

Author version

Published as:

Martins, J. R. R. A., & Hwang, J. T. (2016). Multidisciplinary design optimization of aircraft configurations — part 1: A modular coupled adjoint approach. In Von Karman Institute for Fluid Dynamics Lecture Series, Sint-Genesius-Rode, Belgium, May 2016.
<https://lsdolab.github.io/lsto-publications/publication/martins2016multidisciplinary/>

BibTeX for this reference:

<https://lsdolab.github.io/lsto-publications/publication/martins2016multidisciplinary/>

```
@inproceedings{martins2016multidisciplinary,  
  author = {Joaquim R. R. A. Martins and John T. Hwang},  
  title = {Multidisciplinary Design Optimization of Aircraft Configurations – Part 1: A  
    Modular Coupled Adjoint Approach},  
  booktitle = {Von Karman Institute for Fluid Dynamics Lecture Series,  
    Sint-Genesius-Rode, Belgium, May 2016},  
  year = {2016}  
}
```

Please cite this document as:

J. R. R. A. Martins and John T. Hwang. Multidisciplinary Design Optimization of Aircraft Configurations—Part 1: A modular coupled adjoint approach. Lecture series, Von Karman Institute for Fluid Dynamics, Sint-Genesius-Rode, Belgium, May 2016.

This document can be found at: <http://mdolab.engin.umich.edu>.

Multidisciplinary Design Optimization of Aircraft Configurations

Part 1: A modular coupled adjoint approach

Joaquim R. R. A. Martins and John T. Hwang
University of Michigan

Contents

1	Introduction	2
2	Methods for computing derivatives	4
2.1	Finite differences	4
2.2	Complex-step method	5
2.3	Automatic differentiation	6
2.4	Analytic methods	7
3	Unified mathematical formulation	10
3.1	Notation	11
3.2	Monolithic formulation	11
3.3	Derivative computation	12
4	Hierarchical solution strategy	15
4.1	Mathematical decomposition	18
4.2	Solution algorithm	18
4.3	Data structures	21
4.4	Implementation	22
5	Application to aircraft design	22
5.1	Motivation	22
5.2	Computational models	23
5.3	Problem description	27
5.4	Results	28
6	Summary	32

1 Introduction

Multidisciplinary design optimization (MDO) studies the theory and application of numerical optimization techniques to the design of engineering systems involving multiple disciplines or components. Aircraft are prime examples of multidisciplinary systems, so it is no coincidence that MDO emerged within the aerospace research community. MDO originated in the late 1970s (Haftka, 1977; Haftka and Shore, 1979), following the successful application of numerical optimization to structural design in the 1960s (Schmit, 1960; Schmit Jr., 1981). Aircraft design was one of the first applications of MDO because there is much to be gained by simultaneously considering the various disciplines involved.

One of the first applications of MDO was aircraft wing design, where aerodynamics, structures, and controls are strongly coupled disciplines (Ashley, 1982; Green, 1987; Grossman et al., 1988, 1990; Jansen et al., 2010; Kenway and Martins, 2014). MDO has since been extended to include aircraft sizing (Kroo et al., 1994; Wakayama, 1998) and has been applied to a wide range of other engineering systems (Martins and Lambe, 2013).

One crucial aspect to consider in MDO is how to organize the coupling and optimization of the various disciplines involved. Research on the options for this organization has lead to the development of various MDO *architectures* (Martins and Lambe, 2013). MDO architectures can be either *monolithic* or *distributed*. In a monolithic approach, a single optimization problem is solved. In a distributed approach, the same problem is partitioned into multiple subproblems involving smaller subsets of the variables and constraints. In spite of numerous efforts, distributed architectures have been unable to outperform monolithic architectures in terms of convergence rate (Tedford and Martins, 2010).

In aircraft design optimization problems, the objective and constraint functions are usually nonlinear, and thus a general purpose nonlinear optimizer that can handle constraints is required. There are two main classes of optimization algorithms: gradient-based and gradient-free (or derivative-free) algorithms. Gradient-free algorithms require only the values of the objective and constraint functions, while gradient-based algorithms, also require the gradients of these functions with respect to the design variables. Gradient-based methods utilize the gradient information to find the most promising directions in the design variable space, and converge to the optimum more quickly. This is especially true for problems with large numbers of design variables. Fig. 1 illustrates this by plotting the number of function evaluations required to minimize a multidimensional Rosenbrock function subject to nonlinear constraints. The number of function evaluation for the gradient-free optimizers (ALPSO, NSGA2) scale exponentially, and it becomes infeasible to perform optimization with respect to $\mathcal{O}(10^2)$ design variables or more using this type of algorithms. On the other hand, the gradient-based algorithms (SNOPT, SLSQP) scale much better, exhibiting linear convergence or better depending on the method used to compute the gradients. As we can see, computing the derivatives analytically makes a big difference in the performance. Thus, computing derivatives accurately and efficiently is essential for effective gradient-based optimization. The various methods for computing derivatives are described in Section 2.

One of the issues with gradient-based optimizers is that they are based on the assumption that the functions involved are continuous and differentiable. In practice, a few discontinuities or non-differentiable points do not impede a gradient-based optimizer from converging to a minimum, unless the discontinuity is at a minimum itself. Another issue often cited

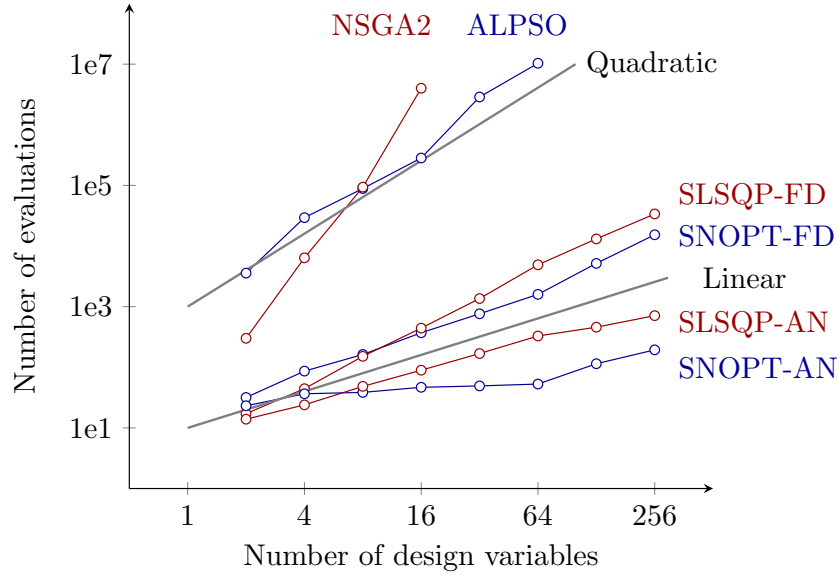


Figure 1: Computational cost of minimizing a constrained multidimensional Rosenbrock function with respect to the number of design variables. The gradient-free optimizers (ALPSO, NSGA2) scale exponentially, while the gradient-based optimizers (SNOPT, SLSQP) scale linearly with finite-difference derivatives (FD), and better than linearly with analytic derivatives (AN).

with gradient-based optimization is that it converges to only one local minimum. While this is true, there is no optimization algorithm that is guaranteed to find the global minimum of a general non-convex function. Some gradient-free algorithms (including the two benchmarked in Fig. 1) do a wider exploration of the design space, which might find a better local minimum and even the global minimum. However, gradient-free algorithms require large numbers of function evaluations, and the computational cost becomes intractable for $\mathcal{O}(10^2)$ design variables or more. In the author’s experience, the existence of multiple local minima (multimodality) has been overstated. In aerodynamic shape optimization, for example, we tried to find multiple local minima, but we were unsuccessful (Lyu et al., 2015). We have had similar experience in other aircraft design and engineering systems optimization problems.

Given the facts above, it is clear that our only hope for solving large-scale aircraft design optimization problems—problems with $\mathcal{O}(10^2)$ design variables or more—is to use a gradient-based optimization. One of the main challenges we tackle in these these lectures is to develop methods for computing gradients as accurately and efficiently as possible. Computing the derivatives of systems where multiple disciplines are coupled introduces another dimension to this challenge.

In the next section of this lecture, we start by introducing the various methods for computing derivatives. Then, we unify these methods in Section 3 using a single equation: the unifying chain rule. This unification leads to a new framework for solving the state of multidisciplinary systems and their derivatives, which we describe in Section 4. To demonstrate this new strategy, we use it in an aircraft application in Section 5, where the mission profile,

airline allocation, and aircraft design are simultaneously optimized.

In Part 2 of this lecture, we develop a framework for the MDO of aircraft configurations based on high-fidelity models, with emphasis on the coupling of the aerodynamics and structures disciplines. This leads to results in the aerostructural design optimization of flexible wings, which we discuss in detail.

2 Methods for computing derivatives

Derivatives play a central role in several numerical algorithms, such as Newton–Krylov methods applied to the solution of the PDEs. In the context of this lecture, we are particularly interested in computing derivatives to inform gradient-based optimization algorithms.

The accuracy of the derivative computation affects the convergence behavior of the solver used in the algorithm. In the case of gradient-based optimization, accurate derivatives are important to ensure robust and efficient convergence, especially for problems with large numbers of design variables and constraints. The precision of the gradients limits how close we can get to the optimum solution, and inaccurate gradients can cause the optimizer to halt or to take a less direct route to the optimum that requires more iterations.

To solve an optimization problem, a gradient-based algorithm requires the derivatives of the objective function with respect to each design variable, and the derivatives of all the constraints with respect to the design variables. In general, we need to compute the derivatives vector-valued function f with respect to a vector of independent variables x , which yields a Jacobian matrix,

$$\frac{df}{dx} = \begin{bmatrix} \frac{df_1}{dx_1} & \cdots & \frac{df_1}{dx_{n_x}} \\ \vdots & \ddots & \vdots \\ \frac{df_{n_f}}{dx_1} & \cdots & \frac{df_{n_f}}{dx_{n_x}} \end{bmatrix}, \quad (1)$$

which is an $n_f \times n_x$ matrix.

2.1 Finite differences

Finite-difference methods are widely used to compute derivatives due to their simplicity and the fact that they can be implemented even when a the function evaluation is provided as black box. Finite-difference formulas are derived by combining Taylor series expansions. Using the right combinations of these expansions, it is possible to obtain finite-difference formulas that estimate an arbitrary order derivative with any required order truncation error. The simplest finite-difference formula can be directly derived from one Taylor series expansion, yielding

$$\frac{df}{dx_j} = \frac{f(x + e_j h) - f(x)}{h} + \mathcal{O}(h), \quad (2)$$

where h is a small perturbation and e_j is a vector of zeros with a unity entry in row j . This formula requires the evaluation of the model at the reference point x and one perturbed point $x + e_j h$, and yields one column of the Jacobian (1). Each additional column requires an additional evaluation of the computational model. Hence, the cost of computing the complete

Jacobian is proportional to the number of input variables, n_x . When the computational model is nonlinear, the constant of proportionality with respect to the number of variables can be less than one if the solutions for the successive steps are warm-started with the previous solution.

Estimating derivatives using finite-difference formulas is subject to the *step-size dilemma*: we want to reduce the truncation error by reducing h but below a certain value of h , errors due to subtractive cancellation become dominant, and for small enough h , the finite-difference formula just yields zero.

Most gradient-based optimizers use finite-differences by default to compute the gradients. For problems with large numbers of design variables, the computation of the derivatives usually becomes the bottleneck in the optimization cycle. In addition, inaccuracies in the derivatives due to subtractive cancellation are often the culprit in cases where gradient-based optimizers fail to converge. Therefore, we need methods that are both more accurate and efficient.

2.2 Complex-step method

The complex-step derivative approximation computes derivatives of real functions using complex variables. This method originated with the work of Lyness and Moler (1967). They developed several methods that made use of complex variables, including a reliable method for calculating the n^{th} derivative of an analytic function. This technique was rediscovered by Squire and Trapp (1998), who derived a simple formula for estimating the first derivative. This estimate is very accurate, extremely robust, and easy to implement (Martins et al., 2003).

The first application of this approach to an iterative solver was by Anderson et al. (2001), who used it to compute derivatives of a Navier–Stokes solver, and later multidisciplinary systems (Newman III et al., 2003). Martins et al. (2003) showed that the complex-step method is generally applicable to any computer program and described the detailed procedure for its implementation. They also presented an alternative way of deriving and understanding the complex step, and connect this to automatic differentiation (explained in Section 2.3). The complex-step method requires access to the source code. Martins et al. (2003) provide a script that facilitates the implementation of the complex-step method to Fortran codes, and explain how to implement this method in Matlab, C/C++, and Python.

The cost of computing a gradient using the complex-step, like finite differences, is proportional to the number of design variables, so it is not recommended for large-scale optimization. However, the complex-step approach has been extremely useful in the verification of high-fidelity aerodynamic (Lyu et al., 2013; Mader et al., 2008) and aerostructural derivatives (Kenway et al., 2014; Martins et al., 2005). The complex-step method is now widely used, with applications not only in engineering, but in the natural sciences as well.

The complex-step derivative approximation, like finite-difference formulas, can also be derived using a Taylor series expansion. Rather than using a real step h , we now use a pure imaginary step, ih . If f is a real function in real variables and it is also analytic, we can expand it in a Taylor series about a real point x as follows:

$$f(x + ihe_j) = f(x) + ih \frac{df}{dx_j} - \frac{h^2}{2} \frac{d^2f}{dx_j^2} - \frac{ih^3}{6} \frac{d^3f}{dx_j^3} + \dots \quad (3)$$

Taking the imaginary parts of both sides of this equation and dividing it by h yields

$$\frac{df}{dx_j} = \frac{\text{Im}[f(x + ihe_j)]}{h} + \mathcal{O}(h^2), \quad (4)$$

Because there is no subtraction operation in the complex-step derivative approximation (4), the only source of numerical error is the $\mathcal{O}(h^2)$ truncation error. By decreasing h to a small enough value, we can ensure that the truncation error is of the same order as the numerical precision of the evaluation of f .

To show how the complex-step method works, we consider an analytic function for which the derivative was computed to 16 digits and then compared the relative error of the complex-step approximation (4) to that of the forward finite-difference (2). As we can see in Fig. 2, the forward-difference estimate initially converges to the exact result at a linear rate since its truncation error is $\mathcal{O}(h)$. However, as the step is reduced below a value of about 10^{-8} , the subtractive cancellation error becomes significant and the estimates are unreliable. When the interval h is so small that no difference exists in the output (for steps smaller than 10^{-16}) the finite-difference estimates eventually yields zero and then the relative error is 1. Comparing the best accuracy of each of these approaches, we can see that by using finite-difference we only achieve a fraction of the accuracy that is obtained by using the complex-step approximation.

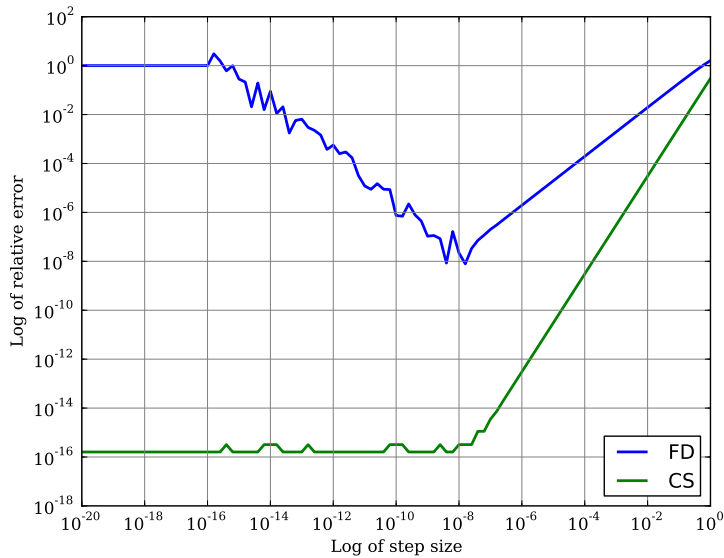


Figure 2: Relative error in the derivative estimate versus step size.

2.3 Automatic differentiation

Symbolic differentiation is another option for computing derivatives. One can differentiate explicit mathematical expressions by hand or by using symbolic differentiation software. However, many engineering models cannot be written explicitly in closed form, relying instead

on iterative numerical procedures expressed as computer programs. Automatic differentiation (AD)—also known as algorithmic differentiation—is based on the systematic symbolic differentiation of each line of a computer program, and the accumulation of total derivatives using the chain rule. The method relies on tools that automatically produce a program that computes user-specified derivatives based on the original program (Griewank, 2000).

When using AD, we consider all variables assigned in a computer program, $t = [t_1, t_2, \dots, t_m]$. We consider the first n variables in this set to be *independent* variables $t_1, t_2, \dots, t_n$, which for our purposes are the design variables, x . We also need to consider the *dependent* variables, $t_{n+1}, t_{n+2}, \dots, t_m$. These are all the intermediate variables in the algorithm, including the outputs, f , which are the functions we want to differentiate. We can then write the sequence of operations in any algorithm as

$$t_i = T_i(t_1, t_2, \dots, t_{i-1}), \quad i = n+1, n+2, \dots, m. \quad (5)$$

The chain rule can be applied to each of these operations and can be written as

$$\frac{dt_i}{dt_j} = \delta_{ij} + \sum_{k=j}^{i-1} \frac{\partial T_i}{\partial t_k} \frac{\partial t_k}{\partial t_j}. \quad (6)$$

Using the forward mode, we choose one j and keep it fixed. We then work our way forward in the index i until we get the desired derivative. The reverse mode, on the other hand, works by fixing i to the index of the quantity we want to differentiate, and working our way backwards in the index j all the way down to the independent variables.

There are two main ways of implementing automatic differentiation: source code transformation and operator overloading. Tools that use source code transformation add new statements to the original source code that compute the derivatives of the original statements. The operator overloading approach consists in defining a new user-defined type that is used instead of real numbers. This new type includes not only the value of the original variable, but its derivative as well. All intrinsic operations and functions have to be redefined (overloaded) to compute the derivatives together with the original computations. The operator overloading approach results in fewer changes to the original code, but is usually less efficient (Griewank, 2000; Pryce and Reid, 1998).

There are automatic differentiation tools available for a variety of programming languages including C/C++, and Fortran (Carle and Fagan, 2000; Giering and Kaminski, 2002; Gockenbach, 2000; Hascoët and Pascual, 2004; Pascual and Hascoët, 2005).

2.4 Analytic methods

Analytic methods are based on the linearization of the numerical model equations. Like AD, the numerical precision of analytic methods is the same as that of the original algorithm. In addition, analytic methods are usually more efficient than AD for a given problem. However, analytic methods are much more involved than the other methods, since they require detailed knowledge of the computational model and a long implementation time. Analytic methods are applicable when we have a quantity of interest f that depends implicitly on the independent variables of interest x as follows

$$f = F(x, y(x)). \quad (7)$$

The implicit relationship between the state variables y and the independent variables is defined by the solution of a set of governing equations that can be written as residuals,

$$r = R(x, y(x)) = 0. \quad (8)$$

By writing the computational model in this form, we have assumed a *discrete* analytic approach. This is in contrast to the *continuous* approach, in which the equations are differentiated before being discretized. We do not discuss the continuous approach in this lecture, but ample literature can be found on the subject (Anderson Jr., 1999; Giles and Pierce, 2000; Jameson, 1988; Liu and Canfield, 2012; Sirkes and Tziperman, 1997), including discussions comparing the two approaches (Dwight and Brezillon, 2006; Nadarajah and Jameson, 2000; Peter and Dwight, 2010).

As a first step toward obtaining the derivatives that we want to compute, we use the chain rule to write the total derivative of f as

$$\frac{df}{dx} = \frac{\partial F}{\partial x} + \frac{\partial F}{\partial y} \frac{dy}{dx}, \quad (9)$$

where the result is an $n_f \times n_x$ Jacobian matrix. It is important to distinguish the total and partial derivatives and define their context. The partial derivatives represent the variation of $f = F(x)$ with respect to changes in x for a fixed y , while the total derivative df/dx takes into account the change in y that is required to keep the residual equations (8) equal to zero. Since the governing equations must always be satisfied, the total derivative of the residuals (8) with respect to the design variables must also be zero. Thus, using the chain rule we obtain

$$\frac{dr}{dx} = \frac{\partial R}{\partial x} + \frac{\partial R}{\partial y} \frac{dy}{dx} = 0. \quad (10)$$

The partial derivatives in Eqs. (9) and (10) can be computed using the methods described earlier (finite differences, complex step, and AD). The computation of the total derivative matrix dy/dx has a much higher computational cost than any of the partial derivatives, since it requires the solution of the residual equations (8). Rearranging the linearized residual equations (10) we can compute the total derivative matrix dy/dx by solving the linear system,

$$\frac{\partial R}{\partial y} \frac{dy}{dx} = -\frac{\partial R}{\partial x}. \quad (11)$$

Substituting this result into the total derivative equation (9), we obtain

$$\frac{df}{dx} = \frac{\partial F}{\partial x} - \underbrace{\frac{\partial F}{\partial y} \left[\frac{\partial R}{\partial y} \right]^{-1}}_{\psi} \frac{\partial R}{\partial x}. \quad (12)$$

The inverse of the square Jacobian matrix $\partial R/\partial y$ is not necessarily calculated explicitly. We use the inverse to denote that this matrix needs to be solved as a linear system with some right-hand-side vector. Equation (12) shows that there are two ways of obtaining the total derivative matrix dy/dx , depending on which right-hand side is chosen for the solution of the linear system.

2.4.1 Direct method

The direct method involves solving the linear system with $-\partial R/\partial x$ as the right-hand side vector, which results in the linear system (11). This linear system needs to be solved for n_x right-hand sides to get the full Jacobian matrix dy/dx . Then, we can use dy/dx in Eq. (9) to obtain the derivatives of interest, df/dx .

As in the case of finite differences, the cost of computing derivatives with the direct method is proportional to the number of design variables, n_x . In a case where the computational model is a nonlinear system, the direct method can be advantageous. Both methods require the solution of a system of the same size n_x times, but the direct method just solves the linear system (11), while the finite-difference method solves the original nonlinear system (8). Even though the various solutions required for the finite-difference method can be warm-started from a previous solution, a nonlinear solution typically requires multiple iterations to converge. The direct method is even more advantageous when a factorization of $\partial R/\partial y$ is available, since each solution of the linear system would then consist in an inexpensive back substitution.

2.4.2 Adjoint method

Adjoint methods for computing derivatives have been known and used for over three decades. They were first applied to solve optimal control problems and thereafter used to perform sensitivity analysis of linear structural finite element models. The first application to fluid dynamics is due to Pironneau (1974). The method was then extended by Jameson (1988) to perform airfoil shape optimization, and later applied to three-dimensional problems, leading to aerodynamic shape optimization of complete aircraft configurations (Lyu and Martins, 2013; Reuther et al., 1999a,b; Vassberg and Jameson, 2002) and shape optimization considering both aerodynamics and structures (Kenway et al., 2012; Martins et al., 2004). The adjoint method has since been generalized for multidisciplinary systems (Martins and Hwang, 2013).

The adjoint equation is encapsulated in the total derivative equation (12), where we observe that there is an alternative option for computing the total derivatives: the linear system involving the large square Jacobian matrix $\partial R/\partial y$ can be solved with $\partial f/\partial y$ as the right-hand side. This results in the *adjoint equations*,

$$\left[\frac{\partial R}{\partial y}\right]^T \psi = -\left[\frac{\partial F}{\partial y}\right]^T, \quad (13)$$

which is a linear systems where ψ is the *adjoint matrix* (of size $n_y \times n_f$). Although this is usually expressed as a vector, we obtain a matrix due to our generalization for the case where f is a vector. This linear system needs to be solved for each column of $[\partial F/\partial y]^T$, and thus the computational cost is proportional to the number of quantities of interest, n_f . The adjoint vector can then be substituted into Eq. (12) to find the total derivative,

$$\frac{df}{dx} = \frac{\partial F}{\partial x} + \psi^T \frac{\partial R}{\partial x}. \quad (14)$$

Thus, the cost of computing the Jacobian $\partial f/\partial x$ using the adjoint method is independent of the number of design variables, n_x , and instead it is proportional to the number of quantities of interest, n_f .

As previously mentioned, the partial derivatives shown in these equations need to be computed using some other method. They can be differentiated symbolically or computed by finite differences, the complex-step method, or even AD. The use of AD to automatically produce the code that computes these partial derivatives has shown to be particularly effective in the development of analytic methods for PDE solvers (Lyu et al., 2013; Mader et al., 2008). More detail on this technique is provided in Part 2 of this lecture.

By comparing the direct and adjoint method, we also notice that all the partial derivative terms that need to be computed are identical, and that the difference in their relative cost comes only from the choice of which right-hand side to use with the residual Jacobian. Figure 3 shows the sizes of the matrices in Eq. (12), which depend on the shape of df/dx . These diagrams illustrate why the direct method is preferable when $n_x < n_f$ and the adjoint method is more efficient when $n_x > n_f$.

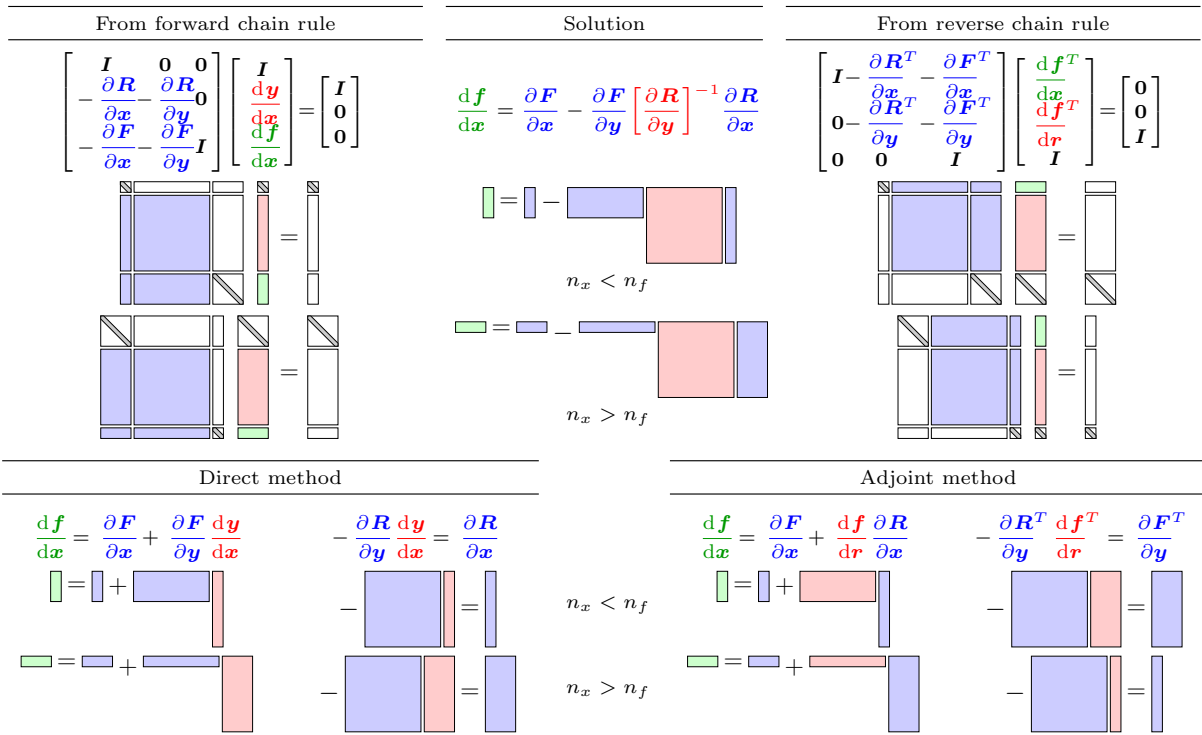


Figure 3: Block matrix diagrams illustrating the structure of the direct and adjoint equations, assuming that $n_y \gg n_x, n_f$. The blue matrices consist of partial derivatives, which are relatively cheap to compute, and the red matrices consist of the total derivatives computed by solving the linear systems.

3 Unified mathematical formulation

In this sections, we derive an equation based on the chain rule of differentiation that unifies all the methods described in Section 2, which inspired a new perspective on the mathematical formulation of systems with multiple numerical models. This new perspective leads to the hierarchical strategy for solving multidisciplinary systems and for computing the derivatives

of these systems that we describe in Section 4.

In our formulation, we use the term *numerical model* to refer to the discrete variables and their explicit or implicit definitions, whereas *computational model* refers to the code that implements the numerical model. Section 3.1 describes the notation for any general numerical model, Sec. 3.2 formulates the general numerical model as a single system of algebraic equations, and in Sec. 3.3 we show how the derivatives of the coupled system are computed.

3.1 Notation

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 for the design variables: We call each scalar varied by the optimizer a separate design variable, to remain consistent with the terminology used in the optimization literature.

Fundamentally, numerical models capture the relationships between quantities, i.e., the response of one or more quantities to independent changes in other quantities. Thus, it is useful to classify the variables in a numerical model as either input, state, or output variables. Input variables are independent variables whose values are set externally by either the designer or the optimizer, so the design variables (x) for an optimization problem are a subset of the input variables. Output variables are the dependent variables of interest, computed explicitly as a function of the input and state variables, and the output variables contain the objective and constraints in an optimization problem (f). State variables (y) are dependent variables computed by the model (8) in the process of computing the output variables, and they are defined by implicit or explicit functions of the design variables and other state variables. Using this nomenclature, a numerical model can be expressed as:

$$\begin{aligned}
y_1 &= Y_1(x_1, \dots, x_m, y_2, \dots, y_p), \\
&\vdots \\
y_p &= Y_p(x_1, \dots, x_m, y_1, \dots, y_{p-1}), \\
f_1 &= F_1(x_1, \dots, x_m, y_1, \dots, y_p), \\
&\vdots \\
f_p &= F_p(x_1, \dots, x_m, y_1, \dots, y_p),
\end{aligned} \tag{15}$$

where some of the state variables also have a residual function R_k associated with them.

3.2 Monolithic formulation

We now reformulate the numerical model (15) as a single system of algebraic equations. We assume that x_k^* is the value of input variable x_k for all $k = 1, \dots, m$, at the point at which the numerical model is being evaluated. The first step is to concatenate the set of input,

state and output variables into a single vector,

$$u = (u_1, \dots, u_n)^T = (x_1, \dots, x_m, y_1, \dots, y_p, f_1, \dots, f_q)^T, \quad (16)$$

where $n = m + p + q$.

The corresponding residuals are given by,

$$\begin{aligned} R_1(u) &= x_1 - x_1^*, \\ &\vdots \\ R_m(u) &= x_m - x_m^*, \\ R_{m+1}(u) &= \begin{cases} y_1 - Y_1(x_1, \dots, x_m, y_2, \dots, y_p) & , y_1 \text{ is explicitly defined} \\ -R_1(x_1, \dots, x_m, y_1, \dots, y_p) & , y_1 \text{ is implicitly defined} \end{cases} , \\ &\vdots \\ R_{m+p}(u) &= \begin{cases} y_p - Y_p(x_1, \dots, x_m, y_1, \dots, y_{p-1}) & , y_p \text{ is explicitly defined} \\ -R_p(x_1, \dots, x_m, y_1, \dots, y_p) & , y_p \text{ is implicitly defined} \end{cases} , \\ R_{m+p+1}(u) &= f_1 - F_1(x_1, \dots, x_m, y_1, \dots, y_p), \\ &\vdots \\ R_{m+p+q}(u) &= f_q - F_q(x_1, \dots, x_m, y_1, \dots, y_p). \end{aligned} \quad (17)$$

The numerical model can be written more compactly as,

$$\left. \begin{aligned} R_1(u_1, \dots, u_n) &= 0 \\ &\vdots \\ R_n(u_1, \dots, u_n) &= 0 \end{aligned} \right\} \Leftrightarrow R(u) = 0. \quad (18)$$

The unified formulation of a numerical model is the algebraic system of equations $R(u) = 0$, which we call the *fundamental system*. Its significance lies in the fact that the vector u^* that solves the fundamental system (18) satisfies the numerical model (15).

3.3 Derivative computation

In the context of optimization, the design variables that the optimizer varies are a subset of the input variables of the numerical model, and the objective and constraint functions given to the optimizer are a subset of the output variables of the numerical model. Thus, the derivatives of interest are a subset of the derivatives of the output variables with respect to the input variables.

Starting from the fundamental system (18), we can derive an equation that unifies all four methods described in Sec. 2. As we will see, each method is determined the choice of the state variables—on one extreme, including no state variables in the vector u results in a finite-step method and on the other extreme, including the variables from every line of code in a computational model results in automatic differentiation (Martins and Hwang, 2013).

To obtain the analytic methods of Sec. 2.4, the fundamental system must be defined with the original residual used for each implicit state variable. For the explicit state variables, the residuals are defined as the value of the state variables minus the output of the explicit

function. We begin by defining

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ \vdots \\ y_p \end{pmatrix}, \quad f = \begin{pmatrix} f_1 \\ \vdots \\ f_q \end{pmatrix} \quad (19)$$

as the input, state, and output vectors and defining the respective functions,

$$Y = \begin{pmatrix} Y_1 \\ \vdots \\ Y_p \end{pmatrix}, \quad F = \begin{pmatrix} F_1 \\ \vdots \\ F_q \end{pmatrix} \quad (20)$$

Then, the fundamental system has the form,

$$u = \begin{pmatrix} x \\ y \\ f \end{pmatrix} \quad \text{and} \quad R(u) = \begin{pmatrix} x - x^* \\ -R(x, y) \\ f - F(x, y) \end{pmatrix}, \quad (21)$$

and the numerical model is encapsulated in the function,

$$G : x \mapsto F(x, Y(x)), \quad (22)$$

which maps the inputs x to the outputs f . The derivatives of interest are $\partial G / \partial x$. The following proposition shows how $\partial G / \partial x$ can be computed from the fundamental system defined by Eq. (21).

Proposition 1. *Let R and u be as defined by Eq. (21). If $\partial R / \partial u$ is invertible and the inverse is defined as*

$$\frac{\partial R}{\partial u}^{-1} = \begin{bmatrix} A^{xx} & A^{xy} & A^{xf} \\ A^{yx} & A^{yy} & A^{yf} \\ A^{fx} & A^{fy} & A^{ff} \end{bmatrix}, \quad (23)$$

then the derivatives we seek are in the bottom left block in this Jacobian, i.e.,

$$\frac{\partial \mathcal{G}}{\partial x} = A^{fx}, \quad (24)$$

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

Proof. By construction, we have

$$\frac{\partial R}{\partial u} \frac{\partial (R^{-1})}{\partial r} = \mathcal{I} \implies \begin{bmatrix} \mathcal{I} & 0 & 0 \\ -\frac{\partial R}{\partial x} & -\frac{\partial R}{\partial y} & 0 \\ -\frac{\partial F}{\partial x} & -\frac{\partial F}{\partial y} & \mathcal{I} \end{bmatrix} \begin{bmatrix} A^{xx} & A^{xy} & A^{xf} \\ A^{yx} & A^{yy} & A^{yf} \\ A^{fx} & A^{fy} & A^{ff} \end{bmatrix} = \begin{bmatrix} \mathcal{I} & 0 & 0 \\ 0 & \mathcal{I} & 0 \\ 0 & 0 & \mathcal{I} \end{bmatrix}. \quad (25)$$

Block-forward substitution for the first block-column yields

$$\begin{bmatrix} A^{xx} \\ A^{yx} \\ A^{fx} \end{bmatrix} = \begin{bmatrix} \mathcal{I} \\ -\frac{\partial R}{\partial y}^{-1} \frac{\partial R}{\partial x} \\ \frac{\partial F}{\partial x} - \frac{\partial F}{\partial y} \frac{\partial R}{\partial y}^{-1} \frac{\partial R}{\partial x} \end{bmatrix}. \quad (26)$$

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

$$\frac{\partial \mathcal{G}}{\partial x} = \begin{bmatrix} \frac{\partial F}{\partial x} & \frac{\partial F}{\partial y} \end{bmatrix} \begin{bmatrix} \mathcal{I} \\ \frac{\partial Y}{\partial x} \end{bmatrix}. \quad (27)$$

Since $\partial \mathcal{R} / \partial y$ is invertible, the implicit function theorem states

$$\frac{\partial Y}{\partial x} = - \frac{\partial R}{\partial y}^{-1} \frac{\partial R}{\partial x}. \quad (28)$$

Combining the two equations above yields

$$\frac{\partial G}{\partial x} = \frac{\partial F}{\partial x} - \frac{\partial F}{\partial y} \frac{\partial R}{\partial y}^{-1} \frac{\partial R}{\partial x}. \quad (29)$$

Therefore,

$$\frac{\partial G}{\partial x} = A^{fx}, \quad (30)$$

as required. $\square$

The application of the inverse function theorem explains why the lower left $n_f \times n_x$ block of $\partial R / \partial u$ is equal to the Jacobian with the derivatives of interest that we denoted in Sec. 2 as df/dx . Assuming $\partial R / \partial u$ is invertible, the inverse function theorem guarantees the existence of a locally defined inverse function $R^{-1} : r \mapsto u | R(u) = r$ that satisfies

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

The concept of a total derivative is used in many settings, but it is difficult to find a clear definition in the literature—they are usually defined in terms of other total derivatives as in Eq. (12). Total derivatives are useful for distinguishing direct and indirect dependence on a variable. In the total derivative (12), df/dx captures both the explicit dependence of F on the argument x and the indirect dependence via other arguments of F (y in this case) that depend on x .

The Jacobian $\partial(R^{-1})/\partial r$ captures a similar relationship because the (i, j) th entry of the matrix $\partial(R^{-1})/\partial r$ captures the dependence of the i th component of u on the j th 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 $\partial R / \partial u$ is invertible at the solution of this system. The matrix of total derivatives du/dr is defined to be

$$\frac{du}{dr} = \frac{\partial(R^{-1})}{\partial r}, \quad (32)$$

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

Following from Eq. (31), the matrix du/dr is also equal to the inverse of $\partial R/\partial u$, leading to

$$\frac{\partial R}{\partial u} \frac{du}{dr} = \mathcal{I} = \frac{\partial R^T}{\partial u} \frac{du^T}{dr}, \quad (33)$$

which we refer to as the *unifying chain rule equation*. The structure of the left inequality in this equation is shown in Fig. 4. The left and right equalities are denoted as the *forward mode* and the *reverse mode*, respectively, drawing inspiration from terminology used in AD (Sec. 2.3). The unifying chain rule (33) was first presented by Martins and Hwang (2013) with different notation and derivation. In that paper, we also show how this equation unifies all methods—how the finite-difference method, complex-step method, AD, direct method, adjoint method, and the chain rule can all be derived from the unifying chain rule (33). Basically, we can derive the equations for the different methods from the unifying chain rule (33) by the appropriate definition of R and u . In the extreme case of the finite-difference or complex-step methods, u is simply the concatenation of the inputs x and outputs f . In the other end of the spectrum, defining u as every single value in a computer program yields AD (Martins and Hwang, 2013).

For the independent variables, the r in the denominator of du/dr can be replaced with the symbol for the variable itself, as shown in Fig. 4. The derivatives of interest are df/dx , which is a sub-block of du/dr . Computing df/dx involves solving a linear system with multiple right-hand sides, so this is more efficient with the left or right equality in Eq. (33), depending on the relative sizes of f and x .

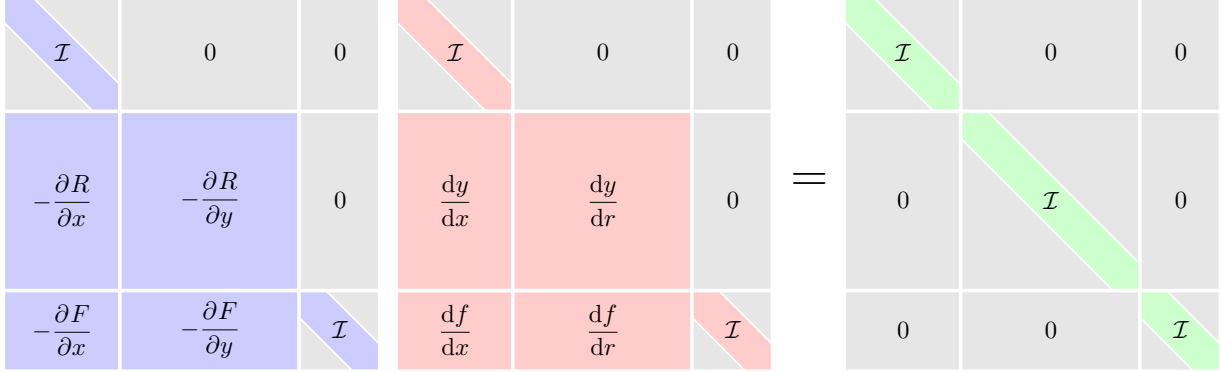


Figure 4: Block structure of the matrices in the left equality of Eq. (33).

4 Hierarchical solution strategy

The unifying chain rule (33) applies to multidisciplinary systems (where R and u contain multiple sets of sub-vectors), and leads to the modular analysis and unified derivatives (MAUD) MDO architecture. The high-level objective of the MAUD architecture is to facilitate two tasks: the evaluation of a computational model with multiple components and the efficient computation of its 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 an elegant way that simplifies the implementation of the computational framework. The task of evaluating a coupled computational model reduces to solving a system of algebraic equations, and the task of computing derivatives reduces to solving a system of linear equations.

At the highest level, the MAUD solves four types of systems:

1. The nonlinear system	2. The Newton system	3. Derivatives in forward mode	4. Derivatives in reverse mode
$R(u) = 0$	$\frac{\partial R}{\partial u} \Delta u = -r$	$\frac{\partial R}{\partial u} \frac{du}{dr} = \mathcal{I}$	$\frac{\partial R^T}{\partial u} \frac{du^T}{dr} = \mathcal{I}$

The MAUD architecture is designed to handle components with distributed data parallelism. This means that a problem in MAUD is not necessarily a single system of equations. Such a monolithic system could result in a significant memory and computational time overhead. For example, if a computational model has two components that can be run in sequence because the second component does not influence the first one, it would be more efficient to run a block Gauss–Seidel iteration over the two components rather than using a Krylov solver for the whole system. Fig. 5 shows several problems with unique structures that MAUD is able to exploit. To exploit problem structure, the MAUD architecture performs a recursive hierarchical algorithm for solving the nonlinear and linear systems. MAUD also adopts a matrix-free approach, since it uses only nonlinear and linear operations on vectors.

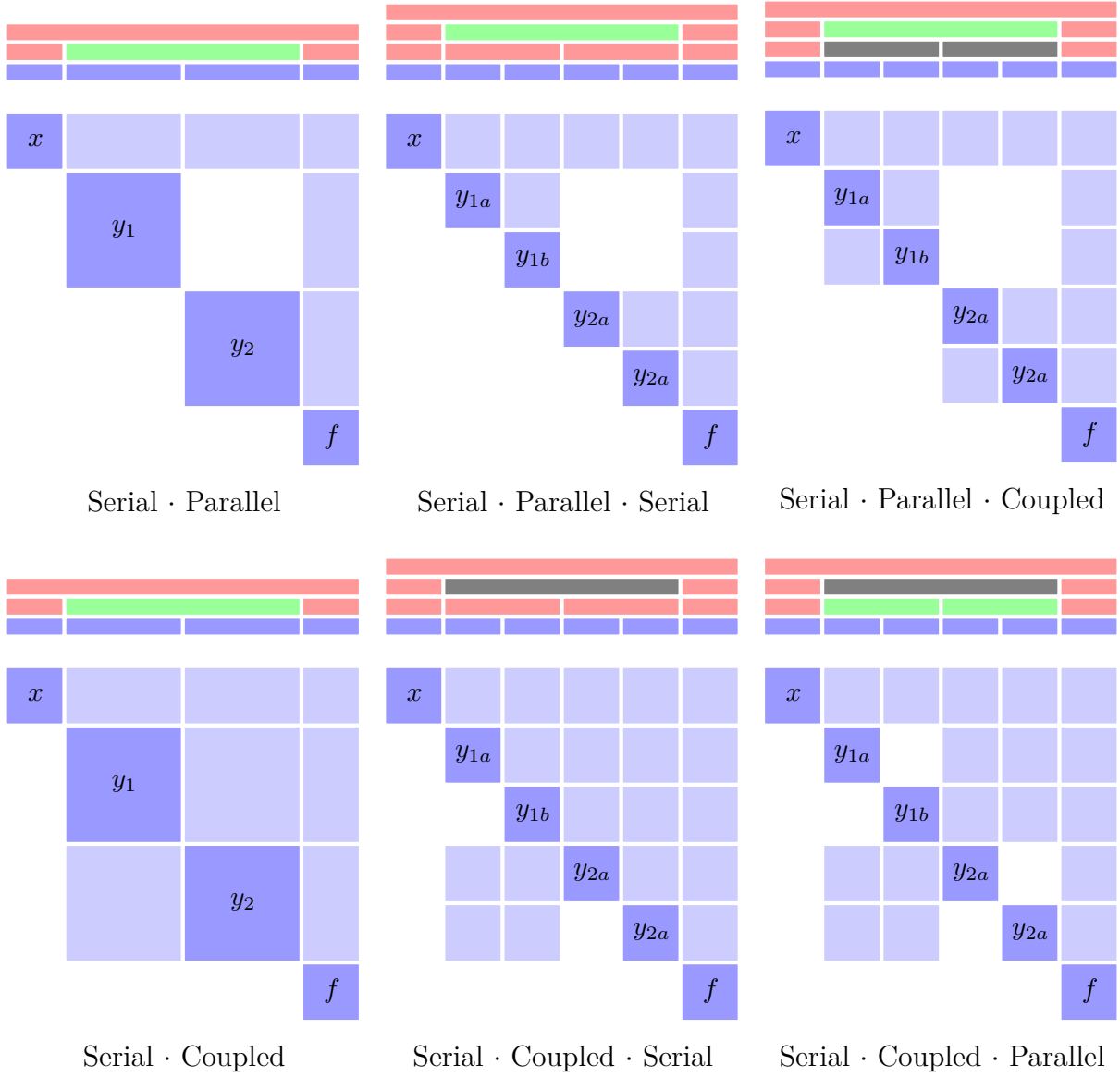


Figure 5: MAUD exploits the structure of multidisciplinary systems by using assemblies of serial, parallel, and coupled solution solvers.

This section describes the MAUD hierarchical solution strategy. Section 4.1 begins by presenting the hierarchical decomposition of the fundamental system (18) into smaller systems of algebraic equations. Section 4.2 presents the MAUD hierarchical decomposition algorithm, and Sec. 4.3 presents the design of MAUD’s data structures.

4.1 Mathematical decomposition

To solve the fundamental system (18) efficiently, we partition the global unknown vector and system of equations into smaller sets. This partitioning is recursive for additional flexibility in handling complex multidisciplinary systems, resulting in a hierarchical decomposition of the fundamental system (18).

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

$$S = \{i + 1, \dots, j\}, \quad 0 \leq i < j \leq n, \quad (34)$$

to represent the indices of the variables that make up the unknowns for this smaller algebraic system. The residual function for this intermediate system is

$$R_S : D_1 \times \dots \times D_n \rightarrow \mathbb{R}^{N_{i+1}} \times \dots \times \mathbb{R}^{N_j}, \quad (35)$$

formed by concatenating the residual functions corresponding to the indices in S . Let $p_S = (u_1, \dots, u_i, u_{j+1}, \dots, u_n)$ and $u_S = (u_{i+1}, \dots, u_j)$ partition u into the *parameter vector* and *unknown vector*, respectively. Then, the intermediate system for index set S implicitly defines u_S as a function of p_S :

$$\left. \begin{array}{l} R_{i+1}(u_1, \dots, u_i, u_{i+1}, \dots, u_j, u_{j+1}, \dots, u_n) = 0 \\ \vdots \\ R_j(\underbrace{u_1, \dots, u_i}_{p_S}, \underbrace{u_{i+1}, \dots, u_j}_{u_S}, \underbrace{u_{j+1}, \dots, u_n}_{p_S}) = 0 \end{array} \right\} \Leftrightarrow R_S(p_S, u_S) = 0. \quad (36)$$

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 owns only one variable or a group of related variables in its unknown vector is analogous to executing a component in the traditional view of MDO—that is, computing the component’s outputs given the values of the inputs, which are contained in p_S in this case. However, this formulation allows the definition of intermediate systems that group together other intermediate systems that may correspond to components, and overall, this enables a hierarchical decomposition of all the variables in the computational model.

4.2 Solution algorithm

We now present the hierarchical solution algorithm at the core of the MAUD architecture. The fundamental system owns the full set of n variables in the numerical model at the hierarchy tree root level. The fundamental system contains a group of intermediate systems that together partition the n variables and form the second level of the hierarchy tree. Each of these intermediate systems can contain other intermediate systems, and so on, until we

reach the leaves of the hierarchy tree, i.e., intermediate systems that do not contain other intermediate systems.

MAUD enables a small and simple interface that captures all operations required from any intermediate system. Thus, an intermediate system defined by an index set S is given by

$$R_S(p_S, u_S) = 0, \quad (37)$$

where u_S is the unknown vector, p_S is the parameter vector, and R_S is the residual function. We assume that there is an implicit function that computes the unknown vector in terms of the parameter vector. Therefore, we can view p_S as the inputs and u_S as the outputs of this system. For a given intermediate system $R_S(p_S, u_S) = 0$, the interface consists of the following operations (where we omit the subscript S for brevity):

apply_nonlinear: $(p, u) \mapsto r = R(p, u)$.

This operation computes the residual functions for the current intermediate system. It is used to compute the right-hand side of the linear system in Newton's method and to check the convergence of the current intermediate system.

solve_nonlinear: $p \mapsto u$.

This operation solves $R(p, u) = 0$ inexactly or to the required convergence tolerance. It is optional to implement this method, since Newton's method is used when the user does not provide a custom solver.

$$\text{apply_linear: } \begin{cases} (dp, du) \mapsto dr = \frac{\partial R}{\partial p} dp + \frac{\partial R}{\partial u} du, & \text{forward mode} \\ dr \mapsto \left(dp = \frac{\partial R^T}{\partial p} dr, du = \frac{\partial R^T}{\partial u} dr \right), & \text{reverse mode.} \end{cases}$$

This operation allows the component to implicitly provide Jacobians $\partial R / \partial(p, u)$ to MAUD's solver. By implementing this as a linear operator, the user does not need to specify how each component stores or computes the Jacobian. This constitutes a single, simple interface that can handle sparse or dense Jacobians, as well as Jacobian-free components. Either the forward or reverse modes can be used for computing the derivatives.

$$\text{solve_linear: } \begin{cases} (dr) \mapsto du \left| \frac{\partial R}{\partial u} du = dr \right., & \text{forward mode} \\ (du) \mapsto dr \left| \frac{\partial R^T}{\partial u} dr = du \right., & \text{reverse mode.} \end{cases}$$

This operation implements the inverse of $\partial R / \partial u$ as a linear operator. Like **solve_nonlinear**, it is optional, since the framework uses a Krylov iterative solver if **apply_linear** is known.

linearize: This operation is part of the interface because many problems require an initial assembly (and in some cases factorization) of the Jacobian, and repeated applications of the matrix or factorization have a substantially lower cost.

The software design for this hierarchical algorithm is naturally object-oriented, with an instance of a **System** class created for each intermediate system. The lowest-level intermediate systems—the leaves in the tree—are instances of the **ElementarySystem** class, while all other intermediate systems are instances of the **CompoundSystem** class, because they contain other **System** instances. In other words, the **ElementarySystem** objects are grouped together by **CompoundSystem** objects, which can in turn be grouped by other **CompoundSystem** objects. Figure 6 illustrates the containment relationships between the **ElementarySystem** class, **CompoundSystem** class, and variables for a general computational model.

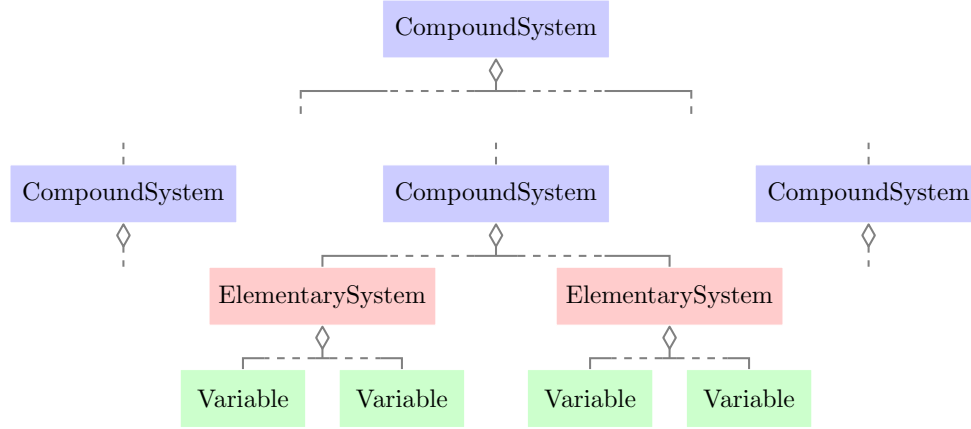


Figure 6: Class containment diagram showing the relationships between objects in a computational model implemented in the MAUD architecture.

The **ElementarySystem** and **CompoundSystem** classes inherit from a base **System** class. The **ElementarySystem** class has three derived classes, reflecting whether the system contains independent, explicitly defined, or implicitly defined variables. The user implements each component in MAUD as a class inheriting from one of these three classes and directly implements operations such as computing the residual function value. Each component contains a subset of the variables, which form the unknown vector in the component’s intermediate system. During initialization, the **ElementarySystem** object must declare its variables as well as its arguments—external variables that depend on the **ElementarySystem** variables.

In contrast, the **CompoundSystem** class is meant to group other **System** objects together, so their operations recursively call those of their children. Moreover, they perform transfer of data potentially distributed across multiple processors. The **CompoundSystem** class has two derived classes that handle parallelism in different ways. In the hierarchy tree in Fig. 6, the root **CompoundSystem** is stored and run on all processors running the job. If a given system object is a **SerialSystem**, it passes all of its processors to the **System** objects it contains and runs its recursive operations sequentially among the contained systems. If a system object is a **ParallelSystem**, it partitions its group of processors among the **System** objects it contains and runs its recursive operations concurrently among the contained systems. The same applies to all **CompoundSystem** objects, and in this manner, each **System** object is assigned a subset of or all of its containing **System** object’s processors. The MAUD class inheritance diagram is shown in Fig. 7. This object-oriented design allows all of MAUD’s

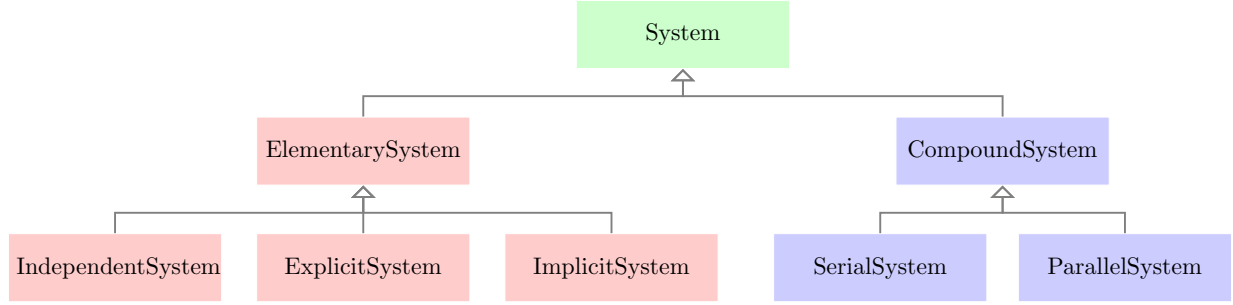


Figure 7: Class inheritance diagram showing the relationships between the **System** classes.

built-in solvers to be implemented as methods of one of the base system classes. Table 1 shows the classes where each built-in solver is implemented.

Table 1: MAUD’s four elementary operations and their implementations in each type of **System** class.

		System classes		
		System	CompoundSystem	ElementarySystem
Virtual methods	apply_nonlinear	—	Recursive	User-implemented
	apply_linear	—	Recursive	User-implemented or FD*
	solve_nonlinear	Newton with line search	Nonlinear block Gauss–Seidel/Jacobi	Optional
	solve_linear	Krylov-type with preconditioning	Linear block Gauss–Seidel/Jacobi	Optional

*FD: finite-difference approximation of the Jacobian.

4.3 Data structures

Efficient data structures are necessary to avoid memory and computing overhead in problems with large numbers of unknowns. For each system, MAUD stores six vectors: u , p , and f for the nonlinear problem and du , dp , and df for the linear problem. The latter three can be interpreted as buffers that contain the data for the solution vector or the right-hand side vector, depending on the situation. Among these six vectors, u , du , f , and df are instances of the **UnknownVec** class, while p and dp are instances of the **ParameterVec** class.

For the **UnknownVec** instances, data is shared with contained or containing systems; that is, the full u , du , f , and df vectors are allocated in the top-level **CompoundSystem**, and all other systems store pointers onto sub-vectors of the global vector. Compared to allocating separate vectors in each system, this approach saves memory as well as computation time, since the subsystems operate directly on sub-vectors of the larger system’s vector A

`ParameterVec` stores only the variables and components that an `ElementarySystem` declares. For illustration, Fig. 8 shows how the u and p vectors are stored. MAUD automates parallel data transfer between `UnknownVec` and `ParameterVec` instances.

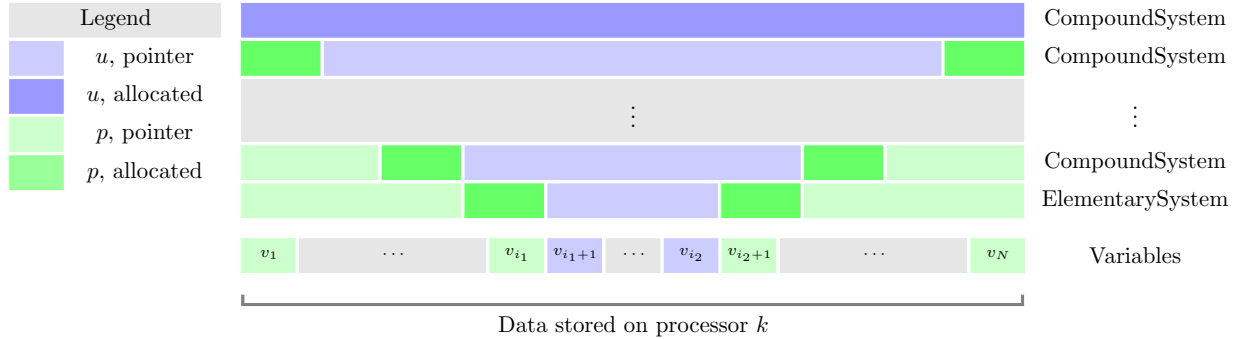


Figure 8: Data storage for the u and p vectors for an `ElementarySystem` object and all of the `CompoundSystem` instances above it in the hierarchy tree.

4.4 Implementation

A minimalistic framework based on the MAUD architecture has been implemented using the Python programming language. This implementation depends only on the `NumPy` package for handling local vectors and on the `petsc4py` package to use the portable, extensible toolkit for scientific computation (PETSc) (Balay et al., 1997). PETSc is used for all parallel data transfers, and its Krylov iterative methods are used as MAUD’s linear solvers, with flexible generalized minimal residual (fGMRES) as the default solver. The entire implementation is contained in a single Python file with about one thousand lines of code, thanks to the MAUD architecture’s monolithic mathematical formulation and the use of PETSc.

The MAUD architecture has been adopted as the algorithmic core of NASA’s OpenMDAO framework (Gray et al., 2013). OpenMDAO is an open-source computational framework designed to facilitate the solution of multidisciplinary optimization problems. Since it uses MAUD, OpenMDAO provides coupled adjoint-based derivative computation, as one of its key features. The mission analysis and allocation models described in the next section have also been implemented in OpenMDAO.

5 Application to aircraft design

We now present an application of MAUD to the large-scale optimization of an aircraft configuration where the airline allocation, mission profile, and aircraft design are all optimized simultaneously.

5.1 Motivation

Typically, when performing aircraft design optimization, only a limited number of flight conditions is considered. For instance, aerodynamic shape optimization using CFD might be run at a fixed, pre-determined set of lift coefficients that represent typical values of similarly-sized aircraft during actual operation (Kenway and Martins, 2015). Similarly, aerostructural

optimization that couples CFD and FEA designs the aerodynamic shape and structural sizing simultaneously, while considering fuel burn for a limited number of flight conditions (Kenway and Martins, 2014; Liem et al., 2015).

Herein we aim to widen the scope of the high-fidelity optimization to include some of the requirement-definition part of conceptual design. Instead of optimizing the aircraft design for a set of representative missions—or a design range, design Mach number, and payload—we let the algorithm choose on which routes in a network it is optimal to fly this next-generation aircraft, and which cruise Mach numbers are optimal on those routes. Thus, we view the problem from the airline’s perspective, since the airlines are the customers who define the requirements in the conceptual-design sense.

With this formulation, the objective function is the profit margin for the airline. We assume an airline network—a set of routes that the airline operates—as well as a finite-sized fleet of existing and next-generation aircraft that the airline can deploy on those routes. The optimizer chooses how to allocate the fleet of aircraft, selects the optimal mission profile for each aircraft flying on each route, and designs the next-generation aircraft, all at the same time. Thus, we call this formulation *allocation-mission-design optimization* (AMD), since all three aspect are optimized simultaneously within a single problem. In this approach, the effective design range and payload for the next-generation aircraft is implicitly determined through optimization. The optimizer has the ability to design a smaller aircraft that is particularly efficient for short-range missions, or it could size the aircraft for long-range missions and cover the short-range routes for which it is over-designed using existing aircraft.

This section presents the first implementation and solution of the AMD optimization problem (Hwang and Martins, 2016). For the design component, we do aerodynamic shape optimization based on 3D Euler CFD. In the mission component, we optimize the shape of the altitude profiles for each mission, and we also make the cruise Mach number for each mission a design variable with a linear Mach number variation during climb and descent. For the allocation, we consider a hypothetical 128-route network with fixed passenger demand on each route and a fleet consisting of 4 existing types of aircraft in addition to the simultaneous design of the next-generation aircraft.

5.2 Computational models

The software components that are integrated to solve the AMD optimization problem include: a CFD mesh deformation algorithm, a CFD solver, a mission analysis model, an aircraft allocation model, and an optimizer.

The CFD mesh deformation algorithm we use propagates the displacements and rotations from the deformed surface to the full CFD volume mesh by using inverse-distance weights (Uyttensprot, 2014).

The CFD solver is SUMad, a structured multi-block finite-volume solver with multigrid (van der Weide et al., 2006). SUMad includes an adjoint implementation using reverse-mode automatic differentiation (Lyu et al., 2013). SUMad solves the Reynolds-averaged Navier–Stokes (RANS) equations or the Euler equations in parallel using the fourth-order Runge–Kutta scheme or the diagonally dominant alternating direction implicit (DDADI) scheme, combined with the Newton–Krylov method. Herein, we solve the Euler equations using SUMad to reduce computation time.

The mission analysis tool discretizes the full mission profile using a collocation approach and solves the vertical and horizontal equilibrium equations, the trim condition, and the ordinary differential equation for fuel weight (Kao et al., 2015). The equations are all coupled to the aircraft performance model, which is represented using a surrogate model for lift and drag coefficients as functions of Mach number and angle of attack. The mission analysis takes the altitude and Mach number profiles as inputs, parametrized using B-splines, and its outputs are fuel burn, block time, and aggregated values for idle and maximum thrust constraints.

There are two approaches used for mission profile optimization: direct and indirect (Betts, 1998). The direct approach applies the optimality conditions after discretizing the equilibrium equations, while the indirect approach differentiates the equilibrium equations first and then discretizes them. Our mission analysis tool takes the direct approach, as it is more amenable to coupling with other disciplines.

One of the advantages of optimizing the altitude profiles is that we are able to consider continuous descent approach, also known as optimized descent profile. Optimized descent is part of the FAA’s next generation air transportation system (NextGen), currently in development to achieve reductions in noise and fuel burn. To this end, continuous descent tests have been conducted recently at several airports (Clarke et al., 2013; Micallef et al., 2014; Shresta et al., 2009).

Figure 9 lists all the variables in the mission analysis, including the shapes of the Jacobians in each connection between variables. The variables are organized into five groups, also known as *assemblies*, both for convenience and to take advantage of the hierarchical solution algorithm in MAUD. We use nonlinear block Gauss–Seidel across the 5 groups and within all but the **CoupledAnalysis** assembly, which internally uses a Newton solver to resolve the coupling.

The inputs are the B-spline control points for the altitudes and their corresponding horizontal positions. The resolution depends on the mission range; typically we use between 10 and 50 control points, with 100 to 250 discretization points. The **Bsplines** assembly then computes the actual values of the altitudes, horizontal coordinates, and slopes. The **AtmosphericProperties** assembly parametrizes the Mach number, temperature, pressure, density, and airspeed in terms of the altitude. The thrust-specific fuel consumption is also a simple function of altitude, but more elaborate models can be used in future work.

The **CoupledAnalysis** assembly contains the core of the mission analysis model. Given an initial guess for the fuel weight at every point in the mission, it applies the vertical equilibrium equation to solve for the required lift coefficient using

$$C_L = \frac{W \cos \gamma}{\frac{1}{2} \rho v^2 S} - C_T \sin \alpha. \quad (38)$$

Next, the surrogate model for C_L is used to solve an implicit function to determine the angle of attack that produces this target C_L with the residual,

$$\tilde{C}_L(\alpha, M) - C_L = 0, \quad (39)$$

where $\tilde{C}_L$ is the surrogate model. The surrogate model for drag coefficient, $\tilde{C}_D$, is then evaluated using the computed angle of attack to determine the resulting drag coefficient

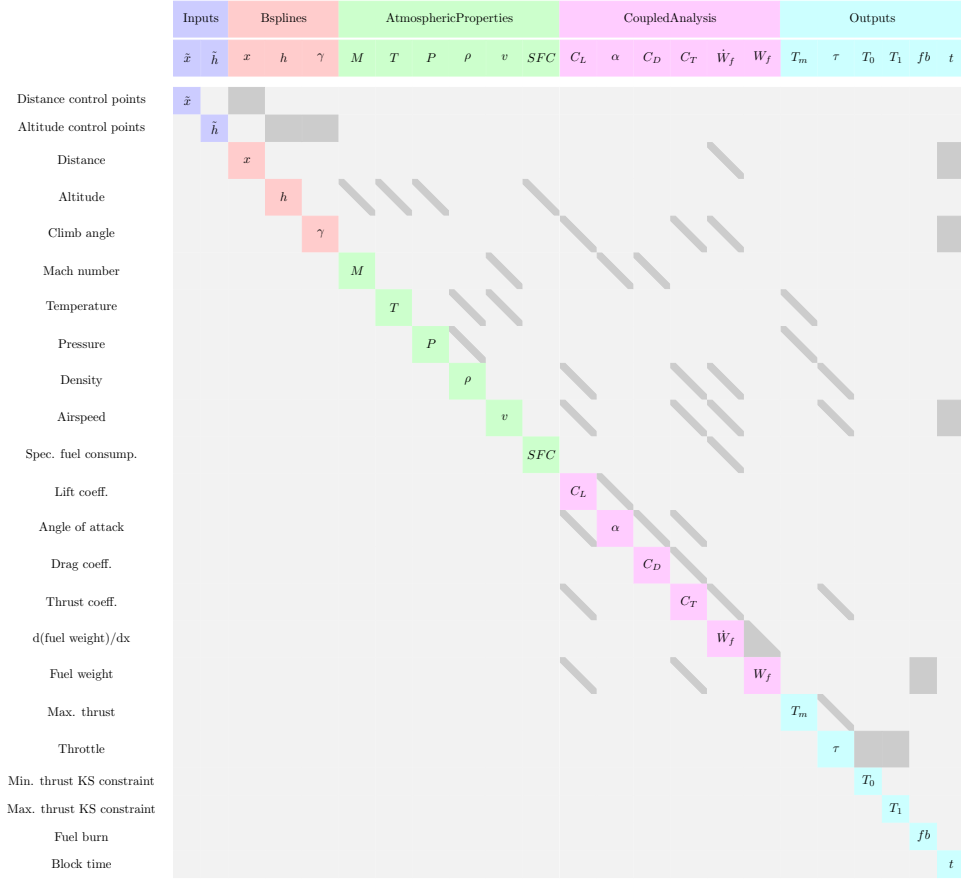


Figure 9: Dependency graph for the variables in a mission analysis.

from

$$C_D = \tilde{C}_D(\alpha, M). \quad (40)$$

Then, the horizontal equilibrium equation is used to compute the thrust coefficient from

$$C_T = \frac{C_D}{\cos \alpha} + \frac{W \sin \gamma}{\frac{1}{2} \rho v^2 S \cos \alpha}. \quad (41)$$

Finally, the ordinary differential equation for the fuel weight is given by

$$\dot{W}_f = SFC \frac{C_T \frac{1}{2} \rho v^2 S}{v \cos \gamma}. \quad (42)$$

This must be integrated backwards from the last mission point that has a small amount of reserve fuel, and this produces a new estimate of aircraft weight that requires us to iterate back to the first step of the **CoupledAnalysis** assembly.

The **Outputs** component performs a series of postprocessing computations, including the maximum thrust estimate at altitude, given by

$$T_m = T_{m,SL} \frac{P}{P_{SL}} \sqrt{\frac{T_{SL}}{T}}, \quad (43)$$

where the maximum thrust is expressed in terms of pressure and temperature, and the respective sea-level values. There is no guarantee that the engine is sized to fly the mission as evaluated by the **CoupledAnalysis** assembly; thus, we define nonlinear constraints that the throttle setting is between idle and maximum, and let the optimizer make sure this is satisfied. However, these thrust constraints must be satisfied at every point in the mission, so we aggregate them using the Kreisselmeier–Steinhauser function (Kreisselmeier and Steinhauser, 1979; Poon and Martins, 2007). The two outputs of the mission analysis that are of interest in the allocation problem are the fuel burn and block time, which are computed at the end.

The allocation problem is formulated with two sets of design variables: the number of flights per day flown by a type of aircraft on a given route, and the number of passengers per flight for a given type of aircraft and a given route. Here, the formulation maximizes profit subject the operational and demand constraints (Hwang et al., 2015). The airline profit is evaluated using estimates for ticket price and operating costs, using

$$\begin{aligned} \text{profit} = & \sum_i^{n_{rt}} \sum_j^{n_{ac}} \left[\text{price_pax}_{i,j} \cdot \text{pax_flt}_{i,j} \cdot \text{flt_day}_{i,j} \right] \\ & - \sum_i^{n_{rt}} \sum_j^{n_{ac}} \left[(\text{cost_flt}_{i,j} + \text{cost_fuel} \cdot \text{fuel_flt}_{i,j}) \cdot \text{flt_day}_{i,j} \right], \end{aligned} \quad (44)$$

where $\text{price_pax}_{i,j}$ is the ticket price per flight, $\text{cost_flt}_{i,j}$ is the total cost of operating a flight minus fuel, cost_fuel is the cost per unit fuel, and fuel_flt is the total fuel burn on a flight.

The allocation problem contributes two inequality constraints to the allocation-mission optimization. The first constraint ensures that the total number of people that fly on a given route on a given day is less than the total demand for that route, and it is expressed as

$$\text{total_pax}_i = \sum_j^{n_{ac}} \left[\text{pax_flt}_{i,j} \cdot \text{flt_day}_{i,j} \right] \leq \text{demand}_i, \quad 1 \leq i \leq n_{rt}. \quad (45)$$

The second constraint takes into consideration how many aircraft of a given type are actually owned by the airline, and it is given by

$$\text{usage}_j = \sum_i^{n_{rt}} \left[\text{flt_day}_{i,j} \cdot (\text{time_flt}_{i,j}(1 + \text{maint}_j) + \text{turn_flt}) \right] \leq 12\text{hr} \cdot \text{num_ac}_j, \quad 1 \leq j \leq n_{ac}, \quad (46)$$

where $\text{time_flt}_{i,j}$ is the block time for a flight, maint_j is the maintenance time required as a multiple of block time, turn_flt is the turnaround time between flights, and num_ac_j is the number of aircraft available for a given type.

The gradient-based optimizer we use to solve this problem is SNOPT (Gill et al., 2002), through the pyOpt Python interface (Perez et al., 2012). SNOPT is an active-set sequential quadratic programming (SQP) algorithm designed to handle sparse nonlinear constrained optimization problems, so it is well-suited to the current application. In addition to the thousands of design variables, the AMD optimization also has tens of thousands of constraints, mostly from the 128 mission analysis, but the tens of thousands by thousands-sized Jacobian is sparse.

5.3 Problem description

We now describe the wing geometry and parametrization, the Mach number parametrization, and the optimization problem.

The geometry that we analyze is a swept, linearly-tapered wing that based on a scaled version of the Boeing 717 wing. Figure 10 shows the result of a CFD analysis of the wing using the Euler equations at a Mach number of 0.78 and an angle of attack of 3° . Since we model only the wing, the additional sources of drag (such as fuselage drag, trim drag, and viscous drag) are accounted for by subtracting the computed drag at a nominal condition from the expected drag for an aircraft of this size.

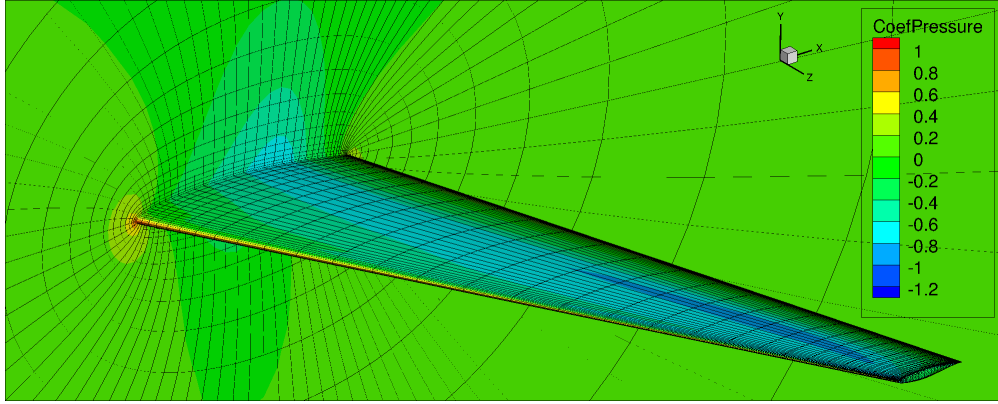


Figure 10: Euler analysis result of the geometry at $M = 0.78$ and $\alpha = 0.3^\circ$.

The full AMD optimization problem is summarized in Table 2. There are 6 twist and 72 shape design variables (shown in Fig. 11) with a minimum wing volume constraint and a 10×10 grid of points on the wing at which a minimum thickness is enforced. Each mission

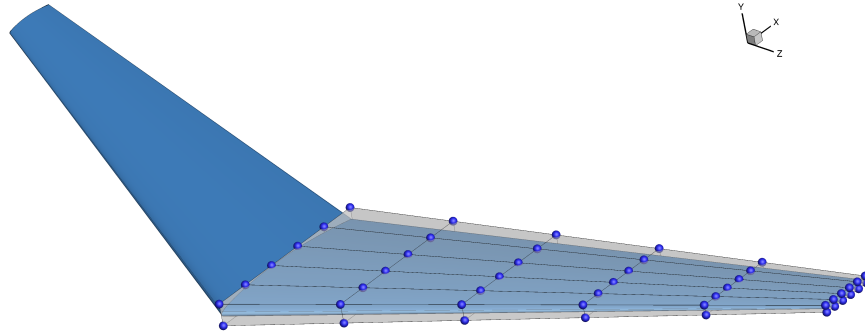


Figure 11: Geometry parametrization using free-form deformation with a tensor-product B-spline volume.

analysis contributes a cruise Mach number design variable and altitude control points, the number of which varies depending on the mission range. The mission analyses also contribute minimum and maximum thrust constraints aggregated using the Kreisselmeier–Steinhauser function and linear constraints on the maximum climb and descent angle. The allocation problem adds passengers per flight and flights per day design variables for each type of

	Variable	Quantity
maximize	profit	
with respect to	twist	6
	shape	72
	cruise Mach number for each route (between 0.6 and 0.82)	1×128
	altitude control points for each route	4,575
	passengers per flight for each aircraft type and route	5×128
	flights per day for each aircraft type and route	5×128
	Total number of design variables	6,061
subject to	wing volume constraint	1
	wing thickness constraints	100
	idle thrust KS constraint for each route	128
	maximum thrust KS constraint for each route	128
	linear climb angle bounds for each mission	22,875
	demand constraint for each route	128
	total flight time constraint for each aircraft	5
	Total number of constraints	23,365

Table 2: The 128-route allocation-mission-design optimization problem.

existing or new aircraft, for each of the 128 routes. There is also a demand constraint that limits the total number of passengers traveling on a route in a given day, and an aircraft utilization constraint that takes into account the finite number of aircraft of each type that the airline owns.

5.4 Results

The main question in this work is whether the AMD optimization is significantly different from the conventional design-only optimization. First, we consider the twist and lift distributions, shown in Fig. 12. In both the AMD and design optimization cases, we observe that the lift distributions are close to elliptical in both cases.

Figure 13 shows slices of the wing, where we can see differences between the AMD and the conventional design results. We observe that they are both thicker overall and have thicker leading and trailing edges when compared to the initial design, though this result would likely be different if we were capturing viscous effects with the RANS equations (Lyu et al., 2015). The AMD and design-only results are sufficiently different to demonstrate that the coupling with allocation and mission analysis is not negligible.

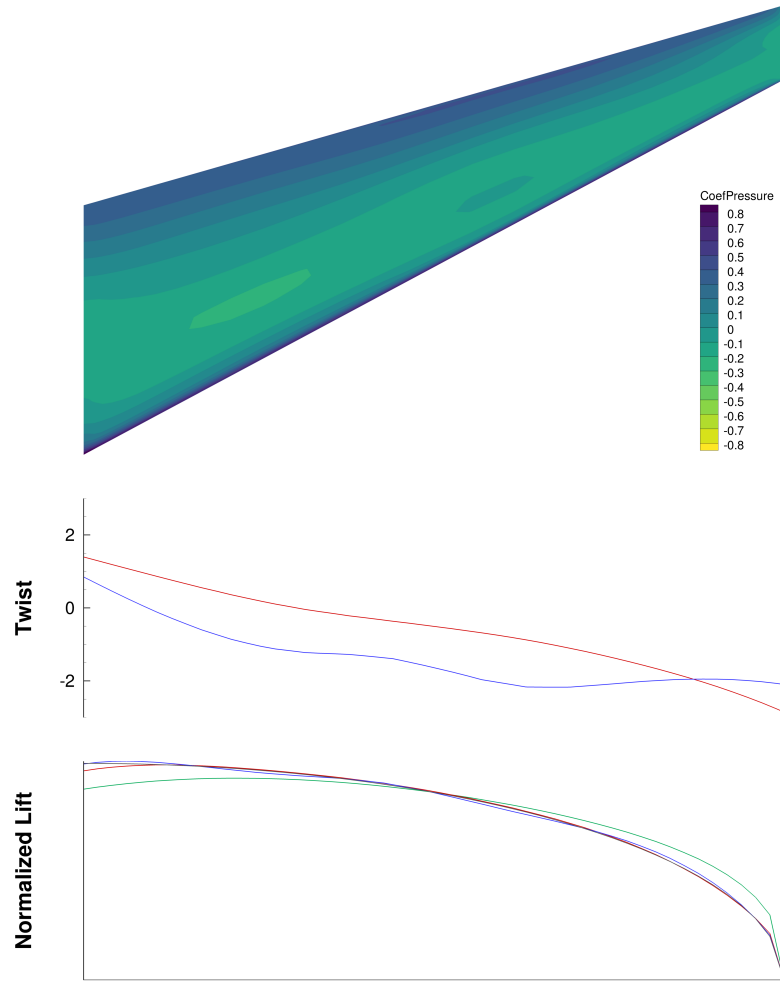


Figure 12: Planform view, optimal twist distribution, and optimized lift distribution; initial design (green), AMD optimization (blue), design optimization (red), elliptical lift distribution (gray).

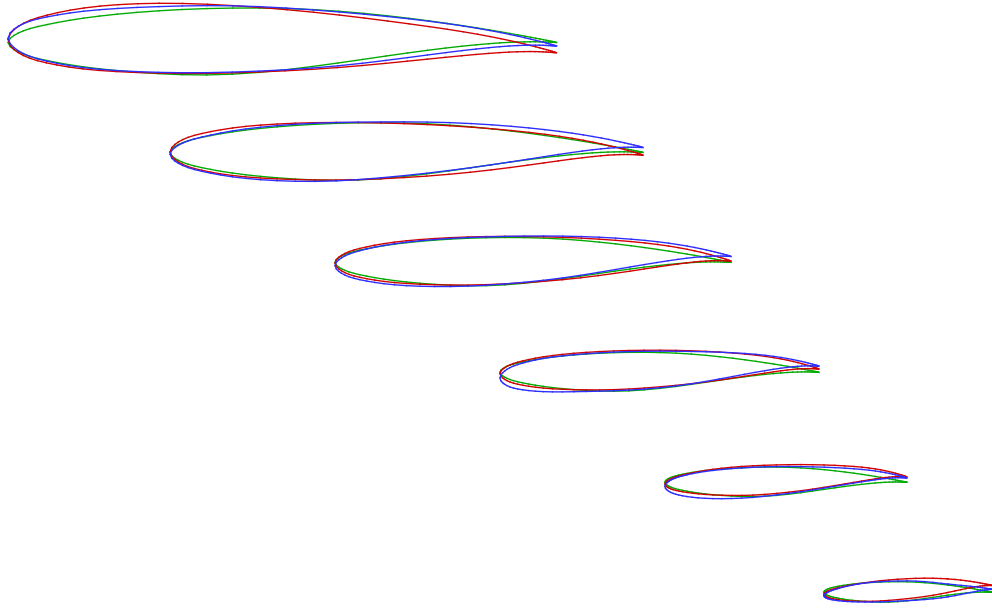


Figure 13: Airfoil shapes for the initial design (green), AMD optimization (blue), and design optimization result (red).

As we can see in Fig. 14, there is a 27% increase in airline profit per day going from the allocation-only optimization to the AMD optimization. The result from the allocation-only optimization yields a daily profit of about \$23.4 million, and the AMD-optimized result has a profit of \$29.8 million.

We can see in Fig. 15 how the allocation-only and AMD optimization results differ. The most noteworthy difference between the two is that the AMD optimization flies the next-generation aircraft on the shorter-range routes much more. We see more directly in Fig. 16 that the next-generation aircraft enjoys the largest share of the total number of passengers flown per day, but this share increases in AMD optimization, as one would expect since the design of the next-generation aircraft improves in the AMD optimization.

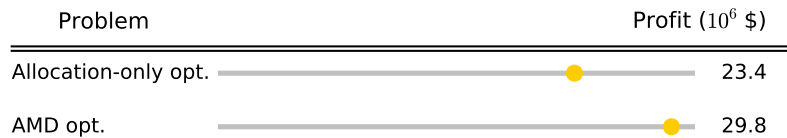


Figure 14: The AMD optimization produces 27% more profit than the allocation-only optimization.

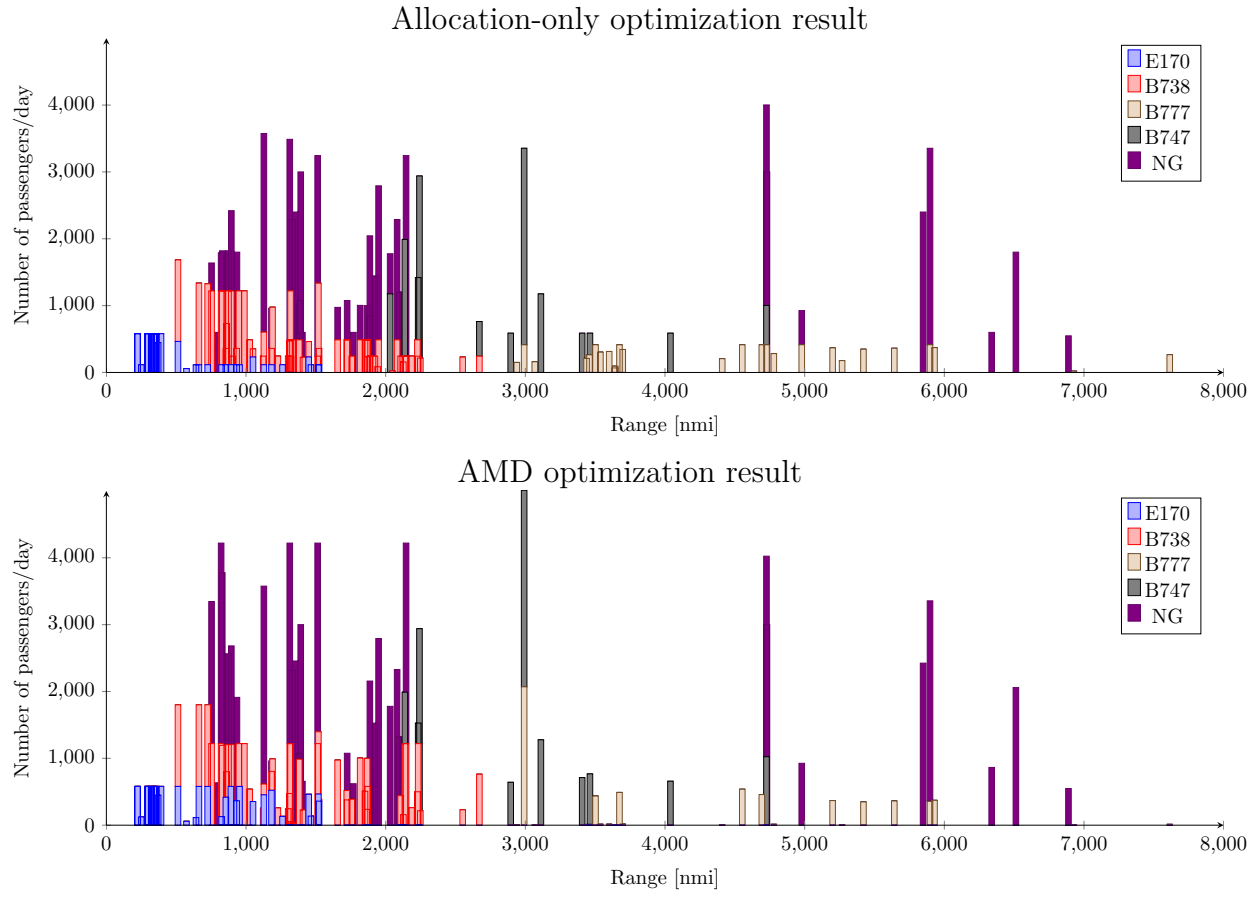


Figure 15: Total number of passengers flown per day for each of the 128 missions shown according to their range.

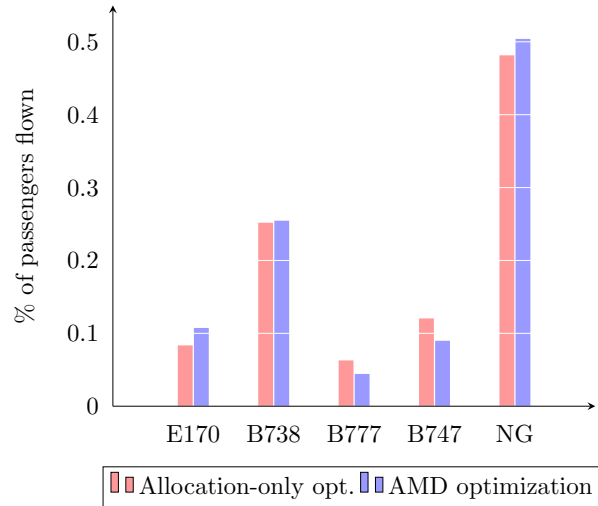


Figure 16: Percentage of the passengers flown on each type of aircraft.

6 Summary

In Part 1 of this lecture we presented a novel way to couple multiple heterogeneous computational methods efficiently—the MAUD architecture. We started by arguing that to solve large-scale aircraft design optimization, we require gradient-based optimization with an efficient way to compute the gradients. We reviewed the various options for computing the gradients and unify these methods using the unifying chain rule equation (33). The equation also applies to multidisciplinary systems, enabling the efficient computation of derivatives of coupled systems. The unifying chain rule leads to the development of MAUD, a framework for modular analysis and derivative computation. We use partitioning using a hierarchical strategy that speeds up the solution using parallel computing. The nonlinear solvers used in MAUD are Newton’s method with a line search, inexact nonlinear block Gauss–Seidel or Jacobi, or any problem-specific user-provided solver. The linear solvers are a Krylov iterative method with variable preconditioning, inexact linear block Gauss–Seidel or Jacobi, or any problem-specific user-provided solver. These nonlinear and linear solvers can be executed at any level of a hierarchical decomposition of the system of equations.

MAUD’s key features can be summarized as follows. First, its parallel data passing greatly facilitates distributed-memory parallel computation. It allows components that have parallel vectors as arguments to not be aware of how those parallel vectors are distributed across their processors. Second, MAUD automatically solves the nonlinear and linear systems that arise. When the user provides a custom solver, the framework uses it; otherwise, it uses its built-in solvers. The third benefit is the automated computation of derivatives given partial derivatives of each component. MAUD uses the analytic methods for computing derivatives, which can compute a full gradient with respect to all inputs at a cost that has the same order of magnitude as that of running the simulation.

MAUD can be applied to any problem that deterministically computes a set of variables as a continuous function of others. The coupled derivative computation provides a significant increase in efficiency when there are implicit state variables and when there is a feedback loop among the dependencies between variables. The core of MAUD has been implemented in OpenMDAO (Heath and Gray, 2012), an open-source computational framework developed by NASA.

To demonstrate the MAUD approach, we solved an aircraft design problem where allocation, mission, and design variables are optimized simultaneously. The optimization problem sought to maximize airline profit by designing a next-generation aircraft and solving an allocation problem to decide how to allocate this aircraft and a fleet of existing aircraft to a 128-route network. We showed that the AMD approach resulted in a 27% increase in airline profit relative to the conventional allocation-only optimization.

In Part 2 of this lecture, we also use gradient-based optimization with efficient methods for computing multidisciplinary derivatives, focusing on the coupling between high-fidelity aerodynamic and structural solvers, and corresponding wing design optimization results.

References

Anderson, W. K., Newman, J. C., Whitfield, D. L., and Nielsen, E. J. (2001). Sensitivity analysis for Navier–Stokes equations on unstructured meshes using complex variables. 39(1):56–63. doi:10.2514/2.1270.

- Anderson Jr., J. D. (1999). *Aircraft Performance and Design*. McGraw–Hill.
- Ashley, H. (1982). On making things the best — aeronautical uses of optimization. *Journal of Aircraft*, 19(1):5–28.
- Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. (1997). *Efficient Management of Parallelism in Object Oriented Numerical Software Libraries*, pages 163–202. Birkhäuser Press.
- Betts, J. T. (1998). Survey of numerical methods for trajectory optimization. *Journal of guidance, control, and dynamics*, 21(2):193–207.
- Carle, A. and Fagan, M. (2000). ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University.
- Clarke, J.-P., Brooks, J., Nagle, G., Scacchioli, A., White, W., and Liu, S. (2013). Optimized profile descent arrivals at los angeles international airport. *Journal of Aircraft*, 50(2):360–369.
- Dwight, R. P. and Brezillon, J. (2006). Effect of approximations of the discrete adjoint on gradient-based optimization. *AIAA Journal*, 44(12):3022–3031.
- Giering, R. and Kaminski, T. (2002). Applying TAF to generate efficient derivative code of Fortran 77-95 programs. In *Proceedings of GAMM 2002, Augsburg, Germany*.
- Giles, M. B. and Pierce, N. A. (2000). An introduction to the adjoint approach to design. *Flow, Turbulence and Combustion*, 65:393–415.
- Gill, P. E., Murray, W., and Saunders, M. A. (2002). SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Journal of Optimization*, 12(4):979–1006. doi:10.1137/S1052623499350013.
- Gockenbach, M. S. (2000). Understanding Code Generated by TAMC. IAAA Paper TR00-29, Department of Computational and Applied Mathematics, Rice University, Texas, USA.
- Gray, J., Moore, K. T., Hearn, T. A., and Naylor, B. A. (2013). Standard platform for benchmarking multidisciplinary design analysis and optimization architectures. *AIAA Journal*, 51(10):2380–2394. doi:10.2514/1.J052160.
- Green, J. A. (1987). Aeroelastic tailoring of aft-swept high-aspect-ratio composite wings. *Journal of Aircraft*, 24(11):812–819. doi:10.2514/3.45525.
- Griewank, A. (2000). *Evaluating Derivatives*. SIAM, Philadelphia.
- Grossman, B., Gurdal, Z., Strauch, G. J., Eppard, W. M., and Haftka, R. T. (1988). Integrated aerodynamic/structural design of a sailplane wing. *Journal of Aircraft*, 25(9):855–860. doi:10.2514/3.45670.

- Grossman, B., Haftka, R. T., Kao, P.-J., Polen, D. M., and Rais-Rohani, M. (1990). Integrated aerodynamic-structural design of a transport wing. *Journal of Aircraft*, 27(12):1050–1056. doi:10.2514/3.45980.
- Haftka, R. T. (1977). Optimization of flexible wing structures subject to strength and induced drag constraints. *AIAA Journal*, 14(8):1106–1177. doi:10.2514/3.7400.
- Haftka, R. T. and Shore, C. P. (1979). Approximate methods for combined thermal/structural design. Technical Report TP-1428, NASA.
- Hascoët, L. and Pascual, V. (2004). Tapenade 2.1 user’s guide. Technical report 300, INRIA.
- Heath, C. and Gray, J. (2012). OpenMDAO: Framework for flexible multidisciplinary design, analysis and optimization methods. In *Proceedings of the 53rd AIAA Structures, Structural Dynamics and Materials Conference*, Honolulu, HI. AIAA-2012-1673.
- Hwang, J. T. and Martins, J. R. R. A. (2016). Allocation-mission-design optimization of next-generation aircraft using a parallel computational framework. In *57th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*. American Institute of Aeronautics and Astronautics.
- Hwang, J. T., Roy, S., Kao, J. Y., Martins, J. R. R. A., and Crossley, W. A. (2015). Simultaneous aircraft allocation and mission optimization using a modular adjoint approach. In *Proceedings of the 56th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, Kissimmee, FL. AIAA 2015-0900.
- Jameson, A. (1988). Aerodynamic design via control theory. *Journal of Scientific Computing*, 3(3):233–260.
- Jansen, P., Perez, R. E., and Martins, J. R. R. A. (2010). Aerostructural optimization of nonplanar lifting surfaces. *Journal of Aircraft*, 47(5):1491–1503. doi:10.2514/1.44727.
- Kao, J. Y., Hwang, J. T., Martins, J. R. R. A., Gray, J. S., and Moore, K. T. (2015). A modular adjoint approach to aircraft mission analysis and optimization. In *Proceedings of the AIAA Science and Technology Forum and Exposition (SciTech)*, Kissimmee, FL. AIAA 2015-0136.
- Kenway, G. K. W., Kennedy, G. J., and Martins, J. R. R. A. (2012). A scalable parallel approach for high-fidelity aerostructural analysis and optimization. In *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Honolulu, HI. AIAA 2012-1922.
- Kenway, G. K. W., Kennedy, G. J., and Martins, J. R. R. A. (2014). Scalable parallel approach for high-fidelity steady-state aeroelastic analysis and derivative computations. *AIAA Journal*, 52(5):935–951. doi:10.2514/1.J052255.
- Kenway, G. K. W. and Martins, J. R. R. A. (2014). Multipoint high-fidelity aerostructural optimization of a transport aircraft configuration. *Journal of Aircraft*, 51(1):144–160. doi:10.2514/1.C032150.

- Kenway, G. K. W. and Martins, J. R. R. A. (2015). AIAA ADODG Case 5: CRM wing-body-tail optimization at flight reynolds number. Technical report, AIAA.
- Kreisselmeier, G. and Steinhauser, R. (1979). Systematic control design by optimizing a vector performance index. In *International Federation of Active Controls Symposium on Computer-Aided Design of Control Systems, Zurich, Switzerland*.
- Kroo, I. M., Altus, S., Braun, R. D., Gage, P., and Sobieski, I. P. (1994). Multidisciplinary optimization methods for aircraft preliminary design. In *5th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*. AIAA Paper 94-4325.
- Liem, R., Kenway, G. K. W., and Martins, J. R. R. A. (2015). Multimission aircraft fuel burn minimization via multipoint aerostructural optimization. *AIAA Journal*, 53(1):104–122. doi:10.2514/1.J052940.
- Liu, S. and Canfield, R. A. (2012). Continuum sensitivity method for aeroelastic shape design problems. In *Proceedings of the 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Indianapolis, IN. AIAA 2012-5480.
- Lyness, J. N. and Moler, C. B. (1967). Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis*, 4(2):202–210.
- Lyu, Z., Kenway, G. K., and Martins, J. R. R. A. (2015). Aerodynamic shape optimization investigations of the Common Research Model wing benchmark. *AIAA Journal*, 53(4):968–985. doi:10.2514/1.J053318.
- Lyu, Z., Kenway, G. K., Paige, C., and Martins, J. R. R. A. (2013). Automatic differentiation adjoint of the Reynolds-averaged Navier–Stokes equations with a turbulence model. In *21st AIAA Computational Fluid Dynamics Conference*, San Diego, CA.
- Lyu, Z. and Martins, J. R. R. A. (2013). Aerodynamic shape optimization of a blended-wing-body aircraft. In *Proceedings of the 51st AIAA Aerospace Sciences Meeting*, Grapevine, TX. AIAA 2013-0283.
- Mader, C. A., Martins, J. R. R. A., Alonso, J. J., and van der Weide, E. (2008). ADjoint: An approach for the rapid development of discrete adjoint solvers. *AIAA Journal*, 46(4):863–873. doi:10.2514/1.29123.
- Martins, J. R. R. A., Alonso, J. J., and Reuther, J. J. (2004). High-fidelity aerostructural design optimization of a supersonic business jet. *Journal of Aircraft*, 41(3):523–530. doi:10.2514/1.11478.
- Martins, J. R. R. A., Alonso, J. J., and Reuther, J. J. (2005). A coupled-adjoint sensitivity analysis method for high-fidelity aero-structural design. *Optimization and Engineering*, 6(1):33–62. doi:10.1023/B:OPTE.0000048536.47956.62.
- Martins, J. R. R. A. and Hwang, J. T. (2013). Review and unification of methods for computing derivatives of multidisciplinary computational models. *AIAA Journal*, 51(11):2582–2599. doi:10.2514/1.J052184.

- Martins, J. R. R. A. and Lambe, A. B. (2013). Multidisciplinary design optimization: A survey of architectures. *AIAA Journal*, 51(9):2049–2075. doi:10.2514/1.J051895.
- Martins, J. R. R. A., Sturdza, P., and Alonso, J. J. (2003). The complex-step derivative approximation. *ACM Transactions on Mathematical Software*, 29(3):245–262. doi:10.1145/838250.838251.
- Micallef, M., Chircop, K., Zammit-Mangion, D., and Sammut, A. (2014). Revised approach procedures to support optimal descents into malta international airport. *CEAS Aeronautical Journal*, pages 1–15.
- Nadarajah, S. and Jameson, A. (2000). A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization. In *Proceedings of the 38th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV. AIAA 2000-0667.
- Newman III, J. C., Whitfield, D. L., and Anderson, W. K. (2003). Step-size independent approach for multidisciplinary sensitivity analysis. *Journal of Aircraft*, 40(3):566–573. doi:10.2514/2.3131.
- Pascual, V. and Hascoët, L. (2005). Extension of TAPENADE towards Fortran 95. In Bücker, H. M., Corliss, G., Hovland, P., Naumann, U., and Norris, B., editors, *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Computational Science and Engineering. Springer, Berlin, Germany.
- Perez, R. E., Jansen, P. W., and Martins, J. R. R. A. (2012). pyOpt: A Python-based object-oriented framework for nonlinear constrained optimization. *Structural and Multidisciplinary Optimization*, 45(1):101–118. doi:10.1007/s00158-011-0666-3.
- Peter, J. E. V. and Dwight, R. P. (2010). Numerical sensitivity analysis for aerodynamic optimization: A survey of approaches. *Computers and Fluids*, 39(3):373–391. doi:10.1016/j.compfluid.2009.09.013.
- Pironneau, O. (1974). On optimum design in fluid mechanics. *Journal of Fluid Mechanics*, 64:97–110.
- Poon, N. M. K. and Martins, J. R. R. A. (2007). An adaptive approach to constraint aggregation using adjoint sensitivity analysis. *Structural and Multidisciplinary Optimization*, 34(1):61–73. doi:10.1007/s00158-006-0061-7.
- Pryce, J. D. and Reid, J. K. (1998). AD01, a Fortran 90 code for automatic differentiation. Report RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 0QX, U.K.
- Reuther, J. J., Jameson, A., Alonso, J. J., Rimlinger, M. J., and Saunders, D. (1999a). Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers, part 1. *Journal of Aircraft*, 36(1):51–60.

- Reuther, J. J., Jameson, A., Alonso, J. J., Rimlinger, M. J., and Saunders, D. (1999b). Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers, part 2. *Journal of Aircraft*, 36(1):61–74.
- Schmit, L. A. (1960). Structural Design by Systematic Synthesis. In *2nd Conference on Electronic Computation*, pages 105–132, New York, NY. ASCE.
- Schmit Jr., L., A. (1981). Structural synthesis—its genesis and development. *AIAA Journal*, 19(10):1249–1263. doi:10.2514/3.7859.
- Shrestha, S., Neskovic, D., and Williams, S. (2009). Analysis of continuous descent benefits and impacts during daytime operations. In *8th USA/Europe Air Traffic Management Research and Development Seminar (ATM2009)*, Napa, CA.
- Sirkes, Z. and Tziperman, E. (1997). Finite difference of adjoint or adjoint of finite difference? *Monthly Weather Review*, 125(12):3373–3378.
- Squire, W. and Trapp, G. (1998). Using complex variables to estimate derivatives of real functions. *SIAM Review*, 40(1):110–112.
- Tedford, N. P. and Martins, J. R. R. A. (2010). Benchmarking multidisciplinary design optimization algorithms. *Optimization and Engineering*, 11(1):159–183. doi:10.1007/s11081-009-9082-6.
- Uyttersprot, L. (2014). *Inverse Distance Weighting Mesh Deformation*. PhD thesis, Delft University of Technology.
- van der Weide, E., Kalitzin, G., Schluter, J., and Alonso, J. J. (2006). Unsteady turbomachinery computations using massively parallel platforms. In *Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV. AIAA 2006-0421.
- Vassberg, J. C. and Jameson, A. (2002). Aerodynamic shape optimisation of a Reno race plane. *International Journal of Vehicle Design*, 28(4):318–338.
- Wakayama, S. (1998). *Multidisciplinary design optimization of the blended-wing-body*. American Institute of Aeronautics and Astronautics.