermediate system in the numerical model.

The first subclass of the *System* class is the *Component* class, and it has three derived classes reflecting

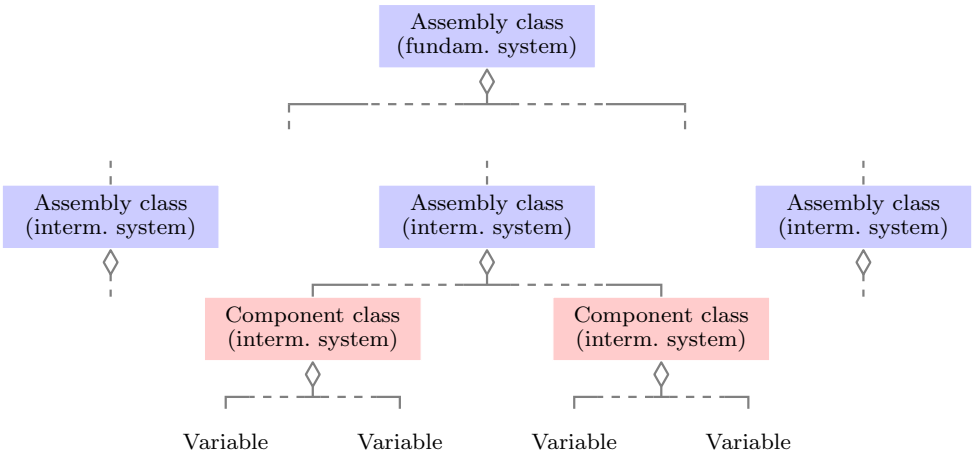


Figure 5: Containment relationships in the computational model hierarchy tree. The top-level assembly corresponds to the fundamental system, and all other assemblies and components correspond to intermediate systems.

whether the system contains independent, explicitly defined, or implicitly defined variables. The three derived classes are *IndepComponent*, *ExplicitComponent*, and *ImplicitComponent*. The user implements each component in the computational framework as a class inheriting from one of these three classes and directly implements operations such as the residual function computation. Each component contains the variables that form the unknown vector in the component's intermediate system. During the initialization, the component must declare its variables as well as its arguments, which are external variables that affect the component's variables.

MAUD enables parallel computation both within a given variable (by distributing the variable over multiple processors) and across variables (by assigning different variables to different processors). Parallelization within a given variable takes place when a component is internally parallelized, and different parts of the variable (a vector) are allocated to different processors. If another component takes entries from this variable as its input, the parallel data transfers that occur during execution abstract the mechanics of determining to which processors the desired entries are allocated and of the necessary interprocessor communication. In the lightweight Python implementation of MAUD described in Sec. 5.1, the parallel data transfers are implemented using PETSc [8] and petsc4py [18] for the underlying mechanics of the scatter operations. Appendix B details the order in which the transfers are called, and the variables for which the transfers take place.

The second subclass of the *System* class is the *Assembly* class, and it is how MAUD supports parallelism across variables. It groups other *System* objects together, so their operations recursively call those of their children. Moreover, they transfer data that is potentially distributed across multiple processors. The *Assembly* class has two derived classes that handle parallelism in different ways. In the hierarchy tree in Fig. 5, the root assembly is stored and run on all the processors. If a given system is a *SerialAssembly*, it passes all of its processors to the systems it contains, and it runs its recursive operations sequentially for those systems. If a given system is a *ParallelAssembly*, it partitions its group of processors among the systems it contains, and it runs its recursive operations concurrently among those systems. This applies to all *Assembly* objects, and so each system is assigned a subset of its parent system's processors.

4.3 Hierarchical solution strategy

As explained in Sec. 4.2, the model is represented using a hierarchy of *System* class instances that contain each other. Mathematically, each instance corresponds to an intermediate system

$$R_S(p_S, u_S) = 0, \tag{39}$$

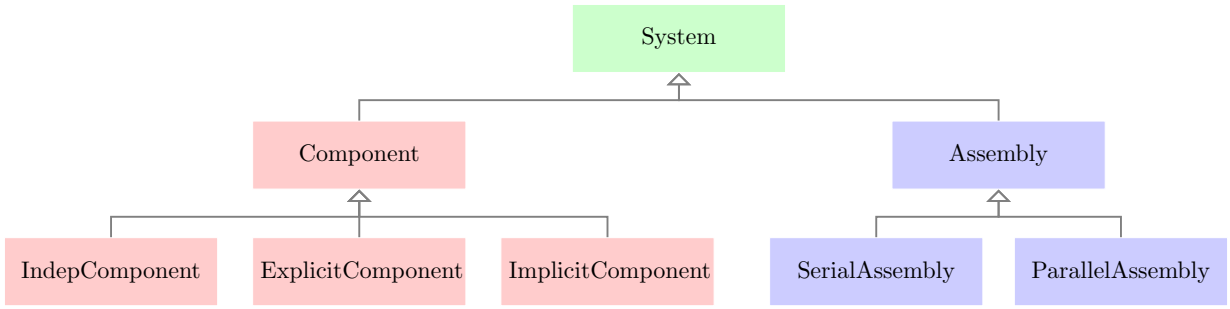


Figure 6: Class inheritance diagram showing the relationships between the *System* classes.

for a particular index set S, as explained in Sec. 4.1. We now present the *System* class methods that represent operations performed on this intermediate system, such as the residual R_S computation and the computation of the solution u_S.

The *System* class has an interface consisting of the following five methods: *apply_nonlinear*, *solve_nonlinear*, *apply_linear*, *solve_linear*, and *linearize*. Table 1 describes these methods, lists their inputs and outputs, and details how they are called. The *apply_nonlinear* method is a nonlinear operator that computes the residuals, and *solve_nonlinear* is a nonlinear operator that solves for the unknowns by converging the residuals to zero. The *apply_linear* method is a linear operator that performs a Jacobian-vector product, and the *solve_linear* method applies the inverse of the square part of the Jacobian as a linear operator. The *linearize* method provides preprocessing steps such as matrix assembly, matrix factorization, and preconditioner computation.

All modern nonlinear and linear solution methods can be implemented using the interface provided by these five methods. For example, a Newton solver assigned to a given assembly (i.e., as the implementation of the *solve_nonlinear* method) would call the assembly's *apply_nonlinear* method to get the nonlinear residual vector for the right-hand side of the linear system, and the *solve_linear* method to solve the linear system. If the user chose a Krylov solver as the implementation of *solve_linear* for this assembly, the Krylov solver would call *apply_linear* multiple times, since this represents a matrix-vector product operation. Furthermore, suppose that the preconditioner used by this solver is linear block Jacobi. Then the linear block Jacobi solver would be another *solve_linear* implementation that would iteratively call the *solve_linear* methods of each of the assembly's subsystems. If the user preferred to use an incomplete factorization for the preconditioner, he/she would perform the sparse matrix assembly and factorization in *linearize* and the back substitution in another *solve_linear* implementation. Note that in these examples there are multiple *solve_linear* implementations because the preconditioner is one linear solver nested inside another (i.e., the Krylov solver). The method named *solve_linear* would run the top-level solver, which is the Krylov solver in this case. More general solution methods such as nonlinear preconditioned solvers, pseudo-transient continuation, and multigrid solvers can be incorporated using custom *solve_nonlinear* or *solve_linear* implementations that override the base implementations that call one of the standard solvers. This can be done not just in components but also in groups, for custom coupled solvers.

While the *solve_nonlinear* (*solve_linear*) implementations can contain any nonlinear (linear) solver, there are four fundamental solvers that play important roles in MAUD: nonlinear block Gauss–Seidel, nonlinear block Jacobi, linear block Gauss–Seidel, and linear block Jacobi. To explain the use of these four solvers, we refer back to Fig. 4. *Assembly* objects with children that have only feed-forward coupling (i.e., red bars in Fig. 4) should be assigned nonlinear (linear) block Gauss–Seidel as their nonlinear (linear) solver, since the nonlinear (linear) system would "converge" in one iteration. For a similar reason, *Assembly* objects with children that have no coupling (i.e., green bars in Fig. 4) should be assigned nonlinear (linear) block Jacobi as their nonlinear (linear) solver. However, all four solvers can be useful in assemblies that have children with coupling (feedback loops) as well, although block Jacobi is known to have poor convergence in this case.

Composable solvers similar to those described above are implemented in PETSc [8] in the nonlinear [13] and linear [12] settings. These approaches hierarchically divide a multidisciplinary system by variable type, discipline, or mesh block [12, 13]. Users can use PETSc's composable solvers to implement *solve_linear* or

solve_nonlinear in MAUD, but they also have the option to use a custom solver, or a solver from any other framework.

Table 2 lists the solvers and operations that can be involved in the implementation of each of the *System* class methods, depending on whether the given *System* object is an *Assembly* or a *Component* object. For instance, *solve_nonlinear* for an assembly could be a custom user-implemented solver, Newton's method, or a block solver; however, a block solver is not appropriate for a component because components cannot be broken down further. The *apply_nonlinear* method must be implemented by the user because it is completely dependent on the problem the user wants to solve. Appendix B presents the algorithms for the key solver implementations.

4.4 Efficient data structures

Efficient data structures are necessary to avoid memory and computing overhead in problems with large vectors. For each system, the MAUD architecture stores six vectors: the output u, input p, and residual r for the nonlinear operations and the corresponding variables for the linear operations, which are du, dp, and dr. The latter three can be considered to be buffers for the linear solution vector or the right-hand side vectors, depending on the situation. For instance, for the Newton system, the right-hand side is stored in dr and the solution vector is stored in du. The size of the vectors u, r, du, and dr is N, the sum of the sizes of all the input, state, and output variables, as defined by Eq. (4). The size of the vectors p and dp depends on the sparsity of the problem. The vectors u, du, r, and dr are instances of the *UnknownVec* class, while the vectors p and dp are instances of the *ParameterVec* class.

To illustrate the *UnknownVec* and *ParameterVec* classes, Fig. 7 shows how the u and p vectors are stored in the MAUD architecture. For the *UnknownVec* instances (u in Fig. 7, in blue), data is shared with the systems above and below in the hierarchy tree. The full u, du, r, and dr vectors are allocated in the top-level *Assembly* object, and the other systems store pointers to subvectors of the global vector (shown in darker blue). Compared to the allocation of separate vectors in each system, this approach saves memory and computational time, since the subsystems operate directly on subvectors of the larger system's vector. When distributed-memory parallel computing is used, each processor stores only its local parts of the u, du, r, and dr vectors; the storage via pointers to subvectors remains the same even in the parallel situation.

The *ParameterVec* instances (p in Fig. 7, in green) store only the variables that a *Component* declares as its arguments. There are two reasons for explicitly storing a separate copy of the arguments of each *Component* as opposed to simply reading the data as needed from the *UnknownVec*. First, if the originating data is stored in a different processor, a buffer is needed for the parallel data communication. Second, the linear and nonlinear block-Jacobi methods require a second copy of the arguments to store their values from the previous Jacobi iteration. The MAUD architecture allows for the use of block Jacobi at multiple levels in the hierarchy tree, and the challenge of updating the arguments at the right times is simplified by incorporating the data transfers into the Jacobi algorithm. In serial mode without block Jacobi, the separate copy results in a memory and computational time penalty, but it is not significant in practice because the components typically have only $\mathcal{O}(1)$ input variables (each can be a vector) and this number does not increase with the number of components.

The MAUD architecture automates parallel data transfer between *UnknownVec* instances and *ParameterVec* instances, providing two benefits. First, if component A depends on variable V from component B, component A does not have to know how many processors component B has, or how much of variable V component B has stored on each processor. For each argument that a *Component* declares, the framework can automatically determine on which processor the requested data is stored and what the local indices are, based on the global indices that were also declared. Second, the data transfer operation is implemented as a method in the *Assembly* class, so it is integrated into the solution algorithms. This operation performs data transfers for multiple *Component* objects simultaneously, improving the parallel performance. A system transfer operation transfers data to a subsystem *ParameterVec* instance from the *UnknownVec* instances of its other subsystems, and vice versa for the reverse mode in a transposed linear system.

4.5 Assumptions and limitations

The most important feature of MAUD is that it enables the computational framework to automatically assemble the total derivatives for a multidisciplinary model given the partial derivatives of each component. Furthermore, the unifying derivative equation (33) adapts to the problem structure and model type, so that

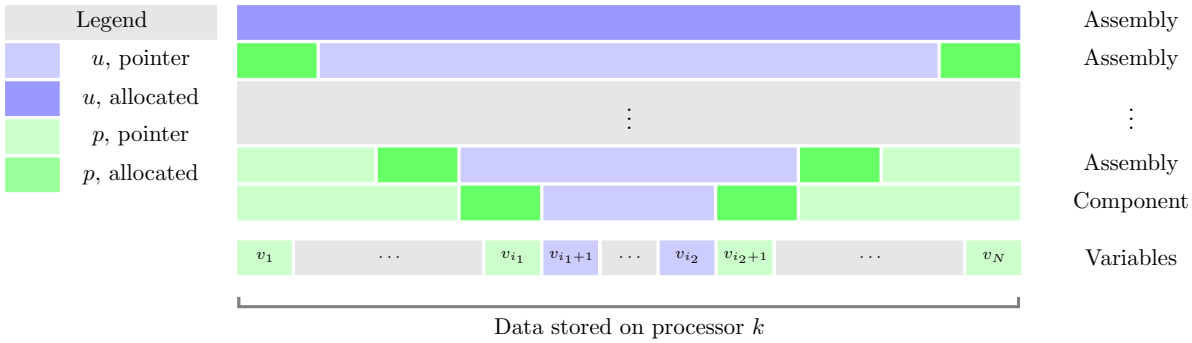


Figure 7: Data storage for the u and p vectors for a *Component* object and all of the *Assembly* instances above it in the hierarchy tree.

solving this equation is equivalent to assembling the total derivatives using the appropriate method among the known methods for computing total derivatives. However, like any method for computing derivatives, the MAUD approach makes assumptions about the underlying models.

The first assumption is that each component is continuous and differentiable. Strictly speaking, both are necessary for the use of the unifying derivative equation (33); however, they are required only locally, so in practice piecewise continuity and differentiability are often sufficient. Continuity and differentiability are also helpful for avoiding convergence issues in the optimization problem. Continuity can be compromised by time-stepping algorithms that have a variable number of time steps, since the number of iterations is inherently discrete. However, if the solution tolerance is sufficiently small, the discontinuities should be small enough that optimization convergence issues can be avoided. An observed rule of thumb is that the solver tolerance must be at least one or two orders of magnitude smaller than the tolerance of the derivative solver for the unifying derivative equation (33), which must in turn be one or two orders smaller than the optimization tolerance.

Although continuity and differentiability are required mathematically, many instances of non-differentiability that the authors have encountered have turned out to be benign. For instance, adjoint-based optimization is commonly used in CFD solvers with upwind discretizations [41]. In cases where point discontinuities might be problematic, the functions can often be smoothed numerically. For instance, the satellite and aircraft operational optimization problems discussed in Secs. 5.2 and 5.3 use smoothing functions to eliminate points that are not continuous and differentiable, enabling the overall optimization problem to be solved using MAUD and a gradient-based optimizer.

The second assumption is that partial derivatives are available for the component. For explicit variables, MAUD requires the partial derivatives of the explicit function with respect to the component's inputs, and for implicit variables, MAUD requires the partial derivatives of the residual with respect to the state variable itself as well as the component's inputs. The multidisciplinary derivative computation can be degraded in both accuracy and computational cost if one of the components wraps a commercial solver that does not compute derivatives, in which case the finite-difference method is the only option. The error in the finite-difference component derivatives propagates to the multidisciplinary derivatives, which can cause optimization convergence issues.

Due to these issues of non-convergence, partial derivatives should be computed accurately using hand differentiation, the complex-step method, or automatic differentiation, when possible. Differentiating by hand requires more effort, but it can be the most efficient method. The manual effort can be reduced by decomposing a larger component into multiple smaller components with simpler mathematical expressions. The complex-step method is accurate and can be relatively labor-free to implement in certain cases, but the cost is proportional to the number of component inputs. The cost of automatic differentiation is proportional to either the number of component inputs or the number of component outputs, but both can be large in some cases.

The third assumption about the underlying models is that in the multidisciplinary Jacobian of partial derivatives, the square diagonal blocks corresponding to intermediate systems that are formed must also

be invertible. This invertibility requirement only applies to the fundamental system and to intermediate systems that are immediate children of systems that use a nonlinear or linear block Gauss–Seidel or Jacobi iteration. Other intermediate systems, including individual components, can have singular Jacobians if one of the groups above them in the hierarchy tree uses monolithic solvers, such as Newton's method or a direct solver.

This invertibility requirement is satisfied if the problem is well-posed, since it is also a requirement for Newton's method. Similar assumptions are discussed by Keyes et al. [42] in the context of solving tightly coupled systems with Newton's method. They assume that the matrix is diagonally dominant and that each block corresponding to a component or discipline is invertible. They state that "these assumptions are natural in the case where the system arises from the coupling of two individually well-posed systems with legacies of being solved separately." When these conditions hold, the matrix is typically invertible in practice.

MAUD has some limitations. Like the adjoint method, MAUD provides a significant speed-up for total derivative computation only when there are implicit state variables, a feedback loop among the components, or both. The presence of coupling in either form requires a nonlinear or linear solver that dominates the computational time. When coupling exists, a straightforward assembly of the partial derivatives of the components is not possible, and the finite-difference method is slower than the adjoint method by roughly a factor of n, where n is the number of design variables. When there is no coupling—i.e., no implicit state variables and no feedback loops—we can just apply the chain rule. In this simpler case, MAUD merely automates the chain-rule assembly of the partial derivatives, using a single linear block Gauss–Seidel iteration to compute the total derivatives. This again highlights the fact that MAUD takes a unified approach because it handles coupled and sequential problems within the same hierarchical solution approach.

5 Results from engineering design applications

We give two examples to demonstrate how the MAUD architecture facilitates the implementation and solution of large-scale engineering design optimization problems: the optimization of a nanosatellite [31] and an aircraft operations optimization [33, 38]. These two problems are larger and more complex than have been solved before in their respective fields. We focus on how MAUD manages the many components and their network of dependencies. The details and physical interpretation of the optimal designs can be found in previous publications [31, 32, 33, 38].

5.1 Python implementation of MAUD

To implement and solve the two demonstration problems, we use a bare-bones Python implementation of MAUD. The MAUD architecture has since been implemented in the OpenMDAO framework [26], but the results herein use the bare-bones version because they were generated prior to OpenMDAO's adoption of the MAUD architecture. The bare-bones implementation depends only on the *NumPy* package for handling local vectors and on the *petsc4py* package [18] as an interface for PETSc [8]. PETSc is used for all parallel data transfers, and its Krylov iterative methods are used as well, with flexible generalized minimal residual (fGMRES) as the default solver. PETSc is not a required dependency, however, since nonparallel problems can be run with *NumPy* data transfers and *SciPy*'s Krylov solvers, in both the bare-bones and OpenMDAO implementations of MAUD.

The entire bare-bones implementation of the framework consists of a single Python file with about one thousand lines of code, thanks to the simplifications enabled by MAUD's monolithic mathematical formulation and the use of PETSc.

5.2 Large-scale optimization of a nanosatellite

A CubeSat is a low-cost nanosatellite with a mass that is less than 1.33 kg. CubeSats are primarily built by university teams for research purposes because they are easier and less costly to design, build, and launch compared to larger satellites. Given that the turnaround time can be as short as a year, the objective of this application of MAUD is to develop a computational model for the CubeSat and to use optimization to expedite the design process, while improving the final design [31].

Part of the motivation for using numerical optimization is the multidisciplinary nature of the problem. The CubeSat stores energy collected by the solar panels in batteries and uses this energy to power three

main systems: the instruments that collect research data, the transmitter that sends this data to stations on Earth, and the momentum wheels that enable attitude control. The modeling of the systems involves several disciplines: orbital dynamics, attitude dynamics, solar cell illumination, heat transfer, solar power generation, energy storage, and communication.

In our formulation, the model for this multidisciplinary system results in 43 separate components (*Component* objects), each of which is responsible for computing a subset of the full system's variables. Some of these variables are determined implicitly through the solution of ODEs using a fourth-order Runge–Kutta solver. The ODE state variables consist of the orbit trajectory, attitude, temperature, and battery charge level. Other variables require interpolating tables of precomputed discrete data to obtain smooth explicit functions. One of the variables—the solar panel exposure variable—is discontinuous because the sunlight exposure on the solar panels rapidly drops to zero when the satellite travels into the Earth's shadow. However, we use cubic interpolation to smooth these transitions. This is done to satisfy the continuity and differentiability requirements discussed in Sec. 4.5.

In total, there are roughly 200 input, state, or output variables, most of which are vector-valued because they are time-dependent. The model contains six 12-hour simulations of the satellite evenly distributed throughout the year, and each of the six simulations takes about 1500 time steps, so these 200 variables result in 2.2 million unknown scalars.

The optimization problem has a total of over 25,000 (scalar) design variables because multiple profiles that vary in time are simultaneously optimized. The optimizer we use is SNOPT [24], which uses sequential quadratic programming to efficiently solve nonlinear constrained optimization problems that are large and sparse. The pyOpt optimization framework [58] is used as the interface to SNOPT.

Figure 8 plots some of the simulation results at the optimized design point, illustrating the scope of a large-scale optimization in a practical design problem. The operational design variables consist of the discretized communication power, roll angle, and solar panel current curves. Figure 8 shows that the optimizer determines that it is optimal to spike the power allocation to the communication module only when there is a line of sight from the satellite to the ground station. The optimizer also determines the optimal roll angle profile to balance the competing considerations of maximizing power generation, shading some of the solar cells when they overheat, and maximizing signal strength during data transfers to the ground station. Optimizing curves rather than scalar values gives rise to tens of thousands of design variables, and these results show that our approach can handle such problems.

The MAUD architecture provides two unique benefits that enable the solution of the satellite design optimization problem. The first is the efficiency in executing the computational model. Since the full coupled system involves 2.2 million unknowns, MAUD's efficient data handling and transfer operations are critical. To solve the large linear systems that arise, the hierarchical decomposition enables the use of an LU decomposition for the Jacobian blocks corresponding to ODE variables. The second benefit is that the coupled derivative computation in MAUD automates what would otherwise be an extremely error-prone and laborious process of combining the derivatives from 43 components and their complex network of dependencies. For this problem, the derivative computation using MAUD amounts to a combination of the adjoint method and the chain rule (since many of the components evaluate explicit functions), where each component is differentiated analytically to obtain the partial derivatives. This problem does not take advantage of MAUD's hierarchically partitioned solvers, because there is no feedback loop among the 43 components so a single iteration of block Gauss–Seidel suffices for both the nonlinear and linear solution.

Figure 9 shows that the optimization problem converged 2 orders of magnitude in optimality and 4 orders in feasibility. This optimization required approximately 100 h using a single 2.93 HGz Core i7-870 processor. This is a low computational cost for a nonlinear optimization problem with so many design variables and states, thanks to the combination of adjoint-based derivative computation and the gradient-based optimizer. The function evaluation and derivative computation times are plotted as a function of problem size in Fig. 10. The plot shows that the additional computational time due to framework overhead does not cause a bottleneck when scaling up this problem, since the overall time scales linearly with the number of degrees of freedom in the problem. More details on this problem can be found in previous work by the authors [31].

5.3 Aircraft operations optimization

The other large-scale optimization application is an airline profit maximization for a fleet of commercial aircraft [33]. Given the design of a next-generation aircraft, the profit maximization problem seeks to

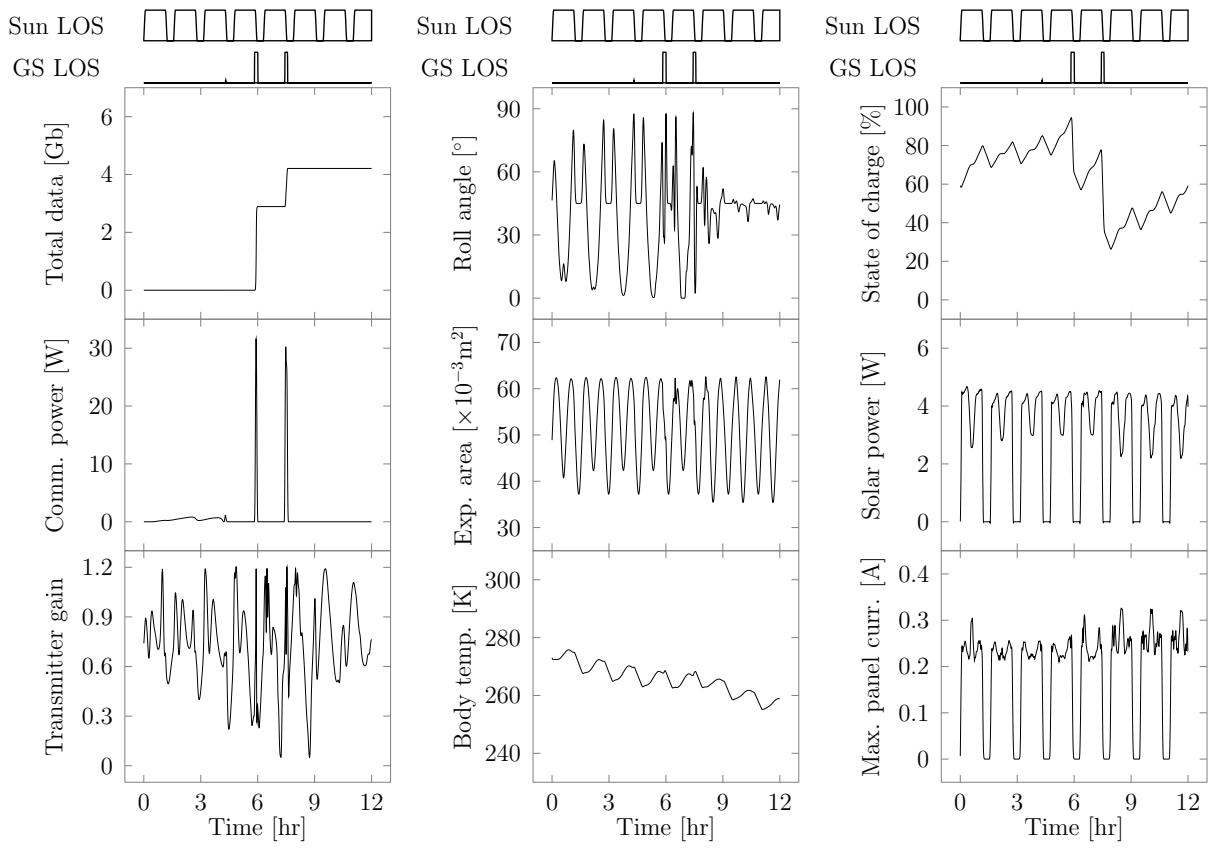
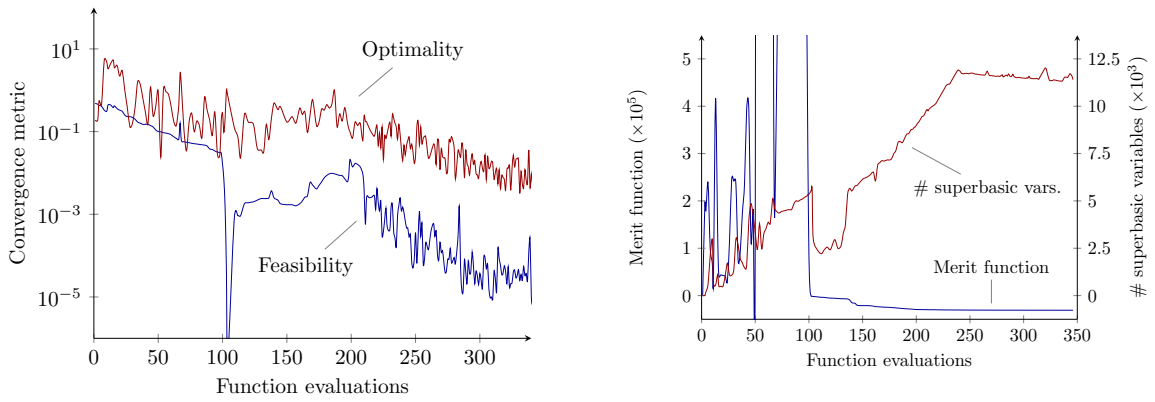
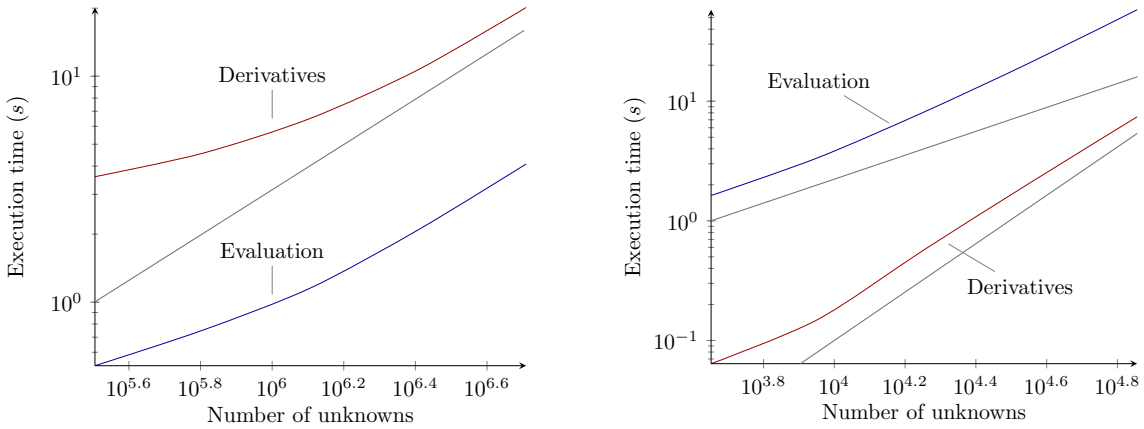


Figure 8: A subset of variables modeled in the nanosatellite design optimization problem. The total data is minimized for a 12-hour period; the battery state of charge curve is constrained to be between 20% and 100% at all times. The communication power profile, roll angle profile, and panel current profile are all parametrized with design variables, and the remaining six quantities are state variables. Sun line-of-sight (LOS) and ground station LOS describe whether the satellite is in view of the Sun and the ground station, respectively.



[31]

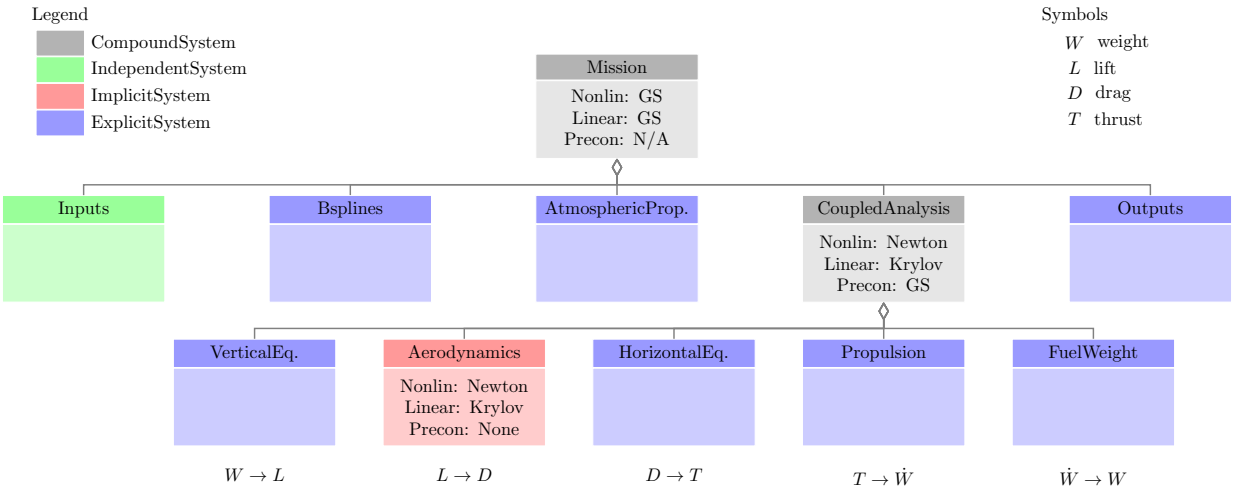
Figure 9: Convergence plots for the CubeSat optimization problem. The superbasic variables are the subset of the design variables that are free (not at their bounds, or used to satisfy equality or active inequality constraints). The merit function is the augmented Lagrangian, which is formed by appending to the objective function both penalty and Lagrange-multiplier terms for the nonlinear constraints. The optimality is the norm of the gradient of the augmented Lagrangian, and the feasibility is the norm of the constraints.



Nanosatellite problem

Aircraft operations problem

Figure 10: Wall times for the functional evaluation and derivative computation (a single coupled adjoint solution) in the nanosatellite and aircraft operations problems with linear and quadratic (right plot only) reference lines.



[38]

Figure 11: Class diagram showing the *System* instances in one of the 128 aircraft mission analyses.

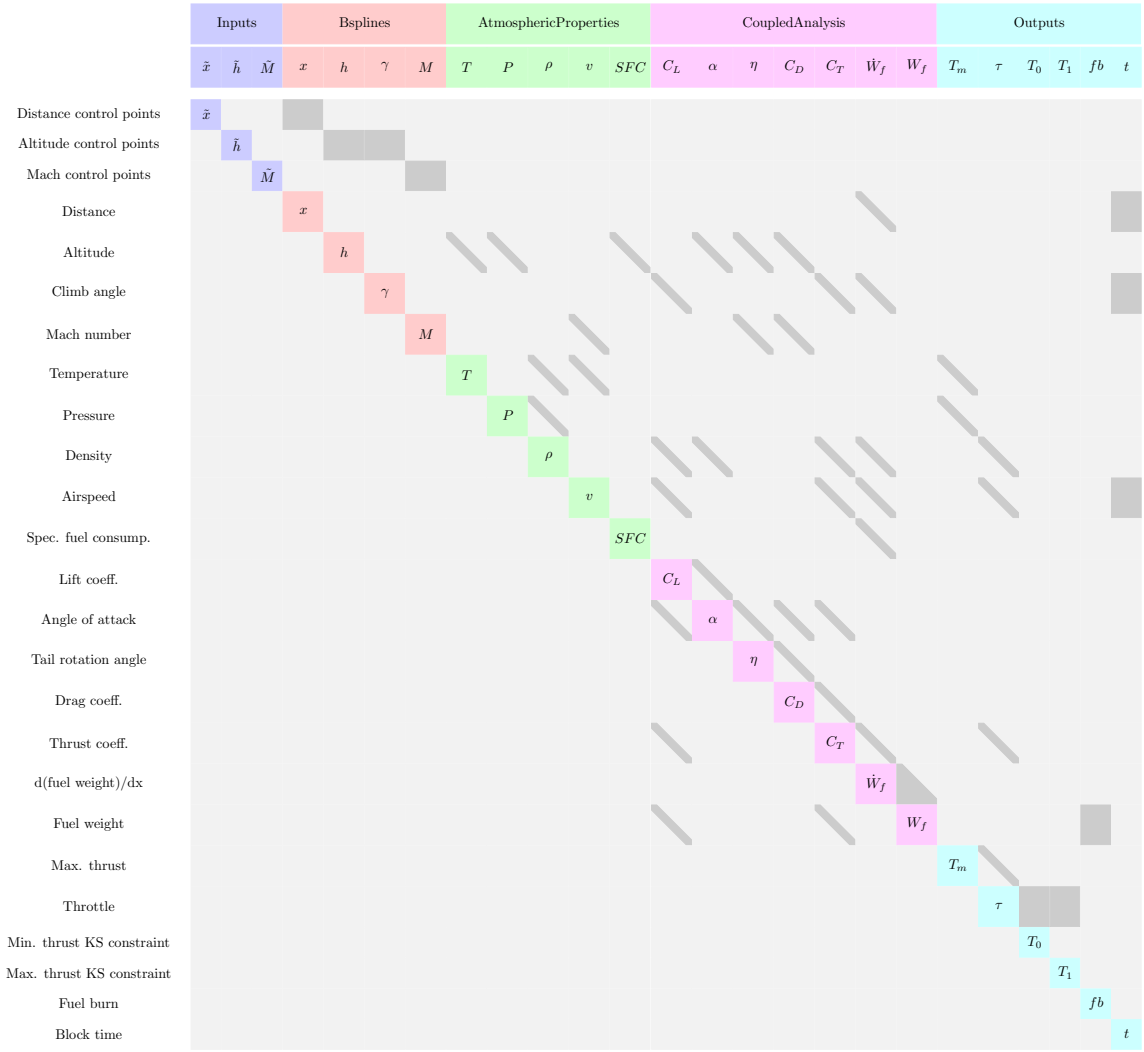
determine on which routes an airline should fly this new aircraft. In addition to the route variables, the profit maximization simultaneously optimizes the altitude profiles for those routes. We assume that the airline has four types of current-generation aircraft in addition to the next-generation aircraft, and that there is a limited number of each type of aircraft. The motivation for this optimization is to evaluate the impact of next-generation aircraft technologies that depend on the altitude profiles, such as morphing wings and continuous descent approaches. This optimization problem is solved using 128 processors.

The computational model consists of 128 mission analyses to predict the fuel efficiency for each aircraft route. Each mission analysis involves several coupled components. Figure 11 illustrates how a single mission instance within the larger problem is partitioned, and the dependencies between the variables are shown in Fig. 12. The *Inputs*, *Bsplines*, *AtmosphericProperties*, *CoupledAnalysis*, and *Outputs* variables are grouped into these categories for convenience, and because the resulting blocks have a sequential chain of dependence that can be solved with a single block Gauss–Seidel iteration, as shown in Fig. 12.

One of the variables in the *AtmosphericProperties* system is the atmospheric temperature at a given point in the mission. The atmospheric temperature can be computed from the altitude, while other quantities such as the pressure, density, and speed of sound are computed from the temperature. However, temperature is a piecewise function of altitude, since it decreases linearly up to 11 km and is constant above that. Since the temperature is not differentiable at 11 km, we use cubic smoothing to ensure that the model satisfies the continuity and differentiability assumption from Sec. 4.5.

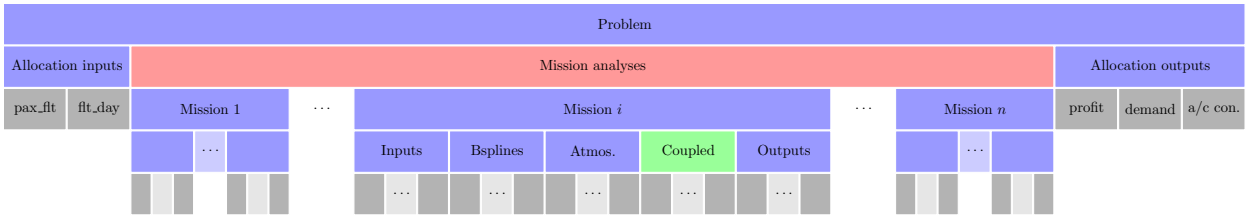
The *CoupledAnalysis* involves feedback loops between fuel burn, thrust, drag, lift, and weight, due to the intrinsic coupling between the propulsion, aerodynamics, weight, and flight dynamics disciplines. The rate of fuel burn at a given point in the flight depends on the engine throttle setting, which is determined by the aerodynamic drag that the engine thrust must overcome. However, the drag depends on how much lift is produced to counteract the weight of the aircraft, which in turn is affected by the total amount of fuel being carried. Newton's method is used to solve this coupled nonlinear system after an optional start-up sequence of nonlinear block Gauss–Seidel iterations. The linear systems that arise are solved using fGMRES [59], and the optional preconditioner consists of a few iterations of linear block Gauss–Seidel.

To evaluate a given set of routes at each optimization iteration, we require 128 instances of the mission analysis we just described (Figs. 11 and 12). These analyses compute the fuel burn for each route, and then the allocation model evaluates how much profit the airline would make given the current aircraft allocation. Figure 13 shows the full hierarchy tree for the problem, which consists of the mission analyses and the allocation model. In Fig. 13, the *Mission analyses* system groups the 128 mission systems. To enable parallel computation of the various missions, we use the block-Jacobi solver for the linear and nonlinear



[33]

Figure 12: Aircraft mission analysis variables (listed on left, top, and diagonal) and their dependencies (shown in off-diagonal dark gray blocks).



[33]

Figure 13: Hierarchy tree showing how the allocation-mission model is structured. The items in gray are the variables, the blue items group their systems in series, and the red item groups its systems in parallel. The coupled analysis is shown in green because its variables are coupled, so it uses Newton's method and GMRES.

systems. Hwang and Martins [33] describe the problem and discuss the results in more detail.

We see from this problem that in practice the hierarchy tree for the partitioning is chosen based on how we want to parallelize and where there is coupling between variables. The desire to parallelize across the 128 missions motivates us to introduce the *Mission analyses* system, and the coupling between the lift, angle of attack, drag, and thrust motivates us to group them into the *CoupledAnalysis* system.

This optimization problem contains over 6,000 design variables and 23,000 constraints. The optimization achieves 3 orders of magnitude of convergence in optimality and feasibility in about 8 hours on 128 processors, as shown in Fig. 14. The optimization was run on 8 nodes, each with 16 E5-2670 2.6 GHz processors and 64 GB RAM.

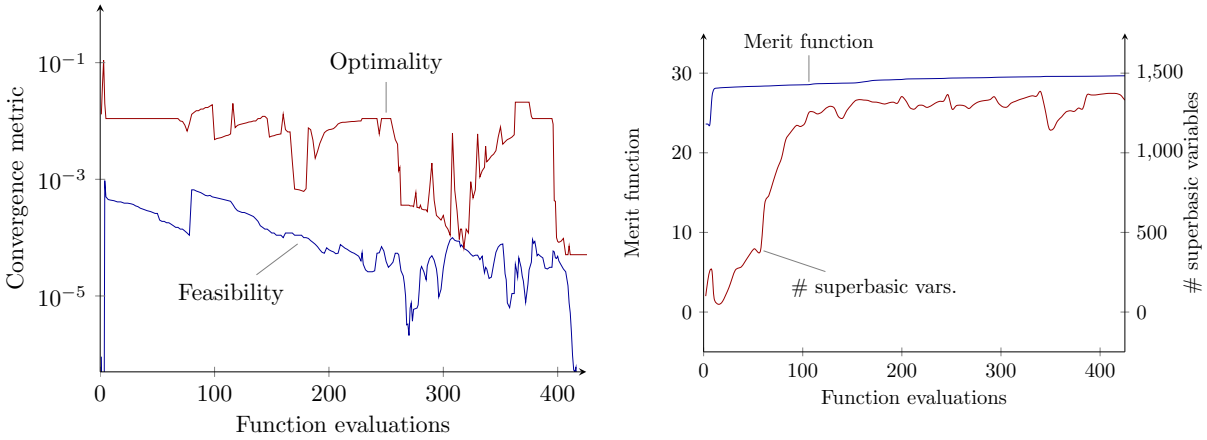
6 Conclusion

In this paper, we have described the MAUD architecture, a novel method for facilitating the coupling of multiple heterogeneous computational models and for computing their coupled solution and coupled derivatives efficiently. We have presented the mathematical formulation for coupling models and computing their derivatives, discussed the algorithmic approach for solving the resulting systems of equations, and given results from the application of MAUD to two multidisciplinary design optimization problems.

The fundamental theory starts by combining all the input, state, and output variables from the various disciplines into a unified vector. The multidisciplinary model is then formulated as a single algebraic system of equations with this unified vector comprising the vector of unknowns, so the task of running the simulation reduces to solving this nonlinear system. Next, we show that the derivatives of the outputs with respect to the inputs of the model can be computed by solving Eq. (33), a linear system with multiple right-hand sides. The nonlinear system is also recursively partitioned to enable hierarchical solution methods.

The software implementation of MAUD is naturally object-oriented. The main base class is a *System* with a compact interface consisting of five methods used to solve the nonlinear and linear systems. The user-defined components that own a subset of the variables are *System* objects, as are the framework-provided *System* objects that group other *System* objects together. In the interface, the components provide Jacobian matrices ultimately as linear operators, and MAUD centrally stores and operates on all the vectors in a concatenated form, making pointers to subvectors available to the components for easy access. This idea of a hierarchy of *System* objects with the five-method interface is an important part of MAUD because it details how one can use the aforementioned fundamental theory and formulation in practical problems and implement modern solution algorithms such as Newton–Krylov solvers and preconditioners without a penalty on computation time.

The key benefits of the MAUD architecture can be summarized as follows. First, its modularity allows the framework to automate parallel data passing between components. Components that have parallel vectors as arguments need not know how those parallel vectors are distributed across their processors. Second, MAUD provides a hierarchical way to organize the components, with the user given a choice of solver at each node in the hierarchy. This is important because MAUD formulates the model as a single system of equations,



[33]

Figure 14: Convergence plots for the aircraft operations optimization. The superbasic variables are the subset of the design variables that are free (not at their bounds, or used to satisfy equality or active inequality constraints). The merit function is the augmented Lagrangian, which is formed by appending both penalty and Lagrange-multiplier terms for the nonlinear constraints to the objective function. The optimality is the norm of the gradient of the augmented Lagrangian, and the feasibility is the norm of the constraints.

and without the hierarchical structure there would be memory and computational overheads. The third and most important benefit is the automated computation of derivatives given the partial derivatives of each component and the appropriate linear solvers. MAUD enables adjoint derivative computation, which can compute a full gradient with respect to all the inputs at a cost that is of the same order of magnitude as that of running the simulation. In particular, MAUD provides a common interface for the use of the chain rule, the coupled adjoint method, and hybrid methods that combine the two; the appropriate method is automatically chosen based on the model.

There are three underlying assumptions for problems that are to be solved with MAUD. First, each component in the multidisciplinary model is assumed to be continuous and differentiable. However, models with some discontinuities can be smoothed in practice, as was done in the two applications presented in this paper. The second assumption is that each component must be able to compute its partial derivatives. If such a computation is not available, one must resort to the finite-difference method, which is inaccurate and inefficient. Finally, MAUD solves linear systems to converge simulations and compute derivatives, so the Jacobian matrices of the multidisciplinary systems must be invertible.

The two applications presented herein demonstrate the power of the adjoint method for problems in which these assumptions hold. The first application is the optimization of a nanosatellite involving 7 disciplines modeled by 43 components and over 2 million total unknowns. The optimization problem contains over 25,000 design variables, and it converges in the equivalent of $\mathcal{O}(100)$ function evaluations, thanks to the coupled adjoint computation that MAUD performs for the 43 components. The second application is an aircraft operations optimization that involves over 6,000 design variables and 23,000 constraints. This problem has fewer components, but the variables that are modeled are coupled at multiple levels, requiring a hierarchical nonlinear solver to perform the multidisciplinary analysis and a hierarchical linear solver to compute the derivatives efficiently using the coupled adjoint method. Moreover, this problem is solved using 128 processors, and MAUD fully automates the parallelization.

MAUD has been implemented in OpenMDAO [26], an open-source computational framework being developed by NASA[1]. Both the CubeSat optimization and the aircraft operations optimization have also been implemented in OpenMDAO. The advantages of MAUD have been tested and demonstrated through its application to aircraft, satellite, wind turbine, airline allocation, and engine design problems, and MAUD can now be easily used via OpenMDAO.

7 Acknowledgments

The authors would like to thank Justin S. Gray, Kenneth T. Moore, and Bret A. Naylor from NASA Glenn Research Center for valuable feedback and discussions, as well as their work on implementing the ideas presented herein in OpenMDAO. The authors would also like to thank the following researchers from the University of Michigan: Dae Young Lee and James W. Cutler for their involvement in the CubeSat problem; Jason Y. Kao for his help in developing the aircraft mission model; and Gaetan K. W. Kenway for insightful discussions. This work was partially supported by NASA (grant number NNX14AC73A) and the National Science Foundation (award number 1435188).

References

[1] 1997. UG—A flexible software toolbox for solving partial differential equations. *Computing and Visualization in Science* 1, 1 (1997), 27–40. DOI:http://dx.doi.org/10.1007/s007910050003

[2] 2000. A component-based architecture for problem solving environments. *Mathematics and Computers in Simulation* 54, 4–5 (2000), 279–293. DOI:http://dx.doi.org/10.1016/S0378-4754(00)00189-0

[3] Jan Albersmeyer and Moritz Diehl. 2010. The Lifted Newton Method and Its Application in Optimization. *SIAM J. on Optimization* 20, 3 (Jan. 2010), 1655–1684. DOI:http://dx.doi.org/10.1137/080724885

[4] Natalia Alexandrov and M. Y. Hussaini (Eds.). 1997. *Multidisciplinary Design Optimization: State-of-the-Art*. SIAM.

[5] Natailia M. Alexandrov and Robert Michael Lewis. 2004a. Reconfigurability in MDO Problem Synthesis, Part 1. In *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. DOI:http://dx.doi.org/10.2514/6.2004-4307

[6] Natalia M. Alexandrov and Robert Michael Lewis. 2004b. Reconfigurability in MDO Problem Synthesis, Part 2. In *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. DOI:http://dx.doi.org/10.2514/6.2004-4308

[7] Vladimir Balabanov, Christophe Charpentier, D. K. Ghosh, Gary Quinn, Garret Vanderplaats, and Gerhard Venter. 2002. VisualDOC: A Software System for General Purpose Integration and Design Optimization. In *9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*. Atlanta, GA. DOI:http://dx.doi.org/10.2514/6.2002-5513

[8] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. 1997. *Efficient Management of Parallelism in Object Oriented Numerical Software Libraries*. Birkhäuser Press, 163–202.

[9] Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat. 2007. deal. II—A general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software (TOMS)* 33, 4 (2007), 24.

[10] F. Bassetti, D. Brown, K. Davis, W. Henshaw, and D. Quinlan. 1998. OVERTURE: An Object-Oriented Framework for High Performance Scientific Computing. In *Supercomputing, 1998.SC98. IEEE/ACM Conference on*. 14–14. DOI:http://dx.doi.org/10.1109/SC.1998.10013

[11] M. W. Beall and M. S. Shephard. 1999. An Object-Oriented Framework for Reliable Numerical Simulations. *Engineering with Computers* 15, 1 (1999), 61–72. DOI:http://dx.doi.org/10.1007/s003660050005

[12] Jed Brown, Matthew G. Knepley, David A. May, Lois Curfman McInnes, and Barry Smith. 2012. Composable linear solvers for multiphysics. In *2012 11th International Symposium on Parallel and Distributed Computing*. IEEE, 55–62.

[13] Peter R. Brune, Matthew G. Knepley, Barry F. Smith, and Xuemin Tu. 2015. Composing scalable nonlinear algebraic solvers. *SIAM Rev.* 57, 4 (2015), 535–565.

[14] Hayoung Chung, John T Hwang, Justin S Gray, and Hyunsun A Kim. 2018. Implementation of topology optimization using OpenMDAO. In *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*. DOI:http://dx.doi.org/10.2514/6.2018-0653

[15] Nathan O. Collier, Lisandro Dalcín, and Victor M. Calo. 2013. PetIGA: High-Performance Isogeometric Analysis. *CoRR* abs/1305.4452 (2013). http://arxiv.org/abs/1305.4452

[16] E. J. Cramer, J. E. Dennis, P. D. Frank, R. M. Lewis, and G. R. Shubin. 1994. Problem Formulation for Multidisciplinary Optimization. *SIAM Journal on Optimization* 4, 4 (1994), 754–776.

[17] Eric C. Cyr, John N. Shadid, and Raymond S. Tuminaro. 2016. Teko: A Block Preconditioning Capability with Concrete Example Applications in Navier–Stokes and MHD. *SIAM Journal on Scientific Computing* 38, 5 (2016), S307–S331.

[18] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. 2011. Parallel distributed computing using Python. *Advances in Water Resources* 34, 9 (2011), 1124–1139.

[19] Robert D Falck, Jeffrey C Chin, Sydney L Schnulo, Jonathan M Burt, and Justin S Gray. 2017. Trajectory optimization of electric aircraft subject to subsystem thermal constraints. In *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. DOI:http://dx.doi.org/10.2514/6.2017-4002

[20] Robert D. Falgout and Ulrike Meier Yang. 2002. hypre: A library of high performance preconditioners. In *International Conference on Computational Science*. Springer, 632–641.

[21] Sam Friedman, Seyede Fatemeh Ghoreishi, and Douglas L Allaire. 2017. Quantifying the impact of different model discrepancy formulations in coupled multidisciplinary systems. In *19th AIAA Non-Deterministic Approaches Conference*. DOI:http://dx.doi.org/10.2514/6.2017-1950

[22] Derek Gaston, Chris Newman, Glen Hansen, and Damien Lebrun-Grandié. 2009. MOOSE: A parallel computational framework for coupled systems of nonlinear equations. *Nuclear Engineering and Design* 239, 10 (2009), 1768–1778. DOI:http://dx.doi.org/10.1016/j.nucengdes.2009.05.021

[23] Pieter Gebraad, Jared J Thomas, Andrew Ning, Paul Fleming, and Katherine Dykes. 2017. Maximization of the annual energy production of wind power plants by optimization of layout and yaw-based wake control. *Wind Energy* 20, 1 (2017), 97–107. DOI:http://dx.doi.org/10.1002/we.1993

[24] P. E. Gill, W. Murray, and M. A. Saunders. 2005. An SQP algorithm for large-scale constrained optimization. *Society for Industrial and Applied Mathematics* 47, 1 (2005). http://www.stanford.edu/group/SOL/papers/SNOPT-SIGEST.pdf

[25] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Massó, Thomas Radke, Edward Seidel, and John Shalf. 2003. The Cactus Framework and Toolkit: Design and Applications. In *Proceedings of the 5th International Conference on High Performance Computing for Computational Science (VECPAR'02)*. Springer-Verlag, Berlin, Heidelberg, 197–227.

[26] Justin Gray, Tristan Hearn, Kenneth Moore, John T. Hwang, Joaquim R. R. A. Martins, and Andrew Ning. 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:http://dx.doi.org/10.2514/6.2014-2042

[27] Andreas Griewank. 2000. *Evaluating Derivatives*. SIAM, Philadelphia.

[28] Eric S Hendricks, Robert D Falck, and Justin S Gray. 2017. Simultaneous propulsion system and trajectory optimization. In *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. DOI:http://dx.doi.org/10.2514/6.2017-4435

[29] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. 2005. An Overview of the Trilinos Project. *ACM Trans. Math. Softw.* 31, 3 (2005), 397–423. DOI: http://dx.doi.org/10.1145/1089014.1089021

[30] Elias N. Houstis and John R. Rice. 2000. Future problem solving environments for computational science. *Mathematics and Computers in Simulation* 54, 4–5 (2000), 243–257. DOI:http://dx.doi.org/10.1016/S0378-4754(00)00187-7

[31] John T. Hwang, Dae Young Lee, James W. Cutler, and Joaquim R. R. A. Martins. 2014. Large-Scale Multidisciplinary Optimization of a Small Satellite's Design and Operation. *Journal of Spacecraft and Rockets* 51, 5 (September 2014), 1648–1663. DOI:http://dx.doi.org/10.2514/1.A32751

[32] John T Hwang and JRRA Martins. 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*. 1662.

[33] John T. Hwang and Joaquim R. R. A. Martins. 2015. Parallel allocation-mission optimization of a 128-route network. In *Proceedings of the 16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. Dallas, TX. DOI:http://dx.doi.org/10.2514/6.2015-2321

[34] John T Hwang and Andrew Ning. 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*. DOI:http://dx.doi.org/10.2514/6.2018-1384

[35] John P Jasa, John T Hwang, and Joaquim Martins. 2018a. Design and trajectory optimization of a morphing wing aircraft. In *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*. DOI:http://dx.doi.org/10.2514/6.2018-1382

[36] John P. Jasa, John T. Hwang, and Joaquim R. R. A. Martins. 2018b. Open-source coupled aerostructural optimization using Python. *Structural and Multidisciplinary Optimization* (2018). (In press).

[37] Xiangmin Jiao, Michael T. Campbell, and Michael T. Heath. 2003. Roccom: An Object-oriented, Data-centric Software Integration Framework for Multiphysics Simulations. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03)*. ACM, New York, NY, USA, 358–368. DOI: http://dx.doi.org/10.1145/782814.782863

[38] Jason Y. Kao, John T. Hwang, and Joaquim R. R. A. Martins. 2015. A modular approach for mission analysis and optimization. In *56th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*. DOI:http://dx.doi.org/10.2514/6.2015-0136

[39] A. J. Keane and P. B. Nair. 2001. Problem solving environments in aerospace design. *Advances in Engineering Software* 32, 6 (2001), 477–487. DOI:http://dx.doi.org/10.1016/S0965-9978(00)00108-3

[40] Graeme J. Kennedy and Joaquim R. R. A. Martins. 2014. A Parallel Finite-Element Framework for Large-Scale Gradient-Based Design Optimization of High-Performance Structures. *Finite Elements in Analysis and Design* 87 (September 2014), 56–73. DOI:http://dx.doi.org/10.1016/j.finel.2014.04.011

[41] Gaetan K. W. Kenway, Graeme J. Kennedy, and Joaquim R. R. A. Martins. 2014. Scalable Parallel Approach for High-Fidelity Steady-State Aeroelastic Analysis and Derivative Computations. *AIAA Journal* 52, 5 (May 2014), 935–951. DOI:http://dx.doi.org/10.2514/1.J052255

[42] David E. Keyes, Lois C. McInnes, Carol Woodward, William Gropp, Eric Myra, Michael Pernice, John Bell, Jed Brown, Alain Clo, Jeffrey Connors, and others. 2013. Multiphysics simulations: Challenges and opportunities. *International Journal of High Performance Computing Applications* 27, 1 (2013), 4–83.

[43] Benjamin S. Kirk, John W. Peterson, Roy H. Stogner, and Graham F. Carey. 2006. libMesh: A C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers* 22, 3-4 (2006), 237–254.

[44] Raymond M. Kolonay and Michael Sobolewski. 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*.

[45] Anders Logg, Kent-Andre Mardal, and Garth Wells. 2012. *Automated solution of differential equations by the finite element method: The FEniCS book*. Vol. 84. Springer Science & Business Media.

[46] Kevin Long, Robert Kirby, and Bart van Bloemen Waanders. 2010. Unified embedded parallel finite element computations via software-based Fréchet differentiation. *SIAM Journal on Scientific Computing* 32, 6 (2010), 3323–3351.

[47] Christopher J. Marriage and Joaquim R. R. A. Martins. 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. DOI:http://dx.doi.org/10.2514/6.2008-5956

[48] Joaquim RRA Martins, Peter Sturdza, and Juan J Alonso. 2003. The complex-step derivative approximation. *ACM Transactions on Mathematical Software (TOMS)* 29, 3 (2003), 245–262.

[49] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. 2005. A Coupled-Adjoint Sensitivity Analysis Method for High-Fidelity Aero-Structural Design. *Optimization and Engineering* 6, 1 (March 2005), 33–62. DOI:http://dx.doi.org/10.1023/B:OPTE.0000048536.47956.62

[50] Joaquim R. R. A. Martins and John T. Hwang. 2013. Review and Unification of Methods for Computing Derivatives of Multidisciplinary Computational Models. *AIAA Journal* 51, 11 (November 2013), 2582–2599. DOI:http://dx.doi.org/10.2514/1.J052184

[51] Joaquim R. R. A. Martins and Andrew B. Lambe. 2013. Multidisciplinary Design Optimization: A Survey of Architectures. *AIAA Journal* 51, 9 (September 2013), 2049–2075. DOI:http://dx.doi.org/10.2514/1.J051895

[52] Joaquim R. R. A. Martins, Christopher Marriage, and Nathan P. Tedford. 2009. pyMDO: An Object-Oriented Framework for Multidisciplinary Design Optimization. *ACM Trans. Math. Software* 36, 4 (Aug. 2009), 20:1–20:25. DOI:http://dx.doi.org/10.1145/1555386.1555389

[53] William F. Mitchell. 2002. The Design of a Parallel Adaptive Multi-level Code in Fortran 90. In *Proceedings of the International Conference on Computational Science—Part III*. Springer-Verlag, London, UK, 672–680.

[54] Andrew Ning and Derek Petch. 2016. Integrated design of downwind land-based wind turbines using analytic gradients. *Wind Energy* 19, 12 (2016), 2137–2152.

[55] Sharon L. Padula and Ronnie E. Gillian. 2006. Multidisciplinary Environments: A History of Engineering Framework Development. In *Proceedings of the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. Portsmouth, VA. DOI:http://dx.doi.org/10.2514/6.2006-7083

[56] Bořek Patzák. 2012. OOFEM—An object-oriented simulation tool for advanced modeling of materials and structures. *Acta Polytechnica* 52, 6 (2012).

[57] Benjamin Peherstorfer, Philip S Beran, and Karen E Willcox. 2018. Multifidelity Monte Carlo estimation for large-scale uncertainty propagation. In *2018 AIAA Non-Deterministic Approaches Conference*. DOI:http://dx.doi.org/10.2514/6.2018-1660

[58] Ruben E. Perez, Peter W. Jansen, and Joaquim R. R. A. Martins. 2012. pyOpt: A Python-Based Object-Oriented Framework for Nonlinear Constrained Optimization. *Structural and Multidisciplinary Optimization* 45, 1 (January 2012), 101–118. DOI:http://dx.doi.org/10.1007/s00158-011-0666-3

[59] Yousef Saad. 1993. A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific Computing* 14, 2 (1993), 461–469.

[60] A. O. Salas and J. C. Townsend. 1998. Framework Requirements for MDO Application Development. In *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*. DOI: http://dx.doi.org/10.2514/6.1998-4740

[61] Jaroslaw Sobieszczanski-Sobieski. 1990. Sensitivity of Complex, Internally Coupled Systems. *AIAA Journal* 28, 1 (1990), 153–160. DOI:http://dx.doi.org/10.2514/3.10366

[62] J. Sobieszczanski-Sobieski and R. T. Haftka. 1997. Multidisciplinary Aerospace Design Optimization: Survey of Recent Developments. *Structural Optimization* 14, 1 (1997), 1–23. DOI:http://dx.doi.org/10.1007/BF011

[63] William Squire and George Trapp. 1998. Using Complex Variables to Estimate Derivatives of Real Functions. *SIAM Rev.* 40, 1 (1998), 110–112.

[64] James R. Stewart and H. Carter Edwards. 2003. The SIERRA Framework for Developing Advanced Parallel Mechanics Applications. In *Large-Scale PDE-Constrained Optimization*, Lorenz T. Biegler, Matthias Heinkenschloss, Omar Ghattas, and Bart van Bloemen Waanders (Eds.). Lecture Notes in Computational Science and Engineering, Vol. 30. Springer Berlin Heidelberg, 301–315. DOI:http://dx.doi.org/10.1007/978-3-642-55508-4_18

[65] S. Tosserams, A. T. Hoftkamp, L. F. P. Etman, and J. E. Rooda. 2010. A Specification Language for Problem Partitioning in Decomposition-Based Design Optimization. *Structural and Multidisciplinary Optimization* 42 (2010), 707–723. DOI:http://dx.doi.org/10.1007/s00158-010-0512-z

[66] P. van der Velde and G. D. Mallinson. 2007. The design of a component-oriented framework for numerical simulation software. *Advances in Engineering Software* 38, 3 (2007), 182–192. DOI:http://dx.doi.org/10.1016/j.advengsoft.2006.05.007

[67] Hiroyuki Yamazaki, Shunji Enomoto, and Kazuomi Yamamoto. 2000. A Common CFD Platform UPACS. In *High Performance Computing*, Mateo Valero, Kazuki Joe, Masaru Kitsuregawa, and Hidehiko Tanaka (Eds.). Lecture Notes in Computer Science, Vol. 1940. Springer Berlin Heidelberg, 182–190. DOI:http://dx.doi.org/10.1007/3-540-39999-2_16

A Nomenclature

List of integers

Numbers of variables

n	total number of input, state, and output variables
m	number of input variables
p	number of state variables
q	number of output variables

Total sizes of variable groups

N	sum of the sizes of all the input, state, and output variables
N^x	sum of the sizes of all the input variables
N^y	sum of the sizes of all the state variables
N^f	sum of the sizes of all the output variables
N_S	sum of the sizes of the output variables of the intermediate system

Sizes of individual variables

N_k	size of the k^{th} variable (among all inputs, states, and outputs)
N_k^x	size of the k^{th} input variable
N_k^y	size of the k^{th} state variable
N_k^f	size of the k^{th} output variable

List of variables

Basic types of variable groups

x	$\in \mathbb{R}^{N^x}$	vector concatenating all m input variables
y	$\in \mathbb{R}^{N^y}$	vector concatenating all p state variables
f	$\in \mathbb{R}^{N^f}$	vector concatenating all q output variables
x^*	$\in \mathbb{R}^{N^x}$	value of the input variable vector at which the model is evaluated

Variable groups used to formulate a single nonlinear system

u	$\in \mathbb{R}^{N}$	vector concatenating all inputs, states, and outputs; short for u_S
r	$\in \mathbb{R}^{N}$	output of the residual function for all the inputs, states, and outputs
r_x	$\in \mathbb{R}^{N^x}$	output of the residual function for all the input variables
r_y	$\in \mathbb{R}^{N^y}$	output of the residual function for all the state variables
r_f	$\in \mathbb{R}^{N^f}$	output of the residual function for all the output variables

Variable groups used to formulate intermediate systems

p	$\in \mathbb{R}^{N-N_S}$	short for p_S
p_S	$\in \mathbb{R}^{N-N_S}$	input for the intermediate system defined by S
u_S	$\in \mathbb{R}^{N_S}$	output for the intermediate system defined by S
r_S	$\in \mathbb{R}^{N_S}$	residual value for the intermediate system defined by S

Individual variables

x_k	$\in \mathbb{R}^{N_k^x}$	k^{th} input variable
y_k	$\in \mathbb{R}^{N_k^y}$	k^{th} state variable
f_k	$\in \mathbb{R}^{N_k^f}$	k^{th} output variable
u_k	$\in \mathbb{R}^{N_k}$	k^{th} variable (among all inputs, states, and outputs)

List of functions

Basic types of functions

$\mathcal{Y}$	$: \mathbb{R}^{N^x} \to \mathbb{R}^{N^y}$	function that computes all the state variables
$\mathcal{R}$	$: \mathbb{R}^{N^x+N^y} \to \mathbb{R}^{N^y}$	residual function for all the state variables
$\mathcal{F}$	$: \mathbb{R}^{N^x+N^y} \to \mathbb{R}^{N^f}$	function that computes all the output variables
$\mathcal{G}$	$: \mathbb{R}^{N^x} \to \mathbb{R}^{N^f}$	function that computes all outputs given inputs

Functions used to formulate a single nonlinear system

R	$: \mathbb{R}^N \to \mathbb{R}^N$	residual function for all the variables
R^{-1}	$: C \to \mathbb{R}^N$	inverse residual defined on an open neighborhood, C, of $0 \in \mathbb{R}^N$

Functions used to formulate intermediate systems

F_S	$: D_1 \times \ldots \times D_{i_1} \times D_{i_2+1} \times \ldots \times D_n \to D_{i_1+1} \times \ldots \times D_{i_2}$	function that solves the intermediate system defined by S
R_S	$: D \to D_{i_1+1} \times \ldots \times D_{i_2}$	residual function for the intermediate system defined by S

Functions associated with individual variables

F_k	$: D_1 \times \ldots \times D_{k-1} \times D_{k+1} \times \ldots \times D_n \to D_k$	function that solves the k^{th} residual function
R_k	$: \mathbb{R}^N \to \mathbb{R}^{N_k}$	residual function for the k^{th} variable (among all inputs, states, and outputs)
$\mathcal{Y}_k$	$: \mathbb{R}^{N^x+N^y-N_k^y} \to \mathbb{R}^{N_k^y}$	function that computes the k^{th} state variable
$\mathcal{F}_k$	$: \mathbb{R}^{N^x+N^y} \to \mathbb{R}^{N_k^f}$	function that computes the k^{th} output variable
$\mathcal{R}_k$	$: \mathbb{R}^{N^x+N^y} \to \mathbb{R}^{N_k^y}$	residual function for the k^{th} state variable

List of symbols for derivatives

$\frac{\mathrm{d}y}{\mathrm{d}x}$	$\in \mathbb{R}^{N^y} \times \mathbb{R}^{N^x}$	total derivative matrix of *variable* y with respect to *variable* x
$\frac{\partial \mathcal{F}}{\partial x}$	$\in \mathbb{R}^{N^f} \times \mathbb{R}^{N^x}$	partial derivative matrix of *function* $\mathcal{F}$ with respect to *argument* x

List of sets

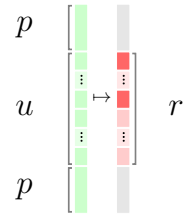
D	$\subseteq \mathbb{R}^N$	Cartesian product of N closed intervals
D_k	$\subseteq \mathbb{R}^{N_k}$	Cartesian product of N_k closed intervals
S	$\subseteq \{1, \ldots, n\}$	contiguous index set defining an intermediate system

B Algorithms

In the following figures, the parts of the vectors in green are arguments; those in pink were computed in previous iterations; those in red are computed in the current iteration; and those in gray are not relevant. The parts of the matrices in light blue were used in previous iterations; those in blue are used in the current iteration; those in light red represent the application of the preconditioner; and those in gray are not relevant.

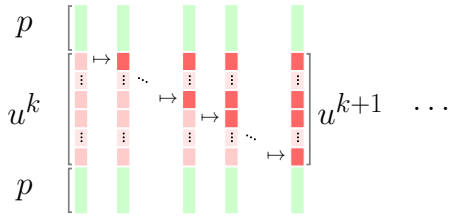
ALGORITHM 1. *apply_nonlinear* [recursive]

input : (p, u)
output: r
transfer u to each *subsys.p*
for *each subsys* **do**
  subsys.apply_nonlinear
end



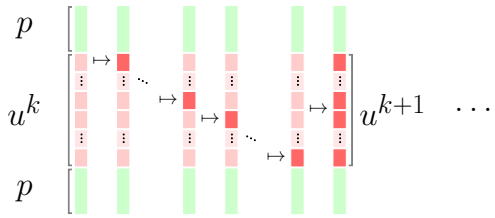
ALGORITHM 2. *solve_nonlinear* [GS]

input : p
output: u
while *not converged* **do**
  for *each subsys* **do**
    transfer u to *subsys.p*
    subsys.solve_nonlinear
  end
end



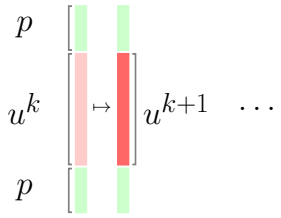
ALGORITHM 3. *solve_nonlinear* [Jacobi]

input : p
output: u
while *not converged* **do**
  transfer u to each *subsys.p*
  for *each subsys* **do**
    subsys.solve_nonlinear
  end
end



ALGORITHM 4. *solve_nonlinear* [Newton]

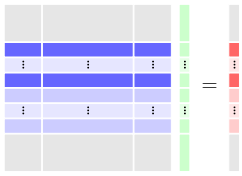
input : p
output: u
while *not converged* **do**
  apply_nonlinear
  $df \leftarrow -f$
  solve_linear
  $\alpha \leftarrow$ *line_search(du)*
  $u \leftarrow u + \alpha \cdot du$
end



In the next four figures, *fwd* indicates the "forward" mode, used to solve the Newton linear system or the left-hand equality of Eq. (33).

ALGORITHM 5. *apply_linear, fwd* [recursive]

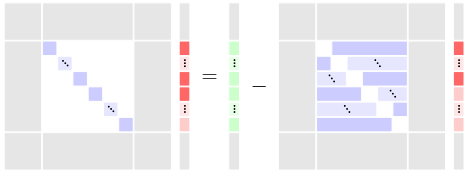
input : (dp, du)
output: dr
transfer du to each $subsys.dp$
for *each subsys* **do**
  $subsys.apply\_linear$
end



$$\frac{\partial R}{\partial (p,u)} \quad \begin{matrix} dp \\ du \end{matrix} \quad dr$$

ALGORITHM 6. *solve_linear, fwd* [GS]

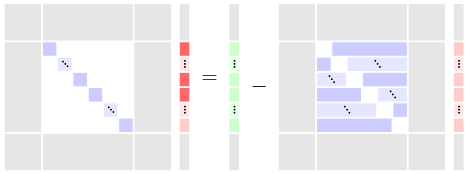
input : dr
output: du
$rhs \longleftarrow dr$
while *not converged* **do**
  for *each subsys* **do**
    transfer du to $subsys.dp$
    $subsys.apply\_linear$
    $subsys.dr \leftarrow rhs - subsys.dr$
    $subsys.solve\_linear$
  end
end



$$\frac{\partial R}{\partial u} \quad du^{k+1} \quad dr \qquad \frac{\partial R}{\partial u} \quad du^{k}$$

ALGORITHM 7. *solve_linear, fwd* [Jacobi]

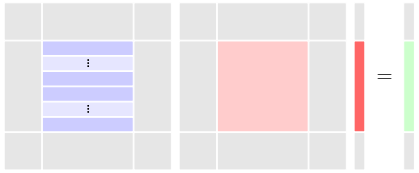
input : dr
output: du
$rhs \longleftarrow dr$
while *not converged* **do**
  transfer du to each $subsys.dp$
  for *each subsys* **do**
    $subsys.apply\_linear$
    $subsys.dr \leftarrow rhs - subsys.dr$
    $subsys.solve\_linear$
  end
end



$$\frac{\partial R}{\partial u} \quad du^{k+1} \quad dr \qquad \frac{\partial R}{\partial u} \quad du^{k}$$

ALGORITHM 8. *solve_linear, fwd* [Krylov]

input : dr
output: du
$rhs \leftarrow dr$
function `linear_operator`(x)
  $dr \leftarrow x$
  $solve\_linear$
  $apply\_linear$
  $y \leftarrow dr$
  return y
$du \leftarrow krylov(rhs, \texttt{linear\_operator})$



$$\frac{\partial R}{\partial u} \quad \sim \frac{\partial R^{-1}}{\partial u} \quad du^{k} \quad dr$$

Table 1: The five methods of the *System* class

Method	Operation		Called by
1. apply_nonlinear computes residuals	Inputs: p, u Outputs: r $r = R(p, u)$		• any nonlinear solver to check convergence • Newton solver to compute RHS
2. solve_nonlinear solves $R(p, u) = 0$ and computes outputs	Inputs: p Outputs: u $u = F(p)$		• run script or optimizer to run the model • parent system during a nonlinear block Gauss–Seidel or Jacobi solution
3. apply_linear provides $\partial R/\partial p$ and $\partial R/\partial u$ as a linear operator	*(forward mode)* Inputs: dp, du Outputs: dr $dr = \frac{\partial R}{\partial p} dp + \frac{\partial R}{\partial u} du$	*(reverse mode)* Inputs: dr Outputs: dp, du $dp = \frac{\partial R}{\partial p}^T dr$ $du = \frac{\partial R}{\partial u}^T dr$	• any linear solver to check convergence • all Krylov solvers
4. solve_linear provides $[\partial R/\partial u]^{-1}$ as a linear operator	*(forward mode)* Inputs: dr Outputs: du $\frac{\partial R}{\partial u} du = dr$	*(reverse mode)* Inputs: du Outputs: dr $\frac{\partial R}{\partial u}^T dr = du$	• run script or optimizer to compute derivatives using the unifying equation • parent system during a linear block Gauss–Seidel or Jacobi solution
5. linearize performs optional assembly or factorization of the Jacobian			• Newton solver after taking a step and just before solving the linear system • run script or optimizer prior to solving the unifying equation

Note: the subscript S in R_S, p_S, and u_S is omitted for brevity.

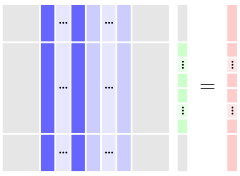
Table 2: Solver or operation executed for each type of method and system

Method	Assembly class	Component class
1. ***apply_nonlinear***	• Recursion	• User-implemented
2. ***solve_nonlinear*** (optional)	• Custom nonlinear solver • Newton's method • Nonlinear block Gauss–Seidel • Nonlinear block Jacobi	• Custom nonlinear solver • Newton's method
3. ***apply_linear***	• Recursion	• User-implemented • FD*
4. ***solve_linear*** (optional)	• Custom linear solver • Krylov method • Direct method • Linear block Gauss–Seidel • Linear block Jacobi	• Custom linear solver • Krylov method • Direct method
5. ***linearize***	• Recursion • User-implemented • Factorization	
	* FD: finite-difference approximation of the Jacobian	

In the next four figures, *rev* indicates the "reverse" mode, used to solve the right-hand equality of Eq. (33).

Algorithm 9. *apply_linear, rev* [recursive]

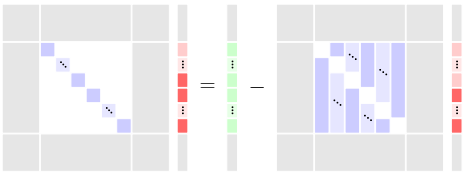
input : dr
output: (dp, du)
for *each subsys* **do**
  subsys.apply_linear
end
transfer each *subsys.dp* to du



$$\frac{\partial R}{\partial (p,u)}^T \quad dr \quad \begin{matrix} dp \\ du \end{matrix}$$

Algorithm 10. *solve_linear, rev* [GS]

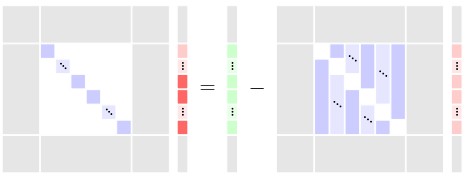
input : du
output: dr
$rhs \longleftarrow du$
while *not converged* **do**
  for *each subsys1* **do**
    for *each subsys2* **do**
      subsys2.apply_linear
    end
    transfer each *subsys2.dp* to du
    $subsys1.du \leftarrow rhs - subsys1.du$
    subsys1.solve_linear
  end
end



$$\frac{\partial R^T}{\partial u} \quad dr^{k+1} \quad du \qquad \frac{\partial R^T}{\partial u} \quad dr^k$$

Algorithm 11. *solve_linear, rev* [Jacobi]

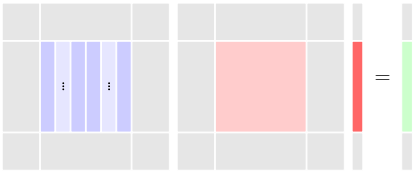
input : du
output: dr
$rhs \longleftarrow du$
while *not converged* **do**
  for *each subsys* **do**
    subsys.apply_linear
  end
  transfer each *subsys.dp* to du
  for *each subsys* **do**
    $subsys1.du \leftarrow rhs - subsys.du$
    subsys.solve_linear
  end
end



$$\frac{\partial R^T}{\partial u} \quad dr^{k+1} \quad du \qquad \frac{\partial R^T}{\partial u} \quad dr^k$$

Algorithm 12. *solve_linear, rev* [Krylov]

input : du
output: dr
$rhs \leftarrow du$
function `linear_operator`(x)
  $du \leftarrow x$
  solve_linear
  apply_linear
  $y \leftarrow du$
  return y
$dr \leftarrow krylov(rhs, \texttt{linear\_operator})$



$$\frac{\partial R^T}{\partial u} \quad \sim \frac{\partial R^{-T}}{\partial u} \quad dr^k \; du$$